An Introduction to

Gödel's Theorems

Peter Smith

Faculty of Philosophy
University of Cambridge

Version date: March 12, 2005
Copyright: ©2005 Peter Smith
Not to be cited or quoted without permission

The book's website is at www.godelbook.net

Contents

Preface v

1 What Gödel's First Theorem Says 1
- 1.1 Incompleteness and basic arithmetic 1
- 1.2 Why it matters 4
- 1.3 What's next? 5

2 The Idea of an Axiomatized Formal Theory 7
- 2.1 Formalization as an ideal 7
- 2.2 Axiomatized formal theories 9
- 2.3 Decidability 12
- 2.4 Enumerable and effectively enumerable sets 16
- 2.5 More definitions 18
- 2.6 Three simple results 20
- 2.7 Negation complete theories are decidable 21

3 Capturing Numerical Properties 23
- 3.1 Remarks on notation 23
- 3.2 L_A and other languages 24
- 3.3 Expressing numerical properties and relations 26
- 3.4 Case-by-case capturing 28
- 3.5 A note on our jargon 30

4 Sufficiently Strong Arithmetics 31
- 4.1 The idea of a 'sufficiently strong' theory 31
- 4.2 An undecidability theorem 32
- 4.3 An incompleteness theorem 33
- 4.4 The truths of arithmetic can't be axiomatized 34
- 4.5 But what have we really shown? 35

5 Three Formalized Arithmetics 37
- 5.1 BA – Baby Arithmetic 37
- 5.2 Q – Robinson Arithmetic 40
- 5.3 Capturing properties and relations in Q 42
- 5.4 Introducing '$<$' and '$\leq$' into Q 44
- 5.5 Induction and the Induction Schema 44
- 5.6 PA – First-order Peano Arithmetic 46
- 5.7 Is PA consistent? 48

5.8 More theories 50

6 Primitive Recursive Functions 51
6.1 Introducing p.r. functions 51
6.2 Defining the p.r. functions more carefully 52
6.3 Defining p.r. properties and relations 55
6.4 Some more examples 55
6.5 The p.r. functions are computable 59
6.6 Not all computable numerical functions are p.r. 61
6.7 PBA and the idea of p.r. adequacy 63

7 More on Functions and P.R. Adequacy 65
7.1 Extensionality 65
7.2 Expressing and capturing functions 66
7.3 'Capturing as a function' 67
7.4 Two grades of p.r. adequacy 69

8 Gödel's Proof: The Headlines 70
8.1 A very little background 70
8.2 Gödel's proof outlined 72

9 Q is P.R. Adequate 74
9.1 The proof strategy 74
9.2 The idea of a β-function 75
9.3 Filling in a few details 77
9.4 The adequacy theorem refined 78

10 The Arithmetization of Syntax 80
10.1 Gödel numbering 80
10.2 Coding sequences 81
10.3 *prfseq* is p.r. 82
10.4 The idea of diagonalization 82
10.5 *diag* and *gdl* are p.r. 83
10.6 Gödel's proof that *prfseq* is p.r. 84
10.7 Proving that *diag* is p.r. 89

11 The First Incompleteness Theorem 90
11.1 Constructing G 90
11.2 Interpreting G 91
11.3 G is unprovable in PA: the semantic argument 91
11.4 'G is of Goldbach type' 92
11.5 G is unprovable in PA: the syntactic argument 92
11.6 Gödel's First Theorem 94
11.7 The idea of ω-incompleteness 96
11.8 First-order True Arithmetic can't be p.r. axiomatized 97

12 Extending Gödel's First Theorem 98
12.1 Rosser's theorem 98
12.2 Another bit of notation 100
12.3 Diagonalization again 101
12.4 The Diagonalization Lemma 102
12.5 Incompleteness again 103
12.6 Provability 104
12.7 Tarski's Theorem 105

13 The Second Incompleteness Theorem 107
13.1 Formalizing the First Theorem 107
13.2 The Second Incompleteness Theorem 108
13.3 Con and ω-incompleteness 109
13.4 The significance of the Second Theorem 110
13.5 The Hilbert-Bernays-Löb derivability conditions 111
13.6 G, Con, and 'Gödel sentences' 114
13.7 Löb's Theorem 115

Interlude 118

14 μ-Recursive and Partial Recursive Functions 121
14.1 Minimization and μ-recursive functions 121
14.2 The Ackermann-Péter function 124
14.3 Diagonalizing out? 128
14.4 Partial computable functions 130
14.5 Partial recursive functions 132

15 Turing Machines 135
15.1 The basic conception 135
15.2 Turing computation defined more carefully 137
15.3 Some simple examples 140
15.4 'Turing machines' and their 'states' 145

16 Turing Computability and Recursiveness 146
16.1 Partial recursiveness entails Turing computability 146
16.2 Turing computability entails partial recursiveness 147
16.3 A corollary – and a general observation 149
16.4 Showing that recursiveness entails Turing computability 150
16.5 Showing that Turing computability entails recursiveness 152

17 Universal Turing Machines 158
17.1 Effectively enumerating Turing programs 158
17.2 Kleene's Normal Form Theorem 159
17.3 Universal machines 161

Bibliography 163

Preface

In 1931, the young Kurt Gödel published his First and Second Incompleteness Theorems; very often, these are simply referred to as 'Gödel's Theorems'. His startling results settled (or at least, seemed to settle) some of the crucial questions of the day concerning the foundations of mathematics. They remain of the greatest significance for the philosophy of mathematics – though just what that significance is continues to be debated. It has also frequently been claimed that Gödel's Theorems have a much wider impact on very general issues about language, truth and the mind. This book gives outline proofs of the Theorems, puts them in a more general formal context, and discusses their implications.

I originally intended to write a shorter book, leaving rather more of the formal details to be filled in from elsewhere. But while that plan might have suited some readers, I very soon decided that it would seriously irritate others to be sent hither and thither to consult a variety of text books with different terminology and different notations. So in the end, I have given more or less fully worked out proofs of most key results.

However, my original plan still shows through in two ways. First, some proofs are only sketched in, and some other proofs are still omitted entirely. Second, I try to make it plain which of the proofs I do give can be skipped without too much loss of understanding. My overall aim – rather as in a good lecture course with accompanying hand-outs – is to highlight as clearly as I can the key formal results and proof-strategies, provide details when these are important enough, and give references where more can be found.[1] Later in the book, I range over a number of intruiging formal themes and variations that take us a little beyond the content of most introductory texts.

As we go through, there is also an amount of broadly philosophical commentary. I follow Gödel in believing that our formal investigations and our general reflections on foundational matters should illuminate and guide each other. So I hope that the more philosophical discussions (though certainly not uncontentious) will be reasonably accessible to any thoughtful mathematician. Likewise, most of the formal parts of the book should be accessible even if you start from a relatively modest background in logic.[2] Though don't worry if you find yourself skimming to the ends of proofs – marked '⊠' – on a first reading: I do that all the time when tackling a new mathematics text.

[1] The plan is for there also to be accompanying exercises on the book's website at www.godelbook.net.

[2] Another plan is that the book will contain a short appendix of reminders about some logical notions and about standard notation: and for those who need more, there will be a more expansive review of the needed logical background on the website

Writing a book like this presents a problem of organization. For example, at various points we will need to call upon some background ideas from general logical theory. Do we explain them all at once, up front? Or do we introduce them as we go along, when needed? Another example: we will also need to call upon some ideas from the general theory of computation – we will make use of both the notion of a 'primitive recursive function' and the more general notion of a 'recursive function'. Again, do we explain these together? Or do we give the explanations many chapters apart, when the respective notions first get put to work?

I've adopted the second policy, introducing new ideas as and when needed. This has its costs, but I think that there is a major compensating benefit, namely that the way the book is organized makes it a lot clearer just what depends on what. It also reflects something of the historical order in which ideas emerged (a little more of the history emerges in footnotes).

I am already accumulating many debts. Many thanks to JC Beall, Hubie Chen, Torkel Franzen, Andy Fugard, Jeffrey Ketland, Jonathan Kleid, Mary Leng, Fritz Mueller, Tristan Mills, Jeff Nye, Alex Paseau, Michael Potter, Wolfgang Schwartz and Brock Sides for comments on draft chapters, and for encouragement to keep going with the book. I should especially mention Richard Zach both for saving me from a number of mistakes, large and small, and for suggestions that have much improved the book. Thanks too to students who have provided feedback, especially Jessica Leech, Adrian Pegg and Hugo Sheppard. I'd of course be very grateful to hear of any further typos I've introduced, especially in the later chapters, and even more grateful to get more general feedback and comments, which can be sent via the book's website.

Finally, like so many others, I am also hugely grateful to Donald Knuth, Leslie Lamport and the LaTeX community for the document processing tools which make typesetting a mathematical text like this one such a painless business.

1 What Gödel's First Theorem Says

1.1 Incompleteness and basic arithmetic

It seems to be child's play to grasp the fundamental concepts involved in the basic arithmetic of addition and multiplication. Starting from zero, there is a sequence of 'counting' numbers, each having just one immediate successor. This sequence of numbers – officially, *the natural numbers* – continues without end, never circling back on itself; and there are no 'stray' numbers, lurking outside this sequence. We can represent this sequence using (say) the familiar arabic numerals. Adding n to m is the operation of starting from m in the number sequence and moving n places along. Multiplying m by n is the operation of (starting from zero and) repeatedly adding m, n times. And that's about it.

Once these basic notions are in place, we can readily define many more arithmetical concepts in terms of them. Thus, for natural numbers m and n, $m < n$ if there is a number $k \neq 0$ such that $m + k = n$. m is a factor of n if $0 < m$ and there is some number k such that $0 < k$ and $m \times k = n$. m is even if it has 2 as a factor. m is prime if $1 < m$ and m's only factors are 1 and itself. And so on.

Using our basic and/or defined notions, we can then make various general claims about the arithmetic of addition and multiplication. There are elementary truths like 'addition is commutative', i.e. for any numbers m and n, we have $m + n = n + m$. And there are yet-to-be-proved conjectures like Goldbach's conjecture that every even number greater than two is the sum of two primes.

That second example illustrates the truism that it is one thing to understand the *language of basic arithmetic* (i.e. the language of the addition and multiplication of natural numbers, together with the standard first-order logical apparatus), and it is another thing to be able to answer the questions that can be posed in that language. Still, it is extremely plausible to suppose that, whether the answers are readily available to us or not, questions posed in the language of basic arithmetic do *have* entirely determinate answers. The structure of the number sequence is (surely) simple and clear: a single, never-ending sequence, with each number followed by a unique successor and no repetitions. The operations of addition and multiplication are again (surely) entirely determinate: their outcomes are fixed by the school-room rules. So what more could be needed to fix the truth or falsity of propositions that – perhaps via a chain of definitions – amount to claims of basic arithmetic?

To put it fancifully: God sets down the number sequence and specifies how the operations of addition and multiplication work. He has then done all he needs to do to make it the case that Goldbach's conjecture is true (or false, as the case may be!).

Of course, that last remark is too fanciful for comfort. We may find it compelling to think that the sequence of natural numbers has a definite structure, and that the operations of addition and multiplication are entirely nailed down by the familiar rules. But what is the real, non-metaphorical, content of the thought that the truth-values of all basic arithmetic propositions are thereby 'fixed'?

Here's one initially rather attractive and plausible way of beginning to sharpen up the thought. The idea is that we can specify a bundle of fundamental assumptions or *axioms* which somehow pin down the structure of the number sequence,[1] and which also characterize addition and multiplication (it is entirely natural to suppose that we *can* give a reasonably simple list of true axioms to encapsulate the fundamental principles so readily grasped by the successful learner of school arithmetic). Second, suppose φ is a proposition which can be formulated in the language of basic arithmetic. Then, the plausible suggestion continues, the assumed truth of our axioms always 'fixes' the truth-value of any such φ in the following sense: either φ is logically deducible from the axioms by a normal kind of proof, and so is true;[2] or $\neg\varphi$ is deducible from axioms, and so φ is false. (We mean, of course, that there exists a proof in principle: we may not stumble on a proof one way or the other. But the picture is that the axioms contain the information from which the truth-value of any basic arithmetical proposition can in principle be deductively extracted by deploying familiar step-by-step logical rules of inference.)

Logicians say that a theory T is *(negation)-complete* if, for every sentence φ in the language of the theory, either φ or $\neg\varphi$ is deducible in T's proof system. So, put into this jargon, the suggestion we are considering is: we should be able to specify a reasonably simple bundle of true axioms which taken together give us a *complete* theory of basic arithmetic – i.e. we can find a theory in which we can prove the truth or falsity of any claim of basic arithmetic. And if that's right, arithmetical truth could just be equated with provability in some appropriate system.

In headline terms, *what Gödel's First Incompleteness Theorem shows is that the plausible suggestion is wrong.* Suppose we try to specify a suitable axiomatic theory T that seems to capture the structure of the natural number sequence and pin down addition and multiplication. Then Gödel gives us a recipe for coming up with a corresponding sentence G_T, couched in the language of basic arithmetic, such that (i) we can show (on very modest assumptions) that neither G_T nor $\neg\mathsf{G}_T$ can be proved in T, and yet (ii) we can also recognize that G_T is true (assuming T is consistent).

This is astonishing. Somehow, it seems, the class of basic arithmetic truths will *always* elude our attempts to pin it down by a set of fundamental assumptions

[1]There are issues lurking here about what counts as 'pinning down a structure' using a bunch of axioms: we'll have to return to some of these issues in due course.

[2]'Normal' is vague, and later we will need to be more careful: but the idea is that we don't want to countenance, e.g., 'proofs' with an infinite number of steps!

from which we can deduce everything else.

How does Gödel show this? Well, note how we can use numbers and numerical propositions to encode facts about all sorts of things (for example, I might number off the students in the department in such a way that one student's code-number is less than another's if the first student is older than the second; a student's code-number is even if the student in question is female; and so on). In particular, we can use numbers and numerical propositions to encode facts about what can be proved in a theory T. And what Gödel did, very roughly, is find a coding scheme and a general method that enabled him to construct, for any given theory T strong enough to capture a decent amount of basic arithmetic, an arithmetical sentence G_T which encodes the thought 'This sentence is unprovable in theory T'.

If T were to prove its 'Gödel sentence' G_T, then it would prove a falsehood (since what G_T 'says' would then be untrue). Suppose though that T is a *sound* theory of arithmetic, i.e. T has true axioms and a reliably truth-preserving deductive logic. Then everything T proves must be true. Hence, if T is sound, G_T is unprovable in T. Hence G_T is then true (since it correctly 'says' it is unprovable). Hence its negation $\neg\mathsf{G}_T$ is false; and so that cannot be provable either. In sum, still assuming T is sound, neither G_T nor its negation will be provable in T: therefore T can't be negation-complete. And in fact we don't even need to assume that T is sound: T's mere consistency turns out to be enough to guarantee that G_T is true-but-unprovable.

Our reasoning here about 'This sentence is unprovable' is reminiscent of the Liar paradox, i.e. the ancient conundrum about 'This sentence is false', which is false if it is true and true if it is false. So you might wonder whether Gödel's argument leads to a paradox rather than a theorem. But not so. Or at least, there is nothing at all problematic about Gödel's First Theorem as a result about *formal* axiomatized systems. (We'll need in due course to say more about the relation between Gödel's argument and the Liar and other paradoxes: and we'll need to mention the view that the argument can be used to show something paradoxical about *informal* reasoning. But that's for later.)

'Hold on! If we can locate G_T, a "Gödel sentence" for our favourite theory of arithmetic T, and can argue that G_T is true-but-unprovable, why can't we just patch things up by adding it to T as a new axiom?' Well, to be sure, if we start off with theory T (from which we can't deduce G_T), and add G_T as a new axiom, we'll get an expanded theory $U = T + \mathsf{G}_T$ from which we *can* trivially deduce G_T. But we now just re-apply Gödel's method to our improved theory U to find a new true-but-unprovable-in-U arithmetic sentence G_U that says 'I am unprovable in U'. So U again is incomplete. Thus T is not only incomplete but, in a quite crucial sense, is *incompletable*.

And note that since G_U can't be derived from $T + \mathsf{G}_T$, it can't be derived from the original T either. And we can keep on going: simple iteration of the same trick starts generating a never-ending stream of independent true-but-unprovable sentences for any candidate axiomatized theory of basic arithmetic T.

1.2 Why it matters

There's nothing mysterious about a theory failing to be negation-complete, plain and simple. For a very trite example, imagine the faculty administrator's 'theory' T which records some basic facts about e.g. the course selections of group of students – the language of T, let's suppose, is very limited and can just be used to tell us about who takes what course in what room when. From the 'axioms' of T we'll be able, let's suppose, to deduce further facts such as that Jack and Jill take a course together, and at least ten people are taking the logic course. But if there's no axiom in T about their classmate Jo, we might not be able to deduce either J = 'Jo takes logic' or $\neg\mathsf{J}$ = 'Jo doesn't take logic'. In that case, T isn't yet a negation-complete story about the course selections of students. However, that's just boring: for the 'theory' about course selection is no doubt completable (i.e. it can be expanded to settle every question that can be posed in its very limited language). By contrast, what gives Gödel's First Theorem its real bite is that it shows that any properly axiomatized theory of basic arithmetic must *remain* incomplete, whatever our efforts to complete it by throwing further axioms into the mix.

This incompletability result doesn't just affect basic arithmetic. For the next simplest example, consider the mathematics of the rational numbers (fractions, both positive and negative). This embeds basic arithmetic in the following sense. Take the positive rationals of the form $n/1$ (where n is an integer). These of course form a sequence with the structure of the natural numbers. And the usual notions of addition and multiplication for rational numbers, when restricted to rationals of the form $n/1$, correspond exactly to addition and multiplication for the natural numbers. So suppose that there were a negation-complete axiomatic theory T of the rationals such that, for any proposition ψ of rational arithmetic, either ψ or $\neg\psi$ can be deduced from T. Then, in particular, given any proposition ψ' about the addition and/or multiplication of rationals of the form $n/1$, T will entail either ψ' or $\neg\psi'$. But then T plus simple instructions for rewriting such propositions ψ' as propositions about the natural numbers would be a negation-complete theory of basic arithmetic – which is impossible by the First Incompleteness Theorem. Hence there can be no complete theory of the rationals either.

Likewise for any stronger theory that either includes or can model arithmetic. Take set theory for example. Start with the empty set $\varnothing$. Form the set $\{\varnothing\}$ containing $\varnothing$ as its sole member. Now form the set $\{\varnothing, \{\varnothing\}\}$ containing the empty set we started off with plus the set we've just constructed. Keep on going, at each stage forming the set of sets so far constructed (a legitimate procedure in any standard set theory). We get the sequence

$$\varnothing, \{\varnothing\}, \{\varnothing, \{\varnothing\}\}, \{\varnothing, \{\varnothing\}, \{\varnothing, \{\varnothing\}\}\}, \ldots$$

This has the structure of the natural numbers. It has a first member (corresponding to zero); each member has one and only one successor; it never repeats. We

can go on to define analogues of addition and multiplication. If we could have a negation-complete axiomatized set theory, then we could, in particular, have a negation-complete theory of the fragment of set-theory which provides us with a model of arithmetic; and then adding a simple routine for translating the results for this fragment into the familiar language of basic arithmetic would give us a complete theory of arithmetic. So, by Gödel's First Incompleteness Theorem again, there cannot be a negation-complete set theory.

In sum, any axiomatized mathematical theory T rich enough to embed a reasonable amount of the basic arithmetic of the addition and multiplication of the natural numbers must be incomplete and incompletable – yet we can recognize certain 'Gödel sentences' for T to be not only unprovable but true so long as T is consistent.

This result, on the face of it, immediately puts paid to an otherwise attractive suggestion about the status of arithmetic (and it similarly defeats parallel claims about the status of other parts of mathematics). What makes for the special certainty and the necessary truth of correct claims of basic arithmetic? It is tempting to say: they are *analytic* truths in the philosophers' sense, i.e. they are logically *deducible* from the very *definitions* of the numbers and the operations of addition and multiplication. But what Gödel's First Theorem shows is that, however we try to encapsulate such definitions in a set of axioms giving us some consistent deductive theory T, there will be truths of basic arithmetic unprovable in T: so it seems that arithmetical truth must outstrip what can be given merely by logic-plus-definitions. But then, how *do* we manage somehow to latch on to the nature of the un-ending number sequence and the operations of addition and multiplication in a way that outstrips whatever rules and principles can be captured in definitions? It can seem that we must have a rule-transcending cognitive grasp of the numbers which underlies our ability to recognize certain 'Gödel sentences' as correct arithmetical propositions. And if you are tempted to think so, then you may well be further tempted to conclude that minds such as ours, capable of such rule-transcendence, can't be machines (supposing, reasonably enough, that the cognitive operations of anything properly called a machine can be fully captured by rules governing the machine's behaviour).

So there's apparently a quick route from reflections about Gödel's First Theorem to some conclusions about arithmetical truth and the nature of the minds that grasp it. Whether those conclusions really follow will emerge later. For the moment, we have an initial if very rough idea of what the Theorem says and why it might matter – enough, I hope, to entice you to delve further into the story that unfolds in this book.

1.3 What's next?

What we've said so far, of course, has been arm-waving and introductory. We must now start to do better – though for the next three chapters our discussions

will remain fairly informal. In Chapter 2, as a first step, we explain more carefully what we mean by talking about an 'axiomatized theory' in general. In Chapter 3, we introduce some concepts relating to axiomatized theories of arithmetic. Then in Chapter 4 we prove a neat and relatively easy result – namely that any so-called 'sufficiently strong' axiomatized theory of arithmetic is negation incomplete. For reasons that we'll explain, this informal result falls well short of Gödel's First Incompleteness Theorem. But it provides a very nice introduction to some key ideas that we'll be developing more formally in the ensuing chapters.

2 The Idea of an Axiomatized Formal Theory

Gödel's Incompleteness Theorems tell us about the limits of axiomatized theories of arithmetic. Or rather, more carefully, they tell us about the limits of axiomatized *formal* theories of arithmetic. But what exactly does this mean?

2.1 Formalization as an ideal

Rather than just dive into a series of definitions, it is well worth quickly reminding ourselves of why we *care* about formalized theories.

So let's get back to basics. In elementary logic classes, we are drilled in translating arguments into an appropriate formal language and then constructing formal deductions of putative conclusions from given premisses. Why bother with formal languages? Because everyday language is replete with redundancies and ambiguities, not to mention sentences which simply lack clear truth-conditions. So, in assessing complex arguments, it helps to regiment them into a suitable artificial language which is expressly designed to be free from obscurities, and where surface form reveals logical structure.

Why bother with formal deductions? Because everyday arguments often involve suppressed premisses and inferential fallacies. It is only too easy to cheat. Setting out arguments as formal deductions in one style or another enforces honesty: we have to keep a tally of the premisses we invoke, and of exactly what inferential moves we are using. And honesty is the best policy. For suppose things go well with a particular formal deduction. Suppose we get from the given premisses to some target conclusion by small inference steps each one of which is obviously valid (no suppressed premisses are smuggled in, and there are no suspect inferential moves). Our honest toil then buys us the right to confidence that our premisses really do entail the desired conclusion.

Granted, outside the logic classroom we almost never set out deductive arguments in a fully formalized version. No matter. We have glimpsed a first ideal – arguments presented in an entirely perspicuous language with maximal clarity and with everything entirely open and above board, leaving no room for misunderstanding, and with all the arguments' commitments systematically and frankly acknowledged.[1]

Old-fashioned presentations of Euclidean geometry illustrate the pursuit of a related second ideal – the (informal) axiomatized theory. Like beginning logic students, school students used to be drilled in providing deductions, though

[1]For an early and very clear statement of this ideal, see Frege (1882), where he explains the point of the first recognizably modern formal system of logic, presented in his *Begriffsschrift* (i.e. *Conceptual Notation*) of 1879.

the deductions were framed in ordinary geometric language. The game was to establish a whole body of theorems about (say) triangles inscribed in circles, by deriving them from simpler results which had earlier been derived from still simpler theorems that could ultimately be established by appeal to some small stock of fundamental principles or *axioms*. And the aim of this enterprise? By setting out the derivations of our various theorems in a laborious step-by-step style – where each small move is warranted by simple inferences from propositions that have already been proved – we develop a unified body of results that we can be confident must hold if the initial Euclidean axioms are true.

On the surface, school geometry perhaps doesn't seem very deep: yet making all its fundamental assumptions fully explicit is surprisingly difficult. And giving a set of axioms invites further enquiry into what might happen if we tinker with these assumptions in various ways – leading, as is now familiar, to investigations of non-Euclidean geometries.

Many other mathematical theories are also characteristically presented axiomatically.[2] For example, set theories are presented by laying down some basic axioms and exploring their deductive consequences. We want to discover exactly what is guaranteed by the fundamental principles embodied in the axioms. And we are again interested in exploring what happens if we change the axioms and construct alternative set theories – e.g. what happens if we drop the 'axiom of choice' or add 'large cardinal' axioms?

However, even the most tough-minded mathematics texts which explore axiomatized theories are typically written in an informal mix of ordinary language and mathematical symbolism, with proofs rarely spelt out in every formal detail. They fall short of the logical ideal of full formalization. That doesn't make them any the less proofs (the contents of a mathematics journal article can be entirely clear and the arguments entirely compelling). But we might reasonably hope that nothing stands in the way our more informally presented mathematical proofs being turned into fully formalized ones – i.e. hope that they could be set out in a strictly regimented formal language of the kind that logicians describe, with absolutely every inferential move made fully explicit and checked as being in accord with some acknowledged formal rule of inference, with all the proofs ultimately starting from our explicitly given axioms. True, the extra effort of laying out everything in this kind of fully formalized detail would almost never be worth the cost in time and ink. In mathematical practice we use enough formalization to convince ourselves that our results don't depend on illicit smuggled premisses or on dubious inference moves, and leave it at that ('sufficient unto the day is the rigour thereof').[3] But still, it *is* essential for good mathematics to

[2]For a classic defence, extolling the axiomatic method in mathematics, see Hilbert (1918).

[3]Bourbaki (1968, p. 8) puts the point like this in a famous passage: 'In practice, the mathematician who wishes to satisfy himself of the perfect correctness or "rigour" of a proof or a theory hardly ever has recourse to one or another of the complete formalizations available nowadays, ... In general he is content to bring the exposition to a point where his experience and mathematical flair tell him that translation into formal language would be no more than

achieve maximum precision and to avoid the use of unexamined inference rules or unacknowledged assumptions. So, putting together the logician's aim of perfect clarity and honest inference with the mathematician's project of regimenting a theory into a tidily axiomatized form, we can see the point of the notion of an *axiomatized formal theory*, if not as a practical day-to-day working medium, then at least as a composite ideal.

Mathematicians (and some philosophical commentators) are apt to stress that there is a lot more to mathematical practice than striving towards the logical ideal. For a start, they observe that mathematicians typically aim not merely for formal correctness but for *explanatory* proofs, which not only show that some proposition must be true, but in some sense make it clear *why* it is true. And such observations are right and important. But they don't affect the point that the business of formalization just takes to the limit features that we expect to find in good proofs anyway, i.e. precise clarity and lack of inferential gaps.

(b) It is certainly not being suggested that only fully regimented proofs – completely set out in some axiomatized formal system – are genuine mathematical proofs. For that would make proofs rare indeed, and almost never to be found in mathematics journals. Informal proofs but be clear and compelling by any reasonably standard.

2.2 Axiomatized formal theories

So, putting together the ideal of formal precision and the ideal of regimentation into an axiomatic system, we have arrived at the concept of an axiomatized formal theory, which comprises (a) a formalized *language*, (b) a set of sentences from the language which we treat as *axioms* characterizing the theory, and (c) some *deductive system* for proof-building, so that we can derive theorems from the axioms. We'll now say a little more about these ingredients in turn.

(a) We'll take it that the general idea of a formalized language is familiar from elementary logic, and so we can be fairly brisk. Note that we will normally be interested in *interpreted* languages – i.e. we will usually be concerned not just with formal patterns of symbols but with expressions which have an intended significance. We can usefully think of an interpreted language as a pair $\langle \mathcal{L}, \mathcal{I} \rangle$, where $\mathcal{L}$ is a syntactically defined system of expressions and $\mathcal{I}$ gives the interpretation of these expressions. We'll follow the standard logicians' convention of calling the first component of the pair an 'uninterpreted language' (or sometimes, when no confusion will arise, simply a 'language').

First, then, on the unintepreted language component, $\mathcal{L}$. We'll assume that this has a finite alphabet of symbols – for we can always construct e.g. an unending supply of variables by standard tricks like using repeated primes (to yield 'x', 'x'', 'x''', etc.). We then need syntactic construction-rules to determine which finite strings of symbols from the given alphabet constitute the vocabulary of

an exercise of patience (though doubtless a very tedious one).'

individual constants (i.e. names), variables, predicates and function-expressions in $\mathcal{L}$. And then we need further rules to determine which finite sequences of these items of vocabulary plus logical symbols for e.g. connectives and quantifiers make up the well-formed formulae of $\mathcal{L}$ (its *wffs*, for short).

Plainly, given that the whole point of using a formalized language is to make everything as clear and determinate as possible, we don't want it to be a disputable matter whether a given sign or cluster of signs is e.g. a constant or one-place predicate of a given system $\mathcal{L}$. And we don't want it to be a disputable matter whether a string of symbols is a wff of $\mathcal{L}$. So we want there to be clear and objective formal procedures, agreed on all sides, for *effectively deciding* whether a putative constant-symbol is indeed a constant, etc., and likewise for deciding whether a putative wff is a wff. We will say more about the needed notion of decidability in Section 2.3.

Next, on the semantic component $\mathcal{I}$. The details of how to interpret $\mathcal{L}$'s wffs will vary with the richness of the language. Let's suppose we are dealing with the usual sort of language which is to be given a referential semantics of the absolutely standard kind, familiar from elementary logic. Then the basic idea is that an interpretation $\mathcal{I}$ will specify a set of objects to be the domain of quantification. For each constant of $\mathcal{L}$, the interpretation $\mathcal{I}$ picks out an element in the domain to be its referent. For each one-place predicate of $\mathcal{L}$, $\mathcal{I}$ picks out a set of elements to be the extension of the predicate; for each two-place predicate, $\mathcal{I}$ picks out a set of ordered pairs of elements to be its extension; and so on. Likewise for function-expressions: thus, for each one-place function of $\mathcal{L}$, the interpretation $\mathcal{I}$ will pick out a suitable set of ordered pairs of elements to be the function's extension;[4] and similarly for many-place functions.

Then there are interpretation rules which fix the truth-conditions of every sentence of $\mathcal{L}$ (i.e. every closed wff without free variables), given the interpretation of the $\mathcal{L}$'s constants, predicates etc. To take the simplest examples, the wff 'Fc' is true if the referent of 'c' is in the extension of 'F'; the wff '$\neg\mathsf{Fc}$' is true if 'Fc' isn't true; the wff '$(\mathsf{Fc} \to \mathsf{Gc})$' is true if either '$\mathsf{Fc}$' is false or '$\mathsf{Gc}$' is true; the wff '$\forall\mathsf{x}\,\mathsf{Fx}$' is true if every object in the domain is in the extension of 'F'; and so on. Note that the standard sort of interpretation rules will make it a mechanical matter to work out the interpretation of any wff, however complex.[5]

(b) Next, to have a theory at all, some wffs of our theory's language need to be

[4] A set is only suitable to be the extension for a function-expression if (i) for every element a in the domain, there is some ordered pair $\langle a, b\rangle$ in the set, and (ii) the set doesn't contain both $\langle a, b\rangle$ and $\langle a, b'\rangle$, when $b \neq b'$. For in a standard first-order setting, functions are required to be total and so associate each argument a with some unique value b – otherwise a term like '$\mathsf{f(a)}$' could lack a reference, and sentences containing it would lack a truth value (contrary to the standard requirement that every first-order sentence is either true or false on a given interpretation).

[5] We can, incidentally, allow a language to be freely extended by new symbols introduced as definitional abbreviations for old expressions – so long as the defined symbols can always be systematically eliminated again in unambiguous ways. Wffs involving definitional abbreviations will, of course, inherit the interpretations of their unabbreviated counterparts.

selected as *axioms*, i.e. as fundamental assumptions of our theory (we'll take it that these are sentences, closed wffs without variables dangling free).

Since the fundamental aim of the axiomatization game is to see what follows from a bunch of axioms, we again don't want it to be a matter for dispute whether a given proof does or doesn't appeal only to axioms in the chosen set. Given a purported proof of some result, there should be an absolutely clear procedure for settling whether the input premisses are genuinely instances of the official axioms. In other words, for an axiomatized formal theory, we must be able to effectively decide whether a given wff is an axiom or not.

That doesn't, by the way, rule out axiomatized theories with infinitely many axioms. We might want to say 'every wff of such-and-such a form is an axiom' (where there is an unlimited number of instances): and that's permissible so long as it is still effectively decidable what counts as an instance of that form.

(c) Thirdly, an axiomatized formal theory needs some deductive apparatus, i.e. some sort of formal *proof-system*. Proofs are then finite arrays of wffs that conform to the rules of the relevant proof-system, and whose only assumptions belong to the set of axioms (note, particular proofs – being finite – can only call upon a finite number of axioms, even if the formal system in question has an infinite number of axioms available).

We'll take it that the core idea of a proof-system is familiar from elementary logic. The differences between various equivalent systems of proof presentation – e.g. old-style linear proof systems vs. natural deduction proofs vs. tableau (or 'tree') proofs – don't matter for our present purposes. What will matter is the strength of the system of rules we adopt. We will predominantly be working with some version of standard first-order logic with identity: but whatever system we adopt, it is crucial that we fix on a set of rules which enable us to settle, without room for dispute, what counts as a well-formed proof in this system. In other words, we require the property of being a well-formed proof from axioms $\varphi_1, \varphi_2, \ldots, \varphi_n$ to conclusion ψ in the theory's proof-system to be a decidable one. The whole point of formalizing proofs is to set out the commitments of an argument with absolute determinacy, so we certainly don't want it to be a disputable or subjective question whether a putative proof does or does not conform to the rules for proof-building for the formal system in use. Hence there should be a clear and effective procedure for deciding whether an array counts as a well-constructed proof according to the relevant proof-system.

Be careful! The claim is only that it should be decidable whether an array of wffs presented as a proof really *is* a proof. This is *not* to say that we can always decide in advance whether a proof exists to be discovered. Even in familiar first-order quantificational logic, for example, it is not in general decidable whether there is a proof from given premisses to a certain conclusion (we'll be proving this undecidability result later).

To summarize then, here again are the key headlines:

> T is an (interpreted) axiomatized formal theory just if (a) T is couched in a formalized language $\langle \mathcal{L}, \mathcal{I} \rangle$, such that it is effectively decidable what counts as a wff of $\mathcal{L}$, etc., (b) it is effectively decidable which $\mathcal{L}$-wffs are axioms of T, and (c) T uses a proof-system such that it is effectively decidable whether an array of $\mathcal{L}$-wffs counts as a proof.

2.3 Decidability

If the idea of an axiomatized formal theory is entirely new to you, it might help to jump forward at this point and browse through Chapter 5 where we introduce some formal arithmetics. The rest of this present chapter continues to discuss formalized theories more generally.

We've just seen that to explain the idea of a properly formalized theory involves repeatedly calling on the concept of effective decidability.[6] But what is involved in saying that some question – such as whether the symbol-string σ is a wff, or the wff φ is an axiom, or the array of wffs Π is a well-formed proof – is decidable? What kind of decision procedures should we allow?

We are looking for procedures which are entirely determinate. We don't want there to be any room left for the exercise of imagination or intuition or fallible human judgement. So we want procedures that follow an *algorithm* – i.e. follow a series of step-by-step instructions (instructions that are pinned down in advance of their execution, and that can be fully communicated), with each small step clearly specified in every detail, leaving no room for doubt as to what does and what does not count as following its instructions. Following the steps should involve no appeal to outside oracles (or other sources of empirical information). There is to be no resort to random methods (coin tosses). And – crucially – the procedure of must be guaranteed to terminate and deliver a result after a finite number of steps.

There are familiar algorithms for finding the results of a long division problem, or for finding highest common factors, or (to take a non-arithmetical example) for deciding whether a propositional wff is a tautology. These algorithms can be executed in an entirely mechanical way. Dumb computers can be programmed to do the job. Indeed it is natural to turn that last point into an informal definition:

> An algorithmic procedure is a *computable* one, i.e. one which a suitably programmed computer can execute and which is guaranteed to deliver a result in finite time.

And then relatedly, we will say:

[6] When did the idea emerge that properties like being a wff or being an axiom ought to be decidable? It was arguably already implicit in Hilbert's conception of rigorous proof. But Richard Zach has suggested that an early source for the *explicit* deployment of the idea is von Neumann (1927).

> A function is *effectively computable* if there is an algorithmic procedure which a suitably programmed computer could use for calculating the value of the function for a given argument.
>
> An *effectively decidable* property is one for which there is an algorithmic procedure which a suitably programmed computer can use to decide whether the property obtains.

So a formalized language is one for which there are algorithms for deciding what strings of symbols are wffs, proofs, etc.

But what kind of computer do we have in mind here when we say that an algorithmic procedure is one that a computer can execute? We need to say something here about the relevant kind of computer's (a) *size and speed*, and (b) *architecture*.

(a) A real-life computer is limited in size and speed. There will be some finite bound on the size of the inputs it can handle; there will be a finite bound on the size of the set of instructions it can store; there will be a finite bound on the size of its working memory. And even if we feed in inputs and instructions it can handle, it is of little practical use to us if the computer won't finish doing its computation for centuries.

Still, we are going to cheerfully abstract from all those 'merely practical' considerations of size and speed. In other words, we will say that a question is effectively decidable if there is a finite set of step-by-step instructions which a dumb computer could use which is *in principle* guaranteed – given memory, working space and time enough – to deliver a decision eventually. Let's be clear, then: 'effective' here does *not* mean that the computation must be feasible for us, on existing computers, in real time. So, for example, we count a numerical property as effectively decidable in this broad, 'in principle', sense even if on existing computers it might take more time to compute whether a given number has it than we have left before the heat death of the universe, and would use more bits of storage than there are particles in the universe. It is enough that there's an algorithm which works in theory and would deliver an answer in the end, if only we had the computational resources to use it and could wait long enough.

'But then,' you might well ask, 'why on earth bother with these radically idealized notions of computability and decidability, especially in the present context? We started from the intuitive idea of a formalized theory, one where the question of whether a putative proof *is* a proof (for example) is not a disputable matter. We made a first step towards tidying up this intuitive idea by requiring there to be some algorithm that can settle the question, and then identified algorithms with procedures that a computer can follow. But if we allow procedures that may not deliver a verdict in the lifetime of the universe, what good is that? Shouldn't we really equate decidability not with idealized-computability-in-principle but with some stronger notion of *feasible* computability?'

That's an entirely fair challenge. And modern computer science has much to say about grades of computational complexity, i.e. about different levels of feasibility. However, we will stick to our idealized notions of computability and decidability. That's because there are important problems for which we can show that *there is no algorithm at all* which is guaranteed to deliver a result: so even without any restrictions on execution time and storage, a finite computer still couldn't be programmed in a way that is guaranteed to solve the problem. Having a weak 'in principle' notion of what is required for decidability means that such impossibility results are exceedingly strong – for they don't depend on mere contingencies about what is practicable, given the current state of our software and hardware, and given real-world limitations of time or resources. They show that some problems are not mechanically decidable, even in principle.

(b) We've said that we are going to be abstracting from limitations on storage etc. But you might suspect that this still leaves much to be settled. Doesn't the 'architecture' of a computing device affect what it can compute?

The short answer is 'no'. And intriguingly, some of the central theoretical questions here were the subject of intensive investigation even before the first electronic computers were built. Thus, in the 1930s, Alan Turing famously analysed what it is for a numerical function to be step-by-step computable in terms of the capacities of a *Turing machine* (an idiot computer following a program built from extremely simple steps: for explanations and examples, see Chapter 15). Now, it is easy to spin variations on the details of Turing's original story. For example, a standard Mark I Turing machine has just a single 'tape' or workspace to be used for both storing and manipulating data: but we can readily describe a Mark II machine which has (say) two tapes – one to be used as a main workspace, and a separate one for storing data. Or we can consider a computer with unlimited 'Random Access Memory' – that is to say, a computer with an unlimited set of registers in which it can store various items of working data ready to be retrieved into its workspace when needed.[7] The details don't matter here and now. What does matter is that exactly the same numerical functions are computable by Mark I Turing machines as are computable by Mark II machines as are computable by register machines, etc. Equivalently, exactly the same questions about whether some numerical property obtains are mechanically decidable by a suitably programmed Mark I Turing machine or Mark II Turing machine or by a register machine, etc. Indeed, *all* the definitions of algorithmic computability by idealized computers that have ever been seriously proposed turn out to be equivalent. In a slogan, algorithmic computability is architecture independent. That's a Big Mathematical Result – or rather, a cluster of results – which can be conclusively proved.

This Big Mathematical Result supports the claim Turing famously makes in his classic paper published in 1936, which we can naturally call

[7]The theoretical treatment of unlimited register machines is first given in (Shepherdson and Sturgis, 1963); there is a very accessible presentation in the excellent (Cutland, 1980).

> *Turing's Thesis* The numerical functions which are computable in the intuitive sense are just those functions which are computable by a Turing machine. Likewise, the numerical questions which are effectively decidable in the intuitive sense are just those questions which are decidable by a suitable Turing machine.

This claim – we'll further explore its content in Chapter **??** – correlates an intuitive notion with a sharp technical analysis. So you might perhaps think it is not the sort of thing we can establish by rigorous proof. But be that as it may: after some seventy years, no successful challenge to Turing's Thesis has been mounted. Which means that we can continue to talk informally about intuitively computable numerical functions and effectively decidable properties of numbers, and be very confident that we are indeed referring to fully determinate classes.

And what about the idea of being computable as applied to non-numerical functions (like truth-functions) or the idea of being effectively decidable as applied to non-numerical properties (like the property of being an axiom of some theory)? Are these ideas determinate too?

Well, think how a real-world computer can be used to evaluate a truth-function or decide whether a wff is an axiom in a formal system. In the first case, we code the truth-values *true* and *false* using numbers, say 0 and 1, and then do a numerical computation. In the second case, we write a program for manipulating strings of symbols, and again – though this time behind the scenes – these strings get correlated with binary codes, and it is these numbers that the computer works on. In the end, using numerical codings, the computations in both cases are done on numbers after all.

Now generalize that thought. A natural suggestion is that *any* computation dealing with Xs can be turned into an equivalent numerical computation via the trick of using simple numerical codes for Xs. More carefully: by a trivial algorithm, we can map Xs to numbers; we can then do the appropriate core computation on the numbers; and then another trivial algorithm translates the result back into a claim about X's. Fortunately, however, we don't need to assess that suggestion in its full generality. For the purposes of this book, the non-numerical computations we are interested in are cases where the Xs are wffs from standard formal languages, or sequences of wffs, etc. And in those cases, there's no doubt that we can algorithmically map claims about wffs etc. to corresponding claims about numbers (see Sections 2.6, 10.1, 10.2). So the question e.g. whether a certain property of wffs is a decidable one can be translated quite uncontentiously into the question whether a corresponding numerical property is a decidable one. Given the Turing Thesis that it is quite determinate what counts as a decidable property of numbers, it follows that it is quite determinate what counts as a decidable property of wffs. And similarly for the other cases we are interested in.

2.4 Enumerable and effectively enumerable sets

Suppose Σ is some set of items: its members might be numbers, wffs, proofs, sets or whatever. Then we say informally that Σ is *enumerable* if its members can – at least in principle – be listed off in some order, with every member appearing on the list; repetitions are allowed, and the list may be infinite. (It is tidiest to think of the empty set as the limiting case of an enumerable set: it is enumerated by the empty list!)

We can make that informal definition more rigorous in various equivalent ways. We'll give just one – and to do that, let's introduce some standard jargon and notation that we'll need later anyway:

i. A function, recall, maps arguments in some domain to unique values. Suppose the function f is defined for all arguments in the set Δ; and suppose that the values of f all belong to the set Γ. Then we write

   $$f\colon \Delta \to \Gamma$$

   and say that f is a (total) function from Δ into Γ.

ii. A function $f\colon \Delta \to \Gamma$ is *surjective* if for every $y \in \Gamma$ there is some $x \in \Delta$ such that $f(x) = y$. (Or, if you prefer that in English, you can say that such a function is '*onto*', since it maps Δ onto the whole of Γ.)

iii. We use '$\mathbb{N}$' to denote the set of all natural numbers.

Then we can say:

> The set Σ is *enumerable* if it is either empty or there is a surjective function $f\colon \mathbb{N} \to \Sigma$. (We can say that such a function enumerates Σ.)

To see that this comes to the same as our original informal definition, just note the following two points. (a) Suppose we have a list of all the members of Σ in some order, the zero-th, first, second, ... (perhaps an infinite list, perhaps with repetitions). Then take the function f defined as follows $f(n) = n$-th member of the list, if the list goes on that far, or $f(n) = f(0)$ otherwise. Then f is a surjection $f\colon \mathbb{N} \to \Sigma$. (b) Suppose conversely that f is surjection $f\colon \mathbb{N} \to \Sigma$. Then, if we successively evaluate f for the arguments 0, 1, 2, ..., we get a list of values $f(0), f(1), f(2) \ldots$ which by hypothesis contains all the elements of Σ, with repetitions allowed.

We'll limber up by noting a quick initial result: If two sets are enumerable, so is the result of combining their members into a single set. (Or if you prefer that in symbols: if Σ_1 and Σ_2 are enumerable so is $\Sigma_1 \cup \Sigma_2$.)

Proof Suppose there is a list of members of Σ_1 and a list of members of Σ_2. Then we can interleave these lists by taking members of the two sets alternately, and the result will be a list of the union of those two sets. (More formally, suppose f_1

enumerates Σ_1 and f_2 enumerates Σ_2. Put $g(2n) = f_1(n)$ and $g(2n+1) = f_2(n)$; then g enumerates $\Sigma_1 \cup \Sigma_2$.) ⊠

That was easy and trivial. Here's another result – famously proved by Georg Cantor – which is also easy, but certainly not trivial:[8]

Theorem 1 *There are infinite sets which are not enumerable.*

Proof Consider the set $\mathbb{B}$ of infinite binary strings (i.e. the set of unending strings like '0110001010011...'). There's obviously an infinite number of them. Suppose, for reductio, that we could list off these strings in some order

$$\begin{array}{ll} 0 & \underline{0}110001010011\ldots \\ 1 & 1\underline{1}00101001101\ldots \\ 2 & 11\underline{0}0101100001\ldots \\ 3 & 000\underline{1}111010101\ldots \\ 4 & 1101\underline{1}11011101\ldots \\ \ldots & \ldots \end{array}$$

Read off down the diagonal, taking the n-th digit of the n-th string (in our example, this produces 01011...). Now flip each digit, swapping 0s and 1s (in our example, yielding 10100...). By construction, this 'flipped diagonal' string differs from the initial string on our original list in the first place, differs from the next string in the second place, and so on. So our diagonal construction yields a string that isn't on the list, contradicting the assumption that our list contained all the binary strings. So $\mathbb{B}$ is infinite, but not enumerable. ⊠

It's worth adding three quick comments.

a. An infinite binary string $b_0b_1b_2\ldots$ can be thought of as characterizing a corresponding set of numbers Σ, where $n \in \Sigma$ just if $b_n = 0$. So our theorem is equivalent to the result that the set of *sets* of natural numbers can't be enumerated.

b. An infinite binary string $b_0b_1b_2\ldots$ can also be thought of as characterizing a corresponding function f, i.e. the function which maps each natural number to one of the numbers $\{0, 1\}$, where $f(n) = b_n$. So our theorem is also equivalent to the result that the set of *functions* from the natural numbers to $\{0, 1\}$ can't be enumerated.

c. Note that non-enumerable sets have to be a *lot* bigger than enumerable ones. Suppose Σ is a non-enumerable set; suppose $\Delta \subset \Sigma$ is some enumerable subset of Σ; and let $\Gamma = \Sigma - \Delta$ be the set you get by removing

[8] Cantor first established this key result in his (1874), using, in effect, the Bolzano-Weierstrass theorem. The neater 'diagonal argument' we give here first appears in his (1891).

> the members of Δ from Σ. Then this difference set must also be non-emumerably infinite – for otherwise, if it were enumerable, $\Sigma = \Delta \cup \Gamma$ would be enumerable after all (by the result we proved above).

Now, note that saying that a set is enumerable in our sense[9] is not to say that we can produce a 'nice' algorithmically computable function that does the enumeration, only that there is *some* function or other that does the job. So we need another definition:

> The set Σ is *effectively* enumerable if it is either empty or there is an *effectively computable* function which enumerates it.

In other words, a set is effectively enumerable if an (idealized) computer could be programmed to start producing a numbered list of its members such that any member will be eventually mentioned – the list may have no end, and may contain repetitions, so long as every item in the set is correlated with some natural number.

A *finite* set of finitely specifiable objects can always be effectively enumerated: any listing will do, and – since it is finite – it could be stored in an idealized computer and spat out on demand. And for a simple example of an effectively enumerable *infinite* set, imagine an algorithm which generates the natural numbers one at a time in order, ignores those which fail the well-known (mechanical) test for being prime, and lists the rest: this procedure generates a never-ending list on which every prime will eventually appear. So the primes are effectively enumerable. Later we will meet some examples of infinite sets of numbers which are enumerable but which *can't* be effectively enumerated.

2.5 More definitions

We now add four definitions more specifically to do with theories:

i. Given a proof of the sentence (i.e. closed wff) φ from the axioms of the theory T using the background logical proof system, we will say that φ is a *theorem* of the theory. And using the standard abbreviatory symbol, we write: $T \vdash \varphi$.

ii. A theory T is *decidable* if the property of being a theorem of T is an effectively decidable property – i.e. if there is a mechanical procedure for determining whether $T \vdash \varphi$ for any given sentence φ of the language of theory T.

iii. Assuming theory T has a normal negation connective '$\neg$', T is *inconsistent* if it proves some pair of wffs of the form φ, $\neg\varphi$.

[9] The qualification 'in our sense' is important as terminology isn't stable: for some writers use 'enumerable' to mean *effectively* enumerable, and use e.g. 'denumerable' for our wider notion.

iv. A theory T is *negation complete* if, for any sentence φ of the language of the theory, either φ or $\neg\varphi$ is a theorem (i.e. either $T \vdash \varphi$ or $T \vdash \neg\varphi$).

Here's a quick and very trite example. Consider a trivial pair of theories, T_1 and T_2, whose shared language consists of the propositional atoms 'p', 'q', 'r' and all the wffs that can be constructed out of them using the familiar propositional connectives, whose shared underlying logic is a standard propositional natural deduction system, and whose sets of axioms are respectively {'¬p'} and {'p', 'q', '¬r'}. Given appropriate interpretations for the atoms, T_1 and T_2 are then both axiomatized formal theories. For it is mechanically decidable what is a wff of the theory, and whether a purported proof is indeed a proof from the given axioms. Both theories are consistent. Both theories are decidable; just use the truth-table test to determine whether a candidate theorem really follows from the axioms. Finally, T_1 is negation incomplete, since the sole axiom doesn't decide whether 'q' or '¬q' holds, for example. But T_2 is negation complete (any wff constructed from the three atoms using the truth-functional connectives has its truth-value decided).

This mini-example illustrates a crucial terminological point. You will be familiar with the idea of a deductive system being '(semantically) complete' or 'complete with respect to its standard semantics'. For example, a natural deduction system for propositional logic is said to be semantically complete when every inference which is semantically valid (i.e. truth-table valid) can be shown to be valid by a proof in the deductive system. *But a theory's having a (semantically) complete logic in this sense is one thing, being a (negation) complete theory is something else entirely.*[10] For example, by hypothesis T_1 has a (semantically) complete truth-functional *logic*, but is not a (negation) complete *theory*. Later we will meet e.g. a formal arithmetic which we label 'PA'. This theory uses a standard quantificational deductive logic, which again is a (semantically) complete *logic*: but we show that Gödel's First Theorem applies so PA is not a (negation) complete *theory*. Do watch out for this double use of the term 'complete', which is unfortunately now entirely entrenched: you just have to learn to live with it.[11]

[10]Putting it symbolically may help. To say that a logic is complete is to say that,

For any set of sentences Σ, and any φ, if $\Sigma \vDash \varphi$ then $\Sigma \vdash \varphi$

where '$\vDash$' signifies the relation of semantic consequence, and '$\vdash$' signifies the relation of formal deducibility. While to say that a theory T with the set of axioms Σ is complete is to say that

For any sentence φ, either $\Sigma \vdash \varphi$ or $\Sigma \vdash \neg\varphi$

[11]As it happens, the first proof of the semantic completeness of a proof-system for *quantificational logic* was also due to Gödel, and indeed the result is often referred to as 'Gödel's Completeness Theorem' (Gödel, 1929). This is evidently not to be confused with his (First) Incompleteness Theorem, which concerns the negation incompleteness of certain *theories of arithmetic*.

2.6 Three simple results

Deploying our notion of effective enumerability, we can now state and prove three elementary results. Suppose T is an axiomatized formal theory. Then:

1. The set of wffs of T can be effectively enumerated.
2. The set of proofs constructible in T can be effectively enumerated.
3. The set of theorems of T can be effectively enumerated.

Proof sketch for (1) By hypothesis, T has a finite basic alphabet; and we can give an algorithm for mechanically enumerating all the possible finite strings of symbols formed from a finite alphabet. For example, start by listing all the strings of length 1, followed by all those of length 2 in some 'alphabetical order', followed by those of length 3 and so on. By the definition of an axiomatized theory, there is a mechanical procedure for deciding which of these symbol strings form wffs. So, putting these procedures together, as we ploddingly generate the possible strings we can throw away all the non-wffs that turn up, leaving us with an effective enumeration of all the wffs. ⊠

Proof sketch for (2) Assume that T-proofs are just linear sequences of wffs. Now, just as we can enumerate all the possible wffs, we can enumerate all the possible *sequences* of wffs in some 'alphabetical order'. One brute-force way is again to start enumerating all possible strings of symbols, and throw away any that isn't a sequence of wffs. By the definition of an axiomatized theory, there is then an algorithmic recipe for deciding which of these sequences of wffs are well-formed proofs in the theory (since for each wff it is decidable whether it is either an axiom or follows from previous wffs in the list by allowed inference moves). So as we go along we can mechanically select out the proof sequences from the other sequences of wffs, to give us an effective enumeration of all the possible proofs. (If T-proofs are more complex, non-linear, arrays of wffs – as in tree systems – then the construction of an effective enumeration of the arrays needs to be correspondingly more complex: but the core proof-idea remains the same.) ⊠

Proof sketch for (3) Start enumerating proofs. But this time, just record their conclusions (when those are sentences, i.e. closed wffs). This mechanically generated list now contains all and only the theorems of the theory. ⊠

Two comments about these proof sketches. (a) Our talk about listing strings of symbols in 'alphabetical order' can be cashed out in various ways. In fact, *any* systematic mechanical ordering will do here. Here's one simple device (it prefigures the use of 'Gödel numbering', which we'll encounter later). Suppose, to keep things easy, the theory has a basic alphabet of less than ten symbols (this is no real restriction). With each of the basic symbols of the theory we

correlate a different digit from '1, 2, ..., 9'; we will reserve '0' to indicate some punctuation mark, say a comma. So, corresponding to each finite sequence of symbols there will be a sequence of digits, which we can read as expressing a number. For example: suppose we set up a theory using just the symbols

$$\neg, \rightarrow, \forall, (,), \mathsf{F}, \mathsf{x}, \mathsf{c}, '$$

and we associate these symbols with the digits '1' to '9' in order. Then e.g. the wff

$$\forall \mathsf{x}(\mathsf{Fx} \rightarrow \neg\forall \mathsf{x}'\neg \mathsf{F}''\mathsf{cx}')$$

(where 'F''' is a two-place predicate) would be associated with the number

374672137916998795

We can now list off the wffs constructible from this vocabulary as follows. We examine each number in turn, from 1 upwards. It will be decidable whether the standard base-ten numeral for that number codes a sequence of the symbols which forms a wff, since we are dealing with an formal theory. If the number does correspond to a wff φ, we enter φ onto our list of wffs. In this way, we mechanically produce a list of wffs – which obviously must contain all wffs since to any wff corresponds some numeral by our coding. Similarly, taking each number in turn, it will be decidable whether its numeral corresponds to a series of symbols which forms a sequence of wffs separated by commas (remember, we reserved '0' to encode commas).

(b) More importantly, we should note that to say that the theorems of a formal axiomatized theory can be mechanically *enumerated* is not to say that the theory is *decidable*. It is one thing to have a mechanical method which is bound to generate every theorem eventually; it is quite another thing to have a mechanical method which, given an arbitrary wff φ, can determine whether it will ever turn up on the list of theorems.

2.7 Negation complete theories are decidable

Despite that last point, however, we do have the following important result in the special case of negation-complete theories:[12]

> **Theorem 2** *A consistent, axiomatized, negation-complete formal theory is decidable.*

[12] A version of this result using a formal notion of decidability is proved by Janiczak (1950), though it is difficult to believe that our informal version wasn't earlier folklore. By the way, it is trivial that an *inconsistent* axiomatized theory is decidable. For if T is inconsistent, it entails every wff of T's language by the classical principle *ex contradictione quodlibet*. So all we have to do to determine whether φ is a T-theorem is to decide whether φ is a wff of T's language, which by hypothesis you can if T is an axiomatized formal theory.

Proof Let φ be any sentence (i.e. closed wff) of T. Set going the algorithm for effectively enumerating the theorems of T. Since, by hypothesis, T is negation-complete, either φ is a theorem of T or $\neg\varphi$ is. So it is guaranteed that – within a finite number of steps – either φ or $\neg\varphi$ will be produced in our enumeration. If φ is produced, this means that it is a theorem. If on the other hand $\neg\varphi$ is produced, this means that φ is not a theorem, for the theory is assumed to be consistent. Hence, in this case, there *is* a dumbly mechanical procedure for deciding whether φ is a theorem. ⊠

We are, of course, relying here on our ultra-generous notion of decidability-in-principle we explained above (in Section 2.3). We might have to twiddle our thumbs for an immense time before one of φ or $\neg\varphi$ to turn up. Still, our 'wait and see' method is guaranteed in this case to produce a result *eventually*, in an entirely mechanical way – so this counts as an effectively computable procedure in our official generous sense.

3 Capturing Numerical Properties

The previous chapter outlined the general idea of an axiomatized formal theory. This chapter introduces some key concepts we need in describing formal arithmetics. But we need to start with some quick ...

3.1 Remarks on notation

Gödel's First Incompleteness Theorem is about the limitations of axiomatized formal theories of arithmetic: if a theory T satisfies some fairly minimal constraints, we can find arithmetical truths which T can't prove. Evidently, in discussing Gödel's result, it will be *very* important to be clear about when we are working 'inside' a formal theory T and when we are talking informally 'outside' the theory (e.g. in order to establish things that T can't prove).

However, (a) we do want our informal talk to be compact and perspicuous. Hence we will tend to borrow the standard logical notation from our formal languages for use in augmenting mathematical English (so, for example, we will write '$\forall x \forall y(x + y = y + x)$' as a compact way of expressing the 'ordinary' arithmetic truth that the order in which you sum numbers doesn't matter).

Equally, (b) we will want our formal wffs to be readable. Hence we will tend to use notation in building our formal languages that is already familiar from informal mathematics (so, for example, if we want to express the addition function in a formal arithmetic, we will use the usual sign '+', rather than some unhelpfully anonymous two-place function symbol like 'f^2_3').

This two-way borrowing of notation will inevitably make expressions of informal arithmetic and their formal counterparts look very similar. And while context alone should no doubt make it pretty clear which is which, it is best to have a way of explicitly marking the distinction. To that end, *we will adopt the convention of using a* sans-serif *font for expressions in our formal languages.* Thus compare ...

$$\forall x \forall y(x + y = y + x) \qquad \forall \mathsf{x} \forall \mathsf{y}(\mathsf{x} + \mathsf{y} = \mathsf{y} + \mathsf{x})$$

$$\exists y\, y = S0 \qquad \exists \mathsf{y}\, \mathsf{y} = \mathsf{S0}$$

$$1 + 2 = 3 \qquad \overline{\mathsf{1}} + \overline{\mathsf{2}} = \overline{\mathsf{3}}$$

The expressions on the left will belong to our mathematicians'/logicians' augmented English (borrowing 'S' to mean 'the successor of'): the expressions on the right are wffs – or abbreviations for wffs – of one of our formal languages, with the symbols chosen to be reminiscent of their intended interpretations.

In talking about formal theories, we need to generalize *about* formal expressions, as when we define negation completeness by saying that for any wff φ, the theory T implies either φ or its negation $\neg\varphi$. We'll mostly use Greek letters for this kind of 'metalinguistic' role: note then that *these* symbols belong to logicians' augmented English: Greek letters will never belong to our formal languages themselves.

So what is going on when we say that the negation of φ is $\neg\varphi$, when we are apparently mixing a symbol from augmented English with a symbol from a formal language? Answer: there are hidden quotation marks, and '$\neg\varphi$' is to be read (of course) as meaning

> the expression that consists of the negation sign '$\neg$' followed by φ.

Sometimes, when being really pernickety, logicians use so-called Quine-quotes or corner-quotes when writing mixed expressions containing both formal and metalinguistic symbols (thus: $\ulcorner\neg\varphi\urcorner$). But this is excessive: no one will get confused by our more casual (and entirely standard) practice. In any case, we'll need to use corner-quotes later for a different purpose.

We'll be very relaxed about ordinary quotation marks too. We've so far been punctilious about using them when mentioning, as opposed to using, wffs and other formal expressions. But from now on, we will normally drop them other than around single symbols. Again, no confusion should ensue.

Finally, we will also be pretty relaxed about dropping unnecessary brackets in formal expressions (and we'll change the shape of pairs of brackets, and occasionally insert redundant ones, when that aids readability).

3.2 L_A and other languages

There is no single language which could reasonably be called *the* language for formal arithmetic: rather, there is quite a variety of different languages, apt for framing theories of different strengths.

However, the core theories of arithmetic that we'll be discussing first are mostly framed in the language L_A, i.e. the interpreted language $\langle\mathcal{L}_A, \mathcal{I}_A\rangle$, which is a formalized version of what we called 'the language of basic arithmetic' in Section 1.1. So here let's begin by characterizing L_A:

1. Syntax: the non-logical vocabulary of $\mathcal{L}_A$ is $\{0, \mathsf{S}, +, \times\}$, where

    a) '0' is a constant;

    b) 'S' is a one-place function-expression[1];

    c) '$+$' and '$\times$' are two-place function-expressions.

[1] In using 'S' rather than 's', we depart from the normal logical and mathematical practice of using upper-case letters for predicates and lower-case letters for functions: but this particular departure is sanctioned by common usage.

The logical vocabulary of $\mathcal{L}_A$ involves at least a standard selection of connectives, a supply of variables, the usual (first-order) quantifiers, and the identity symbol. The details are not critical – though, for convenience, we'll take it that in this and other languages with variables, the variables come in a standard ordered list starting $\mathsf{x}, \mathsf{y}, \mathsf{z}, \mathsf{u}, \mathsf{v}, \ldots$. If we want to use a natural deduction system where inferences can deploy parameters ('arbitrary names'), then we'll need a stock of those too, e.g. $\mathsf{a}, \mathsf{b}, \mathsf{c}, \ldots$

2. Semantics: the interpretation $\mathcal{I}_A$ assigns the set of natural numbers to be the domain of quantification. And it gives items of $\mathcal{L}_A$'s non-logical vocabulary their natural readings, so

   a) '$\mathsf{0}$' denotes zero.

   b) 'S' expresses the successor function (which maps one number to the next one); so the extension of 'S' is the set of all pairs of numbers $\langle n, n+1\rangle$.

   c) '$+$' and '$\times$' are similarly given the natural interpretations.

   Finally, the logical apparatus of $\mathcal{L}_A$ receives the usual semantic treatment.

Variants on L_A which we'll meet later include more restricted languages (which e.g. lack a multiplication sign or even lack quantificational devices) and richer languages (with additional non-logical vocabulary and/or additional logical apparatus). Details will emerge as we go along.

Almost all the variants we'll consider share with L_A the following two features: they (a) include a constant '$\mathsf{0}$' which is to be interpreted as denoting zero, and they (b) include a function-expression 'S' for the successor function which maps each number to the next one. Now, in any arithmetical language with these two features, we can form the referring expressions $\mathsf{0}$, $\mathsf{S0}$, $\mathsf{SS0}$, $\mathsf{SSS0}$, … to pick out individual natural numbers. We'll call these expressions the *standard numerals*. And you might very naturally expect that *any* theory of arithmetic will involve a language with (something equivalent to) standard numerals in this sense.

However, on reflection, this isn't obviously the case. For consider the following line of thought. As is familiar, we can introduce *numerical quantifiers* as abbreviations for expressions formed with the usual quantifiers and identity. Thus, for example, we can say that there are exactly two Fs using the wff

$$\exists_2 \mathsf{x}\mathsf{F}\mathsf{x} =_{\text{def}} \exists\mathsf{u}\exists\mathsf{v}\{(\mathsf{Fu} \wedge \mathsf{Fv}) \;\wedge\; \mathsf{u} \neq \mathsf{v} \;\wedge\; \forall\mathsf{w}(\mathsf{Fw} \rightarrow (\mathsf{w} = \mathsf{u} \vee \mathsf{w} = \mathsf{v}))\}$$

Similarly we can define $\exists_3\mathsf{x}\mathsf{G}\mathsf{x}$ which says that there are exactly three Gs, and so on. Then the wff

$$\{\exists_2\mathsf{x}\mathsf{Fx} \wedge \exists_3\mathsf{x}\mathsf{Gx} \wedge \forall\mathsf{x}\neg(\mathsf{Fx} \wedge \mathsf{Gx})\} \rightarrow \exists_5\mathsf{x}(\mathsf{Fx} \vee \mathsf{Gx})$$

says that if there are two Fs and three Gs and no overlap, then there are five things which are F-or-G – and *this is a theorem of pure first-order logic.*

So here, at any rate, we find something that looks pretty arithmetical and yet the numerals are now plainly *not* operating as name-like expressions. To explain the significance of *this* use of numerals as quantifier-subscripts we do not need to find mathematical entities for them to denote: we just give the rules for unpacking the shorthand expressions like '$\exists_2 x Fx$'. And the next question to raise is: can we regard arithmetical statements such as '$2 + 3 = 5$' as in effect really informal shorthand for wffs like the one just displayed, where the numerals operate as quantifier-subscripts and not as referring expressions?

We are going to have to put this question on hold for a long time. But it is worth emphasizing that it isn't *obvious* that a systematic theory of arithmetic *must* involve a standard, number-denoting, language. Still, having made this point, we'll just note again that the theories of arithmetic that we'll be discussing for the present *do* involve languages like L_A which have standard numerals that operate as denoting expressions.[2]

3.3 Expressing numerical properties and relations

A competent formal theory of arithmetic should be able to talk about a lot more than just the successor function, addition and multiplication. But 'talk about' *how*? Suppose that theory T has a arithmetical language L which like L_A has

[2]The issue whether we need to regiment arithmetic using a language with standard numerals is evidently connected to the metaphysical issue whether we need to regard numbers as in some sense genuine *objects*, there to be referred to by the numerals, and hence available to populate a domain of quantification. (The idea – of course – is not that numbers are *physical* objects, things that we can kick around or causally interact with in other ways: they are *abstract* objects.) But what is the relationship between these issues?

You might naturally think that we need *first* to get a handle on the metaphysical question whether numbers exist; and only then – once we've settled that – are we in a position to judge whether we can make literally true claims using a language with standard numerals which purport to refer to numbers (if numbers don't really exist, then presumably we can't). But another view – perhaps Frege's – says that the 'natural' line gets things exactly upside down. Claims like 'two plus three is five', 'there is a prime number less than five', 'for every prime number, there is a greater one' are straightforwardly correct by everyday arithmetical criteria (and what other criteria should we use?). And in fact we can't systematically translate away number-denoting terms by using numerals-as-quantifier-subscripts. Rather, 'there is' and 'every' in such claims have all the logical characteristics of common-or-garden quantifiers (obey all the usual logical rules); and the term 'three' interacts with the quantifiers as we'd expect from a referring expression (so from 'three is prime and less than five' we can infer 'there is a prime number less than five'; and from 'for every prime number, there is a greater one' and 'three is prime' we can infer 'there is a prime greater than three). Which on the Fregean view settles the matter; *that's just what it takes for a term like 'three' to be a genuine denoting expression referring to an object.*

We certainly can't further investigate here the dispute between the 'metaphysics first' view and the rival 'logical analysis first' position. And fortunately, later discussions in this book don't hang on settling this rather murky issue. Philosophical enthusiasts can pursue one strand of the debate through e.g. (Dummett, 1973, ch. 14), (Wright, 1983, chs 1, 2), (Hale, 1987), (Field, 1989, particularly chs 1, 5), (Dummett, 1991, chs 15–18), (Balaguer, 1998, ch. 5), (Hale and Wright, 2001, chs 6–9).

standard numerals (given their natural interpretations, of course): and let's start by examining how such a theory can express various monadic properties.

First, four notational conventions:

1. We will henceforth use '$\overline{1}$' as an abbreviation for '$\mathsf{S0}$', '$\overline{2}$' as an abbreviation for '$\mathsf{SS0}$', and so on.
2. And since we need to be able to generalize, we want some generic way of representing the standard numeral '$\mathsf{SS}\ldots\mathsf{S0}$' with n occurrences of 'S': we will extend the overlining convention and write $\overline{\mathsf{n}}$.[3]
3. We'll allow ourselves to write e.g. '$(\overline{1} \times \overline{2})$' rather than '$\times(\overline{1}, \overline{2})$'.
4. We'll also abbreviate a wff of the form $\neg\, \alpha = \beta$ by the corresponding wff $\alpha \neq \beta$, and thus write e.g. $\mathsf{0} \neq \overline{1}$.

Consider, for a first example, formal L-wffs of the form

(a) $\quad \psi(\overline{\mathsf{n}}) =_{\text{def}} \exists \mathsf{v}(\overline{2} \times \mathsf{v} = \overline{\mathsf{n}})$

So, for example, for $n = 4$, '$\psi(\overline{\mathsf{n}})$' unpacks into '$\exists \mathsf{v}(\mathsf{SS0} \times \mathsf{v} = \mathsf{SSSS0})$'. It is obvious that

if n is even, then $\psi(\overline{\mathsf{n}})$ is true,
if n isn't even, then $\neg\psi(\overline{\mathsf{n}})$ is true,

where we mean, of course, true on the arithmetic interpretation built into L. Relatedly, then, consider the corresponding open wff

(a′) $\quad \psi(\mathsf{x}) =_{\text{def}} \exists \mathsf{v}(\overline{2} \times \mathsf{v} = \mathsf{x})$

This wff is *satisfied* by the number n, i.e. is true of n, just when $\psi(\overline{\mathsf{n}})$ is true, i.e. just when n is even. Or to put it another way, the open wff $\psi(\mathsf{x})$ has the set of even numbers as its extension. Which means that our open wff *expresses* the property *even*, at least in the sense that the wff has the right extension.

Another example: n has the property of being prime if it is greater than one, and its only factors are one and itself. Or equivalently, n is prime just in case it is not 1, and of any two numbers that multiply to give n, one of them must be 1. So consider the wff

(b) $\quad \psi'(\overline{\mathsf{n}}) =_{\text{def}} (\overline{\mathsf{n}} \neq \overline{1} \;\wedge\; \forall \mathsf{u} \forall \mathsf{v}(\mathsf{u} \times \mathsf{v} = \overline{\mathsf{n}} = \rightarrow\; (\mathsf{u} = \overline{1} \;\vee\; \mathsf{v} = \overline{1})))$

This holds just in case n is prime, i.e.

if n is prime, then $\psi'(\overline{\mathsf{n}})$ is true,
if n isn't prime, then $\neg\psi'(\overline{\mathsf{n}})$ is true.

[3] It might be said that there is an element of notational overkill here: we are in effect using overlining to indicate that we are using an abbreviation convention inside our formal language, so we don't also need to use the sans serif font to mark a formal expression. But ours is a fault on the good side, given the importance of being completely clear when we are working with formal expressions and when we are talking informally.

Relatedly, the corresponding open wff

(b′) $\psi'(\mathsf{x}) =_{\text{def}} (\mathsf{x} \neq \overline{1} \wedge \forall \mathsf{u} \forall \mathsf{v}(\mathsf{u} \times \mathsf{v} = \mathsf{x} \rightarrow (\mathsf{u} = \overline{1} \vee \mathsf{v} = \overline{1})))$

is satisfied by exactly the prime numbers. Hence $\psi'(\mathsf{x})$ expresses the property *prime*, again in the sense of getting the right extension. (For more on properties and their extensions, see Section 7.1.)

In this sort of way, any formal theory with limited basic resources can come to express a whole variety of arithmetical properties by means of complex open wffs with the right extensions. And our examples motivate the following official definition:

> A property P is *expressed* by the open wff $\varphi(\mathsf{x})$ in an (interpreted) arithmetical language L just if, for every n,
> if n has the property P, then $\varphi(\overline{\mathsf{n}})$ is true,
> if n does not have the property P, then $\neg\varphi(\overline{\mathsf{n}})$ is true.

'True' of course continues to mean true on the given interpretation built into L.

We can now extend our definition in the obvious way to cover relations. Note, for example, that in a theory with language like L_A

(c) $\psi(\overline{\mathsf{m}}, \overline{\mathsf{n}}) =_{\text{def}} \exists \mathsf{v}(\mathsf{Sv} + \overline{\mathsf{m}} = \overline{\mathsf{n}})$

is true just in case $m < n$. And so it is natural to say that the corresponding open wff

(c′) $\psi(\mathsf{x}, \mathsf{y}) =_{\text{def}} \exists \mathsf{v}(\mathsf{Sv} + \mathsf{x} = \mathsf{y})$

expresses the relation *less than*, in the sense of getting the extension right. Generalizing again:

> A two-place relation R is expressed by the open wff $\varphi(\mathsf{x}, \mathsf{y})$ in an (interpreted) arithmetical language L just if, for any m, n,
> if m has the relation R to n, then $\varphi(\overline{\mathsf{m}}, \overline{\mathsf{n}})$ is true,
> if m does not have the relation R to n, then $\neg\varphi(\overline{\mathsf{m}}, \overline{\mathsf{n}})$ is true.

Likewise for many-place relations.

3.4 Case-by-case capturing

Of course, we don't merely want various properties of numbers to be *expressible* in the language of a formal theory of arithmetic in the sense just defined. We also want to be able to use the theory to *prove* facts about which numbers have which properties.

Now, it is a banal observation that some arithmetical facts are a lot easier to establish than others. In particular, to establish facts about *individual* numbers typically requires much less sophisticated proof-techniques than proving general truths about *all* numbers. To take a dramatic example, there's a school-room

mechanical routine for testing any given even number to see whether it is the sum of two primes. But while, case by case, every even number that has ever been checked passes the test, no one knows how to prove Goldbach's conjecture – i.e. knows how to prove 'in one fell swoop' that *every* even number greater than two is the sum of two primes.

Let's focus then on the relatively unambitious task of proving that particular numbers have or lack a certain property on a case-by-case basis. This level of task is reflected in the following definition concerning formal provability:

> A property P is *case-by-case captured* by the open wff $\varphi(\mathsf{x})$ of the arithmetical theory T just if, for any n,
>
> if n has the property P, then $T \vdash \varphi(\overline{\mathsf{n}})$,
>
> if n does not have the property P, then $T \vdash \neg\varphi(\overline{\mathsf{n}})$.

For example, in theories of arithmetic T with very modest axioms, the open wff $\psi(\mathsf{x}) =_{\text{def}} \exists\mathsf{v}(\overline{2} \times \mathsf{v} = \mathsf{x})$ not only expresses but case-by-case captures the property *even*. In other words, for each even n, T can prove $\psi(\overline{\mathsf{n}})$, and for each odd n, T can prove $\neg\psi(\overline{\mathsf{n}})$.[4] Likewise, in the same theories, the open wff $\psi'(\mathsf{x})$ from the previous section not only expresses but case-by-case captures the property *prime*.

As you would expect, extending the notion of 'case-by-case capturing' to the case of relations is straightforward:

> A two-place relation R is case-by-case captured by the open wff $\varphi(\mathsf{x}, \mathsf{y})$ of the arithmetical theory T just if, for any m, n,
>
> if m has the relation R to n, then $T \vdash \varphi(\overline{\mathsf{m}}, \overline{\mathsf{n}})$
>
> if m does not have the relation R to n, then $T \vdash \neg\varphi(\overline{\mathsf{m}}, \overline{\mathsf{n}})$.

Likewise for many-place relations.[5]

Now, suppose T is a *sound* theory of arithmetic – i.e. one whose axioms are true on the given arithmetic interpretation of its language and whose logic is truth-preserving. Then T's theorems are all true. Hence if $T \vdash \varphi(\overline{\mathsf{n}})$, then $\varphi(\overline{\mathsf{n}})$ is true. And if $T \vdash \neg\varphi(\overline{\mathsf{n}})$, then $\neg\varphi(\overline{\mathsf{n}})$ is true. Which shows that *if $\varphi(\mathsf{x})$ captures P in the sound theory T then, a fortiori, T's language expresses P.*

[4]We in fact show this in Section 5.3.

[5]This note just co-ordinates our definition with another found in the literature. Assume T is a *consistent* theory of arithmetic, and P is case-by-case captured by $\varphi(x)$. Then by definition, if n does not have property P, then $T \vdash \neg\varphi(\overline{\mathsf{n}})$; so by consistency, if n does not have property P, then not-($T \vdash \varphi(\overline{\mathsf{n}})$); so contraposing, if $T \vdash \varphi(\overline{\mathsf{n}})$, then n has property P. Similarly, if $T \vdash \neg\varphi(\overline{\mathsf{n}})$, then n doesn't have property P. Hence, assuming T's consistency, we could equally well have adopted the following alternative definition (which strengthens two 'if's to 'if and only if'): A property P is case-by-case captured by the open wff $\varphi(\mathsf{x})$ of theory T just if, for any n,

> n has the property P if and only if $T \vdash \varphi(\overline{\mathsf{n}})$,
>
> n does not have the property P if and only if $T \vdash \neg\varphi(\overline{\mathsf{n}})$.

3.5 A note on our jargon

A little later, we'll need the notion of a formal theory's capturing numerical *functions* (as well as properties and relations). But there is a slight complication in that case, so let's not delay over it here; instead we'll immediately press on in the next chapter to apply the concepts that we've already defined.

However, I should just pause to note frankly that my talk of an open wff's 'case-by-case capturing' a numerical property is a bit deviant. It is more usually said that P is 'numeralwise expressed' or that it is 'strongly represented' by a $\varphi(\mathsf{x})$ satisfying our conditions for capture. But I'm unapologetic: the usual jargon is in some danger of engendering confusion.

Whatever your favoured jargon, however, the key thing is to be absolutely clear about the distinction we need to mark – so let's highlight it again. Whether a property P is *expressible* in a theory just depends on the richness of its *language*. Whether a property P can be *case-by-case captured* by that theory depends on the richness of its *proof-system*. It will turn out that for any respectable theory of arithmetic T there are numerical properties that are expressible but not capturable in T (see e.g. Section 12.6).

4 Sufficiently Strong Arithmetics

Starting in Chapter 5, we'll be examining some formal theories of arithmetic 'from the bottom up', in the sense of first setting down the axioms of the theories and then exploring what the various theories are capable of proving. In this present chapter, however, we proceed the other way about. We introduce the concept of a *sufficiently strong* theory of arithmetic, which is a theory that by definition *can* prove what we'd like any moderately competent theory of arithmetic to be able to prove about decidable properties of numbers. We then establish some easy but quite deep results about such theories.

4.1 The idea of a 'sufficiently strong' theory

Suppose that P is any decidable property of numbers, i.e. one for which we have a mechanical algorithm for deciding, given a natural number n, whether n has property P or not (see Section 2.3).

Now, when we construct a formal theory of the arithmetic of the natural numbers, we will surely want deductions inside our theory to be able to track any mechanical calculation that we can already perform informally. After all, we don't want going formal to *diminish* our ability to determine whether n has property P. Formalization aims at regimenting what we can already do: it isn't supposed to hobble our efforts. So – while we might have some passing interest in very limited theories – we will mainly want to aim for a formal theory T which (a) is able to frame some open wff $\varphi(\mathsf{x})$ which expresses the decidable property P, and (b) is such that if n has property P, $T \vdash \varphi(\overline{\mathsf{n}})$, and if n does not have property P, $T \vdash \neg\varphi(\overline{\mathsf{n}})$.

Let's say, for short, that a theory T captures a numerical property P if there is an open wff of the theory which case-by-case captures P (in the sense of Section 3.4). Then the suggestion is that, if P is a decidable property of numbers, we ideally want a competent theory of arithmetic T to be able to capture P. And this thought motivates the following definition:

> A formal theory of arithmetic T is *sufficiently strong* if it case-by-case captures all decidable numerical properties.

The presumption is that it is a reasonable and desirable condition on a formal theory of the arithmetic of the natural numbers that it be sufficiently strong. Much later, when we've done some closer analysis of the idea of decidability, we'll be in a position to specify some theories that do indeed meet this condition, and thereby show that the condition of being 'sufficiently strong' is actually

easily met. This chapter, however, just supposes that there *are* such theories and derives some consequences.

(It is in fact more usual to define being 'sufficiently strong' as a matter of capturing not only all decidable properties but also all decidable relations and all computable functions too. But since we haven't yet defined what it is to capture a function, and since the arguments of this chapter in any case don't depend on that notion, we might as well stick with our weaker definition of sufficient strength.)

4.2 An undecidability theorem

A trivial way of a theory T's being sufficiently strong (i.e. proving lots of wffs about properties of individual numbers) is by being inconsistent (i.e. proving *every* wff about individual numbers). It goes without saying, however, that we are interested in *consistent* theories.

We also like to get *decidable* theories when we can, i.e. theories for which there is an algorithm for determining whether a given wff is a theorem (see Section 2.5). But, sadly, we have the following key result:[1]

> **Theorem 3** *No consistent, sufficiently strong, axiomatized formal theory of arithmetic is decidable.*

Proof We suppose T is a consistent and sufficiently strong axiomatized theory yet also decidable, and derive a contradiction.

By Result 1 of Section 2.6, we know that the set of wffs of T can be effectively enumerated. It is a mechanical business to select from this enumeration the open wffs with (say) 'x' as the sole free variable, and discard the rest. This gives us an effective enumeration of the wffs of T with this one free variable,

$$\varphi_0(\mathsf{x}), \varphi_1(\mathsf{x}), \varphi_2(\mathsf{x}), \ldots$$

And now let's fix on the following definition:

$$n \text{ has the property } D \text{ if and only if wff } T \vdash \neg\varphi_n(\bar{\mathsf{n}})$$

where $\bar{\mathsf{n}}$ is the standard numeral in T which denotes n. Note that the key construction here involves linking the subscripted index with the numeral substituted for the free variable in $\neg\varphi_n(\mathsf{x})$: so this is a cousin of the 'diagonal' construction which we encountered in Section 2.4 (see the comment on the proof of Theorem 1).

We next show that the supposition that T is a decidable theory entails that the 'diagonal' property D is an effectively decidable property of numbers. For

[1] The undecidability of arithmetic was first proved in (Church, 1936). The direct proof given here can be extracted from Theorem 1 of (Tarski et al., 1953, pp. 46–49), but the first published version of our informal version which I know is (Hunter, 1971, pp. 224–225), though the argument was certainly earlier folklore.

given any number n, it will be a mechanical matter to enumerate the open wffs with 'x' as the sole free variable until the n-th one, $\varphi_n(\mathsf{x})$, is produced. Then it is a mechanical matter to form the numeral $\overline{\mathsf{n}}$, substitute it for the free variable in $\varphi_n(\mathsf{x})$, and prefix a negation sign. Now we just apply the supposed mechanical decision procedure for T to decide whether the resulting wff $\neg\varphi_n(\overline{\mathsf{n}})$ is a theorem. Hence there is an algorithmic way of deciding whether n has the property D.

Since, by hypothesis, the theory T is sufficiently strong, it is can capture all decidable numerical properties: so it follows that, in particular, D is capturable, by some wff $\varphi(\mathsf{x})$. This wff must of course occur somewhere in our enumeration of all such wffs. Let's suppose the d-th wff in the enumeration does the trick. That is to say, property D is case-by-case captured by $\varphi_d(\mathsf{x})$.

It is now entirely routine to get out a contradiction. For, by definition, to say that $\varphi_d(\mathsf{x})$ captures D means that for any n,

> if n has the property D, $T \vdash \varphi_d(\overline{\mathsf{n}})$,
> if n doesn't have the property D, $T \vdash \neg\varphi_d(\overline{\mathsf{n}})$.

So taking in particular the case $n = d$, we have

> (i) if d has the property D, $T \vdash \varphi_d(\overline{\mathsf{d}})$,
> (ii) if d doesn't have the property D, $T \vdash \neg\varphi_d(\overline{\mathsf{d}})$.

But now note that our initial definition of the property D implies in particular:

> (iii) d has the property D if and only if $T \vdash \neg\varphi_d(\overline{\mathsf{d}})$.

From (ii) and (iii), it follows that whether d has property D or not, the wff $\neg\varphi_d(\overline{\mathsf{d}})$ is a theorem either way. So by (iii) again, d does have property D, hence by (i) the wff $\varphi_d(\overline{\mathsf{d}})$ must be a theorem too. So a wff and its negation are both theorems of T. Therefore T is inconsistent, contradicting our initial assumption that T is consistent.

In sum, the supposition that T is a consistent and sufficiently strong axiomatized formal theory of arithmetic *and* decidable leads to contradiction. ⊠

Which is a beautiful result: indeed it is one of the delights of our topic: we can get exciting theorems fast!

There's an old hope (which goes back to Leibniz) that can be put in modern terms like this: we might one day be able to mechanize mathematical reasoning to the point that a suitably primed computer could solve all mathematical problems in a domain by deciding theoremhood in an appropriate formal theory. What we've just shown is that this is a false hope: as soon as a theory is strong enough to capture all boringly mechanical reasoning about individual numbers, it must cease to be decidable.

4.3 An incompleteness theorem

Now let's put together Theorem 2 (established in Section 2.7) and Theorem 3.

> **Theorem 2** *A consistent, axiomatized, negation-complete formal theory is decidable.*

> **Theorem 3** *No consistent, sufficiently strong, axiomatized formal theory of arithmetic is decidable.*

These, of course, immediately entail

> **Theorem 4** *A consistent, sufficiently strong, axiomatic formal theory of arithmetic cannot also be negation complete.*

That is to say, for any c.s.s.a. (consistent, sufficiently strong, axiomatic) theory of arithmetic, there will be some pair of sentences φ and $\neg\varphi$, neither of which are theorems. But one of these must be true on the given interpretation of T's language. So, for any c.s.s.a. arithmetic T, there are true-but-unprovable wffs in T.

And adding in new axioms won't help. To re-play the sort of argument we gave in Section 1.1, suppose T is a c.s.s.a. theory of arithmetic, and suppose G_T is a true sentence of arithmetic that T can't prove or disprove. The theory U which you get by adding G_T as a new axiom to T will, of course, now trivially prove G_T, so we've plugged that gap. But note that U is consistent (for if U, i.e. $T + \mathsf{G}_T$, were inconsistent, then by reductio, $T \vdash \neg\mathsf{G}_T$, contrary to hypothesis). And U is sufficiently strong (since it can still prove everything T can prove). It is still decidable which wffs are axioms of U, so the theory still counts as a properly axiomatized formal theory. So Theorem 4 applies, and the new c.s.s.a. theory U must therefore contain a wff G_U (distinct from G_T, of course) which is again true-on-interpretation but unprovable. So T is not only incomplete but in a good sense incompletable.

4.4 The truths of arithmetic can't be axiomatized

Here's another pair of definitions.

i. A set of wffs Σ is *axiomatizable* if there is an axiomatized formal theory T such that, for any wff φ, $\varphi \in \Sigma$ if and only if $T \vdash \varphi$.

ii. An interpreted language L is *sufficiently rich* if it can express every decidable property of numbers.

Then, as an immediate corollary of Theorem 4, we have

> **Theorem 5** *The set of truths of a sufficiently rich language L is unaxiomatizable.*

Proof Suppose that L is sufficiently rich, and we'll suppose – for reductio – that the set of true wffs of L *can* be axiomatized by a theory T. Then T must be negation complete – since for every wff ψ of L, either ψ or $\neg\psi$ is true, and by hypothesis the true one is a theorem.

But let P be any decidable property of numbers. Since L is sufficiently rich, there is some open wff φ such that, for any n,

if n has the property P, $\varphi(\overline{\mathsf{n}})$ is true,
if n doesn't have the property P, $\neg\varphi(\overline{\mathsf{n}})$ is true.

Since T entails all the truths, it follows that for any n

if n has the property P, $T \vdash \varphi(\overline{\mathsf{n}})$,
if n doesn't have the property P, $\neg\varphi(\overline{\mathsf{n}})$.

Since P was an arbitrary decidable property, this means that T must be sufficiently strong (by definition of the notion of sufficient strength). But T is consistent, since by hypothesis it only contains truths. So, by Theorem 4, T is not negation-complete after all. Contradiction! ⊠

Now, the informal idea of (all) 'the truths of arithmetic' is no doubt not a sharp one. But however we refine it, presumably we want it to include at least the truths about the nice, decidable, properties of numbers. So in our jargon, the truths of arithmetic, on any plausible sharpening of that idea, should be the truths of a sufficiently rich language. So our new Theorem warrants the informal claim expressed in the title of this section: the truths of arithmetic can't be axiomatized.

4.5 But what have we really shown?

Back to Theorem 4. This isn't yet Gödel's Theorem, but it is a terrific result.

Or is it? That depends, you might say, on the integrity of the very idea of a sufficient strong theory of arithmetic, i.e. the idea of a theory that captures *all* decidable properties of numbers. But is this idea even coherent? Maybe the notion of capturing 'all' decidable properties is somehow problematic. Compare: perhaps there is something problematic in talking about *all* sets – for it seems that as soon as we have a lot of sets which together purport to be all of them, we can in fact form another set, i.e. the new set containing all those prior sets. Maybe there is something comparably troublesome about the idea of all decidable properties of numbers.

This initial worry, however, turns out to be groundless. As we've already briefly indicated in Section 2.3, there are a number of standard, well-understood, fully coherent ways of formally refining the intuitive notion of decidability, ways that all turn out to locate the same entirely definite and well-defined class of numerical properties (namely those whose application can be decided by a Turing machine). And this formal result, as we noted, prompts what we called Turing's Thesis, namely that the intuitive notion of a decidable property does indeed successfully pick out the determinate class of Turing-decidable properties.

We will later have to return to justify these claims: but for the moment, suppose we buy into Turing's Thesis. This means that the idea of a sufficiently

strong theory is conceptually in good order. It also means that our argument for Theorem 4 which invokes the informal notion of computation and of what can be decided by a computer is cogent. However, couldn't it still be the case that a language for capturing *all* computable properties has to be *very* rich – involving (say) an infinity of different fundamental predicates, all waiting to be governed by their own axioms?[2] So couldn't the moral of our Theorem just be that there can't be complete theories of all the arithmetical truths expressible in certain ultra-rich languages? That would still leave open the possibility that there could be complete theories governing the propositions expressible in (say) the much more restricted language of basic arithmetic. But we announced right back in Section 1.1 that Gödel's own result rules out complete theories even of the truths of basic arithmetic. Hence, if our easy Theorem 4 is to have the reach of Gödel's Theorem, we'll need to show that a theory with the restricted language of basic arithmetic can still be sufficiently strong.

The state of play is therefore this: if our informal style of argument for Theorem 4 is to be used establish something like Gödel's own result, then it needs to be augmented with (i) a defence of Turing's Thesis, *and* (ii) a proof that some axiomatized theory of basic arithmetic is indeed sufficiently strong. And even with (i) and (ii) in play, there would still remain the very significant difference that our easy Theorem only tells us that there is (somewhere or other) an unprovable sentence of a sufficiently strong arithmetic. The proof of the official First Incompleteness Theorem actually tells us how to take a theory T and construct a true but unprovable-in-T sentence of a simple form. And it does this – albeit after an amount of hard work – without needing (i) and without needing all of (ii) either.

In sum, there *is* a very significant gap between our intriguing, quickly-derived, but informal Theorem 4 and the industrial-strength Theorem that Gödel proves. So, while what we have shown in this chapter is highly suggestive, it is time to start turning to Gödel's own arguments. We need first to look at some formal arithmetics in more detail and then to introduce the key notion of primitive recursion.

[2]See Section 6.7 for a description of the theory PBA which has this kind of language.

5 Three Formalized Arithmetics

In this chapter, we move on from the generalities of the previous chapters, and look at three standard formal theories of arithmetic. These theories differ in strength, but they share the following features:

1. Zero, the successor function, addition and multiplication are treated as primitive notions governed by basic axioms, and are not defined in terms of anything more fundamental.
2. The theories' deductive apparatus is no stronger than familiar first-order logic (and in particular doesn't allow second-order quantification, i.e. quantification over properties rather than objects).

It is absolutely standard to start by considering formal theories of arithmetic with these features, though later we'll be looking at some theories which lack them.

5.1 BA – Baby Arithmetic

We begin with a *very* simple formal arithmetic which 'knows' about the addition of particular numbers, 'knows' its multiplication tables, but can't express general facts about numbers at all because it lacks quantifiers and variables. Hence our label *baby arithmetic*, or BA for short.

First, we describe BA's language, $L_B = \langle \mathcal{L}_B, \mathcal{I}_B \rangle$. $\mathcal{L}_B$'s non-logical vocabulary is the same as that of $\mathcal{L}_A$ (Section 3.2): so there is a single individual constant '0', the one-place function symbol 'S', and the two-place function symbols '$+$' and '$\times$'. But $\mathcal{L}_B$'s logical apparatus comprises only the propositional connectives and the identity sign.

The intended interpretation $\mathcal{I}_B$ is the obvious one. '0' denotes zero. 'S' signifies the successor function, and '$+$' and '$\times$' are interpreted as addition and multiplication.

Second, we need to give BA some deductive apparatus. Choose your favourite system of propositional logic to deal with connectives. Then add some standard rules to deal with the identity sign. In particular, we need some version of Leibniz's Law. Let's say that a *term* is either a numeral or an expression built up from numerals by one or more applications of our three functions, as in $\mathsf{S0} + \mathsf{S0}$, $\mathsf{S}(\mathsf{S0} \times (\mathsf{SS0} + \mathsf{SSS0}))$, etc. Then, if τ and ρ are terms, the Law allows us to infer $\varphi(\rho)$ from the premisses $\varphi(\tau)$ and $\tau = \rho$ or $\rho = \tau$.

Third, we have to fix the axioms of BA. To start with, we want to pin down at least the following facts about the structure of the number sequence: (1) Zero is

the *first* number, i.e. isn't a successor; so for every n, $0 \neq Sn$. (2) The number sequence never circles back on itself; so different numbers have different successors – or contraposing, for any m, n, if $Sm = Sn$ then $m = n$.

We haven't got quantifiers in BA's language, however, so we can't express these general facts directly. Rather, we need to employ *schemas*, and say: *any wff that you get from one of the following schemas by substituting standard numerals for the place-holders 'ζ', 'ξ' is an axiom.*

Schema 1 $\quad 0 \neq \mathsf{S}\zeta$

Schema 2 $\quad \mathsf{S}\zeta = \mathsf{S}\xi \rightarrow \zeta = \xi$

(It should go without saying that the substitution rule is: same place-holder, same replacement numeral.)

We'll quickly show that instances of these schemas do indeed entail that different terms in the sequence $\mathsf{0}$, $\mathsf{S0}$, $\mathsf{SS0}$, $\mathsf{SSS0}$, ..., pick out different numbers. Recall, we use '$\overline{\mathsf{n}}$' to represent the numeral $\mathsf{SS}\ldots\mathsf{S0}$ with n occurrences of 'S': so the result we need is that, for any m, n, if $m \neq n$, then $\mathsf{BA} \vdash \overline{\mathsf{m}} \neq \overline{\mathsf{n}}$.

Proof Suppose $m \neq n$ (and let $|m - n| - 1 = j \geq 0$). And assume $\overline{\mathsf{m}} = \overline{\mathsf{n}}$ as a temporary supposition in BA (that's a supposition of the form $\mathsf{SS}\ldots\mathsf{S0} = \mathsf{SS}\ldots\mathsf{S0}$, with m occurrences of 'S' on the left and n on the right). We can now use instances of Schema 2 plus modus ponens to repeatedly strip off initial occurrences of 'S', one on each side of the identity, until either (i) we derive $\mathsf{0} = \mathsf{S}\overline{\mathsf{j}}$, or else (ii) we derive $\mathsf{S}\overline{\mathsf{j}} = \mathsf{0}$ and then use the symmetry of identity to conclude $\mathsf{0} = \mathsf{S}\overline{\mathsf{j}}$. But $\mathsf{0} \neq \mathsf{S}\overline{\mathsf{j}}$ is an axiom (an instance of Schema 1). Contradiction. So, $\overline{m} \neq \overline{n}$ follows by reductio. Which proves that if $m \neq n$, then $\mathsf{BA} \vdash \overline{\mathsf{m}} \neq \overline{\mathsf{n}}$. ⊠

Next we pin down the addition function by saying that any wff that you get by substituting numerals in the following is also an axiom:

Schema 3 $\quad \zeta + \mathsf{0} = \zeta$

Schema 4 $\quad \zeta + \mathsf{S}\xi = \mathsf{S}(\zeta + \xi)$

Instances of Schema 3 tell us the result of adding zero. Instances of Schema 4 with 'ξ' replaced by '0' tell us how to add one (i.e. add $S0$) in terms of adding zero and then applying the successor function to the result. Then, once we know about adding one, we can use further instances of Schema 4 with 'ξ' replaced by 'S0' to tell us how to add two ($SS0$) in terms of adding $S0$. We can then invoke the same Schema again to tell us how to add three ($SSS0$) in terms of adding two: and so on and so forth, thus defining addition for every number.

We can similarly pin down the multiplication function by first defining multiplication by zero, and then defining multiplication by Sn in terms of multiplication by n and addition. Thus we want every numeral instance of the following to be axioms too:

Schema 5 $\quad \zeta \times \mathsf{0} = \mathsf{0}$

Schema 6 $\quad \zeta \times \mathsf{S}\xi = (\zeta \times \xi) + \zeta$

These, together with the previous axioms, can then be used to prove all the familiar basic arithmetical truths about the addition and multiplication of particular numbers.[1]

To illustrate, here's a BA derivation of $\overline{2} \times \overline{1} = \overline{2}$, or rather (putting that in unabbreviated form) of $\mathsf{SS0} \times \mathsf{S0} = \mathsf{SS0}$.

1.	$\mathsf{SS0} \times \mathsf{0} = \mathsf{0}$	Instance of Schema 5
2	$\mathsf{SS0} \times \mathsf{S0} = (\mathsf{SS0} \times \mathsf{0}) + \mathsf{SS0}$	Instance of Schema 6
3.	$\mathsf{SS0} \times \mathsf{S0} = \mathsf{0} + \mathsf{SS0}$	From 1, 2 by LL

('LL' of course indicates the use of Leibniz's Law which allows us to intersubstitute identicals.) To proceed, we now need to show that $\mathsf{0} + \mathsf{SS0} = \mathsf{SS0}$ – and note, this *isn't* an instance of Schema 3. So

4.	$\mathsf{0} + \mathsf{0} = \mathsf{0}$	Instance of Schema 3
5.	$\mathsf{0} + \mathsf{S0} = \mathsf{S}(\mathsf{0} + \mathsf{0})$	Instance of Schema 4
6.	$\mathsf{0} + \mathsf{S0} = \mathsf{S0}$	From 4, 5 by LL
7.	$\mathsf{0} + \mathsf{SS0} = \mathsf{S}(\mathsf{0} + \mathsf{S0})$	Instance of Schema 4
8.	$\mathsf{0} + \mathsf{SS0} = \mathsf{SS0}$	From 6, 7 by LL

Which gives us what we want:

9.	$\mathsf{SS0} \times \mathsf{S0} = \mathsf{SS0}$	From 3, 8 by LL

That's a bit laborious, but it works. And a little reflection on this short proof reveals that similar proofs will enable us to derive the value of *any* sum or product of two numerals.

Now we generalize. Let's say that an *equation* of BA is a wff of the form $\tau = \rho$, where τ and ρ are terms. Then we have the following:

1. If $\tau = \rho$ is true, then $\mathsf{BA} \vdash \tau = \rho$.
2. If $\tau = \rho$ is false, then $\mathsf{BA} \vdash \tau \neq \rho$.

Proof sketch for (1) Our sample proof above illustrates the sort of BA derivation that will prove any true simple equation of the type $\overline{\mathsf{j}} + \overline{\mathsf{k}} = \overline{\mathsf{m}}$ or $\overline{\mathsf{j}} \times \overline{\mathsf{k}} = \overline{\mathsf{n}}$. Given a more complex term τ, involving nested additions and multiplications (or applications of the successor function), we can then prove a true wff of the form $\tau = \overline{\mathsf{t}}$ with a numeral on the right by repeated steps of evaluating inner-most brackets.

To take a mini-example, suppose τ has the shape $((\overline{\mathsf{j}} + \overline{\mathsf{k}}) + \mathsf{S}(\overline{\mathsf{j}} \times \overline{\mathsf{k}}))$. Then we first prove identities evaluating the inner-most bracketed expressions; substituting the results using Leibniz's Law will enable us to derive something like

$$(\overline{\mathsf{j}} + \overline{\mathsf{k}}) + \mathsf{S}(\overline{\mathsf{j}} \times \overline{\mathsf{k}})) = (\overline{\mathsf{m}} + \mathsf{S}\overline{\mathsf{n}})$$

[1] And note, it is evidently decidable whether a wff is an instance of one the six Schemas, and so it is decidable whether a wff is an axiom of BA, as is required if BA is indeed to be an axiomatized theory.

Now evaluate the new, simpler, bracketed expression on the right by proving something of the form $\overline{\mathsf{m}} + \mathsf{S}\overline{\mathsf{n}} = \overline{\mathsf{t}}$. Hence, using Leibniz's Law again, we get

$$(\bar{\mathsf{j}} + \bar{\mathsf{k}}) + \mathsf{S}(\bar{\mathsf{j}} \times \bar{\mathsf{k}})) = \bar{\mathsf{t}}$$

The method of repeated substitutions always works: for *any* complex term τ we'll be able to prove a wff correctly equating its value to that of some numeral.

So, generalizing further, given any *two* terms τ and ρ, if they have the same value so $\tau = \rho$ is true, then we'll be able to prove $\tau = \rho$ by proving each term equal to the same numeral. ⊠

Proof sketch for (2) Suppose that two complex terms τ and ρ have values m and n, where $m \neq n$. By the argument for (1), we'll then be able to prove a pair of wffs of the form $\tau = \overline{\mathsf{m}}$, $\rho = \overline{\mathsf{n}}$. But we've already shown earlier in this section that if $m \neq n$, BA proves $\overline{\mathsf{m}} \neq \overline{\mathsf{n}}$. So, if $m \neq n$, a BA proof of $\tau \neq \rho$ follows using Leibniz's Law twice. ⊠

These two results in turn imply

Theorem 6 BA *is a negation-complete theory.*

Proof sketch for Theorem 6 Note that $\mathcal{L}_B$ has only one primitive predicate, the identity relation. So the only 'atomic' claims expressible in BA are equations involving terms; all other wffs are truth-functional combinations of such equations. But we've just seen that we can (1) prove each true 'atom' and (2) prove the negation of each false 'atom'. So, by ordinary propositional logic, we can derive any true truth-functional combination of atoms (equations), i.e. prove any true wff. Hence, for any wff φ of BA, since either φ or $\neg\varphi$ is true, either φ or $\neg\varphi$ is a theorem. So BA is negation-complete. ⊠

Since BA is complete, it is decidable, by Theorem 2. But of course we don't need a brute-force search through possible derivations in order to determine whether a wff φ is a BA theorem. For note that all BA theorems are true (since the axioms are); and all true BA-wffs are theorems (as we've just seen). Hence deciding whether the BA-wff φ is true decides whether it is a theorem. But any such φ expresses a truth-function of equations, so we can mechanically work out whether it is true or not by using school-room arithmetic for the equations and then using a truth-table.

5.2 Q – Robinson Arithmetic

So far, then, so good. But the reason that Baby Arithmetic manages to prove every correct claim *that it can express* – and is therefore negation complete by our definition – is that it is can't express very much. In particular, it can't express any generalizations at all. BA's completeness comes at the price of being expressively impoverished.

The obvious way to start beefing up BA into something more exciting is to restore the familiar apparatus of quantifiers and variables. So let's keep the same non-logical vocabulary, but now allow ourselves the full resources of first-order logic, so that we are working with the full language $L_A = \langle \mathcal{L}_A, \mathcal{I}_A \rangle$. of basic arithmetic (see Section 3.2).

Since we now have the quantifiers available to express generality, we can replace each metalinguistic Schema (specifying an infinite number of particular axioms) by a single object-language Axiom. For example, we can replace the first two Schemas governing the successor function by

Axiom 1 $\forall x(0 \neq Sx)$

Axiom 2 $\forall x \forall y(Sx = Sy \rightarrow x = y)$

Each instance our earlier Schemas 1 and 2 can be deduced from the corresponding Axiom by one or two applications of Universal Instantiation.

Note, however, that while these Axioms tell us that zero isn't a successor, they leave it open that there are other objects in the domain of quantification (pseudo-zeros, if you like) that aren't successors either. Using quantifiers we can now explicitly rule that out:

Axiom 3 $\forall x(x \neq 0 \; \rightarrow \; \exists y(x = Sy))$

Next, we can similarly replace our previous Schemas for addition and multiplication by universally quantified Axioms:

Axiom 4 $\forall x(x + 0 = x)$

Axiom 5 $\forall x \forall y(x + Sy = S(x + y))$

Axiom 6 $\forall x(x \times 0 = 0)$

Axiom 7 $\forall x \forall y(x \times Sy = (x \times y) + x)$

The formalized theory with language L_A, Axioms 1 to 7, and a standard first-order logic is called *Robinson Arithmetic*, or (very often) simply Q.[2]

Now, the syntactic system $\mathcal{L}_B$ is contained in the syntactic system $\mathcal{L}_A$ in the sense that every $\mathcal{L}_B$-wff is an $\mathcal{L}_A$-wff (but not vice-versa). And since any BA Axiom – i.e. any instance of one of our previous Schemas – can be derived from one of our new Q Axioms, every $\mathcal{L}_B$-wff that can be proved in BA is equally an $\mathcal{L}_A$-wff which can be proved in Q. Hence, for any such *shared* sentence φ, Q does as well as BA; it can either prove φ or prove $\neg\varphi$.

However, Q can't prove or disprove *every* sentence of its full language $\mathcal{L}_A$. We've already announced that Gödel's First Theorem is in fact going to show that *no* axiomatized theory of basic arithmetic can be negation-complete. But

[2]This formal system was first isolated in (Robinson, 1952) and immediately became well-known through the classic (Tarski et al., 1953); its particular interest will emerge in Chapter 9.

we certainly don't need to invoke Gödel to see that Q is incomplete, as we'll now demonstrate.

Given the quantified form of its Axioms, Q can prove *some* generalized claims, such as $\forall x\forall y(x + SSy = SS(x + y))$ or $\forall x(x + S0 = S0 \rightarrow x = 0)$. But there are other equally simple generalizations about numbers that can't be proved in Q. For example, by Theorem 6* we can prove any particular wff of the form $\overline{m} + \overline{n} = \overline{n} + \overline{m}$. But the universally quantified version $\forall x\forall y(x + y = y + x)$ can't be proved from our Axioms.

Proof sketch One standard procedure to show that a wff φ is not a theorem of a given theory T is to find an interpretation (often a deviant, unintended interpretation) for the T-wffs, which makes the axioms of T true and hence all its theorems true, but which makes φ false. So we want to find a deviant, unintended, interpretation of Q's Axioms which would make them true but for which 'addition' fails to commute. Here's an artificial – but still legitimate – example.

Take the domain of our deviant, unintended, interpretation of Q to be the set N^* comprising the natural numbers but with two other 'rogue' elements a and b added (say Gwyneth Paltrow and Jude Law, or any other pair that takes your fancy). Let '0' still to refer to zero. And take 'S' now to pick out the successor* function S^* which is defined as follows: $S^*n = Sn$ for any natural number in the domain, while for our rogue elements $S^*a = a$, and $S^*b = b$. It is immediate that Axioms 1 to 3 are still true on this deviant interpretation.

We now need to extend this model to re-interpret Q's function '+'. Suppose we take this to pick out addition*, where $m +^* n = m + n$ for any natural numbers m, n in the domain, while $a +^* n = a$ and $b +^* n = b$. Further, for any x (whether number or rogue element), $x +^* a = b$ and $x +^* b = a$. It is easily checked that interpreting '+' as addition* still makes Axioms 4 and 5 true. But by construction, $a +^* b \neq b +^* a$, so this interpretation makes $\forall x\forall y(x + y = y + x)$ false.

We are not quite done, however, as we still need to show that we can give a co-ordinate re-interpretation of '$\times$' in Q by some deviant multiplication* function. But we can leave it as an exercise to fill in suitable details. ⊠

So Q cannot prove $\varphi =_{\text{def}} \forall x\forall y(x + y = y + x)$; and since φ is true, and Q (having true axioms and a truth-preserving logic) can only prove truths, Q cannot prove $\neg\varphi$ either. So Q is indeed negation-incomplete.

5.3 Capturing properties and relations in Q

While Q is in some ways very weak since it can't prove many of the true basic generalizations about arithmetical operations, it can still case-by-case capture lots of properties and relations in the sense of Section 3.4. Which shouldn't be at all surprising given that case-by-case capturing doesn't require proving general, quantified wffs: you only need to be able to derive wffs about particular numbers.

To take a simple example, let's show that the *less than* relation is not only expressed but is case-by-case captured by the wff $\exists \mathsf{v}(\mathsf{Sv} + \mathsf{x} = \mathsf{y})$ in Q. That is to say, for any particular pair of numbers, Q can prove that the first is less than the second (if it is) or prove that it isn't (if it isn't).

Proof sketch (1) Suppose $m < n$, so for some $k \geq 0$, $Sk + m = n$. We know that Q can prove everything BA proves and hence, in particular, can prove every true equation. So we have $\mathsf{Q} \vdash \mathsf{S}\overline{\mathsf{k}} + \overline{\mathsf{m}} = \overline{\mathsf{n}}$. But $\mathsf{S}\overline{\mathsf{k}} + \overline{\mathsf{m}} = \overline{\mathsf{n}} \vdash \exists \mathsf{v}(\mathsf{Sv} + \overline{\mathsf{m}} = \overline{\mathsf{n}})$ by existential quantifier introduction. Therefore $\mathsf{Q} \vdash \exists \mathsf{v}(\mathsf{Sv} + \overline{\mathsf{m}} = \overline{\mathsf{n}})$, as was to be shown.

(2) Suppose $m \not< n$. We need to show $\mathsf{Q} \vdash \neg\exists \mathsf{v}(\mathsf{Sv} + \overline{\mathsf{m}} = \overline{\mathsf{n}})$. We'll first demonstrate this in the case where $m = 2$, $n = 1$. And for illustrative purposes, we will work with a standard Fitch-style natural deduction system (where we can introduce 'arbitrary names' and always indent sub-proofs). So consider the following outline argument with some inference steps slightly compressed:

1.	$\quad\exists \mathsf{v}(\mathsf{Sv} + \mathsf{SS0} = \mathsf{S0})$	Supposition
2.	$\qquad\mathsf{Sa} + \mathsf{SS0} = \mathsf{S0}$	Supposition
3.	$\qquad\mathsf{Sa} + \mathsf{SS0} = \mathsf{S(Sa + S0)}$	From Axiom 5
4.	$\qquad\mathsf{S(Sa + S0)} = \mathsf{S0}$	From 2, 3 by LL
5.	$\qquad(\mathsf{Sa} + \mathsf{S0}) = \mathsf{S(Sa + 0)}$	From Axiom 5
6.	$\qquad\mathsf{SS(Sa + 0)} = \mathsf{S0}$	From 4, 5 by LL
7.	$\qquad(\mathsf{Sa} + 0) = \mathsf{Sa}$	From Axiom 4
8.	$\qquad\mathsf{SSSa} = \mathsf{S0}$	From 6, 7 by LL
9.	$\qquad\mathsf{SSSa} = \mathsf{S0} \to \mathsf{SSa} = 0$	From Axiom 2
10.	$\qquad\mathsf{SSa} = 0$	From 8, 9 by MP
11.	$\qquad 0 = \mathsf{SSa}$	From 10
12.	$\qquad 0 \neq \mathsf{SSa}$	From Axiom 1
13.	Contradiction!	From 11, 12
14.	Contradiction!	∃E 1, 2–13
15.	$\neg\exists \mathsf{v}(\mathsf{Sv} + \mathsf{SS0} = \mathsf{S0})$	RAA 1–14.

The only step to explain is at line (14) where we use a version of Existential Elimination – the idea is that if the supposition $\varphi(\mathsf{a})$ leads to contradiction, for arbitrary a, then $\exists \mathsf{v}\varphi(\mathsf{v})$ also leads to contradiction. And inspection of this proof immediately reveals that we can use the same basic pattern of argument to show that $\mathsf{Q} \vdash \neg\exists \mathsf{v}(\mathsf{Sv} + \overline{\mathsf{m}} = \overline{\mathsf{n}})$ whenever $m \not< n$. So we are done. ⊠

That was straightforward. And with a little more effort, we could now go on to show e.g. that the wff $\exists \mathsf{v}(\overline{2} \times \mathsf{v} = \mathsf{x})$ not only expresses but captures the property of being *even*. We won't pause to demonstrate that, however. For we'll see in Chapter 9 that a certain very large class of properties and relations can *all* be case-by-case captured in Q; it just isn't worth giving further proofs in a piecemeal way.

5.4 Introducing '$<$' and '$\leq$' into Q

Given our result in the last section, it is convenient henceforth to use the abbreviatory symbol '$<$', defined so that $\alpha < \beta =_{\text{def}} \exists \mathsf{v}(\mathsf{Sv} + \alpha = \beta)$.[3]

We now can go on to prove inside Q various key facts about the *less than* relation captured by $\mathsf{x} < \mathsf{y}$. For example, we can readily show that $\mathsf{Q} \vdash \neg\exists\mathsf{x}\,\mathsf{x} < 0$, i.e. $\mathsf{Q} \vdash \neg\exists\mathsf{x}\exists\mathsf{v}(\mathsf{Sv} + \mathsf{x} = 0)$. The strategy is to prove that the assumption $\mathsf{Sa} + \mathsf{b} = 0$ leads to contradiction. For by Axiom 3, either $\mathsf{b} = 0$, or $\mathsf{b} = \mathsf{Sc}$ for some c. But we can't have the first, or else $\mathsf{Sa} + 0 = \mathsf{Sa} = 0$, contradicting Axiom 1. And we can't have the second, or else $\mathsf{Sa} + \mathsf{Sc} = \mathsf{S}(\mathsf{Sa} + \mathsf{c}) = 0$, again contradicting Axiom 1.

Note also that we can show that $\exists\mathsf{v}(\mathsf{v} + \mathsf{x} = \mathsf{y})$ captures the relation *less than or equal to* in Q. Which similarly motivates adding the abbreviatory symbol '$\leq$' with the definition $\alpha \leq \beta =_{\text{def}} \exists\mathsf{v}(\mathsf{v} + \alpha = \beta)$.

The addition of '$<$' and '$\leq$' thus defined does make for economy and readability; so henceforth we'll use these symbols in Q and stronger theories without further ado. We should remark, though, that some presentations treat '$<$' and/or '$\leq$' as primitive symbols built into these theories from the start, governed by their own additional axioms. Nothing important hangs on the difference between that approach and our policy of introducing these symbols by definition.

5.5 Induction and the Induction Schema

We saw that Q cannot prove $\forall\mathsf{x}\forall\mathsf{y}(\mathsf{x} + \mathsf{y} = \mathsf{y} + \mathsf{x})$. To derive this wff, and prove many other wffs of L_A that are beyond Q's reach, we need to add some further formal principle for proving quantified wffs.

The obvious candidate is some version of the following *induction principle*:

> Suppose (i) 0 has the numerical property P. And suppose (ii) for any number n, if it has P, then its successor Sn also has P. Then we can conclude that (iii) *all* numbers have property P.

Why does this hold? By hypothesis (i), 0 has P. By (ii), if 0 has P so does $S0$. Hence $S0$ has P. By (ii) again, if $S0$ has P so does $SS0$. Hence $SS0$ has P. Likewise, $SSS0$ has P. And so on and so forth, through all the successors of 0. But the successors of 0 are the only natural numbers. So *all* natural numbers have property P. (The induction principle is therefore underwritten by the basic structure of the number sequence, and in particular by the absence of 'stray' numbers that you can't get to step-by-step from zero.)

Now, our informal induction principle is a generalization covering any genuine numerical property P. Hence to frame a corresponding formal version, we'd ideally like to be able to generalize over all properties, which would mean using

[3] Depending exactly how we set up our formal languages, we might have to be careful about clash of variables when using such an abbreviation; but we're not going to fuss about the details.

a second-order quantifier. But at the moment, as we announced at the beginning of this chapter, we are concentrating on formal theories whose logical apparatus involves no more than the familiar first-order quantifiers which range over a domain of numbers, not their properties. What to do?

We'll have to use a schema again (compare BA where we had to use schemas because we then couldn't even quantify over *objects*: now we are using a schema because even in a full first-order language we can't quantify over *properties*). So – as a first shot – let's say that *any instance of the*

$$\textbf{Induction Schema} \quad (\{\varphi(0) \wedge \forall x(\varphi(x) \rightarrow \varphi(Sx))\} \rightarrow \forall x\varphi(x))$$

is to count as an axiom – where $\varphi(x)$ stands in for some formula of L_A with just 'x' free, and $\varphi(0)$ and $\varphi(Sx)$ are of course the results of replacing occurrences of 'x' in that formula with '0' and 'Sx' respectively.

What is the motivation for this? Well, the wff $\varphi(x)$ is constructed from just the constant term '0', the successor, addition and multiplication functions, plus identity and other logical apparatus. So such a wff surely expresses a perfectly determinate arithmetical property – a property to which the intuitive induction principle therefore applies. Hence the relevant instance of the Induction Schema will be true.

So far, so good. But we will want to use inductive arguments to prove general results about relations as well as about monadic properties. So how can we handle relations? Take, for example, the open wff 'Rxyz'. Simply substituting this into the Schema will yield the wff

$$(\{R0yz \wedge \forall x(Rxyz \rightarrow R(Sx)yz\} \rightarrow \forall xRxyz)$$

which is still *open*, with the variables 'y', 'z' left dangling free. However, back in Section 2.2, we stipulated that axioms for a formal theory should be *closed* wffs (as they need to be if we are to talk without qualification about a bunch of axioms being *true* on the intended interpretation). So our open wff won't do as an induction axiom. Consider, however, its *universal closure* – i.e. the wff which we get by prefixing universal quantifiers to bind the free variables:

$$\forall y\forall z(\{R0yz \wedge \forall x(Rxyz \rightarrow R(Sx)yz\} \rightarrow \forall xRxyz)$$

This is again true, given our informal induction principle. For consider 'Rxab' where 'a' and 'b' are arbitrary constants. Then this open wff will express a monadic property, and induction will apply to this property. Hence the corresponding instance of the Induction Schema where 'Rxab' is substituted will be true. But 'a' and 'b' were arbitrary. So we can generalize into the places held by them. Which gives us the universal closure of the instance of the Schema for 'Rxyz'. And the argument evidently generalizes to to other relational wffs.

So that motivates our second shot at laying down a bunch of induction axioms. We'll say that *the universal closure of any instance of the Induction Schema is an axiom*, where now $\varphi(x)$ stands in for some formula with at least 'x' free, and maybe other variables free as well.

One comment. We will later consider arithmetics with a second-order Induction Axiom which quantifies over all properties of numbers. But what counts as 'all'? At the generous end of the spectrum of possible views here, we might hold that *any* arbitrary set of numbers Σ corresponds to a genuine property P_Σ (where n is defined as having the property P_Σ if and only if $n \in \Sigma$). But we know from Theorem 1, comment (a), that there are non-enumerably many sets of numbers Σ. So, on the generous view, a second-order Induction Axiom covers non-enumerably many properties.

However, the open wffs $\varphi(\mathsf{x})$ of L_A are by contrast enumerable (we enumerate the wffs, and as we go along retain a list of those with just x free). So, the first-order Induction Schema covers only enumerably many numerical properties. Hence we will expect an arithmetic which uses the first-order Schema to be notably weaker than one that uses the second-order Axiom (at least if we take the generous view about properties). This expectation will be confirmed later.

5.6 PA – First-order Peano Arithmetic

Our discussion in the last section motivates moving on from Q – and jumping over some intermediate positions[4] – to adopt the much richer formal theory of arithmetic which we can initially define as follows:

> PA – First-order Peano Arithmetic[5] – is the result of adding to the axioms of Q the universal closures of all instances of the Induction Schema for formulae $\varphi(\mathsf{x})$ of L_A with at least 'x' free.

Plainly, it is decidable whether any given wff has the right shape to be one of the new axioms, so PA is a legitimate formalized theory.

In PA, we *can* now establish generalities like the law $\forall\mathsf{x}\forall\mathsf{y}(\mathsf{x}+\mathsf{y}=\mathsf{y}+\mathsf{x})$. However, we'll first illustrate the use of induction with a preliminary example: consider $\forall\mathsf{x}(0+\mathsf{x}=\mathsf{x})$ – i.e. a mirror version of Axiom 4. This wff is again unprovable in Q, as is shown by the fact that it is falsified on our deviant interpretation with rogue elements (since $0 +^* a \neq a$). But we can prove it by induction. To show this, we'll again assume a Fitch-style natural deduction logic, with the standard rules UI (Universal Instantiation), UG (Universal Generalization on 'arbitrary' names) and CP (Conditional Proof). To derive our target wff $\forall\mathsf{x}(0+\mathsf{x}=\mathsf{x})$, we will evidently need to use an instance of the Induction Schema with $\varphi(\mathsf{x})$ replaced by $(0+\mathsf{x}=\mathsf{x})$ and then aim to prove the two conjuncts in the antecedent of that instance, so we can extract the desired conclusion by a final modus ponens. Here's a formal version:

[4]For a discussion of some theories of arithmetic with various restrictions on the Induction Schema, see the wonderful (Hájek and Pudlák, 1983). These intermediate theories are technically interesting, but are not relevant for us at this point in the book, so we won't delay over them now.

[5]The name is conventional: Giuseppe Peano did publish a list of axioms for arithmetic in Peano (1889) – but they weren't first-order, only explicitly governed the successor relation, and were in any case, as he acknowledged, already to be found in Dedekind (1888).

1.	$(\{0+0=0 \wedge \forall x(0+x=x \to 0+Sx=Sx)\}$ $\to \forall x(0+x=x))$	Instance of Induction
2.	$\forall x(x+0=x)$	Axiom 4
3.	$0+0=0$	From 2 by UI with 0
4.	$\forall x\forall y(x+Sy=S(x+y))$	Axiom 5
5.	$\quad 0+a=a$	Supposition
6.	$\quad \forall y(0+Sy=S(0+y))$	From 4, by UI with 0
7.	$\quad 0+Sa=S(0+a)$	From 6 by UI with a
8.	$\quad 0+Sa=Sa$	From 5, 7 by LL
9.	$0+a=a \to 0+Sa=Sa$	From 5 to 8 by CP
10.	$\forall x(0+x=x \to 0+Sx=Sx)$	From 9 by UG
11.	$\{0+0=0 \wedge \forall x(0+x=x \to 0+Sx=Sx)\}$	From 3, 10 by $\wedge$-intro.
12.	$\forall x(0+x=x)$	From 10, 11 by MP

Now to show $\forall x\forall y(x+y=y+x)$. Put $\varphi(x,y) =_{\text{def}} (x+y=y+x)$, and note we have the following induction axiom:

$$\forall y(\{\varphi(0,y) \wedge \forall x(\varphi(x,y) \to \varphi(Sx,y))\} \to \forall x\varphi(x,y))$$

So given what we have just established, it will follow that $\forall x\forall y\varphi(x,y)$ if we can show that $\forall x\forall y(\varphi(x,y) \to \varphi(Sx,y))$. And we can indeed show this, by another use of induction (hint: changing variables, put $\psi(x) =_{\text{def}} \forall u(\varphi(u,x) \to \varphi(Su,x))$ and then use the instance of the Induction Schema for this open wff).

To be sure, this all rapidly gets more than a trifle tedious. But the point of the present exercise isn't user-friendliness but austere formal rigour: the game is to see what we really *need* by way of absolutely fundamental axioms in order to get standard arithmetical results. And a little investigation and experimentation should convince you that PA does indeed have the resources to establish all the familiar general truths about the addition and multiplication of numbers. True enough, we know – because we've repeatedly trailed the fact – that Gödel's First Theorem is going to show that PA (like any axiomatized theory of basic arithmetic) fails to be negation complete. This time, however, unlike the case with Q, there is no easy way of finding 'ordinary' arithmetic wffs which are true-but-unprovable.

Finally, we should remark on another instance of induction. Suppose we put $\varphi(x) =_{\text{def}} (x \neq 0 \to \exists y(x=Sy))$. Then $\varphi(0)$ is trivial. Likewise, $\forall x\varphi(Sx)$ is also immediate, and that entails $\forall x(\varphi(x) \to \varphi(Sx))$. So we can use an instance of the Induction Schema to derive $\forall x\varphi(x)$. But that's just Axiom 3 of Q. So our original presentation of PA – as having all the Axioms of Q plus the instances of the Induction Schema – involves a certain redundancy.

Given the very natural motivation we have for accepting its unrestricted use of the Induction Schema, PA is the benchmark axiomatized first-order theory of basic arithmetic. So, for neatness, let's bring together all the elements of its specification in one place.

First, the *language* of PA is L_A, a first-order language whose non-logical vocabulary comprises just the constant '0', the one-place function symbol 'S', and the two-place function symbols '+', '×', and whose interpretation is the obvious one.

Second, PA's deductive *proof system* is some standard version of classical first-order logic with identity (differences between versions aren't significant).

And third, its *axioms* – eliminating the redundancy from our original statement of the axioms – are

Axiom 1 $\forall x(0 \neq Sx)$

Axiom 2 $\forall x \forall y(Sx = Sy \to x = y)$

Axiom 3 $\forall x(x + 0 = x)$

Axiom 4 $\forall x \forall y(x + Sy = S(x + y))$

Axiom 5 $\forall x(x \times 0 = 0)$

Axiom 6 $\forall x \forall y(x \times Sy = (x \times y) + x)$

plus the universal closures of all instances of the following

Induction Schema $(\{\varphi(0) \wedge \forall x(\varphi(x) \to \varphi(Sx))\} \to \forall x \varphi(x))$

where $\varphi(x)$ is an open L_A-wff with at least the variable 'x' free.

5.7 Is PA consistent?

PA proves a great deal more than Q. But it wouldn't be much joy to discover that PA's much greater strength is due to the theory's actually tipping over into being *inconsistent* and so entailing every wff. So let's briefly pause over the issue of consistency – not because there is in fact a genuine concern that PA might be in trouble, but because it gives us a chance to mention a topic that will occupy us later.

Here's an argument that our semantic interpretation of PA already implies its consistency. For consider again the given interpretation $\mathcal{I}_A$ which is built into PA's language $L_A = \langle \mathcal{L}_A, \mathcal{I}_A \rangle$. On this interpretation, '0' denotes zero; 'S' represents the successor function, etc.; and the domain of quantification comprises just the natural numbers. Hence on this interpretation (1) the first three axioms inherited from Q are core truths about the operation that takes one number to its successor. And the next four axioms are equally fundamental truths about addition and multiplication. (2) Further, the informal induction principle for arithmetical properties and relations is warranted by our understanding of the structure of the number sequence. So since open wffs of L_A straightforwardly express genuine numerical properties and relations, (the closures of) all the instances of PA's Induction Schema will be true too on the standard interpretation $\mathcal{I}_A$. But (3) the classical first-order deductive logic of PA is truth-preserving so

– given that the axioms are true and PA's logical apparatus is in good order – all its theorems are true on $\mathcal{I}_A$. Hence (4), since that all PA theorems are true on $\mathcal{I}_A$, there cannot be pairs of theorems of the form φ and $\neg\varphi$ (for these of course couldn't both be true together). So (5) not every wff is a theorem, and the theory is therefore consistent.

This argument for the consistency of full PA appears to be compelling. But some might still urge that – despite appearances – we ought be more cautious here. They will want to emphasize the point that an argument for a theory's consistency which appeals to our supposed intuitive grasp of an intended interpretation *can* lead us badly astray. And to support their point, they will refer to one of the most famous episodes in the history of logic, which concerns the fate of the German logician Gottlob Frege's *The Basic Laws of Arithmetic*.[6]

Frege aimed to construct a formal system in which first arithmetic and then the theory of the real numbers can be rigorously developed. He gives axioms for what is in effect a theory of sets, so that the number sequence can be identified as a certain sequence of sets, and then rational and real numbers can be defined via appropriate sets of these sets. Frege takes as one of his Basic Laws the assumption that *for every well-constructed predicate* $\varphi(\mathsf{x})$ *of his language, there is a set (possibly empty) of exactly those things that satisfy this predicate.* And indeed, what could be more plausible? If we can coherently express some condition A, then we should surely be able to talk about the (possibly empty) collection of just those things that satisfy condition A.

But, famously, the assumption is disastrous. As Bertrand Russell pointed out in a letter which Frege received as the second volume of *Basic Laws* was going through the press, the plausible assumption leads to contradiction.[7] Take for example the condition R expressed by '... is a set which isn't a member of itself'. This is, on the face of it, a perfectly coherent condition (the *set of people*, for example, satisfies the condition: the set of people contains only people, so it doesn't contain any sets, so doesn't contain itself in particular). And certainly condition R is expressible in the language of Frege's system. So on Frege's assumption, there will be a set of things that satisfy R. In other words, there is a set Σ_R of all the sets which aren't members of themselves. But now ask: is Σ_R a member of itself? A moment's reflection shows that it is if it isn't, and isn't if it is: contradiction! So there can be no such set as Σ_R; hence Frege's assumption cannot be right, despite its intuitive appeal, and his formal system which embodies that assumption is inconsistent.

This sad tale brings home to us vividly that intuitions of consistency can be mistaken. But let's not rush to make too much of this: the fact that we *can* make mistakes in arguing for the cogency of a formal system on the basis of our supposed grasp of an intended interpretation isn't any evidence that we *have* made a mistake in our argument for the consistency of PA. For a start, Peano Arith-

[6] The first volume of *Basic Laws* was published in 1893, the second in 1903. For a partial translation, see Frege (1964).

[7] See (Russell, 1902).

metic and many stronger theories that embed it have been intensively explored for a century and no contradiction has been exposed.

'But can't we do better,' you might still ask, 'than make the negative point that no contradiction has been found (yet): can't we *prove* that PA is consistent in some other way than by appealing to our supposed grasp of its interpretation?'

Well, let's waive the misleading suggestion that our proof isn't secure as it stands. Then the answer is: yes, there are other proofs. However, we'll have to put further discussion of this intriguing issue on hold until after we have encountered Gödel's *Second* Incompleteness Theorem which does put some interesting limits on the possibilities here. We'll present this crucial result in Chapter 13.2.

5.8 More theories

We stressed at the outset of this chapter that our focus here was going to be pretty narrow. We have, in particular, only looked at three theories which use (no more than) first-order logic. We noted, however, that the intuitively compelling induction principle is naturally read as being second-order (i.e. as involving a generalization over all numerical properties). So one obvious project is to consider what happens when we move to arithmetics constructed in a second-order setting.

Another respect in which this present chapter is narrow is that it confines itself to theories which treat zero, the successor function, addition and multiplication as undefined notions, and the axioms governing them as fundamental. But theorists like Dedekind and Frege, Russell and Whitehead thought we ought to do more: we should seek to *justify* the axioms of arithmetic in terms of further definitions and logical principles. This points to another project, namely to consider the classical attempts to ground Peano Arithmetic by something more basic. And these two projects (considering second-order theories and considering deeper foundations for arithmetic) are related, because some of the classical foundational enterprises essentially use at least second-order logic.

We'll return to all this anon. The point to keep in mind in the meantime is that the theories of arithmetic we've met in this chapter *are* a restricted sample: there's a lot more to be said.

6 Primitive Recursive Functions

The formal theories of arithmetic that we've looked at so far have the successor function, addition and multiplication built in. But why stop there? School arithmetic acknowledges many more numerical functions. This chapter considers a very wide class of computable numerical functions.

6.1 Introducing p.r. functions

We'll start with two more functions from school arithmetic. First, take the factorial – standardly symbolized by '!' written *after* the function's argument, as in '$y!$' – where e.g. $4! = 1 \times 2 \times 3 \times 4$. This function 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 $1!$, $2!$, $3!$, ... Hence our two-clause definition 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 a given value of x and $y = 0$, 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 multiplication and addition functions. Informally, and now presented in the style of everyday mathematics (i.e. without explicit quantifiers), we have:

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

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

Three comments about our examples so far:

1. In each definition, an instance of the second clause can invoke the result of its own previous application to a smaller number, a kind of procedure which is standardly termed 'recursive'. So this sort of two-clause definition of a function is called a *definition by (primitive) recursion*.[1]

2. Note, for example, that $(Sn)!$ is defined as $n! \times Sn$, so it is evaluated by evaluating $n!$ and Sn and 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 examples illustrate *chains* of definitions by recursion and functional composition. Thus addition is defined in terms of the successor function; multiplication is defined in terms of successor and addition; the factorial (or equally, 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.

These examples motivate the following initial gesture towards a definition:

> A *primitive recursive function* is one that can be similarly characterized by a chain of definitions by recursion and composition (a chain ultimately starting with the successor function).[2]

But that is a rather quick-and-dirty characterization, even if it gets across the basic idea. We ought to pause to do better.

6.2 Defining the p.r. functions more carefully

Consider the recursive definition of the factorial again:

[1]Strictly speaking, we need a proof of the claim that recursive definitions really do well-define functions: the proof was first given by Dedekind (1888, §126).

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

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

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 prior to the definition of f (maybe because it is a 'primitive' function which we are allowed to take for granted like the successor function; or perhaps because we've already given recursion clauses to define it; or perhaps because it is a composite function constructed by plugging one known function into another).

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. To get the definition of addition to fit this pattern, we have to take $g(x)$ to be the trivial identity function $I(x) = x$; and 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. A neat way of doing this is to allow more identity functions like $I'(x, y, u) = u$, and then we can put $h(x, y, u) = SI'(x, y, u)$. Finally, to get the definition of multiplication to fit the pattern, $g(x)$ has to be treated as the even more trivial zero function $Z(x) = 0$.

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

> 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.*

If we allow $\vec{x}$ to be empty, so $g(\vec{x})$ is a constant, that subsumes the case of one-place functions like the factorial.

Now for 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 – as when, to

take the simplest case, we set $f(x) = h(g(x))$. Or when, to take a slightly more complex case, we set $f(x, y, z) = h(g(x, y), g'(y, z))$.

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

> 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 $f(\vec{x}, \vec{y}, \vec{z}) = h(\vec{x}, g(\vec{y}), \vec{z})$.

Then we can 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 $h(g(x, y), g'(y, z))$

Finally, 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 the definitional chain, it must bottom out with members of an initial 'starter pack' of trivially simple functions. In the previous section, we just highlighted the successor function among these given functions. But we've since noted that, if we are to get our examples to fit our official account of definition by primitive recursion, we'll have to acknowledge some other, even more trivial, functions. So let's now say that the full set of *initial functions* contains:

i. the successor function S;
ii. zero function, $Z(x) = 0$;
iii. and the identity functions, $I_i^k(\vec{x}) = x_i$ for each x_i in the array of k variables $\vec{x}$.

Hence, putting that all together, we can offer this more formal characterization of the p.r. functions:

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.

Note, by the way, that the initial functions are total functions of numbers (i.e. are defined for every numerical argument); and that primitive recursion and composition both build total functions out of total functions. Which means that all p.r. functions are total functions.

6.3 Defining p.r. properties and relations

As well as talking of p.r. *functions*, we will also need to speak of p.r. *properties* and *relations*.

We can tie talk of functions and talk of properties/relations together by using the very handy notion of a *characteristic function*. Here's a definition.

> 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* is simply for later convenience.

The numerical property P partitions the numbers into two sets, the set of numbers which have the property and the set of numbers which 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.

In what follows, we'll frequently 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, we can now say that a property is *decidable* – i.e. a suitably programmed computer can decide whether the property obtains – just if its characteristic function is *computable* (compare Section 2.3). And without further ado, we can now introduce the idea of a *p.r. property*, meaning – of course – a property with a p.r. characteristic function, and likewise a *p.r. relation* is a relation with a p.r. characteristic function.

6.4 Some more examples

We'll now give a few more examples from the class of p.r. properties and functions. Strictly speaking, you can skip this section since we only pick up its details again in later sections which you can also skip. On the other hand, we will – particularly in Section 10.3 – be making some crucial claims that various functions are p.r.; and these claims will seem a *lot* more plausible if you have already

worked through a handful of simpler cases and seen how to prove that various other properties and functions are primitive recursive. So it is probably worth at least browsing through what follows: but if you find yourself getting bogged down, then just move on.

As a preliminary, we just remark that the functions

$sg(y)$, where $sg(n) = 0$ for $n = 0$, and $sg(n) = 1$ otherwise
$\overline{sg}(y)$, where $\overline{sg}(n) = 1$ for $n = 0$, and $\overline{sg}(x) = 0$ otherwise

are both primitive recursive. To prove $sg(y)$ is p.r. just note the p.r. definition

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

where $SZ(u)$ is p.r. by composition, and $SZ(sg(y)) = S0 = 1$. We prove $\overline{sg}(y)$ is p.r. similarly.

And now we can note four very useful general facts:

Fact 1 Suppose $f(\vec{x})$ 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.*

Proof We illustrate with the case where f is a one-place function. 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 – is given by

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

Here $|u - v|$ is the absolute difference function we showed to be p.r. in Section 6.1, and so the right-hand side is a composition of p.r. functions. ⊠

Fact 2 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. So 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. So, generalizing, *any truth-functional combination of p.r. properties and relations is also p.r..*

Fact 3 As is standard, we'll henceforth use '$(\forall x \leq n)Cx$' as an abbreviation for '$\forall x(x \leq n \rightarrow Cx)$', and '$(\exists x \leq n)Cx$' as short for '$\exists x(x \leq n \wedge Cx)$', where the context C may contain other variables. Similarly, '$(\forall x < n)Cx$' is an abbreviation for '$\forall x(x < n \rightarrow Cx)$', and '$(\exists x < n)Cx$' is short for '$\exists x(x < n \ \wedge \ Cx)$'. And then we have the following: *a property or relation defined from a p.r property*

or relation by bounded quantifications is also p.r. This shouldn't be surprising given Fact 2 and the observation that bounded quantifications are in effect finite conjunctions or disjunctions.

To take the simplest case, suppose the monadic P is p.r.: then we'll show that the new property defined by

$$K(n) \equiv (\exists x \leq n)P(x)$$

is also primitive recursive. (Other cases are similar.)

Proof Suppose again that $p(x)$ is P's p.r. characteristic function. And by composition define the p.r. function $h(u, v) = (p(Su) \times v)$. We put

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

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, and then $k(n)$ goes to zero, and thereafter stays zero. Since k is p.r., K is p.r. too by definition. ⊠

Fact 4 We now introduce the standard *minimization* operator 'μx', to be read: 'the least x such that ...'. Much later, we'll be considering the general use of this operator, but here we will be concerned with *bounded minimization*. So we write, e.g.

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

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. *Then a one-place function defined from a p.r. property by bounded minimization is itself p.r.*

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

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

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

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

And we are done. An easy generalization shows that, if g is any p.r. function, then

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

is also primitive recursive. ⊠

Given these general facts, we can now spring into action to extend our catalogue of particular p.r. properties and functions. Here are some quick examples, mostly concerning prime numbers and prime factorization.

i. The relation $m < n$ is primitive recursive.

ii. The relation $m|n$ which holds when m is a factor of n is primitive recursive.

iii. Let $prime(n)$ be true just when n is a prime number. Then *prime* is a p.r. property.

iv. List the primes as $\pi_0, \pi_1, \pi_2, \ldots$. Then the function $\pi(n)$ whose value is π_n is p.r. (and in future, we will keep writing 'π_n' rather than '$\pi(n)$', just for symbolic neatness.)

v. Let $exp(n, i)$ be the – possibly zero – exponent of the prime number π_i in the factorization of n (this value is determinate because of the so-called Fundamental Theorem of Arithmetic, which says that numbers have a unique factorization into primes). Then *exp* is a p.r. function.

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

Proof for (i) This is an easy warm-up. For the characteristic function of $m < n$ is simply $sg(Sm \dot{-} n)$, which is a composition of p.r. functions. ⊠

Proof for (ii) We have

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

The quantified relation on the right is a truth-functional combination of p.r. relations, so is p.r. by Fact 2. Hence $m|n$ is a p.r. relation by Fact 3. ⊠

Proof for (iii) The property of being *prime* is p.r. because

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

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.) ⊠

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

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

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 4. ⊠

Proof for (v) No exponent in the prime factorization of n is larger than n itself, so we have

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

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 $exp(n, k) = 0$ 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 (vi) $(prime(m) \wedge m|n)$ holds when m is a prime factor of n. This a p.r. relation, and has a p.r. characteristic function $pf(m, n)$. Now consider the function

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

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

$$len(n) = p(0, n) + p(1, n) + \ldots + p(n - 1, n) + p(n, n)$$

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 $len(n) = l(n, n)$. ⊠

It will, by the way, emerge in Chapter 10 why it is of key importance that such functions and properties connected with prime factorization are primitive recursive.

6.5 The p.r. functions are computable

We now turn back from particular examples to general points about primitive recursion.

A p.r. function f must be specifiable by a chain of definitions by recursion and composition leading back ultimately to initial functions. But (a) the initial functions S, Z, and I_i^k are trivially computable. (b) The composition of two computable functions g and h is computable (just feed the output from whatever routine evaluates g as input into the routine that evaluates h). And (c) – the key point – if g and h are computable, and f is defined by primitive recursion from g and h, then f is computable too. So as we build up longer and longer chains

of definitions for p.r. functions, we always stay within the class of computable functions.

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

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

The first clause gives the value of the function for the argument 0; then 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, and corresponds exactly to a certain simple kind of computer routine. Thus compare the definition with the following schematic program:

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

Here *factorial* is a memory register that we initially prime with the value of $0!$. Then the program enters a loop, and on the k-th iteration (counting from 1) it replaces the value in that register with k times the previous value – we'll assume the computer already knows how to do that. When the program exits the loop after n iterations, the value in the register *factorial* will be $n!$.

Generalizing, for a 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 $f(n)$:

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

Thus the one-place function f will be computable by the use of a 'for' loop, assuming that g and h are already computable. Similarly for many-place functions.

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 as we unpick the chain of recursive definitions

factorial $\Rightarrow$ multiplication $\Rightarrow$ addition $\Rightarrow$ successor

we can write a program for the factorial function containing nested 'for' loops which ultimately call the primitive operations of setting the contents of a register to zero, or of applying the successor operation of incrementing the contents of a register by one. Evidently the point generalizes: *primitive recursive functions are computable by nested 'for' loops.*

And the point holds conversely. We can describe a simple programming language LOOP where the only important programming structure is the 'for' loop, which initiates an iterative procedure that is repeated some given number of times (a number which is fixed before the loop is started). Loops can be nested. And commands can be concatenated so that e.g. a loop for evaluating a function g is followed by a loop for evaluating h – so given input n, executing the concatenated loops yields the output $h(g(n))$. Thus a loop corresponds to a definition by primitive recursion, and concatenation corresponds to composition of functions. Hence it is easy to see that every LOOP program defines a p.r. function.[3]

6.6 Not all computable numerical functions are p.r.

So, any p.r. function is mechanically computable. *But not all computable numerical functions are primitive recursive.*[4] In this section, we first make the claim that there are computable-but-not-p.r. numerical functions look plausible. Then we'll actually cook up an example.[5]

First, then, some plausibility considerations. We've just seen that a primitive recursive function f can be computed by a program involving 'for' loops as its main programming structure. Each loop goes through a specified number of iterations. So, just by examining the program for f, we can derive a function $\tilde{f}$, where $\tilde{f}(n)$ gives the number of steps it takes to compute $f(n)$. Moreover, to put it crudely, $\tilde{f}$ will be definable in terms of repeated additions and multiplications corresponding to the way that 'for' loops are chained together and/or embedded inside each other in the program for f: so $\tilde{f}$ will itself be a p.r. function. In sum, *the length of the computation of a p.r. function is given by a p.r. function.*

However, back in Section 2.3 we allowed procedures to count as computational even when don't have nice upper bounds on the number of steps involved. In particular, 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 2.7, 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.

And 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 worlds, they allow some process to be iterated until a given condition is satisfied – *where no prior limit, and so in particular no p.r. limit, is put on the the number*

[3]For a proper specification of LOOP and a proof that every LOOP program does define a p.r. function see Tourlakis (2002); the idea of LOOP programs goes back to Meyer and Ritchie (1967).

[4]We mean of course functions $f\colon \mathbb{N} \to \mathbb{N}$, whose arguments and values are natural numbers.

[5]Probably no one will regard our cooked-up example as one that might be encountered in ordinary mathematical practice: in fact, it requires a bit of ingenuity to come up with a 'natural' example, though we'll give one later, in the Section 14.2.

of iterations to be executed.

If we count what are presented as unbounded searches as computations, then it looks very plausible that not everything computable will be primitive recursive. However, that is as yet only a plausibility consideration: for all we've so far strictly *proved*, it might still be the case that computations presented as unbounded searches can always somehow be turned into procedures with a p.r. limit on the number of steps. But in fact that's false:

> **Theorem 7** *There are algorithmically computable numerical functions which aren't primitive recursive.*

Proof sketch The set of p.r. functions is effectively enumerable. That is to say, we can mechanically produce a list of 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 'recipe' in which it is defined by primitive 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, we select the strings that obey the rules for being a recipe for a p.r. function (that's a mechanical procedure). That generates a list of recipes which effectively enumerates the p.r. functions.

Now take such an effective enumeration f_0, f_1, f_2, ..., of the p.r functions and construct a corresponding *diagonal* function, defined as $d(n) = f_n(n) + 1$ – cf. Section 2.4, and compare the table below. Down the table we list off the

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

p.r. functions. An individual row then gives the values of the p.r. function f_n for each argument. To compute $d(n)$, we just run our effective enumeration of the p.r. functions until we get to f_n. We evaluate that function for the argument n. We then add one. Each step is an entirely mechanically one. So our diagonal function is algorithmically computable.

By construction, however, the function d can't be primitive recursive. For suppose otherwise. Then the function d 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 $d(d)$ is. By hypothesis, the function d is none other than the function f_d, so $d(d) = f_d(d)$. But by the initial definition of the diagonal function, $d(d) = f_d(d) + 1$. Contradiction.

Hence d is a computable function which is not primitive recursive. ⊠

'But hold on! *Why* is the diagonal function not a p.r. function?' Well, as we just noted, if f is a p.r. function, then – as we compute $f(n)$ for increasing values of n – the lengths of the successive computations will be given by the successive values of some function $\tilde{f}(n)$, where $\tilde{f}$ is also primitive recursive. Now contrast evaluating $d(n)$ for increasing values of n. For each new argument, we 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 lengths of these successive computations of different functions f_n. In particular, we have no reason to expect there will be a single p.r. function which gives the length of all those different computations. And our diagonal argument in effect shows that there isn't one.

6.7 PBA and the idea of p.r. adequacy

The value of a p.r. function for any given argument(s) is computable – in p.r. bounded time – in accordance with a step-by-step algorithm. But, as we've said before, the whole aim of formalization is to systematize and regiment what we can already do. And if we can informally calculate the value of a p.r. function for a given input in an entirely mechanical way – ultimately by just repeating lots of school-arithmetic operations – then we will surely want to aim for a formal arithmetic which is able to track these informal calculations. So let's say that a theory T is *p.r. adequate* if it indeed has the resources to express any p.r. function and prove, case-by-case, the correct results about the function's values for specific arguments (we will tidy up that definition in the next chapter).[6]

Here's one way of constructing a p.r. adequate theory. Start from BA, our theory of Baby Arithmetic, again (see Section 5.1). This, recall, is a quantifier free theory which has schemas which reflect the p.r. definitions of addition and multiplication. As we showed, we can use instances of these schemas to prove any true equation using successor, addition and multiplication. Hence BA is adequate for those three functions in the sense that it can evaluate them correctly case-by-case for specific arguments. So far, so good. But suppose we start expanding BA by adding new vocabulary and new schemas. As a first step, we can add the symbol '↑', intended to express the exponential function, and then say that all numeral instances of the following are axioms too:

Schema 7 $\quad \zeta \uparrow 0 = 1$

Schema 8 $\quad \zeta \uparrow \mathsf{S}\xi = (\zeta \uparrow \xi) \times \zeta$

[6] Compare and contrast the informal idea of being 'sufficiently strong' that we met in Section 4.1.

Instances of those schemas enable us to prove the correct result for the value of the exponential function for any arguments. So that makes four functions which can be captured in our expanded BA. For tidiness, let's resymbolize these using the function symbols '$\mathsf{f_0}$', '$\mathsf{f_1}$', '$\mathsf{f_2}$', '$\mathsf{f_3}$'. And now let's keep going: we will add a symbol '$\mathsf{f_n}$' for each n, with the plan that '$\mathsf{f_n}$' should express the n-th p.r. function f_n in a 'good' effective enumeration of the recipes for p.r. functions (where an enumeration is 'good' if the p.r. definition of f_n only involves functions earlier in the enumeration). Then for each '$\mathsf{f_n}$', we write down schemas involving that function expression which reflect f_n's definition in terms of earlier functions. Call the resulting theory PBA.[7]

PBA is still a properly axiomatized theory, because it will be effectively decidable whether any given wff is an instance of one of axiom schemas. Plainly, its language is much richer than BA's, since it has a separate function expression for each primitive recursive function: but for all that, its language remains impoverished in other ways – for it still can't express any general claims. Because it is quantifier-free, we can show that PBA is a negation-complete theory like BA (in fact we just generalize the argument we used to show BA can either prove or disprove every sentence in its limited language). And by construction, PBA is p.r. adequate.

Which all goes to show that we can readily construct a p.r. adequate arithmetic by the brute-force method of pumping up the vocabulary of arithmetic and throwing in axioms for every p.r. function. But do we *need* to do this?

It turns out that we don't. *We only need the language of basic arithmetic in order to frame a p.r. adequate theory.* To put it very roughly, the ground we lose by going back to a language with successor, addition, and multiplication as the only built-in functions, we can make up again by having quantification available. In essence, that was first proved by Gödel in his epoch-making (1931), where he showed in effect that PA is p.r. adequate. Raphael Robinson later (1952) noted that all we need to extract from the induction principle to prove PA's adequacy is the claim that every number is either zero or a successor number. So we can drop all the instances of the Induction Schema and replace them by Q's Axiom 3 and the resulting induction-free arithmetic Q is still p.r. adequate.

We will outline a proof of Q's – and hence PA's – p.r. adequacy in Chapter 9: it is one of the crucial stages in a Gödel-style proof of the First Incompleteness Theorem. But before doing that, we need to pause to say a bit more about functions and relations, and to give an official definition of what it is for a theory to case-by-case capture a function. Then we give an official definition of two grades of p.r. adequacy. That's business for the following chapter.

[7] You can think of that as 'Pumped-up Baby Arithmetic'. I'd call it 'Primitive Recursive Arithmetic' were that label not already standardly used for a rather stronger theory which includes an induction schema as well.

7 More on Functions and P.R. Adequacy

This chapter is rather unexciting housekeeping. But it is brief, and we are going to need the very straightforward notions defined here, so bear with me!

7.1 Extensionality

Throughout this chapter, we are going to be talking about *total* numerical functions $f\colon \mathbb{N} \to \mathbb{N}$: every such function maps each natural number argument n to some unique numerical value $f(n)$.

Two general points:

1. By definition, the *extension* of a numerical function f is the set of ordered pairs of the form $\langle n, f(n)\rangle$.
2. We treat f and g as the *same* function just so long as they map the same arguments to the same values, i.e. just so long as, for each n, $f(n) = g(n)$.

Putting those points together, f and g are the same function if they have the same extension. 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 $2n+1$ is the same function as $(n+1)^2 - n^2$; but the two different modes of presentation indicate different routines for evaluating the function for a given argument.

Now, a p.r. function is one that can be specified by a certain sort of chain of definitions; and one way of presenting such a function is by giving this definitional chain (and thereby making it transparent that the function is indeed p.r.). But the same function can be presented in other ways; and some modes of presentation can completely disguise the fact that the function is recursive. For a dramatic example, consider the function

$fermat(n) = n$ if there are integer solutions to $x^{n+3} + y^{n+3} = z^{n+3}$;
$fermat(n) = 0$ otherwise.

This definition certainly doesn't reveal 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 the very same function as the trivially p.r. function $Z(n) = 0$.

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

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

There is no way (algorithmic or otherwise) of settling what Caesar ate on his third birthday! But despite that, the function $j(n)$ 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 $j(n)$ must be a p.r. function, though we can't determine *which* function it is from this mode of presentation.

Similar points can be made about numerical properties.[1] So, talk about numerical properties is also to be construed extensionally throughout. In other words, we are to treat F and G as being the *same* numerical property just in case Fn holds if and only if Gn, i.e. just in case the properties have the same extension. Of course, just as the same function can be presented in very different ways, so one and the same property may be presented in very different ways, by predicates with different senses. For example, as far as we know, the properties of being *even and greater than two* and the property of being *even and the sum of two primes* are the same property – i.e. the italicized phrases pick out the same extension, though they have different senses.

7.2 Expressing and capturing functions

Suppose that '$f(x)$' specifies some one-place total numerical function. Then consider the expression '$f(x) = y$'; this is a two-place relational expression, and we'll say that it picks out f's *corresponding relation*. So if f is a function and R_f is its corresponding relation, $f(m) = n$ if and only if m has relation R_f to n. Functions and their corresponding relations match up pairs of things in just the same way: hence f and R_f have exactly the same extension, namely the set of ordered pairs $\langle m, f(m)\rangle$. The idea generalizes in the obvious way to many-place functions. And just as the characteristic function trick (Section 6.3) allows us to take ideas defined for functions and apply them to properties and relations, *this* simple tie between functions to their corresponding relations allows us to carry over ideas defined for relations and apply them to functions.

For a start, consider how we can use this tie to extend the idea of *expressing* a relation to cover functions. Here again is the now familiar definition:

> A two-place numerical relation R is expressed by the open wff $\varphi(\mathsf{x}, \mathsf{y})$ in an (interpreted) arithmetical language L just if, for any m, n,
> if m has the relation R to n, then $\varphi(\overline{\mathsf{m}}, \overline{\mathsf{n}})$ is true,
> if m does not have the relation R to n, then $\neg\varphi(\overline{\mathsf{m}}, \overline{\mathsf{n}})$ is true.

[1] If you accept the thesis of Frege (1891), then these are the *same* points. For Frege urges us to treat properties as just a special kind of function – so a numerical property, in particular, is a function which 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.

We can now naturally say that a one-place function f is expressed by the open wff $\varphi(\mathsf{x}, \mathsf{y})$ so long as that wff equally expresses the relation R_f which corresponds to f. Which comes to the following:

> A one-place numerical relation f is expressed by the open wff $\varphi(\mathsf{x}, \mathsf{y})$ in an (interpreted) arithmetical language L just if, for any m, n,
>
> if $f(m) = n$, then $\varphi(\overline{\mathsf{m}}, \overline{\mathsf{n}})$ is true,
>
> if $f(m) \neq n$, then $\neg\varphi(\overline{\mathsf{m}}, \overline{\mathsf{n}})$ is true.

The generalization to many-place functions is immediate.

Similarly, we can extend the idea of *case-by-case capturing* from relations to functions. Here is the definition for a two-place relation:

> A two-place numerical relation R is case-by-case captured by the open wff $\varphi(\mathsf{x}, \mathsf{y})$ of theory T just if, for any m, n,
>
> if m has the relation R to n, then $T \vdash \varphi(\overline{\mathsf{m}}, \overline{\mathsf{n}})$
>
> if m does not have the relation R to n, then $T \vdash \neg\varphi(\overline{\mathsf{m}}, \overline{\mathsf{n}})$.

And we can naturally say that a one-place function f is case-by-case captured by the open wff $\varphi(\mathsf{x}, \mathsf{y})$ so long as that wff captures the corresponding relation R_f. Which comes to the following:

> A one-place function relation f is case-by-case captured by the open wff $\varphi(\mathsf{x}, \mathsf{y})$ of theory T just if, for any m, n,
>
> if $f(m) = n$, then $T \vdash \varphi(\overline{\mathsf{m}}, \overline{\mathsf{n}})$
>
> if $f(m) \neq n$, then $T \vdash \neg\varphi(\overline{\mathsf{m}}, \overline{\mathsf{n}})$.

Again, the generalization to many-place functions is immediate.

7.3 'Capturing as a function'

So far so good. However, although our definition in the last section of what it is for a theory to capture a function will serve perfectly well, it is in fact convenient (and standard) to work with a slightly stronger notion of capturing. This section explains the stronger notion.

Our previous definition might be said to be weak in the following sense. It tells us that T captures a function f if there is some φ which captures the relation that holds between m and n when $f(m) = n$. But it doesn't require that φ – so to speak – captures the function *as a function*, i.e. it doesn't require that T can prove that the capturing wff φ relates a given m to exactly one value n. We will now impose this extra requirement, and say:

> The one-place function f is *captured as a function* by the wff $\varphi(\mathsf{x}, \mathsf{y})$ in T just if
>
> (i) for every m, $T \vdash \exists!\mathsf{y}\varphi(\overline{\mathsf{m}}, \mathsf{y})$
>
> and for any m, n:

(ii) if $f(m) = n$ then $T \vdash \varphi(\overline{\mathsf{m}}, \overline{\mathsf{n}})$,
(iii) if $f(m) \neq n$, then $T \vdash \neg\varphi(\overline{\mathsf{m}}, \overline{\mathsf{n}})$.

Here '$\exists!\mathsf{u}$' is the standard uniqueness quantifier, to be read 'there is exactly one u such that ...'.[2]. So clause (i), as we want, insists that the putative capturing relation can be proved to relate each numerical argument to some unique value: in a phrase, the relation is (provably) functional.

Now, suppose $f(m) \neq n$ because $f(m) = k$, where $n \neq k$. Suppose further that (i) and (ii) hold, so $\varphi(\mathsf{x}, \mathsf{y})$ is provably functional, and also $T \vdash \varphi(\overline{\mathsf{m}}, \overline{\mathsf{k}})$. Then by simple logic, (i) and (ii) imply that $T \vdash (\overline{\mathsf{n}} \neq \overline{\mathsf{k}} \rightarrow \neg\varphi(\overline{\mathsf{m}}, \overline{\mathsf{n}}))$. But as we saw in Section 5.1, in any theory T containing Baby Arithmetic, if $n \neq k$, then $T \vdash \overline{\mathsf{n}} \neq \overline{\mathsf{k}}$. Hence if $f(m) \neq n$ then $T \vdash \neg\varphi(\overline{\mathsf{m}}, \overline{\mathsf{n}})$. So clauses (i) and (ii) imply (iii), in even the most modest theory of arithmetic. So to confirm that φ captures f as a function, we only need to check that conditions (i) and (ii) hold.

Assuming T is consistent and contains Baby Arithmetic, our new definition is now easily seen to be equivalent to the following:

> The one-place function f is captured as a function by the wff $\varphi(\mathsf{x}, \mathsf{y})$ in T just if for all m, n,
> $T \vdash \forall\mathsf{y}(\varphi(\overline{\mathsf{m}}, \mathsf{y}) \equiv \mathsf{y} = \overline{\mathsf{n}})$ if and only if $f(m) = n$.

Likewise, we'll say

> The two-place function f is captured as a function by the wff $\varphi(\mathsf{x}, \mathsf{y}, \mathsf{z})$ in T just if for all m, n, o,
> $T \vdash \forall\mathsf{z}(\varphi(\overline{\mathsf{m}}, \overline{\mathsf{n}}, \mathsf{z}) \equiv \mathsf{z} = \overline{\mathsf{o}})$ if and only if $f(m, n) = o$.

Similarly for the case of many-place functions.

Trivially, if φ captures f as a function in T, then φ captures f in the weaker sense of the previous section. The strict converse doesn't obtain. For example, consider a quantifier-free theory like BA or PBA: that can capture some functions but obviously can't capture them as functions (since the latter involves proving a quantified wff). However, suppose T is at least as strong as Q: then if φ captures f in T, then there will always be a closely related wff φ' which *does* capture f as a function in T. For we just have to set

$$\varphi'(\mathsf{x}, \mathsf{y}) =_{\text{def}} \{\varphi(\mathsf{x}, \mathsf{y}) \wedge \forall\mathsf{z}(\mathsf{z} < \mathsf{y} \rightarrow \neg\varphi(\mathsf{x}, \mathsf{z}))\}$$

Then, for given m, $\varphi'(\overline{\mathsf{m}}, \overline{\mathsf{n}})$ holds as we want for a *unique* n, i.e. the smallest n such that $\varphi(\overline{\mathsf{m}}, \overline{\mathsf{n}})$ is true.

Finally, we should just confirm that our various definitions hang together in the following way: suppose T can prove that $0 \neq \overline{1}$, then *a property is capturable by T if and only if its characteristic function is capturable as a function*:

Proof sketch (i) Suppose P is case-by-case captured in T by the predicate $\varphi(\mathsf{x})$, and consider the wff

[2] '$\exists!\mathsf{u}$' can be defined by taking '$\exists!\mathsf{u}\varphi(\mathsf{u})$' as short for '$\exists\mathsf{u}(\varphi(\mathsf{u}) \wedge \forall\mathsf{v}(\varphi(\mathsf{v}) \rightarrow \mathsf{v} = \mathsf{u}))$'

$$((\varphi(\mathsf{x}) \equiv \mathsf{y} = 0) \land (\neg\varphi(\mathsf{x}) \equiv \mathsf{y} = \bar{1}))$$

It is easily seen that this relational wff captures c_P, the characteristic function of P, and captures it as a function. Conversely, (ii) suppose the wff $\varphi(\mathsf{x}, \mathsf{y})$ captures the characteristic function c_P; then the wff $\varphi(\mathsf{x}, 0)$ captures the corresponding property P. ⊠

So instead of laying down separate conditions for properties/relations and functions being capturable, we could have initially just given conditions for the case of functions, and then let properties and relations look after themselves by saying that they are capturable if their characteristic functions are.

7.4 Two grades of p.r. adequacy

In Section 6.7, we informally said that a theory is p.r. adequate if it can express any p.r. function and prove, case-by-case, the correct results about the function's values for specific arguments. Now we can sharpen up that informal idea in two ways:

> A theory T is *weakly p.r. adequate* if, for every p.r. function f, there is a corresponding open wff φ in T that captures it.
>
> A theory T is *strongly p.r. adequate* if for every p.r. function f, there is a corresponding open wff φ in T that captures it as a function.

And here are four claims involving these notions:

1. PBA is weakly p.r. adequate.
2. PBA is not strongly p.r. adequate
3. Q *is* strongly p.r. adequate.
4. PA is strongly p.r. adequate.

The first claim is trivially true, since we explicitly defined PBA in Section 6.7 in order to make the first of these claims true. (Suppose f_k is the k-th p.r. function, and for simplicity suppose it is one-place. Form the corresponding wff $\mathsf{f_k}(\mathsf{x}) = \mathsf{y}$. Then this case-by-case captures f_k in PBA.) The second claim is also trivially true because being strongly p.r. adequate involves being able to prove a quantified claim, and PBA can't do that. The third claim is far from being trivial: proving that it is true is work for Chapter 9. And given the third claim is true, the fourth immediately follows: PA can prove everything Q can prove, so if Q can prove enough to be p.r. adequate, so can PA.

8 Gödel's Proof: The Headlines

In the last three chapters, we have put in place some necessary background, by setting out some formal theories of arithmetic, defining the notion of a primitive recursive function, and explaining what it takes for a theory to be strongly p.r.adequate. We can now turn to proving Gödel's First Incompleteness Theorem. The business of this very short chapter is to give the headline news about the overall shape of (our reconstruction of) Gödel's proof: the details are for the following three chapters.

We start, though, with ...

8.1 A very little background

Kurt Gödel published his epoch-making paper 'On formally undecidable propositions of *Principia Mathematica* and related systems I' in 1931. He was not yet 25 when it appeared.

We'll discuss the general context of Gödel's paper – and the group of problems in the foundations of mathematics that give his paper its significance – in more detail later. But we ought perhaps to pause before then at least to explain the paper's title, and here is as good a place as any.

As we noted in Section 5.7, Frege set out in *The Basic Laws of Arithmetic* to reconstruct arithmetic and classical analysis (the mathematics of the real numbers) on a secure logical footing. His attempt was sabotaged by its reliance on his fifth 'Basic Law' which postulates the existence of so many sets as to lead to contradiction. And the fatal contradiction that Russell exposed in Frege's system was not the only one to beset various early treatments of the theory of sets (Georg Cantor and Cesare Burali-Forti had already found other paradoxes).

Various responses to these set-theoretic paradoxes were proposed at the beginning of the twentieth century. One suggestion is, in effect, to keep much of Frege's logic but to avoid making the further move that gets him into trouble.

Frege's logical system involves a *type hierarchy*. His system very carefully distinguishes 'objects' (things, in a broad sense) from properties from properties-of-properties from properties-of-properties-of-properties, etc, and every item belongs to a determinate level of the hierarchy. Then the claim is – plausibly enough – that it only makes sense to attribute properties which belong at level l to items at level $l - 1$. For example, the property of *being wise* is a level 1 property, while Socrates is an item at level 0; and it makes sense to attribute that property to Socrates, i.e. to claim that Socrates is wise. Likewise, the property of *applying to people* is a level 2 property, and it makes sense to attribute that property to

the level 1 property of being wise, i.e. to claim that the property of being wise applies to people. But you get nonsense if, for example, you try to apply that level 2 property to Socrates and claim that Socrates applies to people.

Note that this strict stratification of items into types blocks the derivation of the property analogue of Russell's paradox about sets. That paradox, recall, concerned the set of all sets that are not members of themselves. So now consider the putative property of *being a property that doesn't apply to itself*. Does this property apply to itself? It might seem that the answer is that it does if it doesn't, and it doesn't if it does – contradiction! But on Frege's theory of types, there is no real contradiction here. For on such a theory, every genuine property belongs to some particular level of the hierarchy; and a level l property can only sensibly be attributed to items at level $l - 1$, the next level down. So no property can be attributed to itself. (True, every level l property in fact has the property of *being a property at level l that doesn't apply to itself* – but *that* is a respectable level $l + 1$ property, which only applies to level l properties and cannot paradoxically apply to itself.)

We can avoid set-theoretic paradox if we stratify the universe of sets in the way that Frege stratifies the universe of properties. So suppose we now distinguish sets from sets-of-sets from sets-of-sets-of-sets, and so forth; and on one version of this approach we then insist that sets at level l can only have as members items at level $l-1$.[1] Frege himself doesn't take this line, for his disastrous Basic Law V in effect flattens the hierarchy and puts all sets on the same level. But Bertrand Russell and Alfred North Whitehead adopt this hierarchical view of sets in their monumental *Principia Mathematica* (1910–13). They retain and develop Frege's stratification of properties and then link this to the stratification of sets in a very direct way, by treating talk about sets as just lightly disguised talk about their corresponding defining properties. Like Frege in his *Basic Laws*, Russell and Whitehead set out to re-construct all of arithmetic and classical analysis on a secure, paradox-free, footing. And as far as we know, their formal system, unlike Frege's, is indeed consistent.

But now, enter Gödel. He shows that, despite its great power, Russell and Whitehead's system in *Principia* still can't capture all arithmetic truths. As Gödel puts it in the opening words of his paper:

> The development of mathematics toward greater precision has led, as is well known, to the formalization of large tracts of it, so that one can prove any theorem using nothing but a few mechanical rules. The most comprehensive formal systems that have been set up hitherto yet are the system of *Principia Mathematica* on the one hand and the Zermelo-Fraenkel axiom system for set theory ... on the other. These two systems are so comprehensive that in them all methods of

[1] An alternative approach – the now dominant Zermelo-Fraenkel set theory – is more liberal: it allows sets at level l to contain members from *any* lower level. But we still get a paradox-blocking hierarchy.

> proof today used in mathematics are formalized, that is, reduced to a few axioms and rules of inference. One might therefore conjecture that these axioms and rules of inference are sufficient to decide *any* mathematical question that can at all be formally expressed in these systems. It will be shown below that this is not the case, that on the contrary there are in the two systems mentioned relatively simple problems in the theory of integers which cannot be decided on the basis of the axioms. This situation is not in any way due to the special nature of the systems that have been set up, but holds for a very wide class of formal systems; .... (Gödel, 1931, p. 145)

Now, Russell and Whitehead's system, to repeat, is a theory of types: it allows us to talk of, and quantify over, properties, properties-of-properties, properties-of-properties-of-properties, and so on up the hierarchy. Hence the language of *Principia* is immensely richer than the language L_A of first-order PA (where we can only quantify over individuals, and which has no way of representing properties-of-properties-of-properties or higher types). It wouldn't be a great surprise, then, to learn that Russell and Whitehead's axioms don't settle every question that can be posed in their very rich formal language. What *is* a surprise is that there are 'relatively simple' propositions which are 'formally undecidable' in *Principia* – by which Gödel means just that there are arithmetic wffs φ such that we can't prove either φ or $\neg\varphi$ from the axioms.[2] And similarly, there are arithmetical propositions which are 'formally undecidable' in ZF set theory.

As Gödel himself notes, his incompleteness proof only needs to invoke some very modest features of *Principia* and of ZF, and the features are equally shared by PA. So it will do no harm, for our purposes, to indulge in a mild historical fiction and henceforth pretend that he was really talking about PA all along. In what follows, there are also some other deviations from the details of Gödel's original proof; but the basic lines of argument in the next three chapters are all in his great paper. Not surprisingly, other ways of establishing his results (and generalizations and extensions of them) have been discovered since 1931, and we will be looking at some of these later. But there remains a good deal to be said for first introducing the incompleteness theorems by something close to Gödel's own arguments.

8.2 Gödel's proof outlined

There are five main stages in (our reconstruction of) Gödel's proof of his First Incompleteness Theorem:

[2] Don't confuse this sense of undecidability with our earlier talk of algorithmic undecidability. To help see the distinction, consider the mini-theory T_1 we met in Section 2.5. In that theory, 'q' is formally undecidable, i.e. we have neither $T_1 \vdash \mathsf{q}$ nor $T_1 \vdash \neg\mathsf{q}$. But T_1 is an algorithmically decidable theory: we can use truth-tables to determine, for any φ, whether $T_1 \vdash \varphi$

1. He introduces a system of *Gödel numbering*, which systematically associates expressions of PA with numerical codes. Any sensibly systematic coding scheme will do: Gödel's choice is designed, however, to make it relatively easy to prove arithmetical results about the codings.

2. With a coding scheme in place, we can reflect properties and relations of strings of symbols of PA by properties and relations of their Gödel numbers. For example, we can define the numerical properties *term* and *wff* which hold of a number when it is the code number for a symbol sequence which is, respectively, a term or a wff of PA. And we can, crucially, define the numerical relation $prfseq(m, n)$ which holds when m codes for an array of wffs that is a PA proof, and n codes the closed wff that is thereby proved. This project of coding up various syntactic relationships is often referred to as *the arithmetization of syntax*. And what Gödel shows next is that – given a sane system of Gödel numbering – these and a large family of related arithmetical properties and relations are all what he calls '*rekursiv*', i.e. are (in modern parlance) primitive recursive.

3. He proves, in effect, that PA is p.r. adequate. In other words, Gödel shows that every p.r. function $f(\vec{x})$ can be captured by a relational wff $\varphi(\vec{x}, y)$ such that, if $f(\vec{m}) = n$ then, $\mathsf{PA} \vdash \forall \mathsf{y}(\varphi(\vec{\mathsf{m}}, \mathsf{y}) \equiv \mathsf{y} = \bar{\mathsf{n}})$.

4. Next – the exciting bit! – Gödel uses the fact that relations like *prfseq* are p.r. and hence capturable in PA to construct a wff G which, given the coding scheme, 'says' there is no number which is the Gödel number of a proof in PA of the wff which results from a certain substitution construction – where the wff which results from that construction is none other than G itself. So in effect G 'says' of itself 'I am unprovable in PA'. Gödel then shows that G is indeed unprovable, assuming PA is consistent. So we've found an arithmetical wff which is true but unprovable in PA. (And given a slightly stronger assumption than PA's consistency, $\neg$G must also be unprovable in PA.) Moreover, this unprovable wff is, in an important sense, relatively simple.

5. Finally, Gödel notes that the true-but-unprovable sentence G for PA is generated by a method that can be applied and re-applied to other arithmetics that satisfy some modest conditions. So adding G as a new axiom to PA just gives us a revised theory for which we can generate another true-but-unprovable wff G′. So PA is therefore not only incomplete but essentially incompletable.

We'll take the stages of this proof as follows. In the next chapter, we'll prove that Q (and hence PA) is p.r. adequate, which gets stage (3) done. In Chapter 10, we consider the arithmetization of syntax, i.e. stages (1) and (2). Stages (4) and (5) are dealt with in Chapter 11.

9 Q is P.R. Adequate

Back in Section 5.3 we showed that we can case-by-case capture the primitive recursive *less than* relation in Q. But that was just one example. In this chapter, we are going to outline a proof that *any* p.r. function – and hence (via its characteristic function) *any* p.r. property and relation – can be captured in Q (and hence in PA). This result is interesting for its own sake; but we are looking at it here because it is a main load-bearing component in our Gödel-style proof of the First Incompleteness Theorem.

Note, however, that the key proof idea (we can call it 'the β-function trick') is not used again in this book, so it is not necessary to master the argument of this chapter in order to grasp what follows. One option, therefore, is to take a quick look at the first two sections in order to get the basic picture, and leave it at that. Another option is simply to take Q's p.r. adequacy on trust for the moment, and skip this chapter entirely (at least until you've read Chapter 11 and really understand why the result matters).

By the way, the β-function trick is due to Gödel; so the following proof is in essence his. But Gödel, of course, didn't know about Q in his (1931); he gave a proof – in effect – that PA is p.r. adequate: it was Raphael Robinson who first remarked in his (1952) that the proof only depends on that part of PA which follows from the axioms of Q.

9.1 The proof strategy

Recall that a p.r. function is one that can be specified by a chain of definitions by composition and/or primitive recursion, building up from functions in the 'starter pack' of initial functions. Suppose, then, that the following three propositions are all true (and take 'captures' throughout this chapter to mean 'captures-as-a-function'):

1. Q captures the initial functions.
2. If Q captures the functions g and h, then it also captures a function f defined by composition from g and h.
3. If Q captures the functions g and h, then it also captures a function f defined by primitive recursion from g and h.

Now take any p.r. function f. As we follow through the chain of definitions which specifies f, we start with initial functions which are capturable. And – by (2) and (3) – each successive definitional move takes us from capturable functions to capturable functions. So f must be capturable.

Hence: *to show that every p.r. function can be captured in* Q *it is enough to prove (1) to (3).*

Proof sketch for (1) Evidently, the successor function is captured by the wff $\mathsf{Sx = y}$. (It is trivial that, for any m, $\mathsf{Q} \vdash \exists!\mathsf{y}\,\mathsf{S}\overline{\mathsf{m}} = \mathsf{y}$, and it is equally trivial that for any m, n, if $Sm = n$, then $\mathsf{Q} \vdash \mathsf{S}\overline{\mathsf{m}} = \overline{\mathsf{n}}$.) Similarly, the zero function $Z(x) = 0$ is captured by the relational wff $(\mathsf{x = x} \wedge \mathsf{y = 0})$. Thirdly, the identity function $I(x, y) = y$, for example, is captured by $(\mathsf{x = x} \wedge \mathsf{y = z})$; likewise for other identity functions. ⊠

Proof sketch for (2) Suppose g and h are one-place functions, captured by the wffs $\mathsf{G(x, y)}$ and $\mathsf{H(x, y)}$ respectively. Then, we claim, the function $f(x) = g(h(x))$ is captured by the relational wff $\exists\mathsf{z}(\mathsf{H(x, z)} \wedge \mathsf{G(z, y)})$.

It is straightforward to check that that wff is provably functional for numerals. So it remains to show that if $f(m) = n$, then $\mathsf{Q} \vdash \exists\mathsf{z}(\mathsf{H}(\overline{\mathsf{m}}, \mathsf{z}) \wedge \mathsf{G}(\mathsf{z}, \overline{\mathsf{n}}))$. But if $f(m) = g(h(m)) = n$ there must be some number o such that $h(m) = o$ and $g(o) = n$, so $f(m) = g(h(m)) = n$. Then, by assumption, we have $\mathsf{Q} \vdash \mathsf{H}(\overline{\mathsf{m}}, \overline{\mathsf{o}})$, and $\mathsf{Q} \vdash \mathsf{G}(\overline{\mathsf{o}}, \overline{\mathsf{n}})$. And the desired result follows by simple logic.

The argument obviously generalizes to the case where g and h are multi-place functions. ⊠

Starting the proof sketch for (3) Now for the fun part. Consider the primitive recursive definition of the factorial function again:

$$0! = 1$$
$$(Sx)! = Sx \times x!$$

The multiplication and successor functions involved on the right of the equations here are of course capturable: but how can we capture our defined function by a relational wff $\mathsf{F(x, y)}$ in Q?

Think about the p.r. definition for the factorial in the following way. It tells us how to construct a sequence of numbers $0!, 1!, 2!, \ldots, x!$, where we move from the u-th member of the sequence (counting from zero) to the next by multiplying by Su. So, putting $x! = y$, the p.r. definition says

A. There is a sequence of numbers $k_0, k_1, \ldots, k_x$ such that: $k_0 = 1$, and if $u < x$ then $k_{Su} = Su \times k_u$, and $k_x = y$.

So the question of how to reflect the p.r. definition of the factorial inside Q comes to this: how can we express facts about finite sequences of numbers using the limited resources of Q?

9.2 The idea of a β-function

Let's pause the proof sketch for (3), and think about the *kind* of trick we could use here (understand this and you understand the difficult bit of adequacy proof).

Suppose $\pi_0, \pi_1, \pi_2, \pi_3, \ldots$ is the series of prime numbers $2, 3, 5, 7, \ldots$. Now consider the number

$$b = \pi_0^{k_0} \cdot \pi_1^{k_1} \cdot \pi_2^{k_2} \cdots \pi_n^{k_n}$$

This single number b can be thought of as encoding the whole sequence k_0, k_1, k_2, $\ldots$, k_n. For we can extract the coded sequence again by using the (primitive recursive) decoding function $exp(b, i)$ which we met in Section 6.4, which returns the exponent of the prime number π_i in the factorization of b. By the construction of b, then, $exp(b, i) = k_i$ for $i \leq n$.

Now let's generalize. We'll say

> A two-place *β-function* is a function of the form $\beta(b, i)$ such that, for *any* finite sequence of natural numbers $k_0, k_1, k_2, \ldots, k_n$ there is a code b such that for every $i \leq n$, $\beta(b, i) = k_i$.

So the idea is that – for any finite sequence of numbers you choose – you can select a corresponding code number b to be the first argument for β, and then the function will decode it and spit out the members of the required sequence in order as its second argument is increased.[1] We've just seen that there is nothing in the least magical or mysterious about the idea of such a function: *exp* is a simple example.

It turns out, however, to be convenient to generalize our notion of a β-function just a little. So we'll also consider *three*-place β-functions, which take *two* code numbers c and d, as follows:

> A three-place β-function is a function of the form $\beta(c, d, i)$ such that, for *any* finite sequence of natural numbers $k_0, k_1, k_2, \ldots, k_n$ there is a pair of code numbers c, d such that for every $i \leq n$, $\beta(c, d, i) = k_i$.

Continuing the proof sketch for (3) Suppose we have a β-function to hand – and we'll use a three-place function, simply to make a smooth connection with our key example of β-function in the next section. Then we can reformulate

A. There is a sequence of numbers $k_0, k_1, \ldots, k_x$ such that: $k_0 = 1$, and if $u < x$ then $k_{Su} = Su \times k_u$, and $k_x = y$,

as follows:

B. There is some pair c, d such that: $\beta(c, d, 0) = 1$, and if $u < x$ then $\beta(c, d, Su) = Su \times \beta(c, d, u)$, and $\beta(c, d, x) = y$.

Now let's make another assumption: suppose that our three-place β-function can be captured by a four-place open wff of Q which we will abbreviate B. Then we can translate (B) into Q as follows:[2]

[1] For the record, referring to such a function as a 'beta-function' is absolutely standard. The notation was in fact introduced by Gödel himself in his Princeton Lectures (1934, p. 365).

[2] For readability, we temporarily recruit 'c' and 'd' as variables. '∃!' is the familiar uniqueness quantifier again.

C. $\exists c \exists d\{B(c, d, 0, \overline{1}) \wedge$
$\forall u[u < x \rightarrow \exists v \exists w\{(B(c, d, u, v) \wedge B(c, d, Su, w)) \wedge w = Su \times v\}] \wedge$
$B(c, d, x, y)\}$

Let's abbreviate all that by '$F(x, y)$': then this relational wff evidently *expresses* the factorial function. But more than that, we can show that this wff *captures* the factorial function (by proving that the relation is functional, and proving that if $m! = n$, then $Q \vdash F(\overline{m}, \overline{n})$). That takes a bit of effort, but the checking is basically pretty routine, and we won't give the boring details here.

In sum, then, we noted that the p.r. definition of $n!$ tells us that there is a sequence of $(n + 1)$ numbers satisfying a certain condition. Then we used the β-function trick to re-write this as the claim that there is a code number for the sequence – or rather, two code numbers – satisfying a related condition. We can then render this re-written version into Q. But what goes for our p.r. definition of the factorial will fairly obviously go likewise for the p.r. definitions of other functions. Which gives us the desired result (3). ⊠

9.3 Filling in a few details

We'll just quickly round out our proof-sketch for (3) in two ways. First (A), we indicate how to warrant our assumption that there is a β-function that can be captured in Q. Then for the record (B), we'll spell out how to generalize the argument that showed that the factorial can be captured in Q.

(A) Our initial example of a β-function – namely *exp* – is defined in terms of exponentiation, which isn't available (yet) in Q. So we need to find another β-function that *can* be defined using just addition and multiplication, so that we can readily capture it in Q.

Gödel solves our problem like this. Put

$$\beta(c, d, i) =_{\text{def}} \text{the remainder left when } c \text{ is divided by } (d(i + 1) + 1).$$

Then it can be shown – by appeal to what is known as the Chinese Remainder Theorem – that, given any sequence $k_0, k_1, \ldots, k_n$, we can find a suitable pair of numbers c, d such that for $i \leq n$, $\beta(c, d, i) = k_i$.[3] However, the concept of a remainder on division can be elementarily defined in terms of multiplication and addition. We can therefore render this β-function into Q by the following open wff:

$$B(c, d, i, y) =_{\text{def}} \exists u[c = \{S(d \times Si) \times u\} + y \wedge y < S(d \times Si)]$$

[3]This claim should look intrinsically quite plausible. As we divide c by $(d(i + 1) + 1)$ for different values of i for $i \leq n$, we'll get a sequence of $n + 1$ remainders. We just need show that, if we make c big enough, we can find a d that makes the sequence of remainders match a given sequence. This is a fairly straightforward arithmetical result: enthusiasts can consult the classic text by Mendelson (1997, pp. 186–189), where the other claims in this section also get iron-clad proofs, though the details are inevitably tiresome.

And it can again be routinely checked that this wff does capture our three-place β-function in Q.

(B) As we claimed, our neat β-function trick – or rather, *Gödel's* neat β-function trick – generalizes. For the record, suppose the function f is defined from the functions g and h by the standard p.r. recursion equations:

$$f(\vec{x}, 0) = g(\vec{x})$$
$$f(\vec{x}, Sy) = h(\vec{x}, y, f(\vec{x}, y))$$

This definition amounts to fixing the value of $f(\vec{x}, y) = z$ thus:

A* There is a sequence of numbers $k_0, k_1, \ldots, k_y$ such that: $k_0 = g(\vec{x})$, and if $u < y$ then $k_{u+1} = h(\vec{x}, u, k_u)$, and $k_y = z$.

So using a three-place β-function again, that comes to

B* There is some c, d, such that: $\beta(c, d, 0) = g(\vec{x})$, and if $u < y$ then $\beta(c, d, Su) = h(\vec{x}, u, \beta(c, d, u))$, and $\beta(c, d, y) = z$.

Suppose we can already capture the n-place function g by a $(n + 1)$-place relational expression G, and the $(n + 2)$-place function h by the $(n + 3)$-place expression H. Then – using '$\vec{\mathsf{x}}$' to indicate a suitable sequence of n variables – (B*) can be rendered into Q by

C* $\exists \mathsf{c} \exists \mathsf{d}\{\exists \mathsf{w}[\mathsf{B}(\mathsf{c}, \mathsf{d}, 0, \mathsf{w}) \land \mathsf{G}(\vec{\mathsf{x}}, \mathsf{w})] \land$
$\forall \mathsf{u}[\mathsf{u} < \mathsf{y} \to \exists \mathsf{v} \exists \mathsf{w}\{(\mathsf{B}(\mathsf{c}, \mathsf{d}, \mathsf{u}, \mathsf{v}) \land \mathsf{B}(\mathsf{c}, \mathsf{d}, \mathsf{Su}, \mathsf{w})) \land \mathsf{H}(\vec{\mathsf{x}}, \mathsf{u}, \mathsf{v}, \mathsf{w})\}] \land$
$\mathsf{B}(\mathsf{c}, \mathsf{d}, \mathsf{y}, \mathsf{z})\}$

It can then again be checked that this defined wff $\varphi(\vec{\mathsf{x}}, \mathsf{y}, \mathsf{z})$ will serve to capture the p.r. defined function f.

9.4 The adequacy theorem refined

We have now outlined proofs of the three propositions (1) to (3) which we stated at the beginning of Section 9.1. Which suffices to establish

Theorem 8 Q *is strongly p.r. adequate – i.e. can capture-as-a-function every p.r. function.*

But in fact we have proved rather more than we set out to do. For we've just shown not only that every p.r. function is captured by some relational Q wff, but that every such function is captured by a wff which transparently reveals it to *be* a p.r. function.

To repeat once more, a p.r. function f can always be defined by a chain of definitions by composition and primitive recursion, starting from some initial function s. And we've just shown that we can capture f by a corresponding wff *which is built up by steps which recapitulate the definitions.* So we therefore

end up with a wff from which we can read back f's p.r. definition. Let's say that a wff which captures a p.r. function in this revealing way is a *perspicuous* representation of the function. Then we can refine our theorem: Q can capture every p.r. function in a perspicuous way. And we'll take it henceforth that when we talk of a p.r. function being captured by a wff in a formal arithmetic, we have in mind this sort of revealing representation.

Finally, here's an immediate corollary of Theorem 8:

Theorem 8* Q *can capture every p.r. property and relation.*

That's because, a property is p.r. if it has a p.r. characteristic function; and this characteristic function, being p.r., can be captured in Q; but by the result we noted in Section 7.3, if a property's characteristic function is capturable, so is that property itself. Likewise for relations.

Finally, we remark again – as we did at the end of Chapter 7 – that if Q is strongly p.r. adequate, then of course PA, which can prove everything that Q proves, must be strongly adequate too.

10 The Arithmetization of Syntax

This chapter introduces the key idea that we can associates *numbers* (the ostensible subject matter of PA wffs) with *wffs* and *proofs* from our formal arithmetic, and thereby reflect facts about wffs and proofs by numerical claims expressible in arithmetic.

10.1 Gödel numbering

We've already encountered one numbering device in Section 2.6; there we mapped symbols from the alphabet of a theory's language to (base ten) digits, and associated a concatenation of symbols with (the number expressed by) the corresponding concatenation of digits. This sort of thing would work for our present purposes too, but here we'll describe something more like Gödel's original numbering scheme.

Suppose then that our version of PA has the usual logical symbolism (connectives, quantifier symbols, identity, brackets), and symbols for zero and the successor, addition and multiplication functions: associate all those with odd numbers (different symbol, different number, of course). We also need a never-ending supply of variables, say $\mathsf{x}, \mathsf{y}, \mathsf{z}, \mathsf{u}, \ldots$ which we'll associate with even numbers. So, for example, we might have this preliminary series of *symbol codes*:

$\neg$	$\wedge$	$\vee$	$\rightarrow$	$\forall$	$\exists$	$=$	(	)	0	S	$+$	$\times$	x	y	z	u	$\ldots$
1	3	5	7	9	11	13	15	17	19	21	23	25	2	4	6	8	$\ldots$

If our logical apparatus uses parameters ('arbitrary names') then we'll need symbol codes for them too.

A Gödelian numbering scheme for expressions of PA is now defined in terms of this table of preliminary symbol codes as follows:

> Suppose the expression e is a sequence of k symbols $s_0, s_1, s_2, \ldots, s_k$. Then its *Gödel number* (g.n.) is calculated by taking the symbols' correlated code-numbers, using them in turn as exponents for the prime numbers $\pi_0, \pi_1, \pi_2, \ldots, \pi_k$, and then multiplying the results.

For example, the single symbol 'S' has the g.n. 2^{21} (the first prime raised to the appropriate power as read off from our correlation table of symbol codes). The standard numeral SS0 has the g.n. $2^{21} \cdot 3^{21} \cdot 5^{19}$ (the product of the first three primes raised to the appropriate powers). While the wff

$$\exists \mathsf{y}((\mathsf{S0} + \mathsf{y}) = \mathsf{SSS0})$$

has the g.n.

$$2^{11} \cdot 3^4 \cdot 5^{15} \cdot 7^{15} \cdot 11^{21} \cdot 13^{19} \cdot 17^{23} \cdot 19^4 \cdot 23^{17} \cdot 29^{13} \cdot 31^{21} \cdot 37^{21} \cdot 41^{21} \cdot 43^{19} \cdot 47^{17}$$

Which is, of course, *enormous*. So when we say that it is elementary to decode the resulting g.n. again by taking the exponents of prime factors, we don't mean that the computation is quick and easy. We mean that the computational routine required for the task – namely, repeatedly extracting prime factors – involves no more than the mechanical operations of school-room arithmetic.

Two remarks. First, we've allowed the introduction of abbreviatory symbols in PA (like '$\leq$' and '$\overline{3}$'); take the g.n. of an expression including such symbols to be the g.n. of the unabbreviated version.

Second, we will later be assuming there are similar Gödel numbering schemes for the expressions of other theories T with possibly different languages. We can imagine each of these numbering schemes to be built up in the same way but from a different table of preliminary symbol codes to cope with the different basic symbols of T's language. We won't spell out the details.

10.2 Coding sequences

As we said in Section 8.2, the relation $prfseq(m, n)$ will be crucial to what follows, where this relation holds just when m codes for an array of wffs that is a PA proof, and n codes for the closed wff that is thereby proved. But how *do* we code for proof-arrays?

The details will depend on the kind of proof-system we've adopted for our version of PA. To keep things simple, let's assume that the proof-system is a rather old-fashioned linear one (not a tree system), so proof-arrays are simply *sequences* of wffs. Then a nice way of coding these is by what we'll call *super Gödel numbers*. So, given a sequence of PA wffs or other expressions $e_0, e_1, e_2, \ldots, e_n$, we first code each e_i by a regular g.n. g_i, to yield a resulting sequence of regular Gödel numbers. We then encode this sequence into a *super* g.n by repeating the trick of taking powers of primes. For example, the sequence of regular g.n. $g_0, g_1, g_2, \ldots, g_n$ is wrapped up into the super g.n. $2^{g_0} \cdot 3^{g_1} \cdot 5^{g_2} \cdot \cdots \cdot \pi_n^{g_n}$. Hence, decoding a super g.n. involves two steps of taking prime factors: first find the exponents of the prime factors of the super g.n.; then treat those exponents as themselves regular g.n., and take their prime factors to arrive back at a sequence of PA expressions.

We can now define *prfseq* more carefully:

> $prfseq(m, n)$ is true just if m is the *super* g.n. of a sequence of wffs that is a PA proof of the closed wff with *regular* g.n. n.

10.3 *prfseq* is p.r.

Gödel's next main target is to prove that $prfseq(m, n)$ and some cognate functions and relations are primitive recursive.

Writing at the very beginning of the period when concepts of computation were being forged, Gödel couldn't really expect his audience to take anything on trust about what was or wasn't p.r., and he therefore had to do all the hard work of explicitly showing how to define $prfseq(m, n)$ by a chain of definitions by composition and primitive recursion. Assuming, however, that we have even a pretty modest familiarity with the idea of computer programs, we now can perhaps short-cut all that effort and be persuaded by

A very sketchy argument To determine whether $prfseq(m, n)$, proceed as follows. First decode m: that's a mechanical exercise. Now ask: is the result a sequence of PA wffs? That's algorithmically decidable. If it does decode into a sequence of wffs, ask: is this sequence a properly constructed proof? That's decidable too. If it is a proof, ask: is the last wff in the sequence, i.e. the wff proved, a sentence – i.e. does it lack free variables? That's decidable. If it is a sentence, ask: does that final wff have the g.n. n? That's again decidable. So, putting all that together, there is a computational procedure for telling whether $prfseq(m, n)$ holds. Moreover, *the computation involved is at each stage a straightforward, bounded procedure that doesn't involve any open-ended search.* Suppose then that we set out to describe the algorithm a computer could use to determine whether $prfseq(m, n)$. We will be able to do this using nothing more complex than bounded 'for' loops (we don't need to use any open-ended 'do while'/'do until' structure).

That means we should be able to turn our informal algorithm into a numerical LOOP program in the sense of Section 6.5 (a program that – where appropriate – operates on number codes for expressions rather than on the expressions themselves). However, we noted that what a numerical LOOP program computes is primitive recursive. Hence *prfseq* is primitive recursive. ⊠

That is indeed all pretty sketchy, but the argument may well strike you as convincing enough all the same. And if you are then happy to take it on trust that we can make all this rigorous, that's fine. If you aren't, then read Section 10.6!

10.4 The idea of diagonalization

To introduce the next results we want, we need to pause to take a quick look ahead. Gödel is going to tell us how to construct a formal wff G in PA that – putting it crudely – says 'I am unprovable'. We now have an inkling of how he might be able to do that: wffs can contain numerals which refer to Gödel-numbers which are in turn correlated with wffs.

The key step in Gödel's own construction involves taking an open wff that we'll abbreviate U(y), which contains just 'y' free, and substituting *the numeral for that wff's g.n.* for the free variable. So if the g.n. of 'U(y)' is u, which is

denoted by the standard numeral '$\overline{\mathsf{u}}$', then the key step involves forming the wff $\mathsf{U}(\overline{\mathsf{u}})$.[1]

This involves something quite closely akin to the 'diagonal' constructions we encountered in e.g. Sections 4.2, 6.6. In the first case, we matched the index of a wff $\varphi_n(\mathsf{x})$ (in an enumeration of wffs with one free variable) with the numeral substituted for its free variable, to form $\varphi_n(\overline{\mathsf{n}})$. In the second case, we matched the index of a function f_n (in an enumeration of p.r. functions) with the number the function takes as argument, to form $f_n(n)$. Here, in our Gödelian construction, we match $\mathsf{U}(\mathsf{y})$'s Gödel number – and we can think of this g.n. as indexing the wff in a list of wffs – with the numeral substituted for its free variable, and this yields the Gödel sentence G. Hence the standard claim that Gödel's construction again involves a type of diagonalization.

Now just note the following additional point. Given the wff '$\mathsf{U}(\mathsf{y})$' with g.n. u, it can't matter much whether we do the 'diagonalization' construction by forming (i) $\mathsf{U}(\overline{\mathsf{u}})$ (as Gödel himself does) or by forming (ii) $\exists \mathsf{y}(\mathsf{y} = \overline{\mathsf{u}} \land \mathsf{U}(\mathsf{y}))$. For (i) and (ii) are trivially equivalent. But it makes things go very slightly easier if we do things the second way. So here's an official definition, which applies to any expression:

> The *diagonalization* of φ is $\exists \mathsf{y}(\mathsf{y} = \overline{\mathsf{n}} \ \land \ \varphi)$, where '$\overline{\mathsf{n}}$' is numeral for φ's g.n.

10.5 *diag* and *gdl* are p.r.

We next note a couple more results about p.r. functions. First:

> There is a p.r. function $diag(n)$ which, when applied to a number n which is the g.n. of some wff, yields the g.n. of that wff's diagonalization.

Another very sketchy argument Decode the g.n. n to get some expression φ. Then form its diagonalization, $\exists \mathsf{y}(\mathsf{y} = \overline{\mathsf{n}} \ \land \ \varphi)$. Then work out the g.n. of the result. That is a very simple mechanical procedure to compute $diag(n)$, which doesn't involve any unbounded searches. So we again we should be able to turn the procedure into a LOOP program. Hence *diag* is p.r. ⊠

Second, let's consider the numerical relation which we'll dub '*gdl*' (and which will play a starring role in the next chapter):

> $gdl(m, n)$ holds when m is the super g.n. for a PA proof of the diagonalization of the wff with g.n. n.

Again, *gdl* is intuitively computable (without unbounded searches) and so again we expect it to be primitive recursive. And assuming that *prfseq* and *diag* are indeed p.r., then we can actually *prove* in a couple of lines that *gdl* is p.r.:

[1] There's no special significance to using the variable 'y' for the free variable here! But we'll keep this choice fixed, simply for convenience.

Proof Evidently, we can just put

$$gdl(m, n) =_{\text{def}} prfseq(m, diag(n))$$

Then gdl is defined by the composition of p.r. functions, so is itself p.r. ⊠

10.6 Gödel's proof that *prfseq* is p.r.

We've given, then, an informal but hopefully entirely persuasive argument that *prfseq* is primitive recursive. Gödel, however, gives a proper proof of this result by showing how to define a sequence of more and more complex functions and relations by composition and primitive recursion, ending with a p.r. definition of *prfseq*. Inevitably, this is a laborious job: Gödel does it with masterly economy and compression but, even so, it takes him 45 steps of function-building to get there.

We've in fact already traced some of the first steps in Section 6.4. We showed, in particular, that extracting exponents of prime factors – the key operation used in decoding Gödel numbers – involves a p.r. function *exp*. We now need to keep going in the same vein, defining ever more complex functions. What I propose to do here is fill in the next few steps fairly carefully, and then indicate more briefly how the rest go. This should be enough to give you a genuine feel for Gödel's construction and convince you that it can be completed, even though I don't go into every last detail.

Needless to say, you are very welcome to skip the rest of this section and jump to the next chapter: you'll miss nothing of wider conceptual interest.[2]

Our discussion is divided into stages marked (A) to (D).

(A) *Preliminaries* If you are still reading, first quickly revisit Section 6.4. Recall two facts in particular:

v. The function $exp(n, i)$ is p.r., where this returns the exponent of π_i in the prime factorization of n.

vi. The function $len(n)$ is p.r., where this returns the number of distinct prime factors of n.

Note, then, that if n is the g.n. of an expression e which is a sequence of symbols $s_0, s_1, s_2, \ldots, s_k$, then $exp(n, i)$ gives the symbol code of s_i. And if n is the super g.n. of a sequence of wffs or other expressions $e_0, e_1, e_2, \ldots, e_k$, then $exp(n, i)$ gives the g.n. of e_i. Further, note that if n is a g.n., then it consists in multiples of the first n primes: hence – since we count the primes from π_0 – the highest prime factor of n is $\pi_{len(n)-1}$

Now let's introduce a rather cute bit of new notation:

[2]Look at it like this. We argued on general grounds in the previous section that there *is* a LOOP program for determining whether $prfseq(m, n)$ holds. This section in effect describes how to write the program. Which is fun in its way. But once you are are convinced that the programming tricks *can* be done, you can cheerfully forget *how* they are done.

> If φ is an expression of PA, then we'll use '$\ulcorner\varphi\urcorner$' to denote φ's Gödel number,

So, for example: '$\wedge$' and 'SS0' are both PA expressions. And on our numbering scheme, $\ulcorner\wedge\urcorner$, the g.n. of '$\wedge$', is 2^3, and $\ulcorner\mathsf{SS0}\urcorner$ is $2^{21}\cdot 3^{21}\cdot 5^{19}$.

We will now add two more examples of p.r. properties and functions to the six we met before:

vii. Let $var(n)$ be true just when n is the g.n. of a variable. Then var is a p.r. property.

viii. We can define a p.r. function 'star' function such that $(m\star n)$ is the g.n. of the expression which results from stringing together the expression with g.n. m followed by the expression with g.n. n.

Proof for (vii) We just note that

$$var(n)\equiv(\exists x\leq n)(n=2^{2x})$$

The exponential function is p.r., and a definition by bounded quantification preserves primitive recursiveness, so var is p.r. too. Of course, this definition of var depends on our particular coding scheme: alternative methodical schemes will generate other p.r. functions. ⊠

Proof for (viii) Suppose m and n are Gödel numbers, and that $len(m)=j$ and $len(n)=k$. We then want the function $(m\star n)$ to yield the value obtained by taking the first $j+k$ primes, raising the first j to powers that match the exponents (taken in order) of the j primes in the prime factorization of m and then raising the next k primes to powers that match the k exponents in the prime factorization of n. Then $(m\star n)$ will indeed yield the g.n. of the expression which results from stringing together the expression with g.n. m followed by the expression with g.n. n. To take a simple example, if $m=2^{11}\cdot 3^4$ and $n=2^4\cdot 3^{13}\cdot 5^{19}$, then $(m\star n)=2^{11}\cdot 3^4\cdot 5^4\cdot 7^{13}\cdot 11^{19}$. But m is the g.n. of the expression '$\exists$y', and n is the g.n. of 'y = 0', and $(m\star n)$ is, as we want, the g.n. of their concatenation '$\exists$y y = 0'.

Again, suppose a is the g.n. of the wff '$\exists$x(x = S0)'; then $\ulcorner\neg\urcorner\star a$ is the g.n. of '$\neg\exists$x(Sx = 0)'. And suppose b is the g.n. of 'S0 = 0'. Then $\ulcorner(\urcorner\star a\star\ulcorner\rightarrow\urcorner\star b\star\ulcorner)\urcorner$ is the g.n. of '($\exists$x(Sx = 0) $\rightarrow$ S0 = 0)'.[3]

It is readily seen we can define a p.r. $(m\star n)$ which applies to Gödel numbers in the right way. Just put

$$(m\star n)=(\mu x\leq B_{m,n})[(\forall i<len(m))\{exp(x,i)=exp(m,i)\}\wedge (\forall i\leq len(n))\{exp(x,i+len(m))=exp(n,i)\}]$$

To finish, we just need to fix on a suitable $B_{m,n}$ to keep the minimization operator finitely bounded: π_{m+n}^{m+n} is certainly big enough to cover all eventualities! ⊠

[3]Note that $((m\star n)\star o)=(m\star(n\star o))$, which is why we can suppress internal bracketing with the star function.

(B) *Showing* $termseq(m, n)$*,* $term(n)$*, and* $atomic(n)$ *are p.r.* Formal proofs are sequences of wffs; wffs are defined as built up from atomic wffs. And since the only primitive predicate of PA is the identity relation, the atomic wffs of PA are defined as expressions of the kind '$\tau_1 = \tau_2$' where the τ_i are *terms*. So let's focus first on the definition of that last fundamental notion.

A term of PA (cf. Section 5.1) is either '0', or a variable, or is built up from those using the function-symbols 'S', '+', '×' – to give us, for example, the complex term $(\mathsf{S0} \times (\mathsf{SS0} + \mathsf{x}))$.

Now, let's say that a 'constructional history' for a term, or a *term-sequence*, is a sequence of expressions $\langle \tau_0, \tau_1, \ldots, \tau_n \rangle$ such that each expression τ_k in the sequence either (i) is '0', or else (ii) is a variable, or else (iii) has the form $\mathsf{S}\tau_j$, where τ_j is an earlier expression in the sequence, or else (iv) has the form $(\tau_i + \tau_j)$, where τ_i and τ_j are earlier expressions in the sequence, or else (v) has the form $(\tau_i \times \tau_j)$, where τ_i and τ_j are earlier expressions. Since any well-formed term must have the right kind of constructional history, we can adopt as our official definition: *a term is an expression which is the last expression in some term-sequence.*

That last observation motivates our being interested in the numerical relation $termseq(m, n)$, which holds when m is the super g.n. for a term-sequence, and n is the g.n. of the last expression in that term-sequence. And we can easily prove the following interim result: *termseq is primitive recursive.*

Proof We can define *termseq* by something of the following shape:

$$termseq(m, n) =_{\text{def}} \{(\forall k < len(m))[\ldots\ exp(m, k)\ \ldots] \land \\ exp(m, len(m) - 1) = n\}$$

The second conjunct here, i.e. $exp(m, len(m) - 1) = n$, ensures that the last expression in the sequence with super g.n. m indeed has g.n. n. And we now need to fill out the square brackets in the first conjunct in a way that reflects the fact for each $k < len(m)$, $exp(m, k)$ – the g.n. of τ_k in our putative term-sequence – is the g.n. of an expression satisfying one of the five defining conditions for belonging to a term-sequence. So, in those square brackets, we therefore need to say that $exp(m, k)$ is either

(i) the g.n. of '0', so $exp(m, k) = \ulcorner \mathsf{0} \urcorner$;

or (ii) the g.n. of a variable, so $var(exp(m, k))$;

or (iii) the g.n. of '$\mathsf{S}\tau_j$' where τ_j occurs earlier in the term sequence, so
$(\exists j < k)(exp(m, k) = \ulcorner \mathsf{S} \urcorner \star exp(m, j))$;

or (iv) the g.n. of '$(\tau_i + \tau_j)$' where τ_i and τ_j occur earlier, so
$(\exists i < k)(\exists j < k)(exp(m, k) = \ulcorner (\urcorner \star exp(m, i) \star \ulcorner + \urcorner \star exp(m, j) \star \ulcorner) \urcorner)$;

or (v) the g.n. of '$(\tau_i \times \tau_j)$' where τ_i and τ_j occur earlier, so
$(\exists i < k)(\exists j < k)(exp(m, k) = \ulcorner (\urcorner \star exp(m, i) \star \ulcorner \times \urcorner \star exp(m, j) \star \ulcorner) \urcorner)$

All the clauses are p.r. conditions, so their disjunction is a p.r. condition, so *termseq* is p.r., as we wanted to show. ⊠

It is an immediate corollary of this that the property $term(n)$ – which holds when n is the g.n. of a term of PA – *is also primitive recursive.*

Proof Since a term has to be the final member of some term sequence, we can give the following definition:

$$term(n) =_{\text{def}} (\exists x \leq B_n) termseq(x, n)$$

where B_n is a suitably large bound. Given a term with g.n. n, and hence with $l = len(n)$ symbols, its term-sequence will be at most l long: so the super g.n. of any term-sequence constructing it must be less than $(\pi_l^n)^l$. So we can set $B_n = (\pi_l^n)^l$. ⊠

And that leads to another quick result: define *atomic* to be the property of being the g.n. of an atomic wff of PA – then, $atomic(n)$ *is again primitive recursive*:

Proof In PA, the only atomic wffs are expressions of the kind $\tau_1 = \tau_2$. So put

$$atomic(n) =_{\text{def}}$$
$$(\exists x \leq n)(\exists y \leq n)[term(x) \wedge term(y) \wedge n = (x \star \ulcorner = \urcorner \star y)]$$

It is then immediate that $atomic(n)$ is p.r. ⊠

(C) *Showing* $formseq(m, n)$ *and* $wff(n)$ *are p.r.* We can now proceed to define the idea of a *formula-sequence* analogously to the idea of a term-sequence. So: a formula-sequence is a sequence of expressions $\langle \varphi_0, \varphi_1, \ldots, \varphi_n \rangle$ such that each φ_k in the sequence either (i) is an atomic wff, or else (ii) is of the form $\neg\varphi_j$ where φ_j is an earlier expression in the sequence, or else (iii) is of one of the forms $(\varphi_i \wedge \varphi_j)$ or $(\varphi_i \vee \varphi_j)$ or $(\varphi_i \rightarrow \varphi_j)$ where φ_i and φ_j are earlier expressions in the sequence, or else (iv) is of one of the forms $\forall\xi\varphi_j$, or $\exists\xi\varphi_j$ where φ_j is an earlier expression in the sequence and ξ a variable.

We now define the corresponding numerical relation $formseq(m, n)$ which holds when m is the super g.n. of a formula-sequence, and n is the g.n. of the last expression in that sequence. This too is primitive recursive. And to show that, we use exactly the same strategy as in our proof that *termseq* is p.r. – i.e. we put

$$formseq(m, n) =_{\text{def}} \{(\forall k < len(m))[\ldots\ exp(m, k)\ \ldots] \wedge$$
$$exp(m, len(m) - 1) = n\}$$

and this time we fill in the square brackets with clauses that reflect the conditions (i) to (iv) on being members of a formula sequence.

Since a wff by definition is constructed by some formula sequence, then – again by choosing a new suitable bound B_n – we can define the p.r. property

$$wff(n) =_{\text{def}} (\exists x \leq B_n) formseq(x, n).$$

We can also show in similar ways that e.g. *closed* is p.r., where $closed(n)$ holds just when n is the g.n. of a wff with no free variables.

(D) *Showing $prfseq(m,n)$ is p.r.* And now the end is in sight. We outline a proof that *prfseq* is primitive recursive as claimed.

Proof sketch The details here will depend on the type of proof system we are dealing with. To keep things particularly simple, suppose again we've adopted an old-fashioned linear proof-system for PA (which therefore doesn't allow temporary suppositions and sub-proofs). In this kind of system there are logical axioms – e.g. instances of the schemas '$(\varphi \to (\varphi \vee \psi))$' and '$(\forall \xi \varphi(\xi) \to \varphi(\tau))$' – and usually just two rules of inference, modus ponens and universal quantifier introduction. We can then define a PA proof as a sequence of wffs, each one of which is either (i) an axiom (a PA axiom or a logical axiom), or follows from previous wffs in the sequence by (ii) MP or (iii) UI.

The relation $modusponens(m,n,o)$ – which holds when the wff with g.n. o follows from the wffs with g.n. m and n by modus ponens – is obviously p.r., for it is immediate that $modusponens(m,n,o)$ just when

$$m = \ulcorner(\urcorner \star n \star \ulcorner\to\urcorner \star o \star \ulcorner)\urcorner \ \wedge \ wff(n) \ \wedge \ wff(o).$$

Likewise $univintro(m,n)$ is p.r., where this relation holds if n numbers a universal quantification of the wff numbered by m.

Suppose $axiom(n)$ is true just so long as n is the g.n. of an axiom of PA or a logical axiom. Then – following the pattern of our definition of *termseq* – we can say:

> $prfseq(m,n) =_{\text{def}} \forall k < len(m)$, $exp(m,k)$ – i.e. the g.n. of the k-th expression of the sequence – is either
> (i) the g.n. of an axiom, so $axiom(exp(m,k))$;
> or (ii) the g.n. of a wff that follows from two earlier wffs by MP, so
> $(\exists i \leq k)(\exists j \leq k) modusponens(exp(m,i), exp(m,j), exp(m,k))$;
> or (iii) the g.n. of a wff that follows from an earlier wff by UI, so
> $(\exists j \leq k) univintro(exp(m,j), exp(m,k))$:
> and $exp(m, len(m) - 1) = n$ and $closed(n)$.

Which is p.r. if *axiom* is. ⊠

So it just remains to check that the property of being the g.n. of a PA axiom is primitive recursive. Which it is – though showing this is just a bit tricky because we have to deal with the idea of being an instance of the induction schema. But we won't go on to nail down all the details here: enough is probably already more than enough!

One final point in this section. $prfseq(m,n)$ holds when m is the super g.n. of a proof of the wff with g.n. n. So $prov(n) =_{\text{def}} \exists x\, prfseq(x,n)$ holds when the wff with g.n. n is provable. But note that we *can't* read off from n some upper bound on the length of possible proofs for the wff with g.n. n. So we can't define the provability property by some *bounded* quantification of the kind $(\exists x \leq B) prfseq(x,n)$. If we could, the the provability probability would be p.r.: but it isn't – as we will show in Section 12.6.

10.7 Proving that *diag* is p.r.

We are almost done. But the numerical relation which will play a pivotal role when we prove Gödel's First Incompleteness Theorem is not *prfseq* itself but *gdl*, where

$$gdl(m, n) =_{\text{def}} prfseq(m, diag(n))$$

This relation is p.r. if the following function is:

> $diag(n)$ = the g.n. of the result of diagonalizing the expression with g.n. n.

We gave an informal argument that *diag* is indeed p.r. in Section 10.3: to round out this chapter, we can now give a quick proof.

We just need to add one last item to our catalogue of results about p.r. functions:

ix. The function $num(n)$ whose value is the g.n. of the standard numeral for n is p.r.

Proof for (ix) Note that the standard numeral for Sn is of the form 'S' followed by the standard numeral for n. So we have

$$num(0) = \ulcorner 0 \urcorner = 2^{19}$$
$$num(Sx) = \ulcorner \mathsf{S} \urcorner \star num(x) = 2^{21} \star num(x)$$

And we are done. ⊠

Proof that diag is p.r. We now just note that we can put

$$diag(n) = \ulcorner \exists \mathsf{y}(\mathsf{y} = \urcorner \star num(n) \star \ulcorner \wedge \urcorner \star n \star \ulcorner) \urcorner$$

And we get a p.r. function which does the desired job. ⊠

11 The First Incompleteness Theorem

The pieces we need are now all in place: so we can at last give Gödel's central construction which first delivers a true-but-unprovable sentence for PA and then can be generalized to show that this theory – and any other formal arithmetic satisfying some modest constraints – is not only incomplete but incompletable.

11.1 Constructing *G*

We've already trailed the headline news: Gödel constructs a PA sentence G that, when read in the light of the Gödel coding, says 'I am unprovable in PA'. But how does he do this?

First, two reminders. Recall (from Section 10.4):

> The diagonalization of φ is $\exists \mathsf{y}(\mathsf{y} = \overline{\mathsf{n}} \wedge \varphi)$, where '$\overline{\mathsf{n}}$' is numeral for φ's g.n.

And recall (from Section 10.5):

> $gdl(m, n)$ is true when m is the super g.n. for a PA proof of the diagonalization of the wff with g.n. n.

We have shown that this numerical relation gdl is primitive recursive. So by Theorem 8*, there must be an open wff of Q, and hence of PA, which captures it. Further, there will be a wff which captures the relation in a perspicuous way, so that we can read off the relation's (or rather, its characteristic functions's) p.r. definition (see Section 9.4). So let $\mathsf{Gdl}(\mathsf{x}, \mathsf{y})$ abbreviate some such perspicuous representation of gdl. Now consider the PA wff

$$\forall \mathsf{x} \neg \mathsf{Gdl}(\mathsf{x}, \mathsf{y})$$

Abbreviate this wff as $\mathsf{U}(\mathsf{y})$, and let its g.n. be u. Now diagonalize $\mathsf{U}(\mathsf{y})$; that yields

$$\exists \mathsf{y}(\mathsf{y} = \overline{\mathsf{u}} \wedge \forall \mathsf{x} \neg \mathsf{Gdl}(\mathsf{x}, \mathsf{y}))$$

This is the 'Gödel sentence' we've been aiming for: we'll abbreviate it G. And we immediately have the trivial logical equivalence

$$\mathsf{G} \equiv \forall \mathsf{x} \neg \mathsf{Gdl}(\mathsf{x}, \overline{\mathsf{u}})$$

Consider what it takes for G to be true, in the light of the fact that the formal predicate Gdl captures – and hence (assuming PA is sound) expresses – the numerical relation gdl. By the trivial equivalence, G is true if and only if

there is no number m such that $gdl(m, u)$. That is to say, given the definition of gdl, G is true if and only if there is no number m such that m is the code number for a PA proof of the diagonalization of the wff with g.n. u. But by construction, the wff with g.n. u is the wff U(y); and its diagonalization is G. So, G is true if and only if there is no number m such that m is the code number for a PA proof of G. But if G is provable, some number would be the code number of a proof of it. Hence G is true if and only if it is unprovable in PA. Wonderful!

11.2 Interpreting *G*

It is often claimed – incautiously – that G is not merely true if and only if unprovable in PA, but it actually *says* that it is unprovable in PA (indeed, for vividness, we have occasionally put it that way).

But we must be very careful here! The wff G is just another sentence of PA's language L_A, the language of basic arithmetic. It is an enormously long wff involving the first-order quantifiers, the connectives, the identity symbol, and 'S', '+' and '×', which all have the standard interpretation built into L_A. And on that interpretation, G is strictly speaking a complex claim about the results of ordinary arithmetical operations, no more and no less.

Now, when we summarily claim that G 'says' it is unprovable, this is *not* to invoke some radical re-interpretation of its symbols (for doing *that* would just make the claim trivial: if we are allowed radical re-interpretations – like spies fixing on a secret code – then any string of symbols can mean anything). No, it is because the symbols are being given their *standard* interpretation that we can recognize Gdl (when unpacked) as perspicuously expressing gdl, given the background framework of Gödel numbering which is involved in the definition of the relation gdl. And therefore we can recognize G as holding when no number m is such that $gdl(m, u)$: so we can immediately see, without further argument, that G is constructed in such as way as to make it true just when it is unprovable in PA. *That* is the limited sense in which it 'says' – with very heavy scare quotes – that it is unprovable.

11.3 *G* is unprovable in PA: the semantic argument

The argument now runs along the lines we sketched right back in Section 1.1. Assume that the axioms of PA are true on the standard interpretation (which they are, of course). We know that PA's logic – i.e. some version of classical first-order logic – is necessarily truth-preserving. Hence all its theorems are true. But if G (which is true if and only if it is unprovable) can be proved in PA, the theory proves a false theorem. Contradiction! Hence, G is not provable in PA. Hence G is true. So ¬G is false. Hence ¬G cannot be proved in PA either. In Gödel's words, G is a 'formally undecidable' wff of PA. So that gives us

Theorem 9 *If* PA *has true axioms, then there is a closed wff* φ *of* L_A *such that neither* PA $\vdash \varphi$ *nor* PA $\vdash \neg\varphi$.

If we are happy with the semantic assumption that PA's axioms are true on interpretation, the argument for incompleteness is as simple as that, once we have constructed G.[1] However, for reasons that will become clearer when we consider 'Hilbert's programme', it was important to Gödel that incompleteness can be proved without supposing that PA is sound: as he puts it, 'purely formal and much weaker assumptions' suffice.

11.4 '*G* is of Goldbach type'

Before showing how weaker assumptions suffice, let's pause to remark that the unprovable Gödel sentence G, while in one way very complicated when spelt out in all its detail, is in another sense very simple.

Recall Goldbach's conjecture that $\forall x P(x)$, where $P(x)$ expresses the property of being even and the sum of two primes. There is a simple mechanical routine for determining of any given number n whether it is has that property. You just check whether n is even and then look through the $n/2$ different pairs of (non-zero) numbers that which sum to n and test their members for being prime. A simple LOOP program will do the trick. Unsurprisingly, then, the property of being even and the sum of two primes in question is p.r.: i.e. there is a p.r. function which, for the argument n, returns the value 0 if $P(n)$ is true and returns 1 if it isn't. To put it another way, there is a p.r. function which decides whether $P(n)$.

Let's now say more generally that a formal proposition is of 'Goldbach type' if it is equivalent to one of the form $\forall \mathsf{x}\varphi(\mathsf{x})$ where $\varphi(\mathsf{x})$ similarly expresses a p.r. property. And now note that G *is of Goldbach type.* For $\mathsf{G} \equiv \forall \mathsf{x}\varphi(\mathsf{x})$, where $\varphi(\mathsf{x}) = \neg\mathsf{Gld}(\mathsf{x}, \overline{\mathsf{u}})$. But $\mathsf{Gld}(\mathsf{x}, \mathsf{y})$ expresses a p.r. relation, so $\mathsf{Gld}(\mathsf{x}, \overline{\mathsf{u}})$ for fixed u expresses a p.r. property: and recall that applying negation takes us from a p.r. property to a p.r. property. So $\varphi(\mathsf{x})$ does indeed express a p.r. property. Hence our Gödel sentence is in one sense no more complex than a familiar arithmetical claim such as Goldbach's conjecture. And we can strengthen our statement of Theorem 9, to give us

Theorem 9* *If* PA *has true axioms, then there is a closed* L_A*-wff* φ *of Goldbach type such that neither* PA $\vdash \varphi$ *nor* PA $\vdash \neg\varphi$.

11.5 *G* is unprovable in PA: the syntactic argument

We next show, *without* assuming PA's truth, that

[1] Note that Theorem 9 does *not* require the result that the numerical relation *gdl* is *captured* by some open wff $\mathsf{Gdl}(\mathsf{x}, \mathsf{y})$: we so far only need the weaker claim that *gdl* can be *expressed* by some $\mathsf{Gdl}(\mathsf{x}, \mathsf{y})$.

A. If PA is consistent, G is unprovable.

Proof Suppose G is provable in PA. If G has a proof, then there is some super g.n. m that codes its proof. But by definition, G is the diagonalization of the wff with g.n. u. Hence, by definition again, $gdl(m, u)$. Since Gdl captures gdl, it follows that (i) PA $\vdash \mathsf{Gdl}(\overline{\mathsf{m}}, \overline{\mathsf{u}})$.

But since G is logically equivalent to $\forall \mathsf{x} \neg \mathsf{Gdl}(\mathsf{x}, \overline{\mathsf{u}})$, the assumption that G is provable comes to this: PA $\vdash \forall \mathsf{x} \neg \mathsf{Gdl}(\mathsf{x}, \overline{\mathsf{u}})$. However, $\forall \mathsf{x} \neg \mathsf{Gdl}(\mathsf{x}, \overline{\mathsf{u}}) \vdash \neg \mathsf{Gdl}(\overline{\mathsf{m}}, \overline{\mathsf{u}})$. Hence (ii) PA $\vdash \neg \mathsf{Gdl}(\overline{\mathsf{m}}, \overline{\mathsf{u}})$.

So, combining (i) and (ii), the assumption that G is provable entails that PA is inconsistent. Hence, if PA is consistent, there can be no PA proof of G. ⊠

We can also show that ¬G is unprovable, again without assuming PA's axioms are true. But to do this, we need a somewhat stronger assumption than the mere consistency of PA. This takes a bit of explaining.

Let's say

> A theory of arithmetic T is ω-*inconsistent* if, for some wff $\varphi(\mathsf{x})$, $T \vdash \exists \mathsf{x} \varphi(\mathsf{x})$, yet for each number n we have $T \vdash \neg \varphi(\overline{\mathsf{n}})$.

('ω' – *omega* – is the logicians' label for the natural numbers taken in their natural order.) Note that ω-inconsistency, like ordinary inconsistency, is a syntactically defined property – i.e. a property characterized in terms of what wffs can be proved, not in terms of what they mean. Note also that ω-consistency – defined of course as not being ω-inconsistent! – implies plain consistency: that's because T's being ω-consistent is a matter of its *not* being able to prove a certain combination of wffs, which entails that T can't be inconsistent and prove *all* wffs.

On reflection, it should be clear that ω-inconsistency in a theory of arithmetic is a Very Bad Thing (not as bad as outright inconsistency, maybe, but still bad enough). For suppose that the axioms of an arithmetic theory T are true on a 'normal' interpretation (by which we mean an interpretation whose domain comprises just zero and its successors, and on which T's standard numerals are correctly assigned to the corresponding natural numbers). Assuming T has a sound logic, T's theorems will then all be true on this interpretation. So suppose that, for some $\varphi(\mathsf{x})$, T proves $\neg\varphi(\mathsf{0}), \neg\varphi(\overline{\mathsf{1}}), \ldots, \neg\varphi(\overline{\mathsf{n}}), \ldots$ for each and every number n. All these theorems will then be true on the given normal interpretation; so this means that every natural number must satisfy $\neg\varphi(\mathsf{x})$. So there is no object in the domain left over to satisfy $\varphi(\mathsf{x})$. So $\exists \mathsf{x} \varphi(\mathsf{x})$ will be false on this interpretation. Therefore it can't be a theorem. Hence, contraposing, if T is ω-inconsistent – i.e. each of $\neg\varphi(\overline{\mathsf{n}})$ is a theorem and yet $\exists \mathsf{x} \varphi(\mathsf{x})$ is a theorem too – then T cannot have a normal arithmetic interpretation. Given we want a theory of arithmetic to have theorems which are all true on a normal interpretation, we therefore want a theory to be ω-consistent.

With that preamble, we now prove

B. If PA is ω-consistent, ¬G is unprovable.

Proof Suppose that PA is ω-consistent but $\neg$G is provable in PA. If PA is ω-consistent, it is consistent. So if $\neg$G is provable, G is *not* provable. Hence take any m you like, m cannot code for a proof of G. But G is (again!) the wff you get by diagonalizing the wff with g.n. u. Therefore, by definition, our assumptions imply $\neg gdl(m, u)$, for each m. Hence, by the requirement that Gdl captures gdl, we have PA $\vdash \neg\mathsf{Gdl}(\overline{\mathsf{m}}, \overline{\mathsf{u}})$ for each m. But we are assuming that $\neg$G is provable in PA, which is equivalent to assuming PA $\vdash \exists \mathsf{x}\mathsf{Gdl}(\mathsf{x}, \overline{\mathsf{u}})$. And that makes PA ω-inconsistent after all, contrary to hypothesis. Hence, if PA is ω-consistent, $\neg$G is unprovable. ⊠

Putting results (A) and (B) together, we have the following:

> **Theorem 10** *If* PA *is consistent, then there is a closed L_A-wff φ of Goldbach type such that φ is not provable; if* PA *is ω-consistent, $\neg\varphi$ is not provable either.*

And note, by the way, the important point that this existence claim is proved 'constructively' – i.e. by giving us a recipe for actually producing the 'formally undecidable' wff φ.

11.6 Gödel's First Theorem

As we said at the very outset, however, the real bite of Gödelian incompleteness results isn't in the claim that this or that particular theory such as PA happens to be incomplete, but in the fact that his proof-method is general, and applies to any theory satisfying modest conditions.

Looking again at our proof for Theorem 10, we can see that the essential facts underpinning our incompleteness proof are:

1. The relation gdl is primitive recursive.
2. PA is (at least weakly) p.r. adequate, so there is an open wff Gdl such that if $gdl(m, u)$ then PA $\vdash \mathsf{Gdl}(\overline{\mathsf{m}}, \overline{\mathsf{u}})$, and if not-$gdl(m, u)$ then PA $\vdash \neg\mathsf{Gdl}(\overline{\mathsf{m}}, \overline{\mathsf{u}})$.

And – we mustn't forget –

3. We have quantification available in order to form the quantified wff G.[2]

Now, gdl can be defined in terms of *prfseq* (and *diag*). And showing *prfseq* to be p.r. essentially requires just that we don't need to do an open-ended search to check whether the candidate wff is an axiom or an array of wffs is a proof. To put it another way, we require the following: the property of being the code number of a PA axiom (whether a logical axiom or a special axiom) is p.r., and the

[2]In Section 6.7, we outlined the construction of the p.r. adequate theory PBA. Yet, as we remarked, that theory is negation-complete. How can that possibly be, given Gödelian results? Well, note how that the absence of quantifiers in the language of PBA blocks our forming a Gödel sentence for that weak theory.

relation that holds between the code numbers of wffs when one is the immediate consequence of the other(s) is p.r. too. We will call a theory that satisfies this condition *p.r. axiomatized*. So the fundamental fact underlying (1) is

G1. PA is p.r. axiomatized.[3]

Next, recall our definition: a theory is *strongly* p.r. adequate if it captures each p.r. function as a function – where capturing as a function is defined using the universal quantifier (see Sections 7.3, 7.4). So we can in fact can wrap up conditions (2) and (3) into one:

G2. PA is strongly p.r. adequate.

Our argument for PA's incompleteness therefore generalizes to cover any other axiomatized formal theory T which satisfies the same conditions (G1) and (G2). Of course, when we move to consider a different theory T, the set of axioms and/or the set of rules of inference will change. So the details of the corresponding relation $gdl_T(m, n)$ – which holds when m codes for a T-proof of a certain wff – will change too. Hence we'll need a new G_T as the G-style unprovable sentence for T. But we construct this new Gödel sentence in the same way as before, from an open wff which captures the gdl_T relation for T: and this open wff – like any wff for capturing a p.r. relation – need only use basic arithmetical vocabulary. So, even though our construction method produces a different formally undecidable sentence as we change the theory, it will always be the case that

> **Theorem 11** *If T is an ω-consistent, strongly p.r. adequate, p.r. axiomatized theory, then there is an arithmetic T-sentence φ of Goldbach type such that neither $T \vdash \varphi$ nor $T \vdash \neg\varphi$.*

It is this very general result has as much historical right as any to be called *Gödel's First Incompleteness Theorem*. Since we now know that Q is p.r. adequate, we can also assert

> **Theorem 11*** *If T is a consistent, strongly p.r. adequate, which includes the axioms of* Q, *then there is an arithmetic T-sentence φ of Goldbach type such that neither $T \vdash \varphi$ nor $T \vdash \neg\varphi$.*

Though, for the record, these two versions are not quite equivalent because there are other weak p.r. adequate arithmetics which neither contain nor are contained in Q.[4]

[3] Our intuitive characterization of a properly formalized theory said that properties like that of being an axiom of the theory should be *decidable* ones (Section 2.2) – or, as we later put it, should be properties with computable characteristic functions (Section 6.3). Now, we know that not all computable functions are p.r. (Section 6.6). So in principle, we could have an axiomatized theory (according to our intuitive characterization) which isn't p.r. axiomatized. We'll say more about this later. But for the moment, just note that such a theory would have to be very unwieldy if e.g. checking that a wff is an axiom requires a non-p.r. open-ended search. Any ordinary kind of axiomatized theory will certainly be p.r. axiomatized.

[4] For some details, see (Boolos et al., 2002, §§16.2, 16.4), comparing the theories *they* call **Q** and **R**.

Suppose we beef up theory T by adding new axioms: still, so long as the theory remains p.r. axiomatized and ω-consistent (and hence a candidate for having true axioms on a normal arithmetic interpretation), the new theory will also be incomplete. So Theorem 11 implies not just that our theory T is incomplete, but that it is in a good sense *incompletable.*

Take the particular case where the sentence G, constructed as described above, is added to PA as a new axiom. If the axioms of PA are all true on the normal interpretation built into L_A, so too are the axioms of $\mathsf{PA} + \mathsf{G}$, so the augmented theory remains ω-consistent. The augmented theory also remains p.r. adequate (since it can prove everything the original theory proved). It now trivially proves G. But Theorem 11 applies again. So there is a similarly constructed new Gödel sentence G' for our extended theory. Neither G' nor $\neg\mathsf{G}'$ follows from $\mathsf{PA} + \mathsf{G}$. So neither can follow from the unaugmented original theory, hence G' will be another true-but-unprovable sentence of PA, independent of the original Gödel sentence. Repeating the argument gives us an unending stream of such independent unprovable sentences. And we can never 'fix up' PA by adding on enough new axioms to get a p.r. axiomatized, ω-consistent, negation-complete theory: PA *is not just incomplete but incompletable.* Likewise for any p.r. axiomatized theory that includes Q and is therefore strongly p.r. adequate.

11.7 The idea of ω-incompleteness

We'll finish this chapter by pausing to bring out a key feature of PA revealed by our proof of the First Theorem, and then – in the last section – we'll note an important corollary of the Theorem.

Given PA is consistent,

1. For each m, $\neg\mathsf{Gdl}(\overline{\mathsf{m}}, \overline{\mathsf{u}})$ is provable. (We noted this halfway through the proof for (B) in Section 11.5; to repeat, if G is unprovable, then for any m, m cannot code a proof of G, so $\neg gdl(m, u)$, so PA proves $\neg\mathsf{Gdl}(\overline{\mathsf{m}}, \overline{\mathsf{u}})$.)

2. But G – i.e., equivalently, $\forall\mathsf{x}\neg\mathsf{Gdl}(\mathsf{x}, \overline{\mathsf{u}})$ – is unprovable.

A theory T which, for some open wff $\varphi(\mathsf{x})$, can prove each $\varphi(\overline{\mathsf{m}})$ but can't prove $\forall\mathsf{x}\varphi(\mathsf{x})$ is said to be *ω-incomplete.*[5] So the case where $\varphi(\mathsf{x}) =_{\text{def}} \neg\mathsf{Gdl}(\mathsf{x}, \overline{\mathsf{u}})$ shows that PA is ω-incomplete.

In Section 5.2, we noted that Q exhibits a radical case of ω-incompleteness: although it can prove case-by-case all true equations involving numerals, it can't prove most of their universal generalizations. For a simple example, put $\mathsf{Kx} =_{\text{def}} (0 + \mathsf{x} = \mathsf{x})$; then for every n, $\mathsf{Q} \vdash \mathsf{K}\overline{\mathsf{n}}$; but we don't have $\mathsf{Q} \vdash \forall\mathsf{x}\mathsf{Kx}$. In moving from Q to PA by adding the induction axioms, we vastly increase

[5] Compare and contrast: Suppose T can prove each $\varphi(\overline{\mathsf{m}})$. T is *ω-incomplete* if it can't prove something we'd *like* it to prove, namely $\forall\mathsf{x}\varphi(\mathsf{x})$. While – equivalently to our definition in Section 11.5 – T is *ω-inconsistent* if it can actually prove the *negation* of what we'd like it to prove, i.e. it can prove $\neg\forall\mathsf{x}\varphi(\mathsf{x})$, i.e. $\exists\mathsf{x}\neg\varphi(\mathsf{x})$.

our ability to prove generalizations. But we now see that some ω-incompleteness must remain.

And to stress the point we made in Section 11.4, this incompleteness remains at the most basic level. To repeat, the property expressed by $\varphi(\mathsf{x}) =_{\text{def}} \neg\mathsf{Gdl}(\mathsf{x}, \overline{\mathsf{u}})$ is p.r.: so we've shown that PA *isn't even ω-complete for wffs $\varphi(\mathsf{x})$ expressing p.r. properties*. Generalizing the argument, as in the proof of the Theorem 11, shows that this basic level of ω-incompleteness is ineliminable.

11.8 First-order True Arithmetic can't be p.r. axiomatized

Finally, let's draw out an immediate corollary of Theorem 11 (in order to make good another claim we made in informal terms back in Section 1.1, and sharpen the result of Section 4.4).

Suppose L_A is, as before, the language of basic arithmetic (i.e. the first-order language of successor/addition/multiplication, i.e. the language of Q and PA). And let TA be the set of true sentences of L_A, i.e. closed wffs which are true on the standard interpretation built into L_A. So TA is True (Basic) Arithmetic. Then we have

> **Theorem 12** *There is no consistent, p.r. axiomatized theory T whose set of theorems in the language L_A is exactly* TA.

Proof Suppose T entails all of TA. Then it must in particular prove the truths of basic arithmetic required for it to be p.r. adequate (for even Q does that). But then, if T is also consistent and p.r. axiomatized, we can use the Gödel construction to yield a sentence G in the language L_A which is true but unprovable-in-T, so there is a sentence in TA that T can't prove after all, contrary to hypothesis. So if T is consistent, it can't be p.r. axiomatized. ⊠

12 Extending Gödel's First Theorem

Further theorems now come thick and fast. By all means, skim and skip through this chapter on a first reading (as it is rather action-packed with technicalities), though it is worth returning later to explore some of the proofs.

We start by noting a clever trick due to Barkley Rosser, which enables us to weaken the assumption of ω-consistency which Gödel needed in proving his First Incompleteness Theorem.

But our main business here is to generalize the construction underlying the Gödel-Rosser results by proving the Diagonalization Lemma. We then use this elegant result in proving two new results: on the usual kind of assumptions about T, the property of being a *provable* wff of the arithmetical theory T can't be captured in T; and the property of being a *true* wff of T can't even be expressed in T's language.

12.1 Rosser's theorem

The headline news is that Rosser (1936) defines the Gödel-Rosser sentence R by an ingenious variation on the construction of the original Gödel sentence, and then shows that we only need to assume PA's consistency to prove that both R and $\neg\mathsf{R}$ are underivable in PA.

And frankly, that is all you need really to know. How the trick is done isn't that important. But here's a quick introduction just for the record.

First recall our now familiar definition:

> $gdl(m, n)$ holds when m is the super g.n. for a PA proof of the diagonalization of the wff with g.n. n.

And let's next introduce a companion definition:

> $gdl(m, n)$ holds when m is the super g.n. for a PA proof of the *negation* of the diagonalization of the wff with g.n. n.

Both relations are p.r.; so both are capturable in PA, by a pair of wffs that we'll abbreviate by $\mathsf{Gdl}(\mathsf{x}, \mathsf{y})$ and $\overline{\mathsf{Gdl}}(\mathsf{x}, \mathsf{y})$ respectively.

Now consider the wff

$$(\mathsf{Gdl}(\mathsf{x}, \mathsf{y}) \rightarrow (\exists \mathsf{v} \leq \mathsf{x})\overline{\mathsf{Gdl}}(\mathsf{v}, \mathsf{y}))$$

where '$(\exists \mathsf{v} \leq \mathsf{x})\phi$' of course abbreviates '$\exists \mathsf{v}(\mathsf{v} \leq \mathsf{x} \wedge \phi)$'. Roughly, this wff says that if a number x codes a proof of a certain wff, then there is a smaller number v that codes a proof of the negation of the same wff.

And, from this point, we simply echo the Gödel construction. So, first step, we quantify to get

$$\forall \mathsf{x}(\mathsf{Gdl}(\mathsf{x},\mathsf{y}) \rightarrow (\exists \mathsf{v} \leq \mathsf{x})\overline{\mathsf{Gdl}}(\mathsf{v},\mathsf{y}))$$

Abbreviate this open wff as $\mathsf{P}(\mathsf{y})$, and suppose its g.n. is p. Then, second step, diagonalize $\mathsf{P}(\mathsf{y})$ to get

$$\mathsf{R} =_{\text{def}} \exists \mathsf{y}(\mathsf{y} = \overline{\mathsf{p}} \wedge \mathsf{P}(\mathsf{y}))$$

This is the Rosser sentence; it is trivially equivalent to

$$\forall \mathsf{x}(\mathsf{Gdl}(\mathsf{x},\overline{\mathsf{p}}) \rightarrow (\exists \mathsf{v} \leq \mathsf{x})\overline{\mathsf{Gdl}}(\mathsf{v},\overline{\mathsf{p}}))$$

Which reveals that the Rosser sentence is again of Goldbach type.[1]

Let's consider what R 'says' in the light of the fact that the formal predicates Gdl and $\overline{\mathsf{Gdl}}$ express the numerical relations *gdl* and *gdlneg*. As the trivial equivalence shows, R is true when, for every number m, if $gdl(m,p)$ is true then there's a smaller number n such that $gdlneg(n,p)$. But by construction, $gdl(m,p)$ is true if m codes for a PA proof of R (think about it!). Similarly, $gdlneg(n,p)$ is true if n codes for a proof of $\neg\mathsf{R}$. Therefore R says that, for every number m, if m codes for a proof of R, there's a smaller number n that codes for a proof of $\neg\mathsf{R}$. Hence, R says (roughly) 'if R is provable, so is $\neg\mathsf{R}$'.

In the light of this interpretation, R must be unprovable, assuming PA has true axioms and hence is consistent. For if R were provable, then it would be true. In other words, 'if R is provable, so is $\neg\mathsf{R}$' would be true; and this conditional would have a true antecedent, and so we can infer that $\neg\mathsf{R}$ is provable, which makes PA inconsistent, contrary to hypothesis. Therefore R is unprovable. Hence the material conditional 'if R is provable, so is $\neg\mathsf{R}$' has a false antecedent, so is true. In other words, R is true. Hence its negation $\neg\mathsf{R}$ is false, and therefore also unprovable (still assuming PA is sound). Hence R is another true-but-formally-undecidable wff.

However, to show that neither R nor $\neg\mathsf{R}$ is provable, we don't need the semantic assumption that PA is sound; and this time – as we said – we don't even need to assume that PA is ω-consistent. Assuming the mere consistency of PA is enough. And, as with Gödel's original result, all that is required for the proof to go through is that we are dealing with a theory which is p.r. adequate and p.r. axiomatized. Which means that we can generalize, to get the Gödel-Rosser Theorem:

> **Theorem 13** *If T is a consistent, strongly p.r. adequate, p.r. axiomatized theory, then there is an arithmetic T-sentence φ of Goldbach type such that neither $T \vdash \varphi$ nor $T \vdash \neg\varphi$.*

[1] See Section 11.4. For R is equivalent to a quantification of an open wff which expresses a p.r. property – since both $\mathsf{Gdl}(\mathsf{x},\overline{\mathsf{p}})$ and $\overline{\mathsf{Gdl}}(\mathsf{x},\overline{\mathsf{p}})$ separately express p.r. properties, and applying the conditional truth-function and bounded quantification preserves primitive recursiveness (see Section 6.4).

The detailed proof is a close cousin of the proof of Gödel's Theorem. However, it is messier and less intuitive. We really won't miss anything of broader mathematical interest by omitting it here.[2]

Rosser's work nicely sharpens Gödel's original result. But it doesn't really give us any additional insight into the underlying structure of his argument. So now we turn to digging rather deeper.

12.2 Another bit of notation

It is convenient and standard to introduce a really pretty bit of notation (if you tackled Section 10.6, then this is in part a re-introduction):

> If φ is an expression of a formal theory T, then we'll use '$\ulcorner\varphi\urcorner$' *in our logicians' augmented English* to denote φ's Gödel number.

Here we mean, of course, φ's Gödel number with respect to a given scheme for numbering expressions from T's language (see the end of Section 10.1). Borrowing a species of quotation mark is appropriate because the expression '$\ulcorner\varphi\urcorner$' can be read, via our coding scheme, as referring to the expression φ.

We are also going to use this same notation as a placeholder for numerals inside a formal theory T, so that (in our second usage)

> *In formal shorthand for T-wffs*, '$\ulcorner\varphi\urcorner$' holds the place for T's standard numeral for the g.n. of φ.

In other words, inside formal expressions '$\ulcorner\varphi\urcorner$' stands in for the numeral for the number $\ulcorner\varphi\urcorner$.

A simple example to illustrate:

1. '$\mathsf{SS0}$' is a PA expression, the standard numeral for 2.
2. On our numbering scheme $\ulcorner\mathsf{SS0}\urcorner$, the g.n. of '$\mathsf{SS0}$', is $2^{21} \cdot 3^{21} \cdot 5^{19}$.
3. So, by our further convention, we can also use the expression '$\ulcorner\mathsf{SS0}\urcorner$' inside (a definitional extension of) PA, as an abbreviation for the standard numeral for that g.n., i.e. as an abbreviation for '$\mathsf{SSS\ldots S0}$' with $2^{21} \cdot 3^{21} \cdot 5^{19}$ occurrences of 'S'!

This double usage – outside our formal language to denote a g.n. of a formal expression and inside our formal language to take the place of a standard numeral for that g.n. – should by this stage cause no confusion. We could have continued our earlier practice of overlining place-holders for standard numerals, where – for example – the numeral for n is signified by '$\overline{\mathsf{n}}$': we could then indicate the numeral for the g.n. number $\ulcorner\mathsf{SS0}\urcorner$ by '$\overline{\ulcorner\mathsf{SS0}\urcorner}$'. But both aesthetics and common convention count against this.

[2] If you must have the gory details, then you can find them in e.g. (Mendelson, 1997, pp. 208–210).

12.3 Diagonalization again

Let's think again about how our Gödel sentence for PA was constructed. Working backwards, we defined G by diagonalizing the open wff U which contains 'y' as it sole free variable. In other words,

$$\mathsf{G} =_{\mathrm{def}} \exists \mathsf{y}(\mathsf{y} = \ulcorner \mathsf{U} \urcorner \wedge \mathsf{U})$$

where, using our new notation, '$\ulcorner \mathsf{U} \urcorner$' in this formal wff holds the place of the standard numeral for U's g.n. $\ulcorner \mathsf{U} \urcorner$. Here,

$$\mathsf{U} =_{\mathrm{def}} \forall \mathsf{x} \neg \mathsf{Gdl}(\mathsf{x}, \mathsf{y})$$

In turn, $\mathsf{Gdl}(\mathsf{x}, \mathsf{y})$ captures the relation *gdl*, where – as we saw in Section 10.5 – we can put

$$gdl(m, n) =_{\mathrm{def}} prfseq(m, diag(n))$$

Now, the one-place p.r. function *diag* can be captured in PA by some two-place open wff $\mathsf{Diag}(\mathsf{x}, \mathsf{y})$; and the two-place p.r. relation *prfseq* can be captured by a two-place open wff $\mathsf{Prf}(\mathsf{x}, \mathsf{y})$. So we can therefore give the following definition:

$$\mathsf{Gdl}(\mathsf{x}, \mathsf{y}) =_{\mathrm{def}} \exists \mathsf{z}(\mathsf{Prf}(\mathsf{x}, \mathsf{z}) \wedge \mathsf{Diag}(\mathsf{y}, \mathsf{z}))$$

Putting these steps together, we therefore have

$$\begin{aligned}
\mathsf{U} &=_{\mathrm{def}} \forall \mathsf{x} \neg \mathsf{Gdl}(\mathsf{x}, \mathsf{y}) \\
&=_{\mathrm{def}} \forall \mathsf{x} \neg \exists \mathsf{z}(\mathsf{Prf}(\mathsf{x}, \mathsf{z}) \wedge \mathsf{Diag}(\mathsf{y})) \\
&\equiv \quad \forall \mathsf{x} \forall \mathsf{z} \neg(\mathsf{Prf}(\mathsf{x}, \mathsf{z}) \wedge \mathsf{Diag}(\mathsf{y}, \mathsf{z})) \\
&\equiv \quad \forall \mathsf{z} \forall \mathsf{x} \neg(\mathsf{Prf}(\mathsf{x}, \mathsf{z}) \wedge \mathsf{Diag}(\mathsf{y}, \mathsf{z})) \\
&\equiv \quad \forall \mathsf{z}(\mathsf{Diag}(\mathsf{y}, \mathsf{z}) \to \neg \exists \mathsf{x}(\mathsf{Prf}(\mathsf{x}, \mathsf{z})) \\
&\equiv \quad \forall \mathsf{z}(\mathsf{Diag}(\mathsf{y}, \mathsf{z}) \to \neg \exists \mathsf{v}(\mathsf{Prf}(\mathsf{v}, \mathsf{z})) \\
&=_{\mathrm{def}} \forall \mathsf{z}(\mathsf{Diag}(\mathsf{y}, \mathsf{z}) \to \neg \mathsf{Prov}(\mathsf{z})) \\
&=_{\mathrm{def}} \mathsf{U}'
\end{aligned}$$

Here, the penultimate step simply introduces a new abbreviation, $\mathsf{Prov}(\mathsf{z}) =_{\mathrm{def}} \exists \mathsf{v}\, \mathsf{Prf}(\mathsf{v}, \mathsf{z})$.

Three remarks about this: (1) $\mathsf{Prov}(\overline{\mathsf{n}})$ is true just so long as there is a number v such that $prfseq(v, n)$, i.e. just so long n is the g.n. of wff that has a PA proof. In particular, $\mathsf{Prov}(\ulcorner \varphi \urcorner)$ is true just so long as $\mathsf{PA} \vdash \varphi$. That means $\mathsf{Prov}(\mathsf{x})$ expresses the provability property *prov* (where $prov(n)$ is true just when n numbers a theorem: cf. the end of Section 10.6).

(2) When abbreviatory definitions are unpacked, our equivalences are provable within PA, since we've just derived them by simple logical manipulation. So, given that G is equivalent to the result of substituting '$\ulcorner \mathsf{U} \urcorner$' for 'y' in U, we have

$$\mathsf{PA} \vdash \mathsf{G} \equiv \forall \mathsf{z}(\mathsf{Diag}(\ulcorner \mathsf{U} \urcorner, \mathsf{z}) \to \neg \mathsf{Prov}(\mathsf{z}))$$

(3) Since U and U' are trivially equivalent, then it shouldn't matter whether we consider G, the diagonalization of U, or G', the diagonalization of U'. Those diagonalizations involve different numerals (denoting the different Gödel numbers $\ulcorner\mathsf{U}\urcorner$ and $\ulcorner\mathsf{U}'\urcorner$): but it is a very easy exercise to check that, indeed, G' will do just as well as G for proving Gödel's theorem.

For example, suppose that G' is provable. Then some number m is the g.n. of a proof of the diagonalization of U', i.e. $gdl(m, \ulcorner\mathsf{U}'\urcorner)$. So $\mathsf{PA} \vdash \mathsf{Gdl}(\overline{\mathsf{m}}, \ulcorner\mathsf{U}'\urcorner)$. But to suppose G' provable is — by our equivalences – tantamount to to supposing $\mathsf{PA} \vdash \forall \mathsf{x} \mathsf{Gdl}(\mathsf{x}, \ulcorner\mathsf{U}'\urcorner)$, and contradiction is immediate.

12.4 The Diagonalization Lemma

Our chain of equivalences, together with the fact that Diag captures *diag*, quickly gives us

> **Theorem 14** $\mathsf{PA} \vdash \mathsf{G} \equiv \neg\mathsf{Prov}(\ulcorner\mathsf{G}\urcorner)$.

Proof Since diagonalizing U yields G, $diag(\ulcorner\mathsf{U}\urcorner) = \ulcorner\mathsf{G}\urcorner$, by the definition of *diag*. Hence, since by hypothesis Diag captures *diag*, we have

$$\mathsf{PA} \vdash \forall \mathsf{z}(\mathsf{Diag}(\ulcorner\mathsf{U}\urcorner, \mathsf{z}) \equiv \mathsf{z} = \ulcorner\mathsf{G}\urcorner)$$

But we have just seen that

$$\mathsf{PA} \vdash \mathsf{G} \equiv \forall \mathsf{z}(\mathsf{Diag}(\ulcorner\mathsf{U}\urcorner, \mathsf{z}) \to \neg\mathsf{Prov}(\mathsf{z}))$$

Hence, substituting a proven equivalent for '$\mathsf{Diag}(\ulcorner\mathsf{U}\urcorner, \mathsf{z})$', it follows that

$$\mathsf{PA} \vdash \mathsf{G} \equiv \forall \mathsf{z}(\mathsf{z} = \ulcorner\mathsf{G}\urcorner \to \neg\mathsf{Prov}(\mathsf{z})).$$

From which the desired theorem is immediate. ⊠

What this shows is that the informal claim 'G is true if and only if it is unprovable' can itself be formally proved within PA. Very neat!

A little reflection now shows that the basic construction that allows us to prove Theorem 14 can be generalized radically to yield the following much broader result – standardly called *The Diagonalization Lemma*:[3]

> **Theorem 15** *If T is a strongly p.r. adequate, p.r. axiomatized, theory and $\varphi(\mathsf{z})$ is an open wff with one free variable, then there is some sentence γ such that $T \vdash \gamma \equiv \varphi(\ulcorner\gamma\urcorner)$*

Proof Since T is a nicely axiomatized theory, we can define a p.r. function $diag_T$ which takes us from the g.n. of a T-wff to the g.n. of its diagonalization.[4]

[3] In a footnote added to later reprintings of his (1934), Gödel says that this Lemma 'was first noted by Carnap (1934)': first noted in print, yes; but Gödel himself had probably already got there in 1930.

[4] Of course, we are now talking in the context of some appropriate scheme for Gödel-numbering expressions of T – see the final remark of Section 10.1.

Since T is p.r. adequate and can capture any p.r. function, we can construct a two-place T-predicate $\mathsf{Diag_T}$ which captures the function $diag_T$ in T.

We can assume without loss of generality that $\varphi(\mathsf{z})$ doesn't contain occurrences of 'y'. And now let's do to '$\varphi(\mathsf{z})$' what the Gödelian construction does to '$\neg\mathsf{Prov}(\mathsf{z})$'. So, first step, put $\psi =_{\text{def}} \forall\mathsf{z}(\mathsf{Diag_T}(\mathsf{y},\mathsf{z}) \rightarrow \varphi(\mathsf{z}))$.

Next, construct γ, the diagonalization of ψ. So $\gamma \equiv \forall\mathsf{z}(\mathsf{Diag_T}(\ulcorner\psi\urcorner,\mathsf{z}) \rightarrow \varphi(\mathsf{z}))$.

Our theorem now follows speedily. Note that $diag_T(\ulcorner\psi\urcorner) = \ulcorner\gamma\urcorner$. Hence, $T \vdash \forall\mathsf{z}(\mathsf{Diag_T}(\ulcorner\psi\urcorner,\mathsf{z}) \equiv \mathsf{z} = \ulcorner\gamma\urcorner)$, since by hypothesis $\mathsf{Diag_T}$ captures $diag_T$. Hence $T \vdash \forall\mathsf{z}(\mathsf{Diag_T}(\ulcorner\psi\urcorner,\mathsf{z}) \rightarrow \varphi(\mathsf{z})) \equiv \forall\mathsf{z}(\mathsf{z} = \ulcorner\gamma\urcorner \rightarrow \varphi(\mathsf{z}))$, since the antecedents of the two conditionals are proven equivalents. But the left-hand side of our new biconditional is equivalent to γ, and the right-hand side is in turn equivalent to $\varphi(\ulcorner\gamma\urcorner)$. So $T \vdash \gamma \equiv \varphi(\ulcorner\gamma\urcorner)$. ⊠

We will use this beautiful Lemma three times over in the next three sections, first in revisiting the incompleteness result, and then to prove two further results.

But first a quick remark about jargon. Suppose that function f maps the argument a back to a itself, so that $f(a) = a$, then a is said to be a *fixed-point* for f; and a theorem to the effect that, under certain conditions, there is fixed point for f is a *fixed-point theorem*. By analogy, Theorem 15 is often also referred to as a fixed point theorem, with γ as (so to speak) a fixed-point for the predicate $\varphi(\mathsf{z})$.

12.5 Incompleteness again

We can think of the Incompleteness Theorem as derived by combining the very general Diagonalization Lemma with the following specific result:

> **Theorem 16** *Let T be a p.r. axiomatized, p.r. adequate theory, and let γ be any fixed point for $\neg\mathsf{Prov_T}(\mathsf{z})$. Then, if T is consistent, γ is unprovable, and if T is ω-consistent, then $\neg\gamma$ is unprovable.*

Here, $\mathsf{Prov_T}$ of course expresses provability in T – i.e. $\mathsf{Prov_T}(\mathsf{x}) =_{\text{def}} \exists\mathsf{v}\mathsf{Prf_T}(\mathsf{v},\mathsf{x})$, where $\mathsf{Prf_T}$ captures the p.r. $prfseq_T$ relation (and where $prfseq_T(m,n)$ is true when m is the super g.n. of at T-proof of the closed wff with g.n. n, on our chosen scheme for numbering T-expressions).

For future reference, it is useful first to separate out two simple preliminary results (and simply to promote readability, we'll now suppress subscript T's for the rest of this section, so for the moment read 'Prov' as '$\mathsf{Prov_T}$' etc.):

P1. For any sentence φ, if $T \vdash \varphi$, then $T \vdash \mathsf{Prov}(\ulcorner\varphi\urcorner)$.

P2. For any sentence φ, if not-($T \vdash \varphi$), then, for all n, $T \vdash \neg\mathsf{Prf}(\overline{\mathsf{n}},\ulcorner\varphi\urcorner)$.

Proof for (P1) Suppose $T \vdash \varphi$. Then there is a T proof of the wff with g.n. $\ulcorner\varphi\urcorner$. Let this proof have the super g.n. m. Then, by definition, $prfseq(m,\ulcorner\varphi\urcorner)$. Hence, since $prfseq$ for T is captured by Prf, it follows that $T \vdash \mathsf{Prf}(\overline{\mathsf{m}},\ulcorner\varphi\urcorner)$. So by existential quantifier introduction, $T \vdash \exists\mathsf{v}\,\mathsf{Prf}(\mathsf{v},\ulcorner\varphi\urcorner)$, i.e. $T \vdash \mathsf{Prov}(\ulcorner\varphi\urcorner)$. ⊠

Proof for (P2) Suppose φ is not provable in T, so no number is the g.n. of its proof – and therefore, by definition, for all n not-$prfseq(n, \ulcorner\varphi\urcorner)$. Then, since $prfseq$ for T is captured by Prf, it follows that for all n, $T \vdash \neg\mathsf{Prf}(\bar{\mathsf{n}}, \ulcorner\varphi\urcorner)$. ⊠

Proof for Theorem 16 γ is a fixed point for '$\neg\mathsf{Prov}$', i.e. $T \vdash \gamma \equiv \neg\mathsf{Prov}(\ulcorner\gamma\urcorner)$. But by (P1), we also have if $T \vdash \gamma$ then $T \vdash \mathsf{Prov}(\ulcorner\gamma\urcorner)$. So, if T is consistent, we can't have $T \vdash \gamma$.

Now assume T is ω-consistent (and hence consistent). Suppose $T \vdash \neg\gamma$. Since $T \vdash \gamma \equiv \neg\mathsf{Prov}(\ulcorner\gamma\urcorner)$, it follows that $T \vdash \mathsf{Prov}(\ulcorner\gamma\urcorner)$, i.e. (a): $T \vdash \exists \mathsf{v}\mathsf{Prf}(\mathsf{v}, \ulcorner\gamma\urcorner)$. By consistency, T cannot also prove γ. So, by (P2), we have (b): for all n, $T \vdash \neg\mathsf{Prf}(\bar{\mathsf{n}}, \ulcorner\gamma\urcorner)$. But (a) and (b) together make T ω-inconsistent after all. Contradiction. So, assuming T *is* ω-consistent, we can't have $T \vdash \neg\gamma$. ⊠

In sum, Theorem 16 tells us that if there is a fixed point for $\neg\mathsf{Prov}_T$, then T can't be negation-complete (assuming it is ω-consistent); and Theorem 15 tells us that such a fixed point must exist. Put the results together and we are back with Gödel's First Theorem.

12.6 Provability

Consider again the T-wff $\mathsf{Prov_T}(\mathsf{x})$ which *expresses* the property $prov_T$ of being the g.n. of a T-theorem. The obvious next question to ask is: does this wff also case-by-case *capture* that property? The following theorem shows that it can't:

> **Theorem 17** *No open wff in a consistent, p.r. adequate theory T can capture the numerical property* prov_T*.*

Proof Suppose for reductio that $\mathsf{P}(\mathsf{x})$ abbreviates an open wff – not necessarily identical to $\mathsf{Prov}(\mathsf{x})_\mathsf{T}$ – which captures $prov_T$. By the Diagonalization Lemma applied to $\neg\mathsf{P}(\mathsf{z})$, there is some wff γ such that

(i) $T \vdash \gamma \equiv \neg\mathsf{P}(\ulcorner\gamma\urcorner)$.

By the general assumption that P captures $prov_T$, we have in particular

(ii) if $T \vdash \gamma$, i.e. $prov_T(\ulcorner\gamma\urcorner)$, then $T \vdash \mathsf{P}(\ulcorner\gamma\urcorner)$
(iii) if not-($T \vdash \gamma$), i.e. not-$prov_T(\ulcorner\gamma\urcorner)$, then $T \vdash \neg\mathsf{P}(\ulcorner\gamma\urcorner)$

Contradiction quickly follows. By (iii) and (i), if not-($T \vdash \gamma$), then $T \vdash \gamma$. Hence $T \vdash \gamma$. So by (ii) and (i) we have both $T \vdash \mathsf{P}(\ulcorner\gamma\urcorner)$ and $T \vdash \neg\mathsf{P}(\ulcorner\gamma\urcorner)$ making T inconsistent, contrary to hypothesis. ⊠

Hence $prov_T$ cannot be captured in T, although T – being p.r. adequate – can capture any p.r. property. So $prov_T$ isn't a p.r. property. In particular, our original $prov$ property for PA is not primitive recursive.

To say that $prov_T$ isn't a p.r. property is to say that there is no p.r. function of n which returns 0 if $prov_T(n)$ is true, and 1 otherwise. In other words, there is no p.r. function which decides what is a theorem of T.

Now in fact, we can actually show that a consistent, p.r. adequate theory T isn't decidable *at all*, whether by a p.r. computation or by any other kind of algorithmic procedure. But we aren't yet in a position to establish this stronger claim, because we haven't yet got a general theory of algorithmic procedures.[5] Later we'll see how to beef up the argument in this section in order to get the stronger result.

12.7 Tarski's Theorem

Suppose L is any interpreted language, and suppose that we have some system for Gödel-numbering expressions of L. Then there is a corresponding numerical property $true_L$ such that $true_L(n)$ if and only if n is the g.n. of a true L-sentence.

Now, suppose we are dealing with the language L_A (the language of basic arithmetic), which can express numerical properties. Then a natural question arises: is there an L_A-wff $\mathsf{T}(\mathsf{x})$ which *expresses* the numerical property $true_{L_A}$?

Well suppose there *is*. Suppose that, for all φ, $\mathsf{T}(\ulcorner\varphi\urcorner)$ is true if and only if $true_{L_A}(\ulcorner\varphi\urcorner)$, i.e. if and only if φ is true. In other words, suppose that each instance of

1. $\mathsf{T}(\ulcorner\varphi\urcorner) \equiv \varphi$

is true. Now, since Q satisfies the conditions of the Diagonalization Lemma, then – applying the Lemma to the negation of $\mathsf{T}(\mathsf{x})$ – we know that there must be some L_A wff L such that $\mathsf{Q} \vdash \mathsf{L} \equiv \neg\mathsf{T}(\ulcorner\mathsf{L}\urcorner)$.[6] But everything Q proves is true (since Q's axioms are of course true), so

2. $\mathsf{L} \equiv \neg\mathsf{T}(\ulcorner\mathsf{L}\urcorner)$

would then be true. But as a special case of (1), we have

3. $\mathsf{T}(\ulcorner\mathsf{L}\urcorner) \equiv \mathsf{L}$

is also true. But (2) and (3) immediately lead to $\mathsf{L} \equiv \neg\mathsf{L}$ and hence contradiction. So our supposition that (1) is always true has to be rejected. Hence no predicate of L_A can express the numerical property $true_{L_A}$.

And now the argument evidently generalizes. Take any language L rich enough for us to be able to formulate in L something equivalent to the very elementary arithmetical theory Q (that's so we can use the Diagonal Lemma again): call that an adequate arithmetical language. Then by the same argument, we have

> **Theorem 18** *No predicate of an adequate arithmetical language L can express the numerical property* true_L *(i.e. the property of numbering a truth of L).*

[5] Though compare our informal Theorem 3 back in Section 4.2, which shows that, for any consistent sufficiently strong T, $prov_T$ can't be an algorithmically decidable property of numbers.

[6] 'L' is for Liar!

In short: while you can express *syntactic* properties of a sufficiently rich formal theory of arithmetic (like provability) inside the theory itself via Gödel numbering, you can't express some key *semantic* properties (like arithmetical truth) inside the theory.[7] This is *Tarski's Indefinability Theorem*.[8]

We'll return to issues about truth later. But for the moment, just note that we have the following situation. Take Q as an example of a sound, p.r. adequate theory, then (1) there are some numerical properties that Q can capture (the p.r. ones for a start); (2) there are some properties that Q can express but not capture (for example $prov_{\mathsf{Q}}$); and (3) there are some properties that Q's language cannot even express (for example $true_{\mathsf{Q}}$).

It is not, we should hasten to add, that the property $true_{\mathsf{Q}}$ is mysteriously ineffable, and escapes all formal treatment: a richer theory with an extended language may perfectly well be able to capture $true_{\mathsf{Q}}$. But the point remains that, however rich a given theory of arithmetic is, there will be limitations, not only on what numerical properties it can capture but even on which numerical properties that particular theory's language can express.

[7] An observation which of course gives us incompleteness again: truth isn't provability in some fixed formal theory T – so assuming that T is sound and everything provable in it is true, this means that there must be truths which it can't prove.

[8] Named in honour of Alfred Tarski who proved this in his (1933); though Gödel again had already noted the result, e.g. in a letter to Zermelo written in October, 1931 (Gödel, 2003, pp. 423–429).

13 The Second Incompleteness Theorem

In this chapter, we continue our review of the swathe of results that follow once we've got the First Theorem – and the Diagonalization Lemma that underlies it – in place. In particular, we (at last!) introduce Gödel's *Second* Incompleteness Theorem.

13.1 Formalizing the First Theorem

Six initial points (the first four are definitions which will either be reminders or news, depending on how much of the last chapter you skipped):

1. We define $prov(n)$ to be true just when n is the g.n. of a PA theorem, i.e. just when $\exists m\ prfseq(m, n)$.[1]

2. We use $\ulcorner\varphi\urcorner$ in our logicians' English to denote φ's g.n.: so $prov(\ulcorner\varphi\urcorner)$ just when $\mathsf{PA} \vdash \varphi$.[2]

3. We use $\mathsf{Prf(x, y)}$ as an abbreviation for the wff of PA's language that captures the relation $prfseq$. And we define $\mathsf{Prov(y)} =_{\text{def}} \exists \mathsf{v}\mathsf{Prf(v, y)}$. Hence $\mathsf{Prov(y)}$ expresses the numerical property $prov$.[3]

4. We also use $\ulcorner\varphi\urcorner$ *inside a formal language* as a place-holder for the standard numeral for φ's g.n.: thus $\mathsf{Prov}(\ulcorner\varphi\urcorner)$ is true if and only if $\mathsf{PA} \vdash \varphi$.

5. PA's Axiom 1 immediately entails the wff $\mathsf{0} \neq \bar{\mathsf{1}}$; so if PA also proved $\mathsf{0} = \bar{\mathsf{1}}$, it would be inconsistent. Contraposing, if PA is consistent, it can't prove $\mathsf{0} = \bar{\mathsf{1}}$. On the other hand, if PA is inconsistent, we can derive anything, and hence derive $\mathsf{0} = \bar{\mathsf{1}}$ in particular. So the statement that PA is consistent is equivalent to the claim that it *can't* prove $\mathsf{0} \neq \bar{\mathsf{1}}$. But that's equivalent to saying it *isn't* the case that $prov(\ulcorner \mathsf{0} = \bar{\mathsf{1}} \urcorner)$.

6. Finally, the consistency claim that it is *isn't* the case that $prov(\ulcorner \mathsf{0} = \bar{\mathsf{1}} \urcorner)$ can be expressed inside PA by the wff $\neg\mathsf{Prov}(\ulcorner \mathsf{0} = \bar{\mathsf{1}} \urcorner)$. So that motivates the following new definition:[4]

   $$\mathsf{Con} =_{\text{def}} \neg\mathsf{Prov}(\ulcorner \mathsf{0} = \bar{\mathsf{1}} \urcorner)$$

[1] See the end of Section 10.6, and the end of Section 12.3, (v).

[2] For a little more explanation, see Section 12.2.

[3] See Section 12.3, (v) again.

[4] If your preferred system of first-order logic for PA has the absurdity constant '$\bot$', then you could instead put $\mathsf{Con} =_{\text{def}} \neg\mathsf{Prov}(\ulcorner\bot\urcorner)$.

Now to put these definitions to work. In Section 11.5 we proved Gödel's First Theorem, i.e. showed that

A. If PA is consistent then G is not provable in PA.

But the claim (A) is straightforwardly equivalent to

B. If it's not the case that $prov(\ulcorner 0 = \overline{1} \urcorner)$, then it's not the case that $prov(\ulcorner \mathsf{G} \urcorner)$.

And this numerical proposition (B) can now be expressed *inside* PA by the formal wff

C. $\mathsf{Con} \rightarrow \neg\mathsf{Prov}(\ulcorner \mathsf{G} \urcorner)$

So far, however, this doesn't really tell us anything new. The exciting claim is the next one: *the formal wff* (C) *is itself a* PA *theorem.* In other words, the content of the First Theorem is not merely *expressible* in PA but is actually *provable* in PA too.

Gödel, in his classic paper, invites us to observe the following:

> All notions defined or statements proved [in establishing the First Theorem] are also expressible or provable in P. For throughout, we have used only the methods of definition and proof that are customary in classical mathematics, as they are formalizable in P. (Gödel, 1931, p. 193)

Here, however, the system P is Gödel's simplified version of Russell and Whitehead's theory of types, which is much richer than first-order PA (see Section 8.1). It perhaps isn't so obvious that all the reasoning needed in the proof of the First Theorem can be formally reflected in our weaker system. To check that we can indeed formalize the proof inside PA requires a lot of work. But let's suppose the hard work has been done. Then we will have arrived at the *Formalized First Theorem*, i.e. established (FFT):

$$\mathsf{PA} \vdash \mathsf{Con} \rightarrow \neg\mathsf{Prov}(\ulcorner \mathsf{G} \urcorner)$$

13.2 The Second Incompleteness Theorem

Assuming the Formalized First Theorem, suppose now (for reductio) that

$$\mathsf{PA} \vdash \mathsf{Con}$$

Then, from (FFT) by modus ponens, we'd have

$$\mathsf{PA} \vdash \neg\mathsf{Prov}(\ulcorner \mathsf{G} \urcorner)$$

But we know that

$$\mathsf{PA} \vdash \mathsf{G} \equiv \neg\mathsf{Prov}(\ulcorner \mathsf{G} \urcorner)$$

(that's Theorem 14, Section 12.4). So it would follow that

$$\mathsf{PA} \vdash \mathsf{G}$$

contradicting the First Theorem. Therefore our supposition must be false, unless PA is inconsistent. Hence, assuming (FFT), we have

> **Theorem 19** *If* PA *is consistent, the wff* Con *which expresses its consistency is not provable in* PA.

Which is Gödel's *Second Incompleteness Theorem* for PA. Assuming that the axioms of PA are true on the standard interpretation and hence that PA is consistent (Section 5.7), Con is another true-but-unprovable wff.

And like the First Theorem, the result generalizes. Here's one version:

> **Theorem 20** *If T is a p.r. axiomatized theory which includes the axioms of* PA*, then the wff* $\mathsf{Con_T}$ *which expresses its consistency is not provable in T.*

This holds because including the axioms of PA is enough to for T to be able to prove the Formalized First Theorem for T (we'll say a bit more about this at the end of Section 13.5). And then – by the same argument as above – the Second Theorem will follow.

13.3 *Con* and ω-incompleteness

How surprising is the Second Theorem?

We know that – once we have introduced Gödel numbering – lots of arithmetical truths coding facts about provability-in-PA *can* themselves be proved in PA. So, initially, it is at least on the cards that Con should turn out to be provable too.

Now, assuming PA is consistent, $\mathsf{0} = \bar{\mathsf{1}}$ can't be a theorem: so no number is the super g.n. of a proof of it – i.e. for all n, it isn't the case that $prfseq(n, \ulcorner \mathsf{0} = \bar{\mathsf{1}})$. Since Prf captures *prfseq*, we therefore have – still assuming consistency –

(i) for any n, $\mathsf{PA} \vdash \neg\mathsf{Prf}(\bar{\mathsf{n}}, \ulcorner \mathsf{0} = \bar{\mathsf{1}} \urcorner)$

If we could prove Con, then (unpacking the abbreviation) we'd have

(ii) $\mathsf{PA} \vdash \forall \mathsf{v} \neg\mathsf{Prf}(\mathsf{v}, \ulcorner \mathsf{0} = \bar{\mathsf{1}} \urcorner)$

The unprovability of Con means, however, that we can't get from (i) to (ii) – *which is another example of the failure of ω-completeness.*[5]

In so far as the ω-incompleteness of PA was originally unexpected, the unprovability of Con in particular might be deemed to have been unexpected too. But now we know that examples of ω-incompleteness are endemic in p.r. axiomatized, p.r. adequate theories, Theorem 19 is perhaps no surprise.

[5] See Section 11.7: indeed, since Con is of Goldbach type, in the sense of Section 11.4, this is another example of the failure of ω-completeness for wffs of Goldbach type.

13.4 The significance of the Second Theorem

You might think: 'Suppose that for some reason we hold that there is a genuine question about the consistency of PA. Still, the fact that we can't derive Con is no new evidence *against* consistency, since the First Theorem tells us that lots of true claims about provability are underivable. While if, *per impossibile*, we could have given a PA proof of the consistency of the theory, that wouldn't really have given us special evidence *for* its consistency – for we could simply reflect that if PA were inconsistent we'd be able to derive anything from the axioms, and so we'd still be able to derive Con! Hence the derivability or otherwise of PA's consistency *inside* PA *itself* isn't especially interesting.'

But in fact, on reflection, the Theorem *does* tell us two evidently important and interesting things. It tells us (1) that *even* PA isn't enough to derive the consistency of PA so we certainly can't derive the consistency of PA using a *weaker* theory. And it tells us (2) that PA isn't enough to derive the consistency of *even* PA so we certainly can't use PA to demonstrate the consistency of *stronger* theories. We'll comment on those points in turn.

(1) Here's an immediate corollary of our Theorem:

> **Theorem 19*** *If T is a consistent sub-theory of* PA *then the wff* Con *which expresses the consistency of* PA *is not provable in T.*

A sub-theory of PA is any theory which shares some or all of the language of PA, shares some or all of the axioms, and shares some or all of the standard first-order deductive system.

Evidently, if the sub-theory T doesn't have all the language of basic arithmetic, then it won't even be able to frame the PA-wff we've abbreviated Con; so it certainly won't be able to prove it. The more interesting case is where T is a theory which does share the language of PA but doesn't have all the induction axioms and/or uses a weaker deductive system than classical first-order logic. Such a theory T can't prove *more* than PA. So, by Theorem 19, assuming it is consistent, T can't prove Con either.

Recall our brief discussion in Section 5.7, where we first raised the issue of PA's consistency. We noted that arguing for consistency by appealing to the existence of an apparently coherent intended interpretation might be thought to be risky (the appeal is potentially vulnerable to the discovery that our intuitions are deceptive and that there is a lurking incoherence in the interpretation). So the question naturally arises whether we can give a demonstration of PA's consistency that depends on something supposedly more secure. And once we've got the idea of coding up facts about provability using Gödel numbering, we might perhaps wonder whether we could, so to speak, lever ourselves up to establishing PA's consistency by assuming the truth of some weaker and supposedly less problematic arithmetic.

Let's take an example. Suppose we are tempted by radically *constructivist*

reservations about classical mathematics,[6] and think e.g. that we get into problematic territory once we start considering quantified propositions $\exists x P(x)$ or $\forall x P(x)$ when P isn't a decidable property of numbers. Then we might feel more comfortable starting with an arithmetic CA which drops the classical law of excluded middle and uses only intuitionistic logic, and which only allows instances of the Induction Schema which use open wffs $\varphi(\mathsf{x})$ which express *decidable* properties of numbers. But could we then use the constructively 'safe' arithmetic CA to prove that classical PA is at least consistent? Our Theorem shows that we can't.

(2) Here's another corollary of our Theorem:

> **Theorem 19**** *If T is a stronger theory which includes* PA *as a sub-theory, then the wff* $\mathsf{Con_T}$ *which expresses the consistency of T is not provable in* PA.

That's because, if PA could establish the consistency of the stronger theory T, it would thereby establish the consistency of PA as part of that theory, contrary to the Second Theorem.

We'll later be returning to consider the significance of this corollary at some length: it matters crucially for the assessment of 'Hilbert's Programme', about which much more anon. But the basic point is this. Suppose that we are faced with some classical 'infinitary' mathematical theory T into which arithmetic can be embedded as some some small part. For example T might, at the extreme, be the standard set theories ZF or ZFC with their staggeringly huge ontology of sets.[7] We may reasonably enough have anxieties about the very consistency of T, let alone its veracity. But Hilbert's hope is that we should be able to prove at least the consistency of T by using some modest and relatively uncontentious reasoning.[8] Given the idea of coding up facts about provability using Gödel numbering, we might try to make this hope come true by showing that T is consistent using relatively uncontentious arithmetical reasoning in (say) PA, applying it to provability facts about T. Theorem 19** shows that this can't be done either.

13.5 The Hilbert-Bernays-Löb derivability conditions

The rest of this chapter concerns technicalities: you can skip them (though some of the details are rather intriguing, especially Löb's Theorem in Section 13.7). We'll break this section into three parts.

(i) We haven't yet *proved* the Second Incompleteness Theorem, for we haven't proved the Formalized First Theorem (FFT):

$$\mathsf{PA} \vdash \mathsf{Con} \rightarrow \neg\mathsf{Prov}(\ulcorner \mathsf{G} \urcorner)$$

[6] As in, for example, (Bishop and Bridges, 1985, ch. 1).
[7] For a provocative response to that level of ontological commitment, see (Boolos, 1997).
[8] See, for example, his classic paper Hilbert (1925).

And Gödel himself didn't fill in the details to warrant his corresponding claim that $P \vdash \mathsf{Con}_P \to \neg\mathsf{Prov}_P(\ulcorner \mathsf{G}_P \urcorner)$, where G_P is the Gödel sentence for his original system of arithmetic P, Con_P is the consistency statement for that system, and so on. David Hilbert and Paul Bernays first gave a full proof in their *Grundlagen der Mathematik*: as we noted, this takes a lot of (rather tedious!) work, which we certainly won't reproduce here.[9]

However it is worth bringing out a key feature of the Hilbert and Bernays proof. As before (in Section 12.5), we use $\mathsf{Prov_T}$ as the provability predicate for whatever formal system T is in question – so n satisfies $\mathsf{Prov_T}(\mathsf{x})$ just in case there is the wff with g.n. n is provable in T. And Hilbert and Bernays isolated three conditions on the predicate $\mathsf{Prov_T}$, conditions whose satisfaction is enough for T to prove (FFT), so the Second Incompleteness Theorem applies to T. In his (1955), Martin H. Löb gave a rather neater version of these conditions. Here they are in Löb's version:

C1. if $T \vdash \varphi$, then $T \vdash \mathsf{Prov_T}(\ulcorner \varphi \urcorner)$

C2. $T \vdash \mathsf{Prov_T}(\ulcorner \varphi \to \psi \urcorner) \to (\mathsf{Prov_T}(\ulcorner \varphi \urcorner) \to \mathsf{Prov_T}(\ulcorner \psi \urcorner))$

C3. $T \vdash \mathsf{Prov_T}(\ulcorner \varphi \urcorner) \to \mathsf{Prov_T}(\ulcorner \mathsf{Prov_T}(\ulcorner \varphi \urcorner) \urcorner)$

Condition (C1) says that T can 'reflect' about its own proofs in the following minimal sense – if T can prove φ, then it 'knows' that φ is provable. Condition (C2) says that T can also 'reflect' on its own statements about provability, and can establish that if $\varphi \to \psi$ and φ are both provable, then so is ψ. Condition (C3) again says that T can 'reflect' on its own provability claims, and show that if φ is provable, then it is provable that it is provable.

(ii) We'll next show that (for any T strong enough to prove the Diagonalization Lemma), conditions (C) do indeed together entail the corresponding version of (FFT). In fact, we'll do just a bit better than that, and we'll actually prove (E)

$$T \vdash \mathsf{Con_T} \equiv \neg\mathsf{Prov_T}(\ulcorner \mathsf{G_T} \urcorner)$$

which strengthens (FFT)'s one-way conditional to a biconditional. $\mathsf{G_T}$ is of course the Gödel sentence for T constructed by following the same route we used in constructing the Gödel sentence for PA; and $\mathsf{Con_T} =_{\text{def}} \mathsf{Prov_T}(\ulcorner \mathsf{0} = \bar{\mathsf{1}} \urcorner)$.

However, before we give the proof, two notational matters. First, for the rest of this chapter – to improve readability – we'll drop subscripts from $\mathsf{Prov_T}$, $\mathsf{G_T}$, etc., leaving it to context to make clear which formal system is in question. And

[9] Gödel's (1931) is entitled 'On formally undecidable propositions of *Principia Mathematica* and related systems I': the plan was for a part II, giving a detailed proof of (FFT) for his logical system P, but Gödel never wrote it. Hilbert and Bernays do the job – though for a rather different logical system – in their (1939). For a modern proof of (FFT) for PA itself, see Tourlakis (2003).

second, to improve readability further, we'll introduce some even snappier (and now conventional) notation: we'll henceforth abbreviate '$\mathsf{Prov}(\ulcorner\varphi\urcorner)$' by '$\Box\varphi$'.[10]

With this new notation available, Con (i.e. $\neg\mathsf{Prov}(\ulcorner 0 = \overline{1}\urcorner)$) is $\neg\Box\, 0 = \overline{1}$, and we can rewrite the Hilbert-Bernays-Löb (HBL) conditions as follows:

C1. if $T \vdash \varphi$, then $T \vdash \Box\varphi$

C2. $T \vdash \Box(\varphi \to \psi) \to (\Box\varphi \to \Box\psi)$

C3. $T \vdash \Box\varphi \to \Box\Box\varphi$

And in the same notation, the result to be proved from these conditions is (E):

$$T \vdash \mathsf{Con} \equiv \neg\Box\mathsf{G}$$

Proof Let's note that – given $T \vdash 0 \neq \overline{1}$ – simple logic shows that, for any wff φ, T proves both

$$0 = \overline{1} \to \varphi \qquad (*)$$
$$\neg\varphi \to (\varphi \to 0 = \overline{1})$$

Given the latter and (C1), this means that T proves

$$\Box(\neg\varphi \to (\varphi \to 0 = \overline{1}))$$

So given (C2) and using modus ponens, it follows that T also proves each instance of the following:

$$\Box\neg\varphi \;\to\; \Box(\varphi \to 0 = \overline{1}) \qquad (**)$$

Hence we can argue as follows (where these are truncated versions of arguments to be spelt out *inside* our formal theory T). We assume T is strong enough for us to show (1), e.g. because it contains Q and is therefore p.r. adequate:

1.	$(\mathsf{G} \equiv \neg\Box\mathsf{G})$	(See Thm. 14)
2.	$(\mathsf{G} \to \neg\Box\mathsf{G})$	From 1
3.	$\Box(\mathsf{G} \to \neg\Box\mathsf{G})$	From 2, given C1
4.	$\Box\mathsf{G} \to \Box\neg\Box\mathsf{G}$	From 3, given C2
5.	$\Box\neg\Box\mathsf{G} \;\to\; \Box(\Box\mathsf{G} \to 0 = \overline{1})$	Instance of $(**)$
6.	$\Box\mathsf{G} \to \Box(\Box\mathsf{G} \to 0 = \overline{1})$	From 4 and 5
7.	$\Box\mathsf{G} \to (\Box\Box\mathsf{G} \to \Box\, 0 = \overline{1})$	From 6, given C2
8.	$\Box\mathsf{G} \to \Box\Box\mathsf{G}$	Instance of C3
9.	$\Box\mathsf{G} \to \Box\, 0 = \overline{1}$	From 7 and 8
10.	$\neg\Box\, 0 = \overline{1} \to \neg\Box\mathsf{G}$	Contraposing
11.	$\mathsf{Con} \to \neg\Box\mathsf{G}$	Defn. of Con

[10]If you are familiar with modal logic, then you will recognize the standard symbol for the necessity operator: and the parallels and differences between 'φ is provable (in T)' and 'φ is necessarily true' are highly suggestive. These parallels and differences are the topic of 'provability logic', the subject of a contemporary classic, namely (Boolos, 1993).

And to get the reverse conditional, we argue

1.	$(0 = \overline{1} \to \mathsf{G})$	Instance of $(*)$
2.	$\Box(0 = \overline{1} \to \mathsf{G})$	From 1, given C1
3.	$\Box 0 = \overline{1} \ \to\ \Box\mathsf{G}$	From 2, given C2
4.	$\neg\Box\mathsf{G} \ \to\ \neg\Box 0 = \overline{1}$	Contraposing
5.	$\neg\Box\mathsf{G} \ \to\ \mathsf{Con}$	Defn. of Con

Which demonstrates (E). ⊠

So, assuming T proves the Diagonalization Lemma (and so $T \vdash \mathsf{G} \equiv \neg\Box\mathsf{G}$), and assuming its provability predicate satisfies the HBL derivability conditions, then T proves (E), and hence (FFT), and so we get the Second Incompleteness Theorem for T.[11]

(iii) Given the HBL conditions get us to the Second Theorem, the natural next question to ask is: what does it take to prove those conditions hold for a given theory T?

Well, it is easy to see that (C1) holds so long as T can capture the appropriate numerical relation *prfseq* for T – to show this, we just generalize the argument for (P1) in Section 12.5. However, (C2) and (C3) are significantly more troublesome to prove. We aren't going to go into details, except to make one observation.

Consider (C3), which is equivalent to

$$T \vdash \forall \mathsf{v}(\mathsf{Prf}(\mathsf{v}, \ulcorner\varphi\urcorner) \to \exists \mathsf{w}\, \mathsf{Prf}(\mathsf{w}, \ulcorner\mathsf{Prov}(\ulcorner\varphi\urcorner)\urcorner)$$

where Prf captures the relation *prfseq* for T. So to prove (C3), we need to show that a certain universally quantified expression is a T-theorem: and it turns out that to demonstrate this we need to use induction inside T. *Which means that* (C3) *isn't derivable in an induction-free system like* Q. Similarly for (C2). However, since induction *is* freely available in PA, the conditions (C2) and (C3) can (and do) obtain there. That's why our current statement of Theorem 20, the general version of Gödel's second Theorem, applies to theories that include PA.[12]

13.6 *G*, *Con*, and 'Gödel sentences'

The Diagonalization Lemma established (in our new notation) that there is a G such that

(D) $\mathsf{PA} \vdash \mathsf{G} \equiv \neg\Box\mathsf{G}$.

[11] We set aside for the present the question whether satisfying the HBL conditions are also necessary for the unprovability of consistency.

[12] In fact, to get (C2) and (C3), we just need to be able to use induction for predicates like Prf that express p.r relations – so we can in fact get versions of the Second Theorem which apply to theories in between Q and PA: but we won't pause over this question of fine tuning here.

And – assuming we can make good the claim that the HBL conditions hold for provability in PA – we've just established

(E) $\mathsf{PA} \vdash \mathsf{Con} \equiv \neg\Box\mathsf{G}$

So it follows trivially that

(F) $\mathsf{PA} \vdash \mathsf{Con} \equiv \mathsf{G}$

Now consider the following pair of outline arguments within PA:

1.	$(\mathsf{Con} \to \mathsf{G})$	Given F
2.	$\Box(\mathsf{Con} \to \mathsf{G})$	From 1, given C1
3.	$\Box\mathsf{Con} \to \Box\mathsf{G}$	From 2, given C2
4.	$\neg\Box\mathsf{G} \to \neg\Box\mathsf{Con}$	Contraposing
5.	$\mathsf{Con} \to \neg\Box\mathsf{Con}$	From 4 and E

1.	$(\mathsf{G} \to \mathsf{Con})$	From F
2.	$\Box(\mathsf{G} \to \mathsf{Con})$	From 1, given C1
3.	$\Box\mathsf{G} \to \Box\mathsf{Con}$	From 2, given C2
4.	$\neg\Box\mathsf{Con} \to \neg\Box\mathsf{G}$	Contraposing
5.	$\neg\Box\mathsf{Con} \to \mathsf{Con}$	From 4 and E

So we have

(K) $\mathsf{PA} \vdash \mathsf{Con} \equiv \neg\Box\mathsf{Con}$

In other words, we have $\mathsf{PA} \vdash \mathsf{Con} \equiv \neg\mathsf{Prov}(\ulcorner\mathsf{Con}\urcorner)$, so Con is also a 'fixed point' for $\neg\mathsf{Prov}$ (see Section 12.4 again for the notion of a fixed point).

Suppose we keep the phrase 'Gödel sentence' for a wff which is constructed in the standard way and (in a reasonable sense) 'says' it is unprovable. We have now established that there are fixed points for $\neg\mathsf{Prov}$ that are *not* Gödel sentences.

13.7 Löb's Theorem

(i) Suppose T is a sound theory of arithmetic with true axioms. Then if φ is a *provable* wff, then it is *true*. We can express this inside T by the wff $\Box\varphi \to \varphi$. Such a wff will always be true on interpretation, for any φ. Hence we'd like T to be able to prove all such wffs (since we'd like T to prove as many arithmetic truths as possible). The obvious next question is: can it?

This is answered by *Löb's Theorem:*

Theorem 21 *If '$\Box$' for T satisfies the Hilbert-Bernays-Löb derivability conditions, then if $T \vdash \Box\varphi \to \varphi$ then $T \vdash \varphi$.*

But, if T is sound, it isn't always the case that $T \vdash \varphi$ (indeed, it won't prove any false φ). So it isn't always the case that $T \vdash \Box\varphi \to \varphi$.[13]

(ii) We can prove Löb's Theorem theorem as a consequence of the Second Incompleteness Theorem.[14] But it is a lot more fun to proceed the other way around – i.e. to use the HBL derivability conditions to prove Löb's Theorem directly, and then re-establish the Second Incompleteness Theorem as a simple corollary.

Proof Assume that, for some particular φ,

1. $\Box\varphi \to \varphi$

is already a T-theorem. Now consider the wff $(\mathsf{Prov}(\mathsf{z}) \to \varphi)$. By the Diagonalization Lemma, for some wff γ, T proves $\gamma \equiv (\mathsf{Prov}(\ulcorner\gamma\urcorner) \to \varphi)$. So, in our new notation,

2. $\gamma \equiv (\Box\gamma \to \varphi)$

is also a theorem. Continuing to argue within T (imagine particular fillings for 'γ' and 'φ'), we have

3.	$\gamma \to (\Box\gamma \to \varphi)$	From 2
4.	$\Box(\gamma \to (\Box\gamma \to \varphi))$	From 3, given C1
5.	$\Box\gamma \to \Box(\Box\gamma \to \varphi)$	From 4, given C2
6.	$\Box\gamma \to (\Box\Box\gamma \to \Box\varphi)$	From 5, given C2
7.	$\Box\gamma \to \Box\Box\gamma$	Given C3
8.	$\Box\gamma \to \Box\varphi$	From 6 and 7
9.	$\Box\gamma \to \varphi$	From 1 and 8
10.	γ	From 2 and 9
11.	$\Box\gamma$	From 10, given C1
12.	φ	From 9 and 11

Hence, assuming $T \vdash \Box\varphi \to \varphi$, we can conclude $T \vdash \varphi$. But φ was an arbitrarily selected wff; so the result generalizes, giving us Löb's Theorem ⊠

[13] Löb's Theorem also settles a question asked by Henkin (1952). By the Diagonalization Lemma applied to the unnegated wff $\mathsf{Prov}(\mathsf{z})$, there is a sentence H such that $T \vdash \mathsf{H} \equiv \mathsf{Prov}(\ulcorner\mathsf{H}\urcorner)$ – this will be a sentence that 'says' that it is provable. Henkin asked: *is* it provable? Löb's Theorem shows that it is. For by hypothesis, $T \vdash \mathsf{Prov}(\ulcorner\mathsf{H}\urcorner) \to \mathsf{H}$, i.e. $T \vdash \Box\mathsf{H} \to \mathsf{H}$; and the Theorem tells us that this entails $T \vdash \mathsf{H}$.

[14] Here's the idea. Assume we are dealing with an arithmetic to which the Second Theorem applies. We'll suppose $T \vdash \mathsf{Prov_T}(\ulcorner\varphi\urcorner) \to \varphi$, and derive $T \vdash \varphi$. So make that supposition and now consider the theory T' you get by adding $\neg\varphi$ as an axiom to T. Then trivially $T' \vdash \neg\varphi$, and $T' \vdash \mathsf{Prov_T}(\ulcorner\varphi\urcorner) \to \varphi$, so $T' \vdash \neg\mathsf{Prov_T}(\ulcorner\varphi\urcorner)$. But to prove that φ is unprovable in T is to prove that adding $\neg\varphi$ to T doesn't lead to contradiction, i.e. is to prove that T' is consistent. So for T' to prove $\vdash \neg\mathsf{Prov_T}(\ulcorner\varphi\urcorner)$ is tantamount to proving its own consistency. But by the Second Incompleteness Theorem, T' can't prove its own consistency if it is itself consistent. So T' is inconsistent. So adding $\neg\varphi$ to T leads to inconsistency. So $T \vdash \varphi$.

(iii) Now, we noted at the very outset that the reasoning for Gödel's First Theorem has echoes of the Liar Paradox; the proof of Löb's theorem echoes another logical paradox.

For suppose we temporarily reinterpret the symbol '$\Box$' as expressing the truth-predicate, so we read $\Box S$ as *'S' is true.* And let φ express any proposition you like, e.g. *Santa Claus exists.* Then

$1'$. $\Box\varphi \to \varphi$

is a truism about truth. And thirdly, suppose that the sentence γ says: *if γ is true, then Santa Claus exists.* Then we have

$2'$. $\gamma \equiv (\Box\gamma \to \varphi)$

holding by definition.

From here on, we can argue *exactly* as before – though this time we are not working within the formal theory T, but are reasoning informally, appealing to conditions (C1) to (C3) reinterpreted as intuitive principles about truth, i.e.

C1$'$. If we can establish φ that establishes that φ is true.

C2$'$. If $\varphi \to \psi$ is true, then if φ is true, so is ψ.

C3$'$. If φ is true, then it is true that it is true.

Using these evidently sound principles we again arrive by the same route at

$12'$. φ

which is the conclusion that *Santa Claus exists.* So, from truisms about truth and a definitional equivalence like ($2'$) we can, it seems, conclude anything! This line of reasoning is nowadays usually known as 'Curry's Paradox' (after Haskell B. Curry who presented it in his (1942)) though close relations of it were certainly known to medieval logicians such as Albert of Saxony.

It isn't obvious what the best way is to block Curry's paradox (any more that it is obvious what the best way is to block the Liar). There is *something* fishy about allowing a sentence γ such that ($2'$) holds: but what exactly? Fortunately answering that question is not our business. We merely remark that Löb's Theorem, like Gödel's, is not a paradox but a limitative result, this time another result about a theory's ability to prove propositions about its own provability properties.

(iv) Finally, we quickly apply Löb's Theorem to re-derive the Second Incompleteness Theorem. Note that if $T \vdash \mathsf{Con}$ then, as an immediate consequence, $T \vdash \neg\mathsf{Con} \to 0 = \overline{1}$, i.e. $T \vdash \Box 0 = \overline{1} \to 0 = \overline{1}$ But then, by Löb's Theorem, we can conclude $T \vdash 0 = \overline{1}$. But for any relevant T, $T \vdash 0 \neq \overline{1}$. Hence, if $T \vdash \mathsf{Con}$, T is inconsistent.

Interlude

We have now given a proof – at least in detailed outline – of Gödel's First Incompleteness Theorem and explained the content of the Second Theorem. What next? Five topics fairly immediately suggest themselves.

(1) In the arguments of Chapters 3 and 4, we relied on informal notions of computability and decidability. We showed, for example, that any consistent, sufficiently strong, axiomatized formal theory of arithmetic is undecidable – i.e. no algorithmic procedure can be used to settle what's a theorem (see Theorem 3, Section 4.2). In our later presentation of Gödel's First Theorem and its implications, we proved that no consistent, p.r. adequate, p.r. axiomatized formal theory of arithmetic is decidable by a p.r. function – i.e. there is no p.r. function which returns the output 0 if the input n is the g.n. of a theorem, and returns 1 otherwise (see Section 12.6). But as we noted there is an interesting gap between the informal result and the technical one. For we now know that there are algorithmic procedures that aren't p.r.; so an obvious question is whether we can beef up the technical result to show that *no* algorithm (whether or not p.r.) can decide whether an arbitrary wff is the theorem of a p.r. adequate arithmetic.

Exploring that issue involves reconsidering the notion of an algorithmic procedure, and coming up with a formal account which – unlike our story about p.r. functions – covers the *whole* class of intuitively computable functions. We have already signalled that such an account is available (see Section 2.3, and recall our brief remarks about Turing's Thesis). We didn't need to invoke this account in proving Gödel's Theorems; but if we are to press on and prove further limitative results about algorithmic undecidability, then we need to deal with general notions of computation.

(2) It is familiar that standard deductive systems for first-order logic are sound and complete. That means, in particular, that $\mathsf{PA} \vdash \varphi$ if and only if every interpretation of PA's language which makes the axioms true makes φ true.[1] So if PA doesn't prove φ, there must in particular be some interpretation which makes the axioms of PA true and yet φ false. Since PA proves neither G nor $\neg$G (assuming consistency), it follows there must be a interpretation of PA's language which makes the axioms all true and makes the Gödel sentence G false and another which also makes the axioms true and makes G true. Here there must be variant interpretations which agree in making all the axioms of PA true, but which diverge enough to give different truth-values to the 'formally undecidable' sentences. Hence, the first-order theory PA *doesn't* pin down a unique type of

[1] Recall, the language of PA is L_A, i.e. the interpreted language $\langle \mathcal{L}_A, \mathcal{I}_A \rangle$ (Section 3.2): so we are talking here about spinning variations on the built-in interpretation $\mathcal{I}_A$.

interpretative structure as the only possible one which makes its axioms true.[2]

On the other hand, it's a familiar result – encountered early in a university mathematics course – that (what mathematicians usually call) Peano Arithmetic *does* pin down a single type of structure. The argument is essentially this. Take the property R of being either zero or a successor of zero. Zero has the property R. If any number is R, so is its successor. So by induction, every number is R. So the domain of interpretation for Peano Arithmetic must contain a zero, its successors, and nothing else. The axioms for addition and multiplication then completely define the addition and multiplication functions on this domain. So that completely fixes the structure of any interpretation for Peano Arithmetic.

How do we reconcile these results? By noting that our formal theory PA and the mathematicians' informal Peano Arithmetic are crucially different theories. The former is a *first-order* theory (i.e. involves only quantification over a domain of numbers). The latter is *second-order* (i.e. is presented as embodying the intuitive second-order induction principle which quantifies over all properties of numbers – see Section 5.5).

An obvious task, then, is to explore further the differences between the first-order and the – perhaps more natural – second-order versions of Peano Arithmetic; and we'll also want to consider some theories of arithmetic of intermediate strength. We'll need to reflect on the impact of the Gödelian limitative results on these various theories.

(3) We've found a whole flotilla of true wffs that are unprovable in PA (assuming it is sound and therefore consistent). There's the original Gödel sentence G. There's the Gödel sentence G′ that you construct by adding G as a new axiom and running the Gödel construction on the new theory PA + G. Then there are all the further sentences G″, G‴, ... that you get by repeating that procedure. We also know that Con is unprovable. And for every false φ, the true wff $\mathsf{Prov}(\ulcorner\varphi\urcorner) \to \varphi$ is unprovable too (by Löb's Theorem).

Now, these various wffs are (when everything is unpacked) just enormously convoluted sentences of basic arithmetic. But faced with the fully unpacked versions, they surely wouldn't strike us as being of intrinsic *arithmetic* interest. The wffs are interesting for a different reason – i.e. because, via a given Gödel-numbering scheme, they encode meta-arithmetical claims about provability-in-PA. We might naturally ask, however, whether there are other truths of arithmetic which can also be shown to be independent of PA, where these other truths *are* of more purely arithmetical interest to number theorists. (To put the question rather tendentiously: are there incompleteness results of concern to mathematicians, or is this just a game for logicians?)

(4) The First Incompleteness Theorem shows that there is a gap between truth and provability in PA (and between truth and provability in any other p.r. ade-

[2] In fact, we don't need Gödel's Theorems to show that: as we'll explain later, the Löwenheim-Skolem Theorem is already enough. But as we'll also explain, Gödelian incompleteness does give an extra bite to this result.

quate, p.r. axiomatized theory). Is this because truth is somehow ineffable, and we can't theorize about it?

Tarksi's theorem (Section 12.7) tells us, for example, that the notion of truth-in-PA can't be expressed in the arithmetical language of PA. But that *doesn't* mean that it is inexpressible. There is nothing to stop us enriching PA's language with a truth predicate, i.e. a predicate $\mathsf{T}(\mathsf{x})$ which is such that $\mathsf{T}(\overline{\mathsf{n}})$ just when the wff with g.n. n is a truth of PA. And now suppose that we lay down some appropriate collection of axioms – call them Tr – governing the use of this truth-predicate. Then we might wonder whether we might not be able to show that $\mathsf{PA} + \mathsf{Tr} \vdash \mathsf{T}(\ulcorner \mathsf{G} \urcorner)$. And since we'd expect any competent theory of the truth-predicate to be such that $\mathsf{Tr} \vdash \mathsf{T}(\ulcorner \mathsf{G} \urcorner) \equiv \mathsf{G}$, that would mean $\mathsf{PA} + \mathsf{Tr} \vdash \mathsf{G}$.

In sum, although the PA can't prove its Gödel-sentence G, perhaps PA *plus a suitable theory of truth* can prove G.[3] So an obvious question to ask is: does any kind of theory of truth enable us to prove this? An answer should throw some light on the issue of just what we are committing ourselves to when we say that the Gödel-sentence is indeed *true*-but-unprovable-in-PA.

(5) That takes us to questions about the broader implications of Gödel's Theorems. Recall the speculations in Section 1.2 to the effect that a rule-transcending cognitive grasp of the numbers is needed to explain our ability to recognize the truth of Gödel sentences, and – since we have such a rule-transcending cognitive ability – minds like ours can't be machines. What should we make of such ideas?

We have also mentioned in a fleeting way the Frege-Russell programme for putting arithmetic and classical analysis on a secure logical footing (see Sections 5.7 and 10): just how do Gödel's Theorems impact on that programme? Gödel himself was rather more concerned to exhibit difficulties for Hilbert's Programme, which we haven't so far only fleetingly mentioned. So something, finally, needs to be said about that.

That gives us, then, the following menu of topics for discussion:

1. Computability and decidability
2. First-order vs. second-order and other arithmetics
3. Arithmetically interesting claims unprovable in PA
4. Truth and Gödel's Theorems
5. Wider philosophical implications

Which will be more than enough to keep us going!

[3]Of course, PA + T can't prove its *own* new Gödel-sentence $\mathsf{G}_{\mathsf{PA}+\mathsf{T}}$!

14 μ-Recursive and Partial Recursive Functions

We have now sketched a proof of Gödel's First Incompleteness Theorem (Chapter 11), and explained the content of the Second Theorem (Chapter 13). Yet the ingredients used in our discussions so far have been very modest. We needed the basic idea of capturing properties and functions in a formal axiomatized theory of arithmetic like PA, the idea of a primitive recursive function, and the idea of coding up claims about relations between wffs into claims about relations between their code-numbers. And we needed to show that various numerical relations coding proof relations among PA wffs are p.r., and hence can be captured in PA. We then worked Gödelian wonders with these very limited ingredients, and we haven't yet had to make use of any of the more sophisticated tools from the mathematical logician's bag of tricks. Nor have we had to call upon a general theory of computable functions.

However, if we are to press on to get some sense of how the two Theorems fit into the wider mathematical landscape and discuss their broader implications, then we really do need to fill in some additional technical background. So in the next few chapters, we'll say some more about ideas of computability, and then we'll explain a few rather more advanced ideas from logical theory. We'll be concentrating on those topics which are eventually most relevant to developing our understanding of Gödel's Theorems; proofs will often be pretty sketchy.

14.1 Minimization and μ-recursive functions

We've so far explored one (very large) class of computable functions, namely the primitive recursive ones. But we've also proved, by a neat diagonal argument, that these aren't the only computable functions defined over the natural numbers (see Chapter 6, especially Section 6.6).[1]

The p.r. functions are those which can be defined using *composition* and *primitive recursion*, starting from the successor, zero, and identity functions. So the next question to ask is: what other mode(s) of definition should we throw into the mix in order to get a broader class of computable functions (hopefully, to get *all* of them)? And given our earlier discussions, it is obvious what our first candidate answer should be. As we explained in Section 6.6, p.r. functions are calculated using *bounded* computational procedures using 'for' loops (as we enter each loop, we put a limit in advance on how many iterations are required). However, we also count *unbounded* search procedures – implemented by 'do until'

[1] We won't keep repeating: we are concerned throughout this chapter with functions whose arguments and values are natural numbers.

loops – as computational. So, the obvious way to try extending the class of p.r. functions is by allowing functions to be defined by means of some sort of 'do until' procedure. We'll explain how to do this, in four steps.

(a) Let's start with the simplest case. Suppose we know that G is a decidable numerical property, and that it has at least one instance. Or, what comes to the same, suppose that G's characteristic function g is computable and, for some n, $g(n) = 0$. Then we can do a systematic search to find a number f which has the property G by running the following program:

1. $y := 0$
2. Do until $g(y) = 0$
3. $\quad y := y + 1$
4. Loop
5. $f := y$

Here, the memory register y is initially set to contain 0. We then enter a loop. At each iteration, we compute $g(y)$ to see if its value is zero; if it does, we exit the loop and put f equal to the current value of y; otherwise we increment y by one and do the next test. By hypothesis, we do eventually hit a value for y such that $g(y) = 0$: so the program is bound to terminate. This 'do until' program plainly calculates the number f which is *the least y such that $g(y) = 0$*.

(b) Now generalize this idea. Recall the notational device we introduced in Section 6.2, where we wrote $\vec{x}$ as short for the array of n variables $x_1, x_2, \ldots, x_n$. We'll say that

> The $(n + 1)$-place function $g(\vec{x}, y)$ is *regular* if it is a total function and for all values of $\vec{x}$, there is a y such that $g(\vec{x}, y) = 0$.

Then here's the key definition:

> Suppose $g(\vec{x}, y)$ is an $(n + 1)$-place regular function. Let $f(\vec{x})$ be the n-place function which, for each $\vec{x}$, takes as its value the least y such that $g(\vec{x}, y) = 0$. Then we say that f is defined by *regular minimization* from g.

(If we allow n to be zero so $\vec{x}$ is empty, our general definition covers our introductory case where f is a zero-place function, i.e. a constant!)

The crucial observation is this: if f is defined from the computable function g by regular minimization, then f is computable too. For we can set out to compute $f(\vec{x})$ for any particular arguments using the kind of simple 'do until' loop that we just illustrated. And our stipulation that g is regular ensures that a computation of the value of $f(\vec{x})$ terminates for each $\vec{x}$. In other words, f is a total computable function, defined for all arguments.

(c) We'll introduce some standard symbolism. When f is defined from g by minimization, we'll write

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

where the minimization operator 'μy' can be read 'the least y such that ...'.[2] Compare the operation of bounded minimization which we met in Section 6.4: we are now concerned with a species of *unbounded* minimization.

(d) Now we put these ideas together. We said that we can expect to expand the class of computable functions beyond the p.r. ones by considering functions that are computed using a 'do until' search procedure. We've just seen that when we define a function by regular minimization, this in effect specifies that its value is to be computed by just such a search procedure. Which suggests that a third mode of definition to throw into the mix for defining computable (total) functions, alongside composition and primitive recursion, is definition by regular minimization.

With that motivation, let's say:

> The *μ-recursive* functions are those that can defined from the initial functions by a chain of definitions by composition, primitive recursion and/or regular minimization.[3]

Or putting it more carefully, in the style of our official account of the p.r. functions in Section 6.2, we can say

1. The initial functions S, Z, and I_i^k are μ-recursive;
2. if f can be defined from the μ-recursive functions g and h by composition, then f is μ-recursive;
3. if f can be defined from the μ-recursive functions g and h by primitive recursion, then f is μ-recursive;
4. if g is a regular μ-recursive function, and f can be defined from g by regular minimization, then f is μ-recursive;
5. nothing else is a μ-recursive function.

Since regular minimization yields total functions, the μ-recursive functions – as we've just defined them – are always total computable functions.[4]

[2]We follow a fairly common convention in inserting the square brackets to promote readability, even though they are strictly speaking redundant.

[3]Most writers nowadays use plain 'recursive' instead of 'μ-recursive'. But for an important warning about the variable terminology hereabouts, see Section 14.5, footnote 12.

[4]Note that a 'for' loop – i.e. a programming structure which instructs us to iterate some process as a counter increments from 0 to n – can be recast as a 'do until' loops: just iterate the same process while incrementing the counter until its value equals n. So definitions by primitive recursion, which call 'for' loops, can be subsumed under definitions by minimization, which call 'do until' loops. Hence you might well expect that clause (4) in our definition is redundant. And you'd *almost* be right. By a theorem of Kleene's (1936b), if you add addition, multiplication and the characteristic function of the less-than relation to the list of initial functions, then indeed you can drop (4). And some books define recursive functions this way; see e.g. (Shoenfield, 1967, 109). Still, I don't find this approach as illuminating, so I stick to the more conventional mode of presentation as given here.

14.2 The Ackermann-Péter function

Every primitive recursive function is, of course, a μ-recursive function: but are there μ-recursive functions which aren't primitive recursive?

Yes. For a start, the computable-but-not-p.r. function $d(n)$ that we constructed by the diagonalization trick in Section 6.6 is an example. But in this section we'll look at another example, which is both more tractable and also mathematically more natural. The basic idea of the example is due to Wilhelm Ackermann (in his 1928).

First a simple observation. Any p.r. function f can be specified by a chain of definitions in terms of primitive recursions and compositions leading back to initial functions. This definition won't be unique: there will always be various ways of defining f (for a start, by throwing in unnecessary detours). But take the shortest definitional chain – or, if there are ties for first place, take one of the shortest. Now, the length of this shortest definitional chain for f will evidently put a limit on how fast $f(n)$ can grow as n grows (and likewise for two-place functions). That's because it puts a limit on how complicated the computation can be – in particular, the number of loops-within-loops-within-loops that we have to play with – and so limits the number of times we can get to apply the successor function depending on the initial input argument n.

For example, consider the two-place functions f_1, i.e. *sum* (repeated applications of the successor function), f_2, i.e. *product* (repeated sums), f_3, i.e. *exponentiation* (repeated products). These functions have increasingly long full definitional chains, and the full programs for computing them involve 'for' loops nested increasingly deeply. And as their respective arguments grow, the value of f_1 of course grows slowly, f_2 grows faster, f_3 faster still.

This sequence of functions can obviously be continued. Next comes f_4, the *super-exponential*, defined by repeated exponentiation:

$$x \uparrow 0 = x$$
$$x \uparrow Sy = x^{x \uparrow y}$$

Thus, for example, $3 \uparrow 4$ is $3^{3^{3^{3^{3}}}}$ with a 'tower' of 4 exponents. Similarly, we can define f_5 (super-duper-exponentiation, i.e. repeated super-exponentiation), f_6 (repeated super-duper-exponentiation), and so on. The full chain of definitions for each f_k gets longer and longer as k increases – and the values of the respective functions grow faster and faster as their arguments are increased.[5]

But now consider the function $a(x) = f_x(x, x)$. The value of $a(n)$ grows *explosively*, running away ever faster as n increases. So take any given p.r. function: this has a maximum rate of growth determined by the length of its definition, and $a(x)$ eventually grows faster. Hence $a(x)$ isn't primitive recursive: but it is evidently computable.

[5]NB The claim, of course, isn't that longer definitions always *entail* faster growth, only that our examples show how longer definitions *permit* faster growth.

This idea of Ackermann's is *very* neat, and is worth pausing over. So consider again the recursive definitions of our functions f_1 to f_4 (compare Section 6.1, and our definition of '$\uparrow$' above). We can rewrite the second clauses in each of those definitions as follows:

$$
\begin{aligned}
f_1(y, Sz) &= Sf_1(y, z) \\
&= f_0(y, f_1(y, z)) \text{ – if we cunningly define } f_0(y, z) = Sz \\
f_2(y, Sz) &= f_1(y, f_2(y, z)) \\
f_3(y, Sz) &= f_2(y, f_3(y, z)) \\
f_4(y, Sz) &= f_3(y, f_4(y, z))
\end{aligned}
$$

There's a pattern here! So now suppose we put

$$f(x, y, z) =_{\text{def}} f_x(y, z)$$

Then the value of f gets fixed via a *double* recursion:

$$f(Sx, y, Sz) = f(x, y, f(Sx, y, z))$$

However, nothing very exciting happens to the second variable, 'y'. So we'll let it drop out of the picture, to get a variant on the Ackermann's construction due to Rósza Péter (1935). Consider, then, the function p governed by the clause

$$p(Sx, Sz) = p(x, p(Sx, z))$$

Of course, this single clause doesn't yet fully define p – it doesn't tell us, e.g., the value of $p(0, 0)$. So we will round out the definition to yield the following three equations

$$
\begin{aligned}
p(0, z) &= Sz \\
p(Sx, 0) &= p(x, S0) \\
p(Sx, Sz) &= p(x, p(Sx, z))
\end{aligned}
$$

To see how these equations work together to determine the value of p for given arguments, consider the following calculation:

$$
\begin{aligned}
p(2, 1) &= p(1, p(2, 0)) \\
&= p(1, p(1, 1)) \\
&= p(1, p(0, p(1, 0))) \\
&= p(1, p(0, p(0, 1))) \\
&= p(1, p(0, 2)) \\
&= p(1, 3) \\
&= p(0, p(1, 2)) \\
&= p(0, p(0, p(1, 1))) \\
&= p(0, p(0, p(0, p(1, 0)))) \\
&= p(0, p(0, p(0, p(0, 1)))) \\
&= p(0, p(0, p(0, 2))) \\
&= p(0, p(0, 3)) \\
&= p(0, 4) \\
&= 5
\end{aligned}
$$

To evaluate the function, the recipe is as follows. At each step look at innermost occurrence of p, and apply whichever of the definitional clauses pertains (only one can); keep on going until at last you reach something of the form $p(0, m)$ and can apply the first clause one last time and halt. A little reflection will convince you that this procedure *does* always terminate (hint: look at the pattern of numbers in each vertical column, and also down diagonals). And note that our recipe involves a *do until* procedure: so it shouldn't be a surprise to hear that we have the following

> **Theorem 22** *The Ackermann-Péter function is μ-recursive but not primitive recursive.*

Sketch of a proof sketch: $p(x, z)$ is not p.r. We'll not pause long over this as we've already indicated the main proof idea. Given the function p's origin in the sequence of functions sum, product, exponential, ..., it is evident that the functions $p(0, z)$, $p(1, z)$, $p(2, z)$, ... grow faster and faster as z increases. And we can fairly easily confirm that for any primitive recursive function $f(z)$ there is some n such that $p(n, z)$ grows faster – i.e. after some threshold d, then for all $z > d$, $f(z) > p(n, z)$. Hence p can't be primitive recursive.

For suppose otherwise: then $q(z) = p(z, z)$ is p.r. too. Hence there is some n and some d such that, if $z > d$ and $z > n$, then $q(z) < p(n, z) < p(z, z) < q(z)$. Contradiction! ⊠

Proof sketch: $p(x, z)$ is μ-recursive We'll take longer over this result as the proof illustrates a very important general idea that we'll make use of in later chapters. The proof has three stages.

(i) *Introducing more coding* Consider the successive terms in our calculation of the value of $p(2, 1)$. We can introduce code numbers representing these terms by a simple, two-step, procedure:

1. Transform each term like $p(1, p(0, p(1, 0)))$ into a corresponding sequence of numbers like $\langle 1, 0, 1, 0\rangle$ by the simple expedient of deleting the brackets and occurrences of the function-symbol 'p'.
2. Code the resulting sequence $\langle 1, 0, 1, 0\rangle$ by Gödel numbering, e.g. by using powers of primes. So we put e.g.

   $$\langle l, m, n, o\rangle \Rightarrow 2^{l+1} \cdot 3^{m+1} \cdot 5^{n+1} \cdot 7^{o+1}$$

   (where we need the +1 in the exponents to handle the zeros).

Thus, step (1), we can think of our computation of $p(2, 1)$ as generating in turn the sequences

$$\langle 2, 1\rangle, \langle 1, 2, 0\rangle, \langle 1, 1, 1\rangle, \langle 1, 0, 1, 0\rangle, \ldots$$

Then, step (2), we code up each sequence; so the successive steps in the calculation of $p(2, 1)$ will respectively receive the code numbers

72, 540, 900, 2100, ...

(ii) *Coding/decoding functions* So far, that's just routine coding. Now we put it to work by defining a couple of coding/decoding functions as follows:

1. $c(x, y, z)$ is the code number of the sequence of numbers corresponding to the z-th term (counting from zero) in the calculation of $p(x, y)$ if the calculation hasn't yet halted by step z; and otherwise $c(x, y, z) = 0$. So, for example, $c(2, 1, 3) = 2100$; and $c(2, 1, 20) = 0$.

2. $fr(x)$ = one less than the exponent of 2 in the prime factorization of x. Hence, if n codes a sequence of numbers, $fr(n)$ recovers its first member.

(iii) *Facts about our coding functions* Next, some claims about our coding functions, which together establish the desired result that the Ackermann-Péter function is μ-recursive.

1. The coding function c is primitive recursive. That's a little tedious though not difficult to prove.[6] But the claim should in any case look entirely plausible, because the evaluation of c for given arguments involves a step-by-step computation using a simple rule that takes us from one step to the next. No step-to-step move involves flying off on an open-ended search. So a LOOP program will suffice to give an algorithm for the computation (compare Section 6.5), and such a program will always determine a p.r. function.

2. The calculation of the function p for given arguments x, y eventually halts at the z-th step for some z, and then $c(x, y, Sz) = 0$. Therefore g is regular.

3. $\mu z[c(x, y, Sz) = 0]$ is therefore the step-number of the step in the calculation which delivers the value of $p(x, y)$.

4. Hence $c(x, y, \mu z[c(x, y, Sz) = 0])$ is the code number of the final value of $p(x, y)$.

5. Hence, decoding, $p(x, y) = fr(c(x, y, \mu z[c(x, y, Sz) = 0]))$.

6. But the function fr is primitive recursive (in fact $fr(x) =_{\text{def}} exp(x, 0) \dot{-} 1$, where exp is as introduced in Section 6.4).

Hence, putting those facts together, since c and fr are p.r., and the minimization in our redefinition of $p(x, y)$ in (5) is regular, $p(x, y)$ is μ-recursive.[7] ⊠

[6] We won't do it here; but we *do* give details of how prove that a similar coding function is p.r. in Section 16.5.

[7] The proof that the computable-but-not-p.r. diagonal function $d(n)$ is μ-recursive is similar. Details can be found e.g. in Péter's classic book (1951, with a revised edition rather badly translated as her 1967). The key trick is to use double recursion again to define a function $\varphi(m, n)$ such that, for a given m, $\varphi(m, n) = f_m(n)$, where running through the f_i gives us the

14.3 Diagonalizing out?

So: there are computable functions which are μ-recursive but not primitive recursive. The obvious next question is: are there computable functions which aren't μ-recursive either?[8]

Familiarity with some programming languages may well suggest that the answer should be 'no'. For when things are reduced to basics, we see that the main programming structures available in such languages are (in effect) 'for' loops and 'do until' loops, which correspond to definitions by primitive recursion and minimization. So – given that our modern general-purpose programming languages have proved sufficient for specifying algorithms to generate any computable function we care to construct – it doesn't seem a big leap to conjecture that *every* computable function should be definable in terms of composition (corresponding to the chaining of program modules), primitive recursion, and minimization.

That rough and ready argument certainly gives some initial plausibility to the following Thesis: *a function is an intuitively computable total function just in case it is μ-recursive.* For reasons that will emerge in Chapter **??**, we can call this 'Church's Thesis' in honour of Alonzo Church.[9]

However, don't we already have materials for a knock-down argument against this Thesis? For in Section 6.6, we proved that not every computable function is primitive recursive by 'diagonalizing out' – that is to say, we used a diagonal construction which took us from a list of all the p.r. functions to a further computable function which isn't be on the list. Well, why shouldn't we now just use the same trick again?

The argument would go:

---

desired effective enumeration of the p.r. functions. Since φ is definable by a double recursion it is μ-recursive – by the same kind of argument which showed that that Ackermann-Péter function is μ-recursive. Hence $d(n) = \varphi(n, n) + 1$ is μ-recursive too.

By the way, it would be quite wrong to take away from our discussion above the impression that the μ-recursive-but-not-p.r. functions must all suffer from explosive growth. Péter gives a beautiful counter-example. Take our enumeration f_i of p.r. functions, and now consider the functions $g_i(n) =_{\text{def}} sg(f_i(n))$, where sg is as defined in Section 6.4 (i.e. $sg(n) = 0$ for $n = 0$, and $sg(n) = 1$ otherwise). Evidently, running through the g_i gives us an effective enumeration – with many repetitions of course – of all the p.r. functions which only take the values 0 and 1. Now consider the μ-recursive function $\psi(n) = |1 - sg(\varphi(n, n))|$. This function too only takes the values 0 and 1; but it can't be primitive recursive. For suppose otherwise. Then for some k, $\psi(n) = g_k(n) = sg(\varphi(k, n))$. So we'd have

$$sg(\varphi(k, n)) = \psi(n) = |1 - sg(\varphi(n, n))|$$

and hence

$$sg(\varphi(k, k)) = \psi(k) = |1 - sg(\varphi(k, k))|$$

Which is impossible. Therefore there are μ-recursive-but-not-p.r. functions which only ever take the values 0 and 1, and hence do not suffer value explosion!

[8] Note, we are still – for the moment – talking about computable *total* functions from $\mathbb{N}$ into $\mathbb{N}$.

[9] Compare 'Turing's Thesis' which we introduced in Section 2.3. Over the next chapters, it will become clear that these two Theses are in fact equivalent – but we aren't in a position to show that yet, so (for the moment) we must distinguish them.

> Take an effective enumeration of the μ-recursive functions, f_0, f_1, f_2, ..., and define the diagonal function $D(n) = f_n(n) + 1$. Then D differs from each f_j (at least for the argument j). But D is computable (since to evaluate it for argument n, you just set a computer to enumerate the f_j until it reaches the n-th one, and then by hypothesis the value of $f_n(n) + 1$ is computable). So D is computable but not μ-recursive.

But this argument fails, and it is very important to see why. The crucial point is that we are not entitled to its initial assumption. While the p.r. functions are effectively enumerable, *we have no reason to suppose that there is an effective enumeration of the μ-recursive functions* (and, it will transpire, every reason to suppose that there isn't one).

But what makes the critical difference between the cases? Well, remind yourself of the informal argument (in Section 6.6) that shows that the p.r. functions are effectively enumerable. If we now try to run a parallel argument argument for the claim that the μ-recursive functions are effectively enumerable, things go just fine at the outset:

> Every μ-recursive function has a 'recipe' in which it is defined by primitive recursion or composition or regular minimization from other functions which are defined by recursion or composition or regular minimization 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 ...

But at this point the parallel argument breaks down, since we *can't* continue

> ... and as we go, we can mechanically select the strings that obey the rules for being a recipe for a μ-recursive function.

To determine mechanically whether a series of definitions obey the rules for being the recipe for a μ-recursive function we would need to determine, in particular, whether each application of the minimization operator is a *regular* minimization. Take the first occurrence of the minimization operator in the chain of definitions for a function. Up to that point, the definitions are characterizing a p.r. function: so to tell whether this minimization operator is correctly applied to a regular function we must be able to tell whether a p.r. function is regular. Hence, in sum, in order to have an effective algorithm for telling whether a set of definitions obey the rules for being the recipe for a μ-recursive function, we'd need already to have an effective algorithm for telling whether a p.r. function given by a certain definition is regular. But is there such an algorithm?

We can put this question in another, equivalent, way. Suppose $f_0, f_1, f_2, \ldots$ is an effective enumeration of p.r. functions; and suppose we define the test

function *reg*, where $reg(n) = 0$ if f_n is regular and $reg(n) = 1$ otherwise (thus *reg* is the characteristic function of the property of numbering a regular function in the enumeration). There is an algorithm for testing regularity just in case *reg* is a computable function. But is it?

Well, as we'll see in Section **??**, we can in fact conclusively prove at least that *reg* isn't a *μ-recursive* function. So this means that in order to use the diagonalization argument against Church's Thesis, we must first show that *reg* is computable even though not μ-recursive. But there isn't a smidgen of evidence for that – and of course, if we *could* show that the *reg* is computable even though not μ-recursive, this would already give us a direct counterexample against Church's Thesis, and we wouldn't need to use the diagonal argument!

Perhaps we should summarize, as the dialectical situation here is a bit delicate. We asked: can we argue against the Thesis that every intuitively computable (total) function is μ-recursive by using a diagonal argument modelled on our arguments against the claim that every computable function is primitive recursive? We saw that such a diagonal argument would only be available if the function *reg* is computable in the intuitive sense. But we also claimed – and will prove – that that function is not μ-recursive. So we can only use the diagonal argument against Church's Thesis if we *already* have demonstrated that *reg* is a counterexample to the Thesis. In short, we can't use the diagonal argument to get any independent purchase against Church's Thesis where we hadn't produced a counterexample before.

The Thesis lives to fight another day.

14.4 Partial computable functions

The use of 'do until' loops is potentially dangerous. Suppose we set out to iterate some loop until condition C obtains, but in fact C never obtains. Then our computation gets stuck in an infinite cycle and never terminates.

Likewise – indeed, it is the same point – the use of unbounded minimization is potentially dangerous. Suppose we set out to search for the least y such that $g(\vec{x}, y) = 0$, but g isn't regular. Then, for some arguments $\vec{x}$, our search may not terminate.

Our insistence so far that we use only regular minimization was precisely to guarantee that we wouldn't ever get stuck in a never-ending loop. Still, you might say,

> the intuitive notion [of a computation] does not require that a mechanical procedure should always terminate or succeed. A sometimes unsuccessful procedure, if sharply defined, is still a procedure, i.e. a well determined manner of proceeding.[10]

[10] The quotation is from Hao Wang (1974, p. 84), who is reporting Gödel's views in conversation. The earliest discussions of computable functions did, however, concentrate on total functions: partial computable functions first step into the limelight in Kleene (1938).

Suppose then that we relax the notion of a definition by minimization to allow us (roughly) to define $f(\vec{x})$ as the least y such that $g(\vec{x}, y) = 0$, even though g isn't regular. Then $f(\vec{x})$ may be undefined for some values, yet be defined *and* computable by a 'do until' procedure for some other values. So f, in short, could be a *partial* computable function.

A general account of computability ought, it seems, to cover such partial computable functions. In the rest of this section, we gather together some preliminary definitions. Then in next section we extend the notion of minimization to allow for partial recursive functions. There are some subtleties.

Here then are the initial definitions that we'll need: we first give them for one-place functions:

1. f is a *partial* numerical function – in standard symbols, we write $f\colon \mathbb{N} \rightharpoondown \mathbb{N}$ – if f is defined for arguments just in some domain $\Delta \subseteq \mathbb{N}$, and for each $n \in \Delta$, it returns a value $f(n) \in \mathbb{N}$. Note that we allow the case where $\Delta = \mathbb{N}$, so f is in fact total, i.e. defined for all numbers n. But ...

2. f is *strictly* partial if it is partial but not total, i.e. is undefined for some $n \in \mathbb{N}$.

3. f is the same partial function as g if f and g are defined for the same domain Δ, and for all $n \in \Delta$, $f(n) = g(n)$.

4. f^c is the completion of f – where f is a partial function with domain $\Delta \subseteq \mathbb{N}$ – if, for all $n \in \Delta$, $f^c(n) = f(n)$, and for all $n \notin \Delta$, $f^c(n) = 0$.[11] So f^c is a total function which – so to speak – fills in the gaps left by places where f is undefined by setting the value of the function to zero in those cases.

The generalizations to many-place functions are immediate. For example,

1′. f is a two-place partial numerical function – in standard symbols, we write $f\colon \mathbb{N}^2 \rightharpoondown \mathbb{N}$ – if there is some domain Δ which is a subset of $\mathbb{N}^2$, the set of ordered pairs of natural numbers, and for each pair $\langle m, n\rangle \in \Delta$, f is defined and returns a value $f(m, n) \in \mathbb{N}$.

Finally, we have the following key definition (which again we give first for the case of a one-place function):

5. The *partial* function f is *computable* if there is some algorithm which delivers the output $f(n)$ for each n in f's domain Δ, and which delivers no output otherwise.

In other words, a partial function f is computable if there is an algorithmic procedure which we can follow to calculate the right value whenever f is defined, and which doesn't properly terminate when f isn't defined. Again, the generalization to many-place functions is obvious.

[11] '$n \notin \Delta$' is of course the negation of '$n \in \Delta$'.

14.5 Partial recursive functions

We are now going to move on from the intuitive notion of a partial computable function to introduce the formally defined notion of a partial recursive function. To do this, we need to liberalize the use of minimalization as follows:

> Suppose $g(\vec{x}, y)$ is an $(n+1)$-place partial function. Then we say that f is defined by *minimization* from g just in case
> either (i) $f(\vec{x})$ is the least y such that: $g(\vec{x}, y) = 0$ and for all $z < y$, $g(\vec{x}, z)$ is defined and non-zero;
> or else (ii) if such a y doesn't exists, $f(\vec{x})$ is undefined.

Again, we allow the case where $n = 0$, so f is just a constant.

A definition by regular minimization is, a fortiori, also a definition by minimization in this broader sense. Henceforth, then, we'll extend our use of the 'μ' notation so that we write $f(\vec{x}) = \mu y[g(\vec{x}, y) = 0]$ when f is defined by minimization from g, whether g is regular or not.

Why don't we just say that $f'(\vec{x})$ is defined by minimization from $g(\vec{x}, y)$ if either (i$'$) it yields the least y such that $g(\vec{x}, y) = 0$ if that exists, or else (ii$'$) is undefined? Well, to take the simplest example, suppose g is a partial computable function such that $g(0)$ is undefined but $g(1) = 0$. Then $f' = 1$ (the least y such that $g(y) = 0$). *But we can't compute f' by any search procedure*: that's because, if when we put $y = 0$ and try to test whether $g(y) = 0$, we get stuck since $g(0)$ is undefined. The moral is: if we want minimization applied to a computable function g to give us another computable function f, we must ensure that, when $f(\vec{x})$ is defined, we can indeed find its value by repeatedly testing whether $g(\vec{x}, y) = 0$, incrementing y until we hit a value which passes the test *without getting stuck en route*. That's precisely what we ensure by the two-pronged clause (i).

And now here is the key definition (the natural one, given what's gone before):

> The *partial recursive* functions are those that can defined from the initial functions by a chain of definitions by composition, primitive recursion and/or minimization.

Or putting it more carefully in the now familiar style, we can say

1. The initial functions S, Z, and I^k_i are partial recursive;
2. if f can be defined from the recursive functions g and h by composition, then f is partial recursive;
3. if f can be defined from the recursive functions g and h by primitive recursion, then f is partial recursive;
4. if g is a partial recursive function, and f can be defined from g by minimization, then f is partial recursive;
5. nothing else is a partial recursive function.

Since, in particular, a function defined by minimization from a (partial) computable function is itself partial computable, the partial recursive functions are all partial computable.

Four more comments:

(i) The point should be stressed again: the so-called partial recursive functions needn't be *strictly* partial. For a start, all p.r. functions also count as partial recursive functions.

(ii) In addition, the μ-recursive functions (obtained when minimization is applied to regular functions) are also total partial recursive functions. That's immediate from our definitions. And the converse also holds: the total partial recursive functions are all μ-recursive. *However, that isn't trivial, it's a theorem!* – in fact it's Theorem 25, Section 16.3.[12]

Why isn't it trivial? Well, suppose that $g(n)$ is a *total* partial recursive function whose values all belong to the set Δ. And suppose that $h(n)$ is a strictly partial recursive function defined just for values $n \in \Delta$ (and, let's imagine, h is given to us by a definition that involves the application of minimization to a non-regular function). Then (a) the function $f(n) = h(g(n))$ is partial recursive (since it is a composition of two partial recursive functions). But (b) f is in fact a total function (since h is defined for every output from g). However, (c) the total partial recursive function f, defined as the composition of g and h, is not given to us by a definition that involves only regular minimizations. So (d), if f is to be μ-recursive by our definition in Section 14.1, then we need to prove that it can always be given some *other* definition which reveals it to be μ-recursive. And that is certainly non-trivial.

(iii) You might think, all the same, that the issue of partial functions is of no great interest, speculating that we could always in principle tidy things up by replacing a partial function f with e.g. its completion f^c. But not so: a function's partiality often can't be eliminated while retaining computability. Or more precisely – as we'll show later – there can be strictly partial recursive functions whose completion is not μ-recursive.

(iv) As we noted before, a partial recursive function f is evidently partial computable. But what about the converse claim?

Corresponding to the Church's Thesis for total functions, we have the Thesis for partial functions: *computable partial functions are partial recursive functions.*

[12]This is the point, perhaps, for an *important warning about terminology*, to aid comparison with other treatments. So note that a majority of writers use 'recursive function' (plain and simple) to mean μ-recursive function: see e.g. Cutland (1980, pp. 49–50), Cohen (1987, pp. 60–61), or Mendelson (1997, p. 175). But others use 'recursive function' to mean 'total partial recursive function': see for example Cooper (2004, p. 18). That's why we've stuck to our unambiguous terminology.

Note also a more superficial difference in terminology that might still confuse the unwary browser glancing at other books: some writers – e.g. (Cooper, 2004) again – use the abbreviation 'p.r. function' to mean not *primitive* recursive, but *partial* recursive.

We'll argue for this generalized Thesis over the coming chapters. But for the moment, we make just one observation.

Note that – in contrast to the μ-recursive functions – recipes for *partial* recursive functions *are* effectively enumerable. The argument is as before in the case of p.r. functions. We can choose some standard formal specification language for representing chains of definitions in terms of composition, recursion and minimization. We can effectively enumerate the possible strings of symbols from this language and, as we do so, we can effectively select out the strings that obey the rules for being a recipe for a partial recursive function (because this time, we don't need to decide whether minimization is applied to a regular function). But if partial recursive functions can be specified in some effectively enumerable sequence $f_0, f_1, f_2, \ldots$, why can't we continue the argument as we did for the primitive recursive functions and 'diagonalize out'?

Well, how is the diagonal function supposed to be constructed? If, as before, we put $D(n) = f_n(n) + 1$, then D is indeed a partial computable function, but we can't prove that D is a new function, not already on our list. Perhaps it is none other than f_k, where $f_k(k)$ is undefined, so $D(k)$ is also undefined, which allows D and f_k to be the same function, agreeing on the values for which they *are* defined.

If, alternatively, we put $D(n) = f_n(n) + 1$ if $f_n(n)$ is defined, and otherwise put $D(n) = 0$ (for example), then D must indeed be distinct from each of the f_i, but now – by observation (iii) – we don't know that D is computable.

So either way, we can't diagonalize out of the class of partial recursive functions.

In sum, the situation is this. The double-pronged Church's Thesis is that the intuitively computable total functions and the intuitively computable partial functions are respectively just the μ-recursive and partial recursive functions. In each case, there is a potential challenge: can't we diagonalize out to construct a computable but not recursive function? And in each case, we have a blocking response – but the lines of the responses are different in the two cases.

15 Turing Machines

Our overall project is to get clear about the content and implications of Gödel's Theorems: and, as far ideas of computability are concerned, the concepts of primitive recursive, μ-recursive, and partial recursive functions are all we need for the job. However, having reached this point, it is well worth saying something about Alan Turing's classic analysis of computability,[1] and then establishing the crucial result that the Turing computable total functions are exactly the μ-recursive functions. This result is interesting in its own right, is historically important, and enables us later to establish some further results about recursiveness in a particularly neat way. Further, it warrants us running together Turing's Thesis (Section 2.3: the computable functions are those computable by a Turing machine) with Church's Thesis (Section 14.3: the computable functions are just the μ-recursive ones). These pay-offs make the next three chapters worth the effort.

15.1 The basic conception

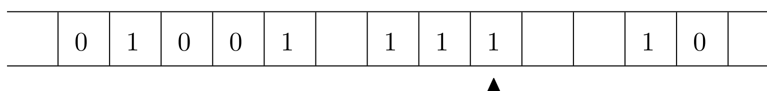
Think of executing an algorithmic computation 'by hand', using pen and paper. We follow strict rules for writing down symbols in various patterns. To keep things tidy, let's write the symbols neatly one-by-one in the squares of some suitable square-ruled paper. Eventually – if all goes well, and we don't find ourselves spiralling around in a circle, or marching off to infinity – the computation halts and the result of the computation is left written down in some block of squares on the paper.

Now, using a two-dimensional grid for writing down the computation is not of the essence. Imagine cutting up the paper into horizontal strips a square deep, and pasting these together into one long tape. We could use that as an equivalent work-space.

Using a rich repertoire of symbols is not of the essence either. Suppose some computational system uses 27 different symbols. Number these off using a five-binary-digit code (so the 14th symbol, for example, gets the code '01110'). Then divide each of the original squares on our work-space tape into a row of 5 small cells. Instead of writing one of original symbols into one of the original big squares, we could just as well write its binary code digit-by-digit into a block of 5 cells.

So – admittedly at some cost in user-friendliness – we can think of our original hand computation as essentially equivalent to following an algorithm (a set of

[1] See his (1936), which is handily reprinted with a very useful long introduction in (Copeland, 2004).



instructions) for doing a computation by writing down or erasing binary digits one-at-a-time in the cells on a linear tape, as in our diagram. The arrow-head indicates the position of the *scanned cell*, i.e. the one we are examining as we are about to apply the next computational instruction. (We'll assume, by the way, that we can paste on more work-space as and when we need it – so, in effect, the tape is unlimited in length in both directions).

Let's now consider the instructions governing our computation. We will list them as labelled or numbered lines, where each line tells us what to do depending on the contents of the scanned cell. So we can think of a single line from our algorithm as having the form

> q: if the scanned cell contains '0', do action A^0, then go to q^0;
> if the scanned cell contains '1', do action A^1, then go to q^1;
> if the scanned cell is blank, do action A^B, then go to q^B.

where q and the q^j are line-numbers or labels (and perhaps one or two of the conditional clauses are redundant).

It is convenient to distinguish two special labels, 1 and 0: the first will be used to mark the initial line of a program, while the second is really a pseudo-label and 'go to 0' will actually mean 'end the execution of the program'. Note, by the way, that program execution can stop for two reasons: we can get that explicit instruction to stop, or we can run out of program lines to tell us what to do next. In the first case, we'll say that the execution *halts*; in the second case, we'll say that it *freezes*.

What are the possible actions A^j here? There are two types. We can write in the scanned cell – i.e. over-write any contents in the scanned cell with '0' or '1', or 'write a blank' (i.e. erase the current contents). Or else we can move along the tape so that a new cell becomes the scanned cell. We'll take it that any moving is to be done step-wise, one cell at a time; and more complex actions – like e.g. copying the contents of the scanned cell into the next four cells to the right – are to be performed as a sequence of basic actions. There are five possible minimal actions A, which we can indicate as follows:

> 0: write a '0' in the scanned cell (overwriting any scanned content);
> 1: write a '1';
> B: write a blank;
> L: make the next cell to the left the scanned cell;
> R: make the next cell to the right the scanned cell.

Let's now say, as a first shot, that a *Turing program* is just a collection of program lines of the very simple three-clause form we illustrated, which tell us which action to perform and which line to jump to next.

In executing such a program, we typically start with some number or numbers written as *input* on the work-tape (written in binary digits, of course). And we begin by following the relevant instruction given at the program line labelled 1. We then follow the further instructions we encounter as we are told to jump from line to line. If and when the execution halts because we get to the instruction 'go to 0', we look at the block of binary digits containing the scanned cell, and those digits give the numerical *output* of our calculation.

So we can say, e.g., that a one-place function f is computed by a Turing program Π if, given n on the tape as input, executing the program eventually yields $f(n)$ as output just in case $f(n)$ is defined. In the general case, we'd expect a Turing program to compute a *partial* function. Indeed, it will often be the null function, which is nowhere defined: but even if program execution halts properly for some inputs it may freeze or go into an infinite loop for other inputs.

The idea of a Turing computation, then, is extraordinarily simple – it's basically a matter of running a program whose sole programming structure is the 'go to line q' instruction. In Section 15.3, we'll give some mini-examples of Turing programs in operation. But first we really need to refine the ideas we've just sketched. By the end of the next section, then, we will have introduced some sharper terminology and also cleaned up some details. Along the way, there's a number of fairly arbitrary and non-significant choices to be made. We won't comment on these, but don't be surprised to find other choices made in other treatments: the basic conception, however, always stays much the same. (Suggestion: it might help to read the next two sections in tandem, using theory and practice to illuminate each other.)

15.2 Turing computation defined more carefully

(i) *I-quadruples* We define an *instruction-quadruple* (or 'i-quadruple' for short) to be an ordered quadruple of the form

$$\langle q_1, S, A, q_2 \rangle$$

whose elements are as follows:

1. q_1 is a numeral other than '0'; we'll refer to this first element of an i-quadruple as its *label* – to emphasize the point that the numerals here aren't doing arithmetical work. An i-quadruple labelled '1' is an *initial* quadruple.
2. S – representing the contents of the scanned cell – is one of the symbols $\{0, 1, B\}$. 'B', of course, represents a blank cell.
3. A is one of the symbols $\{0, 1, B, L, R\}$: these represent the five possible minimal actions.
4. q_2 is a numeral.

An i-quadruple is to be read as giving a labelled, conditional, two-part instruction as follows:

q_1: *if* the scanned cell contains S, *do* the action indicated by A, then *go to* the instructions with label q_2 – unless q_2 is '0', in which case halt the program.

Evidently, we can now compress our verbose instruction line

q: if the scanned cell contains '0', do action A^0, then go to q^0;
if the scanned cell contains '1', do action A^1, then go to q^1;
if the scanned cell is blank, do action A^B, then go to q^B.

into three i-quadruples which share the same initial label, thus:

$$\langle q, 0, A^0, q^0\rangle,\ \langle q, 1, A^1, q^1\rangle,\ \langle q, B, A^B, q^B\rangle$$

(ii) *Turing programs* We will say that a set Π of i-quadruples is *consistent* if it doesn't contain i-quadruples with the same label which issue inconsistent instructions. More formally:

A set Π of i-quadruples is consistent if there's no pair of i-quadruples $\langle q_1, S, A, q_2\rangle$, $\langle q_1, S, A', q_2'\rangle$ in Π such that $A \neq A'$ or $q_2 \neq q_2'$.

Which leads to the following sharpened official definition:

A *Turing program* is a finite consistent set of i-quadruples.[2]

(iii) *Executing Turing programs* We execute a Turing program Π as follows:

1. We start with the work-tape in some *initial configuration* – i.e. with digits occupying a *finite* number of cells, and with some particular cell being scanned. Suppose the content of that initial scanned cell is S.

2. We then look for some appropriate initial quadruple to execute. That is to say, we look for an i-quadruple in Π of the form $\langle 1, S, A, q_2\rangle$: by consistency there is at most one distinct such i-quadruple. We perform action A and jump to the instructions with label q_2.

[2]Some presentations say that Turing programs are sets of *i-quintuples*. The idea is that we can wrap up e.g. the pair of instructions

$\langle 10, 1, 0, 11\rangle$
$\langle 11, 0, R, 12\rangle$

into a single instruction of the form

$\langle 10, 1, 0, R, 12\rangle$

For consider: executing the two quadruples involves first replacing the digit '1' in the scanned cell by '0' and jumping to instruction 11, and secondly – since we now have '0' in the scanned cell – moving on to scan the next cell to the right and then jumping to instruction 12. That has the same effect as executing the more complex single instruction '10: if there is '1' in the scanned cell, then replace it with '0', move right, and jump to instruction 12', which is summarized by our quintuple. There is really nothing to chose between presenting Turing programs as sets of quadruples or sets of quintuples other than aesthetics.

3. We next look for an i-quadruple in Π of the form $\langle q_2, S', A', q_3\rangle$, where S' is the content of the currently scanned cell. We perform action A' and jump to the instructions with label q_3. And we keep on going ...

4. ... unless and until either (a) the execution *halts* because we are explicitly told to halt – i.e. we encounter a 'jump to 0' instruction – or (b) the execution *freezes* because we have simply run out of instructions we can apply.

We will be particularly interested in cases where we run a program starting and finishing in what we'll call *standard* modes. To explain:

a. We start with the work-tape in a *standard initial configuration* if the tape is blank except for containing as input one or more blocks of binary digits, with the blocks separated by single blank cells, and the initial scanned cell is the left-most cell of the left-most block.

b. A run of the program is said to *halt gracefully* if (a) it halts properly (rather than freezes), and (b) it leaves the work-tape cleaned up and blank apart from a single block of binary digits, with (c) the scanned cell being the left-most cell of this block.

(iv) *Turing computable functions* Suppose that $f\colon \mathbb{N} \rightharpoondown \mathbb{N}$, i.e. f is a one-place partial function, which is defined for numbers in some domain Δ (where Δ is either $\mathbb{N}$, the set of natural numbers, or some proper subset of $\mathbb{N}$). Then,

> The Turing program Π computes the function f just if, for all n, when Π is executed starting with the tape in a standard initial configuration with n in binary digits on the tape (and nothing else), then (i) if $n \in \Delta$, the execution halts gracefully with $f(n)$ in binary digits on the tape, and (ii) if $n \notin \Delta$, the execution never halts.

Now suppose that $g\colon \mathbb{N}^2 \rightharpoondown \mathbb{N}$, i.e. g is a two-place partial function, which is defined for pairs of numbers in some domain Δ (where Δ is either $\mathbb{N}^2$, the set of all ordered pairs of natural numbers, or some proper subset of $\mathbb{N}^2$). Then,

> The Turing program Π computes the function g just if, for all m, n, when Π is executed starting with the tape in a standard initial configuration with m in binary digits on the tape, followed by a blank cell, followed by n in binary digits (and nothing else), then (i) if $\langle m, n\rangle \in \Delta$, the execution halts in standard configuration with $f(m, n)$ in binary digits on the tape; and (ii) if $\langle m, n\rangle \notin \Delta$ the execution never halts.

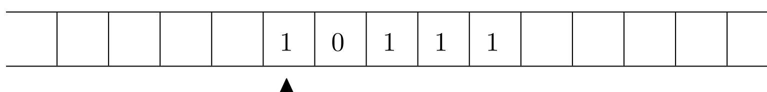
The generalization to many-place functions is obvious. And we say

> A function is *Turing computable* if there is a Turing program that computes it.

(v) *Not all programs compute functions* Finally, note that we've said that a program Π is to count as computing some monadic function (say) when, for any input n, Π either halts gracefully or never halts. Suppose however that Π sometimes halts but not gracefully. Then by our definition, Π doesn't compute a function.

15.3 Some simple examples

(i) *The successor function* We'll now construct a little Turing program that computes the successor function. So suppose the initial configuration of the tape is as follows

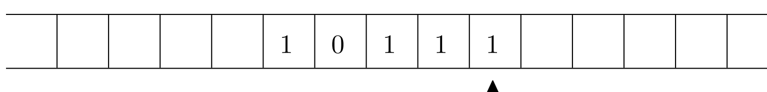


Then the program needs to deliver the result '11000'. The program task can be broken down into three stages:

Stage A We need to make the scanned cell the last cell in the initial block of digits. Executing the following i-quadruples does the trick:

$\langle 1, 0, R, 1\rangle$
$\langle 1, 1, R, 1\rangle$
$\langle 1, B, L, 2\rangle$

These initial instructions move the scanned cell to the right until we overshoot and hit the blank at the end of the initial block of digits; then we shift back one cell, and look for the instructions with label '2':

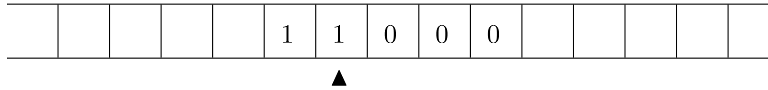


Stage B Now for the core computation of the successor function. So if the scanned cell contains '0' (or is blank), we change it to '1', and then go to the final Stage C. If the scanned cell contains '1', we change that to '0', and 'carry 1' – i.e. we move to the next cell to the left, and repeat Stage B:

$\langle 2, 0, 1, 4\rangle$
$\langle 2, 1, 0, 3\rangle$
$\langle 3, 0, L, 2\rangle$
$\langle 2, B, 1, 4\rangle$

Note that we only jump to a quadruple with label '3' if we've just written '0' in the scanned cell, so we only need the quadruple that starts $\langle 3, 0, \ , \ \rangle$. The fourth

quadruple is to deal with the case where we keep on ‘carrying 1’ until we hit the blank at the front of the initial block of digits. Executing these instructions gets the tape in our example into the following state:

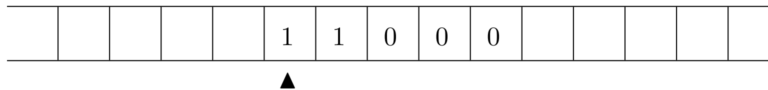


And we now look for instructions with label ‘4’.

Stage C Finishing up. We need to ensure that the scanned cell returns to be at the front of the block, so that the computation halts in standard configuration. Analogously to Stage A, we can write:

$\langle 4, 0, L, 4\rangle$
$\langle 4, 1, L, 4\rangle$
$\langle 4, B, R, 0\rangle$

Following these instructions, the scanned cell moves leftwards until it overshoots the block and moves right one cell, and then the execution halts gracefully.



The scanned cell is now the first in the block of digits. There is nothing else on the tape. So we have halted gracefully.

Thus our i-quadruples together give us a program which computes the successor function.

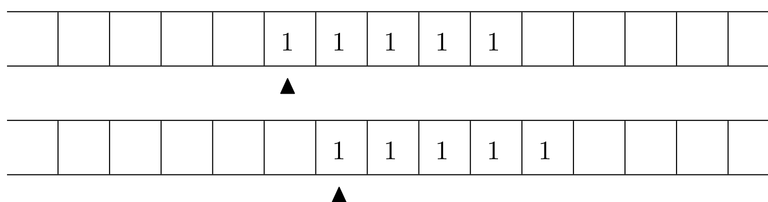
(ii) *Another program for the successor function* We’ll give two more examples of Turing programs in this section. First, note that our successor program, applied to ‘111’, changes those digits to ‘000’, then prefixes a ‘1’ to give the correct output ‘1000’. So in this case, the output block of digits starts *one cell to the left* of the position of the original input block. We’ll now – for future use – describe a variant of the successor program, which this time always neatly yields an output block of digits starting in exactly the *same* cell as the original input block.

What we need to do, clearly, is to prefix a routine at the beginning of the program which, if but only if it detects an unbroken block of ‘1’s, shifts that block one cell to the right (by adding a ‘1’ at the end, and deleting a ‘1’ at the beginning). The following will do the trick, and also – like Stage A of our previous program – it moves the current cell to the end of the block:

$\langle 1, 0, R, 20\rangle$
$\langle 1, 1, R, 1\rangle$
$\langle 1, B, 1, 10\rangle$
$\langle 10, 1, L, 10\rangle$

$\langle 10, B, R, 11\rangle$
$\langle 11, 1, B, 11\rangle$
$\langle 11, B, R, 20\rangle$
$\langle 20, 0, R, 20\rangle$
$\langle 20, 1, R, 20\rangle$
$\langle 20, B, L, 2\rangle$

The initial instructions start us off scanning through the input block to the right. If we encounter a '0' then we continue by following the instructions labelled '20' (which take the scanned cell to the end of the block). If all we meet are '1's then we put another '1' at the end of the block, and then follow the instructions labelled in the tens, which first take us back to the beginning of the block, and then get us to delete the initial '1' and move one cell right. You can check that executing the program up to this point changes the state of the tape from the first to the second state illustrated here:



We then just follow the instructions labelled '20' again, so we end up scanning the cell at the end of the block (and now looking for instructions labelled '2').

Since we've used new labels in these revised i-quadruples, we can now add on the same Stage B i-quadruples from our previous successor program to perform the task of adding one, and the same Stage C i-quadruples to get the scanned cell back to the beginning of the resulting block. Putting all those together gives us the desired program.

Phew! That's a lot of effort — but then, programming at the level of 'machine code' (which is in effect what we are doing) *is* hard work. That's why we ordinarily use high-level programming languages and rely on compilers that work behind the scenes to translate our perspicuous and manageable programs into the necessary instructions for manipulating binary bits.

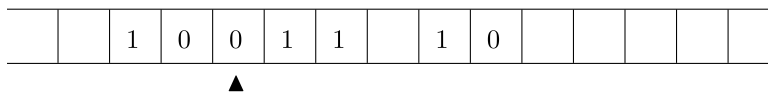
(iii) Our remaining example is a simple copying program which takes a block of input digits, and produces as output the same block (in the same place), followed by a blank, followed by a duplicate of the original block.

We need somehow to keep track of where we are in the copying process. We can do this by successively deleting a digit in the original block, going to the new block and writing a copy of the deleted digit, returning to the 'hole' we made to mark our place, replacing the deleted digit, and then moving on to copy the next digit.

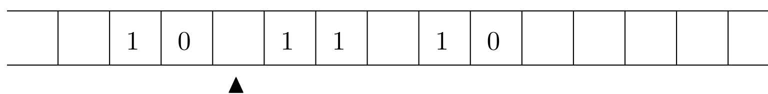
Our program for doing all this can be broken down into four sub-programs:

1. *Choosing what to do* Examine the scanned cell. If it contains a '0', delete it, and go to sub-program (2). If it contains a '1', delete it, and go to sub-program (3). If it is blank, then we've got to the end of the digits in the original block, so we just need to call sub-program (4) to ensure that we halt gracefully.
2. *Copying a '0'* This routine 'remembers' that we've just deleted a '0'. We scan on to the right until we find the *second* blank – which marks the end of our duplicate block (if and when it exists) – and write a '0'. Then we scan back leftwards until we again find the second blank (the blank created when we deleted the '0'). Rewrite a '0' there, which finishes copying that digit. So now move on to scan the next cell to the right, and return to sub-program (1).
3. *Copying a '1'* Just like sub-program (2), except that this routine 'remembers' that we've just deleted a '1'.
4. *Finish up* Move the scanned cell back to the beginning.

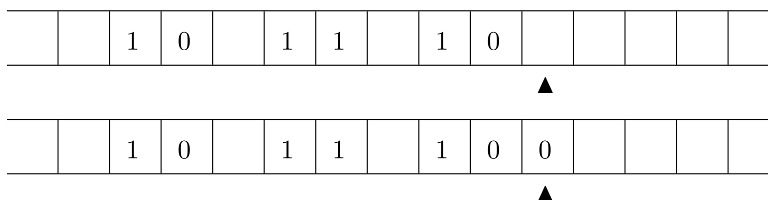
To illustrate, suppose the current state of the tape is like this:

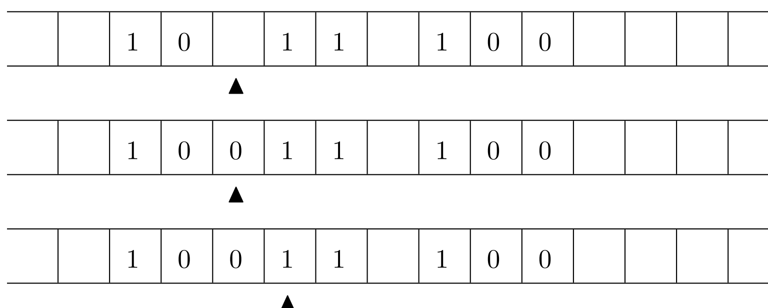


Sub-program (1) instructs us to delete the currently scanned digit and start executing sub-program (2):



Sub-program (2) then takes us through the following stages: (a) we scan to the right to find the end of the second block), (b) we write '0' there, (c) we scan back to the left to find the 'hole' in the first block that we just created, (d) we rewrite '0' there, (e) we move one cell right. So these five stages produce these successive states of the tape:





And we've thereby copied another digit.

Here – just for the fun of it – is how to code up our outlined program strategy into i-quadruples (we've marked the beginnings of the four sub-programs):

$\langle 1, 0, B, 10\rangle$ (1)
$\langle 1, 1, B, 20\rangle$
$\langle 1, B, L, 30\rangle$

$\langle 10, 0, R, 10\rangle$ (2)
$\langle 10, 1, R, 10\rangle$
$\langle 10, B, R, 11\rangle$
$\langle 11, 0, R, 11\rangle$
$\langle 11, 1, R, 11\rangle$
$\langle 11, B, 0, 12\rangle$
$\langle 12, 0, L, 12\rangle$
$\langle 12, 1, L, 12\rangle$
$\langle 12, B, L, 13\rangle$
$\langle 13, 0, L, 13\rangle$
$\langle 13, 1, L, 13\rangle$
$\langle 13, B, 0, 14\rangle$
$\langle 14, 0, R, 1\rangle$

$\langle 20, 0, R, 20\rangle$ (3)
$\langle 20, 1, R, 20\rangle$
$\langle 20, B, R, 21\rangle$
$\langle 21, 0, R, 21\rangle$
$\langle 21, 1, R, 21\rangle$
$\langle 21, B, 1, 22\rangle$
$\langle 22, 0, L, 22\rangle$
$\langle 22, 1, L, 22\rangle$
$\langle 22, B, L, 23\rangle$
$\langle 23, 0, L, 23\rangle$
$\langle 23, 1, L, 23\rangle$
$\langle 23, B, 1, 24\rangle$
$\langle 24, 0, R, 1\rangle$

$\langle 30, 0, L, 30\rangle$ (4)
$\langle 30, 1, L, 30\rangle$
$\langle 30, B, R, 0\rangle$

Which all just reinforces the point that writing Turing programs for performing even simple tasks very quickly becomes *very* painful. So we won't give any more detailed examples.[3] After all, our concern here isn't with practical computing but rather with Turing's analysis of what a computation, at bottom, consists in. If he is right, then any genuinely algorithmic step-by-step computation can be replicated using a Turing program (compare 'Turing's Thesis' as we stated it in Section 2.3).

[3] Masochists can try their hands at programming one of the many on-line Turing machine simulators which are available: but it is a game that quickly palls!

15.4 'Turing machines' and their 'states'

A final remark. We have so far imagined a human 'computer' executing a Turing program 'by hand', writing and erasing symbols from a paper tape, mechanically following the program's instructions. Evidently, a machine could be programmed to do the same job. And any mechanism for running a Turing program might naturally be referred to as a 'Turing machine' (at least if we pretend that its 'tape' is inexhaustible).

But the theory of Turing computations doesn't care about the hardware implementation. What matters about a Turing machine is always its program. Hence one standard practice is to think of a Turing machine as an idealized computer individuated by the Turing program it is running (i.e. same program, same Turing machine: different program, different Turing machine). Another, equally standard, practice is simply to *identify* a Turing machine with its program (it is common enough to read 'A Turing Machine is a set of quadruples such that . . . '). Nothing at all hangs on this. When we occasionally talk of Turing machines we'll in fact be thinking of them in the first way. But mostly – for clarity's sake – we'll keep talking about programs rather than machines.

Suppose that a Turing machine (in our sense) is in the middle of executing a program. The state-of-play is that it is about to execute some i-quadruple in its program, while scanning a particular cell on a tape which has some configuration of cells filled with digits. We can think of this overall state-of-play as characterized by the 'internal' state of the machine (it is about to execute a quadruple with label q) combined with the 'external' state of the tape (the configuration of the tape, with one cell picked out as the 'current' cell). That's why q labels are standardly said to identify internal *states* of the Turing machine. Again, it is fine to talk this way for vividness: but don't read too much into it.

16 Turing Computability and Recursiveness

We are not going to write any more programs to show, case by case, that this or that particular function is Turing computable, not just because it gets painfully tedious, but because we can easily establish that *every* partial recursive function is Turing computable and, conversely, *every* Turing computable function is partial recursive. (It follows from our proofs that the μ-recursive functions are also just the total Turing computable functions.) This equivalence between our two different characterizations of computable functions is of key importance, and we'll be discussing its significance in Chapter 17. In the present chapter, we outline proof-strategies for both halves of our result, and note a corollary of our proofs: we then – mostly for enthusiasts – indicate how to fill in the outlines.

16.1 Partial recursiveness entails Turing computability

Every μ-recursive function can be evaluated 'by hand', using pen and paper, prescinding from issues about the *size* of the computation. But we have tried to build into the idea of a Turing computation the essentials of any hand-computation. So we should certainly hope and expect that μ-recursive functions will turn out always to be Turing computable. And indeed – as just announced – we can prove something stronger:

Theorem 23 *Every partial recursive function is Turing computable*

Since every μ-recursive function is a partial recursive function – see Section 14.5, (ii) – it immediately follows that the μ-recursive functions are Turing computable.

Proof outline We'll say that a Turing program is *dextral* (i.e. 'on the right') if

a. in executing the program, we never have to write in any cell to the *left* of the initial scanned cell (or scan any cell more than one to the left of that initial cell);
b. if and when the program halts gracefully, the final scanned cell is the *same* cell as the initial scanned cell (in other words, the input block(s) of digits at the beginning of a computation and final the output block start in the same cell);

The key point about a dextral program, then, is that we can run it while storing data safely on a leftwards portion of the tape, because the program doesn't touch that portion. So complicated computations can proceed by running a

series of dextral sub-programs using leftwards portions of the tape to preserve data between sub-programs.

If a function is computable by a dextral Turing program, we'll say it is *d-Turing computable*. Suppose now that the following four propositions are all true:

1. The initial functions are d-Turing computable.
2. If the functions g and h are d-Turing computable, then so is a function f defined by composition from g and h.
3. If the functions g and h are d-Turing computable, then so is a function f defined by primitive recursion from g and h.
4. If the function g is d-Turing computable, so is the function f defined by minimization.

Then take any partial recursive function f. This can be defined by some chain of definitions by composition and/or primitive recursion and/or minimization, beginning with initial functions. So as we follow through f's chain of definitions, we start with initial functions which are d-Turing computable – by (1) – and each successive definitional move takes us from d-Turing computable to d-Turing computable functions – by (2), (3), and (4). So f must be d-Turing computable. So a fortiori, the partial recursive function f must be plain Turing computable.

Hence to establish Theorem 23, it is enough to establish (1) to (4). But each of those is more or less easy to prove. ⊠

If this overall proof strategy seems familiar, that's because we used the same proof idea in Chapter 9 when showing that PA is p.r. adequate. The details needed to fill in the proof outline are given in Section 16.4 – though these are technicalities without much conceptual interest, so by all means skip them.

16.2 Turing computability entails partial recursiveness

As we said, it was only to be hoped and expected that we could show that all μ-recursive functions are Turing computable. What about the converse claim that the Turing computable total functions are all μ-recursive?

This is a more substantial and perhaps rather surprising result. For Turing computability involves entirely 'free form', unstructured, computation (at the level of 'machine code'): we place no restrictions on the way we stack up i-quadruples into a program, other than brute consistency. By contrast, μ-recursive functions are defined (in effect) in terms of the algorithms that can be described by a higher-level computer language where programs have to be structured using just 'for' loops and 'do until' loops. So we might well wonder whether every function which is computable using an arbitrarily structured Turing program can also be computed using just our two types of looping structure. But we can now show the following:

Theorem 24 *All Turing computable functions are partial recursive*

So, in particular, the Turing computable total functions will be total partial recursive, and hence (as we'll prove in the next section) μ-recursive.

Proof outline Take the case where f is a monadic Turing computable function. Consider a computation of $f(n)$ by the program Π: we'll execute the instructions in one i-quadruple after each tick of the clock: the clock keeps ticking, however, even if the program's execution has stopped because it has halted or frozen.

At the j-th tick of the clock, the current 'state-of-play' of the Turing computation is given by (i) a description of the contents of the tape; (ii) a specification of which cell is the currently scanned cell; (iii) the label for the i-quadruple we will execute next. Note that we start with only a finite number of occupied cells on the tape, and each step makes only one modification; so at every step there are only a finite number of occupied cells; so giving the description (i) is always a finite task.

Suppose we use some kind of sensible Gödel-style numbering in order to encode the state-of-play by a single code-number s. And now consider the coding function

> $c(n, j) = s$, where s is the code describing the state-of-play at time j of the computation which starts at time 0 with input n. There are three possibilities. The program is still running, so $s > 0$ is the code describing the state as we enter the jth step in our Turing computation of $f(n)$. Or the execution of the program has already halted, in which case we put $c(n, j) = 0$. Or the program has already frozen in which case $c(n, j)$ is frozen in time too, i.e. we put $c(n, j) = c(n, j - 1)$.

Then we can show that *the function* $c(n, j)$ *is primitive recursive*. By this stage in the game, this shouldn't at all be a surprising claim. Just reflect that getting your laptop computer to simulate a Turing computation step by step involves getting it at each step to check through a finite list of instructions (so any searches are bounded by the length of the Turing program): and a LOOP program suffices for that.

If the computation halts at step j, then for all $t \leq j$, $c(n, t) > 0$ and $c(n, Sj) = 0$. So $c(n, \mu z[c(n, Sz) = 0])$ gives the code describing the state-of-play – and in particular the contents of the tape – at the point where the computation halts, if it ever does. Therefore,

$$f(n) = decode(c(n, \mu z[c(n, Sz) = 0]))$$

where *decode* is a function that decodes a state-of-play description s and returns the number encoded in the output block of binary digits on the tape at the end of the computation (both sides of the equation will be undefined if the computation doesn't halt gracefully for input n). Assuming the Gödel numbering is sensible

so that *decode* is also primitive recursive, it follows immediately that f is partial recursive. ⊠

The argument generalizes in the obvious way to many-place functions. And if the strategy of our proof-outline here also seems familiar, that's because we used much the same proof idea in Section 14.2 when sketching a proof that the Ackermann-Péter function is μ-recursive. We fill in the proof outline in Section 16.5. Again, you can certainly just skim through the details – especially as this time there's no way of making them particularly pretty. What matters most is that you grasp the strategy in our proof outline above.

16.3 A corollary – and a general observation

Suppose $f(n)$ is a one-place partial recursive function. By Theorem 23, f is computed by some Turing program. But then, as we noted in the proof of Theorem 24, there is a two-place p.r. function $c(n, j)$ whose value codes for the state-of-play as we enter the jth step in the Turing computation for input n. And $f(n) = decode(c(n, \mu z[c(n, Sz) = 0]))$, where *decode* is also a p.r. function.

Suppose now that $f(n)$ is also a *total* function. Then $\mu z[c(n, Sz) = 0]$ must yield a value for each n. But $c(x, Sz)$ – being a composition of p.r. functions – is a total function. Hence the application of the minimization operator here is regular. Which shows that f has a definition involving regular minimization, and is μ-recursive. Generalizing to many-place functions, it follows – as promised in Section 14.5 – that

Theorem 25 *Every total partial recursive function is μ-recursive.*

And hence, combining Theorems 24 and 25, the total Turing computable functions are just the μ-recursive ones.

So much for our corollary, now for the general observation. We've defined Turing machines as dealing with *binary* symbols, using a *one-tape* work-space, moving its focus of operations *one cell* at a time, and also reading/writing *one cell* at a time. Variations are evidently possible. For example, we could use a larger repertoire of symbols, or we could consider a machine with more than one tape. *But such changes don't make any difference to what our machines can compute* – i.e. they don't take us outside the class of recursive functions. We've already argued for that informally in the case of changing the size of the symbol set (see Section 15.1); and we can similarly argue, case by case, that e.g. working with two tapes doesn't make a difference either, by sketching a way of transforming a program for our two-tape Turing machine into an equivalent Turing program of the original type.

However, such transformations can be rather messy to effect. So note that the proof-strategy of the last section can adopted to get nicer equivalence proofs. For all we need to show is that the appropriate coding function $c_T(n, j) = s$ is primitive recursive, where s is now a suitable code describing the state-of-play

at time j of the computation of a given $f(n)$ on our modified machine T. And then it follows by just the same reasoning that f will still be recursive.

16.4 Showing that recursiveness entails Turing computability

We'll now explain how to fill in the details of our proof-sketch for Theorem 23 by establishing points (1) to (4) as listed in Section 16.1.

Proof sketch for (1) We proved in Section 15.3 (ii) that the successor function is not just Turing computable but is d-Turing computable. It's trivial that the zero function $Z(x)$ is computable by a dextral program – just write a program that takes any block of digits, erases it from the right, and leaves a single '0' on the tape. It's also easily seen that the identity functions are d-Turing computable by erasing and moving blocks of digits. ⊠

Proof sketch for (2) For simplicity, we'll just consider the case of monadic functions. Suppose the program Π_g d-Turing computes g, and the program Π_h d-Turing computes h. Then to d-Turing compute the composite function $f(n) = h(g(n))$, we can just run Π_g on the input n to give $g(n)$, and then run Π_h on that output to calculate $h(g(n))$. (The computation will halt, as we want, just when $h(g(n))$ has a value and so $f(n)$ is defined.)

How exactly do we chain together two programs Π_g and Π_h into one composite program? We need to do two things. We first ensure that there is no clash of labels by changing the q-numbers in the i-quadruples in Π_h (doing it systematically, of course, to preserve the all-important cross-references between quadruples). And then we must just ensure that – rather than using the 'halt' label – Π_g ends by telling us to process the first instruction in our re-labelled Π_h. ⊠

Proof sketch for (3) We'll suppose f is defined by the recursion clauses

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

where both g and h are d-Turing computable. We need to show that f is d-Turing computable. (Simplifying the discussion for the case where the x-variable drops out of the picture, or generalizing it to cover the case where we have an array of variables $\vec{x}$, is straightforward.)

It is convenient to introduce an abbreviated way of representing the contents of a tape. We'll use $\boxed{n}$ to indicate a block of cells containing the binary digits for n, we'll use 'B' to indicate a single blank cell, and then e.g. $\boxed{m}B\boxed{n}$ represents a tape which is blank except for containing m in binary followed by a blank followed by n. So, what we need to describe is a dextral program that takes $\boxed{m}B\boxed{n}$ as input and delivers $\boxed{f(m,n)}$ as output. Here's a sketch:

1. Given the input $\boxed{m}B\boxed{n}$, use a combination of a copying program and a program for subtracting one to get the tape eventually to read as follows:

$$\boxed{m}\,B\,\boxed{n}\,B\,\boxed{m}\,B\,\boxed{n-1}\,B\,\boxed{m}\,B\,\boxed{n-2}\,B\ldots$$
$$\ldots B\,\boxed{m}\,B\,\boxed{2}\,B\,\boxed{m}\,B\,\boxed{1}\,B\,\boxed{m}\,B\,\boxed{0}\,B\,\boxed{m}$$

2. Now move to scan the first cell of the *last* block of digits $\boxed{m}$. Run our program for evaluating g starting in that position. Since this program is by hypothesis dextral, it doesn't visit the portion of the tape any further left than the blank before that last block. *So it is just as if the preceding tape is completely empty.* Hence the program for evaluating g will run normally on the input m, and it will calculate $g(m)$ – i.e. calculate $f(m, 0)$. So after running g the tape will end up reading

$$\boxed{m}\,B\,\boxed{n}\,B\,\boxed{m}\,B\,\boxed{n-1}\,B\ldots\,B\,\boxed{m}\,B\,\boxed{1}\,B\,\boxed{m}\,B\,\boxed{0}\,B\,\boxed{f(m,0)}$$

3. Scan the first cell of the concluding three blocks $\boxed{m}\,B\,\boxed{0}\,B\,\boxed{f(m,0)}$. Run our program for evaluating h starting in that position. Since this program too is by hypothesis dextral, it ignores the leftwards contents of the tape and the program will run normally on the three inputs m, 0, $f(m, 0)$ to calculate $h(m, 0, f(m, 0))$ – i.e. calculate $f(m, 1)$. So after running h this first time, the tape will end up reading

$$\boxed{m}\,B\,\boxed{n}\,B\,\boxed{m}\,B\,\boxed{n-1}\,B\ldots\,B\,\boxed{m}\,B\,\boxed{1}\,B\,\boxed{f(m,1)}$$

   Now repeat the same operation. So next, scan the first cell of the last three blocks $\boxed{m}\,B\,\boxed{1}\,B\,\boxed{f(m,1)}$). Run the program for evaluating h again, and the tape will end up containing the shorter row of blocks

$$\boxed{m}\,B\,\boxed{n}\,B\,\boxed{m}\,B\,\boxed{n-1}\,B\ldots\,B\,\boxed{f(m,2)}$$

   Keep on going, each time running h using the last three blocks of digits as input, and eventually we will – as desired – be left with just

$$\boxed{f(m,n)}$$

   on the tape if $f(m, n)$ is defined – or else executing the program will have got stuck somewhere en route.

So we've outlined the shape of a program that gives a d-Turing computation of the recursively defined f. ⊠

Proof sketch for (4) It remains to show that if the function $g(x, y)$ is d-Turing computable, then so is the function $f(x) = \mu y\, g(x, y)$. Hence we want to specify a program that takes $\boxed{m}$ as input and delivers the output $\boxed{\mu y\, g(m,y)}$.

Our task is to run the program g with successive inputs of the form $\boxed{m}\,B\,\boxed{n}$, starting with $n = 0$ and incrementing n by one on each cycle; and to keep on going unless and until we get the output '0', when we return the value of n. But note, we need to do this while 'remembering' at each stage the current values of m and n. So here's a four-stage strategy for doing the job. (Again, simplifying to cover the case where the x-variable drops out of the picture, or generalizing to cover the case where we have an array of variables $\vec{x}$, is straightforward.)

1. We are given $\boxed{m}$ on the tape. Use a modified copier to produce

   $\boxed{m}B\boxed{0}B\boxed{m}B\boxed{0}$

   to be fed into Stage (2).

2. Given any input of the kind

   $\boxed{m}B\boxed{n}B\boxed{m}B\boxed{n}$

   move to scan the first occupied cell of the second block $\boxed{m}$. Now run the dextral program for g from that starting point, i.e. on the input $\boxed{m}B\boxed{n}$. See whether the result is 0. If it isn't, go to Stage (3). If it is, finish up with Stage (4).

3. The state of the tape is now

   $\boxed{m}B\boxed{n}B\boxed{g(m,n)}$

   Delete the final block, so we are left with just $\boxed{m}B\boxed{n}$ on the tape. Next increment the $\boxed{n}$ block by one, and then use a copier to yield

   $\boxed{m}B\boxed{Sn}B\boxed{m}B\boxed{Sn}$

   And repeat Stage (2) on this input.

4. The state of the tape is now

   $\boxed{m}B\boxed{n}B\boxed{0}$

   We need, in effect. to delete the blocks of digits either side of $\boxed{n}$, and shift the remaining block to the left (so it starts in the position where the $\boxed{m}$ block started). Exercise: write a d-Turing program that does the trick!

These stages can clearly be combined into a composite d-Turing program so, once more, we are done. ⊠.

Which is, all in all, a really rather pretty proof – which is why I couldn't resist giving it here!

16.5 Showing that Turing computability entails recursiveness

To fill in the proof details for Theorem 24 for masochists, we need to confirm that the state-of-play coding function c which we introduced in Section 16.2 is primitive recursive. The details will depend on how we code up a description of the state-of-play in a computation. So after (i) proving an important general fact about p.r. functions which we'll need, we'll (ii) illustrate *one* method of coding, and then (iii) sketch a proof that the corresponding c is indeed primitive recursive.

(i) *Definition by cases* Here, then, is another key fact about p.r. functions to add to those we met in Sections 6.4, 10.6. Suppose the function f is *defined by cases* from k other p.r. functions f_i, i.e.

$$
\begin{aligned}
&f(n) = f_0(n) \text{ if } C_0(n)\\
&f(n) = f_1(n) \text{ if } C_1(n)\\
&\quad\vdots\\
&f(n) = f_k(n) \text{ if } C_k(n)\\
&f(n) = 0 \text{ otherwise}
\end{aligned}
$$

where the conditions C_i are exclusive and express p.r. properties (i.e. have p.r. characteristic functions c_i). *Then f is also primitive recursive.*

Proof Just note that

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

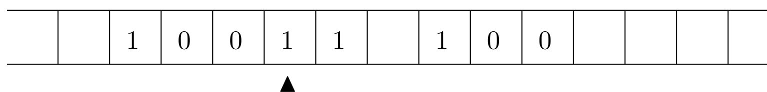
since $\overline{sg}(c_i(n)) = 1$ when $C_i(n)$ and is otherwise zero (cf. Section 6.4 for the definition of the p.r. function $\overline{sg}$). ⊠

(ii) *Coding* We'll divide the work-tape into three (non-overlapping) parts – the left part, the scanned cell, the right part – and then we'll code up the contents of the cells in these three parts using the following revised, all-numerical, code (which is sadly less memorable than the one we used before when we introduced i-quadruples, but is now more convenient):

1: indicates a cell containing 0;
2: indicates a cell containing 1;
3: indicates a blank cell.

So, firstly, the contents of the single current scanned cell can be indicated by the code $cc = 1, 2$ or 3.

Next, the right part of the tape contains a finite number of occupied cells: suppose that the r-th cell to the right of the scanned cell is the last occupied one. We'll code for the contents of these r cells in the now familiar Gödelian way – we take the first r primes, and raise the n-th prime to the power of the basic code for the contents of the n-th cell counting rightwards. Multiply the results to give the Gödel number rt. For example, if the current state of the tape is



then $cc = 2$ and $rt = 2^2 \cdot 3^3 \cdot 5^2 \cdot 7^1 \cdot 11^1 = 7700$. We set $rt = 0$ if the right part of the tape is empty.

Thirdly, lt is the Gödel-number for the contents of the occupied cells in the left part of the tape. Take the first l primes (where l is such that the l-th cell

to the left is the last occupied one): raise the n-th prime to the power of the numerical code for the contents of the n-th cell, now counting *leftwards*, and multiply the results. Hence, in our example, $lt = 2^1 \cdot 3^1 \cdot 5^2 = 150$. Again, we set $lt = 0$ if the left part of the tape is empty.

Suppose, then, that the state-of-play is as follows (while we are executing the program before it halts): we are about to execute an i-quadruple with label lb, and the tape contents are describable by the triplet of codes cc, lt, rt. We can code this overall state-of-play by one super-Gödel number s:

$$s = \ulcorner lb, cc, lt, rt \urcorner =_{\text{def}} 2^{lb} \cdot 3^{cc} \cdot 5^{lt} \cdot 7^{rt}$$

To round out our definition of s there are two more cases to consider. First, suppose the computation has halted. Then we'll set the state-of-play code to be

$$s = 0$$

and further 'steps' have no effect. And second suppose that the computation freezes (we run out of i-quadruples to execute). Then we freeze the value of s at the value it had when the program 'hung'.

(iii) *Defining the state-of-play function* To repeat, we want to show that the function c is primitive recursive, where $c(n, j)$ returns the state-of-play code at time j in computing $f(n)$.

To prove that $c(n, j)$ is p.r., it is enough to find two p.r. functions, *initial* and *trans* such that we can set

$$c(n, 0) = initial(n)$$
$$c(n, Sj) = trans(c(n, j))$$

$c(n, 0)$ gives the code for the state-of-play at the very beginning of the computation of $f(n)$: so $initial(n)$ needs to encode, in particular, the initial configuration of the tape. Whilst *trans* needs to encode the transition from the j-th to the Sj-th step in our computation of $f(n)$.

Let's think a little bit more about *trans*. Consider the following four functions

> $label(lb, cc, lt, rt)$ yields the value lb giving the label in the next state-of-play, i.e. the one we move on to after the current state coded by $\ulcorner lb, cc, lt, rt \urcorner$. (If the program freezes at this point because it gives us no instruction for what to do, then the 'next' state-of-play is identical to the current one.)
>
> $scan(lb, cc, lt, rt)$ yields the value of cc coding the contents of the scanned cell in the next state of play.
>
> $left(lb, cc, lt, rt)$ yields the value of lt coding the contents of the left part of the tape in the next state of play.
>
> $right(lb, cc, lt, rt)$ yields the value of rt coding the contents of the left part of the tape in the next state of play.

It is relatively routine to define the transition function *trans* in terms of these four functions, and thereby show that *trans* is p.r. if the new functions are.

Proof We have to consider cases:

1. $c(n, j) > 0$. Then use the p.r. function *exp* to extract the exponents of $c(n, j)$'s prime factors, i.e. the current values of *lb*, *cc*, *lt*, *rt*. Then there are two sub-cases:
   a) $lb > 0$: Apply our four new functions to calculate the updated values of our four code numbers. Use these updated values as exponents of the first four primes and multiply to get $c(n, Sj)$.
   b) Otherwise we've just halted so put $c(n, Sj) = 0$
2. Otherwise $c(n, j) = 0$, so put $c(n, Sj) = 0$ too.

So *trans* is defined by cases in a way that makes it p.r if our four new functions are. ⊠

(iv) *Completing the proof that the state-of-play function is p.r.* To complete the proof, we need to show that five simpler functions are p.r., namely *initial* and the additional functions we've just introduced. You may very well be prepared to take that on trust – but if not, here are some proof sketches:

(1) *Proof sketch that initial(n) is p.r.* Suppose $bi(n)$ is the number of digits in the binary representation of n; and $digit(m, n)$ returns 2 if the $(m + 1)$-th digit of n in binary is '1', returns 1 if the $(m+1)$-th digit is '0', and returns '0' if there is no such digit. Both these functions are easily shown to be primitive recursive.

Now, we start a computation of $f(n)$ with the work-tape in a standard initial configuration – i.e. with n in binary on the tape, and scanning the first digit, with the rest of n's binary digits to the right – and about to execute an initial i-quadruple with label '1'. So we have the initial values

$$label_i = 1$$
$$cc_i = digit(0, n)$$
$$lt_i = 0$$
$$rt_i = (\mu x \leq B)(\forall k < bi(n))(exp(k, x) = digit(k + 1, n))$$

for some suitable bound B. So, we can put

$$initial(n) = \ulcorner 1, cc_i, 0, rt_i \urcorner = 2^1 \cdot 3^{cc_i} \cdot 5^0 \cdot 7^{rt_i}$$

where on the right we have a p.r. function of n. ⊠

(2) *Proof sketch that the label function is p.r.* The way that the state-of-play of our computation changes as we compute $f(n)$ will depend on the Turing program Π which, by hypothesis, computes f. Now, each of its i-quadruples $\langle q_1, S, A, q_2 \rangle$ can be thought of as contributing to the definition of two functions:

$action(q_1, S)$ which takes a label number and (the number code for) the contents of the scanned cell, and yields (a number code for) the next action A if $\langle q_1, S, A, q_2\rangle$ is in Π, and yields zero otherwise – take the action codes to be 1, 2, 3 for writing '0', '1' or a blank; 4 for moving right, 5 for moving left;

$jump(q_1, S)$ which similarly takes a label number and (the number code for) the contents of the scanned cell and yields q_2 if $\langle q_1, S, A, q_2\rangle$ is in Π, and yields q_1 if there is no quadruple $\langle q_1, S, A, q_2\rangle$ is in Π

Since the program is finite, we can write down a finite set of conditions for defining $action(q_1, S)$ and for defining $jump(q_1, S)$. Hence these functions are p.r., by the general fact we established in (i). But trivially,

$$label(lb, cc, lt, rt) = jump(lb, cc)$$

So *label* is also p.r. ⊠

(3) *Proof sketch that the scan, left and right functions are p.r.* We'll deal explicitly with *scan* and *left*; the case of *right* is exactly similar to *left*. There are three cases to consider, depending on the next action we are told to perform:

1. $action(lb, cc) = 0$ indicating that no 'next action' is defined, and the program freezes. So we freeze the state-of-play code by putting

   $$scan(lb, cc, lt, rt) = cc$$
   $$left(lb, cc, lt, rt) = lt$$

2. $action(lb, cc)$ yields a code for writing a blank, '0, or '1' in the scanned cell. Then the left part of the tape is not touched. So,

   $$scan(lb, cc, lt, rt) = action(lb, cc)$$
   $$left(lb, cc, lt, rt) = lt$$

3. $action(lb, cc) = 4$, i.e. codes for moving the scanned cell right. Then the new scanned contents will be the current contents of the first cell on the right; and contents of the new leftwards portion of the tape after we've moved are the contents of the old leftwards portion prefixed by the contents of the current scanned cell. So we can put

   $$scan(lb, cc, lt, rt) = exp(0, rt)$$
   $$left(lb, cc, lt, rt) = 2^{cc} \star lt$$

   For recall, $exp(0, rt)$ returns the exponent of $\pi_0 = 2$ in the factorization of rt – i.e. returns, as we want, the code for the current contents of the first cell on the right. And the 'star' function is the one we introduced in Section 10.6 precisely for the job of coding up the results of adding symbols to a string.

4. $action(lb, cc) = 5$, i.e. codes for moving the scanned cell left. Then

$$scan(lb, cc, lt, rt) = exp(0, lt)$$
$$left(lb, cc, lt, rt) = (\mu x \leq lt)(\exists y \leq 3)(lt = 2^y \star x)$$

Hence, putting all that together, we can define each of *scan* and *left* by cases. So, again by the result (i), they are primitive recursive. ⊠

Phew! Finally, we'll leave it as an exercise to explain how to define the decoding function *decode*, which reads the final state of a tape, so we can put $f(n) = decode(c(n, \mu z[c(n, Sz) = 0]))$. If all has gone well, and the program has just halted gracefully at step $h = \mu z[c(n, Sz) = 0]$, the scanned cell contains the first of a single block of binary digits. So we want a function that extracts the codes cc and rt from the final value of $c(n, h)$, and then reconstructs the number whose binary digits are in the scanned cell followed by the occupied rightward cells. If you've been really following this far, then that's left as a fairly easy exercise: enough is enough!

17 Universal Turing Machines

This chapter introduces a further key idea already in Turing's great 1936 paper, namely the idea of *universal* Turing machines which can simulate the operation of *any* Turing machine. En route to showing that there are such universal machines, we prove a famous theorem due to Kleene.

17.1 Effectively enumerating Turing programs

Turing programs (or equivalently, the Turing machines identified by their programs) can be effectively enumerated by using Gödel-numbers to code sets of consistent quadruples.

Here's one way of doing this. Recall, an i-quadruple has the form $\langle q_1, S, A, q_2\rangle$ where S is one of $\{0, 1, B\}$, A is one of $\{0, 1, B, L, R\}$, and q_1 and q_2 are numerals (which can of course include '0' and '1' again). So suppose first we set up the following basic coding relating these elements and the integers:

$$
\begin{aligned}
bc(B) &= 1\\
bc(L) &= 2\\
bc(R) &= 3\\
bc(q) &= 4 + q \text{ for each number } q
\end{aligned}
$$

Then we form the g.n. of a quadruple $\langle q_1, S, A, q_2\rangle$ in the obvious way:

$$gn(\langle q_1, S, A, q_2\rangle) = 2^{bc(q_1)} \cdot 3^{bc(S)} \cdot 5^{bc(A)} \cdot 7^{bc(q_2)}$$

And we can form the super g.n. of a collection of i-quadruples $\{i_0, i_1, i_2, \ldots, i_n\}$ as follows:

$$sgn(\{i_0, i_1, i_2, \ldots, i_n\}) = 2^{gn(i_0)} \cdot 3^{gn(i_1)} \cdot 5^{gn(i_2)} \cdot \ldots \cdot \pi_n^{gn(i_n)}$$

where π_n is the $n+1$-th prime.

For any number e, it's a mechanical matter to check whether e is the super g.n. of a Turing program Π (just take prime factors of e, then prime factors of the exponents; and if this reveals that e is the super g.n. of a finite set of i-quadruples, then check that it is a consistent set and hence a Turing program). So here's how we can construct an effective enumeration of Turing programs Π_e:

> If e is the super g.n. of a Turing program Π, put $\Pi_e = \Pi$; otherwise put $\Pi_e = \varnothing$ (i.e. the empty program with no i-quadruples in it). Then $\Pi_0, \Pi_1, \Pi_2, \ldots$ is an effectively generated list of all the Turing programs (with repetitions allowed).

17.2 Kleene's Normal Form Theorem

In Section 16.2, we considered the computation of a particular one-place function f by a given Turing machine Π. At each stage of the computation, we can characterize the state-of-play by a single code-number s. And we defined the two-place coding function

> $c(n, j) = s$, where $s > 0$ is the code describing the state-of-play at time j – i.e is the code describing the state as we enter the jth step in our Turing computation of $f(n)$, unless the computation has already halted in which case $c(n, j) = 0$, or it has frozen in which case $c(n, j) = c(n, j - 1)$.

Now we are going to define a more general three-place coding function:

> $c'(e, n, j) = s$, where $s > 0$ is the code describing the state-of-play at time j – i.e is the code describing the state as we enter the jth step in a run of the Turing program Π_e given the initial input n, unless the computation has already halted in which case $c'(e, n, j) = 0$, or it has frozen in which case $c'(e, n, j) = c'(e, n, j - 1)$.

The two-place coding function c is primitive recursive: we gave a very brief motivating argument for that claim in Section 16.2, and proved it in rather gruesome detail in Section 16.5. Similarly, the three-place coding function c' is primitive recursive.

Here's a motivating argument. To evaluate $c'(e, n, j)$ we first need to decode the index number e to find the corresponding program Π_e. But that's just a matter of taking prime factors twice over, which is a p.r. procedure. Then we need to evaluate the corresponding two-place function $c_e(n, j)$ which codes the stay-of-play as we enter the jth step in a run of the Turing program Π_e from initial input n; and we've already shown that such a function is primitive recursive. Hence, given that both stages in evaluating $c'(e, n, j)$ are p.r., so is the overall function.[1]

Given $c(n, j) = s$ is the state-of-play coding function for the computation of $f(n)$ via a particular Turing program, we saw that we can put

$$f(n) = decode(c(n, \mu z[c(n, Sz) = 0]))$$

[1] That motivating argument can be pumped up into another full-dress proof; but we won't do that here. Let's just give the merest hint of how to proceed. To show that $c(n, j)$ is p.r., we needed to show e.g. that the function $action(q_1, S)$ is p.r., where $action(q_1, S)$ takes a label number and the contents of the scanned cell, and yields the next action if $\langle q_1, S, A, q_2\rangle$ is defined, and yields zero otherwise. Similarly, to show that $c'(e, n, j)$ is p.r., we need to show that $action'(e, q_1, S)$ is p.r., where $action'(e, q_1, S)$ takes a label number and the contents of the scanned cell, and yields the next action if $\langle q_1, S, A, q_2\rangle$ is an i-quadruple in the program Π_e, and yields zero otherwise. But we can readily give a p.r. condition which is satisfied just if $\langle q_1, S, A, q_2\rangle$ is an i-quadruple in the (consistent) program with code number e. And so it goes.

where *decode* is a p.r. function which decodes a state-of-play description s and returns the number encoded in the output block of binary digits. Now let's use the symbol $f_e(x)$ for the one-place function Turing-computed by Π_e, if there is one. (By Theorem 23, every partial recursive function is computed by some Turing machine. So every monadic partial recursive function is one of the f_e.) Then exactly similarly, we have

$$f_e(n) = decode(c'(e, n, \mu z[c'(e, n, Sz) = 0]))$$

Let's define two functions by composition as follows:

$$t(x, y, z) =_{\text{def}} c'(x, y, Sz)$$
$$u(x, y, z) =_{\text{def}} decode(c'(x, y, z))$$

Both t and u are p.r. functions: hence we have established the following result:

> There is a pair of three-place p.r. functions t and u such that any one-place partial recursive function can be given in the standard form
>
> $$f_e(n) =_{\text{def}} u(e, n, \mu z[t(e, n, z) = 0])$$

Which is almost, but not quite, Kleene's Normal Form theorem (first proved in his 1936a). For here's the official version for one-place functions:

> **Theorem 26** *There is a three-place p.r. function T and a one-place p.r. function U such that any one-place partial recursive function can be given in the standard form*
>
> $$f_e(n) =_{\text{def}} U(\mu z[T(e, n, z) = 0])$$

So the only difference in the official version is that, by a bit of not-quite-trivial juggling, we can get all the dependence on e and n packed into the function T, allowing U to be a one-place function. This difference between our almost-Kleene version and the real thing is a distinction which doesn't matter at all in most applications, and the rationale for almost-Kleene is probably more transparent. However, just for the record, here's a

Proof of the official version Consider the four-place relation $R(e, n, r, j)$ which holds when the Turing program Π_e, run with input n initially on the tape, halts gracefully with the result r on the tape, no later than time j. This relation is p.r., since it holds when $t(e, n, j) = 0$, and $r = u(c, n, (\mu z \leq j)[t(e, n, z) = 0])$ – and both those conditions are p.r.

Now put $T(e, n, z) = 0$ when $(\exists r \leq z)(\exists j \leq z)(z = 2^r \cdot 3^t \wedge R(e, n, r, j))$ and put $T(e, n, z) = 1$ otherwise. Then T is p.r. (see Section 16.5 on definition by cases). The computation halts at the least value of z such that $T(e, n, z) = 0$. And we can extract the value of r, i.e. the result of the computation, by using the familiar factorizing function $exp(z, 0)$ which gives the exponent of π_0, i.e. 2, in the factorization of z. So put $U(z) =_{\text{def}} exp(z, 0)$. Then, as we wanted, $f(n) = U(\mu z\, T(e, n, z))$, with U and T p.r. functions. ⊠

We can readily generalize the argument to get a Normal Form theorem to cover many-place partial recursive functions. But we won't bother with the details.[2]

Note, by the way, that $U(\mu z\, T(e, n, z))$ is a partial recursive function for *every* index number e. So, running through the indices e gives us *an effective enumeration of the one-place partial recursive functions* (it includes *all* the monadic partial recursive functions since, to repeat, every such function is computed by some Turing machine with some index e, and $f_e(n)$ is the function that machine computes).

17.3 Universal machines

'Turing machines', as we noted in Section 15.4, are identified by their programs: machines running different programs count as different machines (thus we can talk of *the* machine with program Π_e, or simple the machine Π_e). You might suppose that this makes for a big gap between Turing machines and the 'general purpose' computers you are familiar with (where the same machine can load and run many programs). However, we can close the apparent gap with a remark and a theorem.

The remark is that, if a general purpose computer is to be able to interpret the programs it loads, it must of course already be running an interpreter program. And even if it e.g. loads up the interpreter from its hard disk when booting up, the computer has to start with *some* software hard-wired in (to enable it to load the interpreter). Our general purpose computers, too, can in effect be individuated by their necessary built-in software.

The theorem is that there are Turing machines – universal machines, in the jargon – which function as general purpose machines. That is to say, there are Turing machines which can 'load' arbitrary programs and then 'interpret' and run them. More carefully, there are Turing machines which, given a specification of the machine Π_e together with one or more numbers $\vec{n}$ as input (on its tape), delivers just the same output as the machine Π_e itself running on input $\vec{n}$.

How are we going to specify a machine on the input tape? There are various ways, but one is just to use its program's code-number e. Taking that line, and concentrating on the case where we are computing monadic functions, we have the following:

> **Theorem 27** *There is a universal Turing machine which, given numbers e and n as input, yields as output (if any) the result of machine Π_e operating on n.*

Proof Consider the two-place function $U(\mu z[T(x, y, z) = 0])$ used in Kleene's Normal Form theorem. This is partial recursive. Hence, by Theorem 23, there is a Turing machine Π_u which computes it. Think how Π_u behaves. Run it from

[2] Note that we *can* prove Kleene's theorem about recursive functions more directly, i.e. without going via the different idea of a Turing program. But our indirect proof is more accessible.

the initial state where the super g.n. for Π_e is on the tape followed by n; then the output will be $U(\mu z[T(e,n,z) = 0])$, i.e. will be the same as the output of Π_e run from the initial state where just n is on the tape. So Π_u is the desired 'universal' program.[3] ⊠

Again, we can readily generalize this result to the case of machines computing many-place functions. But again we won't bother with the details.

That's a beautiful (and perhaps surprising) Theorem: it emphasizes how powerful a Turing machine can be. Here's a way of putting the point dramatically. Given a Turing program Π_e, then *we* can follow the program instructions, using paper and pencil, for some given input argument n (of course, there may be no final output, if we get stuck in a loop and e.g. find ourselves writing an infinite sequence of '1's on the tape). *A universal Turing machine can do the same*, given (the code for) Π_e on its tape: it too can follow the instructions for applying Π_e to some argument n. Hence, at least as far as our ability to follow algorithmic instructions like Turing programs is concerned, we are no better than universal Turing machines. So maybe – to raise a question that will concern us later – we just *are* embodied Turing machines?

[3] Just as we proved Kleene's theorem about recursive functions by going via Turing machines, now we've proved the existence of a universal machine for computing all monadic Turing-computable functions by going via a theorem about recursive function. And just as we can prove Kleene's theorem more directly, we can prove the existence of universal machines more directly – as Turing himself did (or almost did – there were some bugs in his original version). But having done all the hard work earlier of relating recursiveness to Turing computability, why make use of that now to give our very neat, if less direct, derivations?

Bibliography

Gödel's papers are identified by their dates of first publication; but translated titles are used and references are to the versions in the *Collected Works*, where details of the original publications are to be found. Similarly, articles or books by Frege, Hilbert etc. are identified by their original dates, but references are whenever possible to standard English translations.

Ackermann, W., 1928. On Hilbert's construction of the real numbers. In van Heijenoort 1967, pp. 495–507.

Balaguer, M., 1998. *Platonism and Anti-Platonism in Mathematics.* New York: Oxford University Press.

Bishop, E. and Bridges, D., 1985. *Constructive Analysis.* Berlin: Springer-Verlag.

Boolos, G., 1993. *The Logic of Provability.* Cambridge: Cambridge University Press.

Boolos, G., 1997. Must we believe in set theory? In Boolos 1998, pp. 120–132.

Boolos, G., 1998. *Logic, Logic, and Logic.* Cambridge, MA: Harvard University Press.

Boolos, G., Burgess, J., and Jeffrey, R., 2002. *Computability and Logic.* Cambridge: Cambridge University Press, 4th edn.

Bourbaki, N., 1968. *Elements of Mathematics: Theory of Sets.* Paris: Hermann.

Cantor, G., 1874. On a property of the set of real algebraic numbers. In Ewald, Vol. 2, pp. 839–843.

Cantor, G., 1891. On an elementary question in the theory of manifolds. In Ewald, Vol. 2, pp. 920–922.

Carnap, R., 1934. *Logische Syntax der Sprache.* Vienna: Springer. Translated into English as Carnap 1937.

Carnap, R., 1937. *The Logical Syntax of Language.* London: Paul, Trench.

Church, A., 1936. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58: 345–363.

Cohen, D. E., 1987. *Computability and Logic.* Chichester: Ellis Horwood.

Cooper, S. B., 2004. *Computability Theory.* Boca Raton, Florida: Chapman and Hall/CRC.

Copeland, B. J. (ed.), 2004. *The Essential Turing.* Oxford: Clarendon Press.

Curry, H. B., 1942. The inconsistency of certain formal logics. *Journal of Symbolic Logic*, 2: 115–117.

Cutland, N. J., 1980. *Computability.* Cambridge: Cambridge University Press.

Dedekind, R., 1888. Was sind und was sollen die Zallen? In Ewald 1996, Vol. 2, pp. 790–833.

Dummett, M., 1973. *Frege: Philosophy of Language.* London: Duckworth.

Dummett, M., 1991. *Frege: Philosophy of Mathematics.* London: Duckworth.

Ewald, W. (ed.), 1996. *From Kant to Hilbert.* Oxford: Clarendon Press.

Field, H., 1989. *Realism, Mathematics and Modality.* Oxford: Basil Blackwell.

Frege, G., 1882. On the scientific justification of a conceptual notation. In T. W. Bynum (ed.), *Conceptual Notation and Related Articles*, pp. 83–89. Oxford: Clarendon Press, 1972.

Frege, G., 1891. Function and concept. In Frege 1984, pp. 137–156.

Frege, G., 1964. *The Basic Laws of Arithmetic.* Berkeley and Los Angeles: University of California Press.

Frege, G., 1984. *Collected Papers.* Oxford: Basil Blackwell.

Gödel, K., 1929. On the completeness of the calculus of logic. In Gödel 1986, Vol. 1, pp. 60–101.

Gödel, K., 1931. On formally undecidable propositions of *Principia Mathematica* and related systems I. In Gödel 1986, Vol. 1, pp. 144–195.

Gödel, K., 1934. On undecidable propositions of formal mathematical systems. In Gödel 1986, Vol. 1, pp. 346–371.

Gödel, K., 1986. *Collected Works, Vol. 1: Publications 1929–1936.* New York and Oxford: Oxford University Press.

Gödel, K., 2003. *Collected Works, Vol. 5: Correspondence H–Z.* Oxford: Clarendon Press.

Hájek, P. and Pudlák, P., 1983. *Metamathematics of First-Order Arithmetic.* Berlin: Springer.

Hale, B., 1987. *Abstract Objects.* Oxford: Basil Blackwell.

Hale, B. and Wright, C., 2001. *The Reason's Proper Study.* Oxford: Clarendon Press.

Henkin, L., 1952. A problem concerning provability. *Journal of Symbolic Logic*, 17: 160.

Hilbert, D., 1918. Axiomatic thought. In Ewald 1996, Vol. 2, pp. 1107–1115.

Hilbert, D., 1925. On the infinite. In van Heijenoort 1967, pp. 369–392.

Hilbert, D. and Bernays, P., 1939. *Grundlagen der Mathematik, Vol II.* Berlin: Springer.

Hunter, G., 1971. *Metalogic.* London: Macmillan.

Janiczak, A., 1950. A remark concerning the decidability of complete theories. *Journal of Symbolic Logic*, 15: 277–279.

Kleene, S. C., 1936a. General recursive functions of natural numbers. *Mathematische Annalen*, 112: 727–742.

Kleene, S. C., 1936b. A note on recursive functions. *Bulletin of the American Mathematical Society*, 42: 544–546.

Kleene, S. C., 1938. On notation for ordinal numbers. *Journal of Symbolic Logic*, 3: 150–155.

Löb, M. H., 1955. Solution of a problem of Leon Henkin. *Journal of Symbolic Logic*, 20: 115–118.

Mendelson, E., 1997. *Introduction to Mathematical Logic.* Chapman and Hall, 4th edn.

Meyer, A. R. and Ritchie, D., 1967. Computational complexity and program structure. Tech. Rep. RC-1817, IBM.

Peano, G., 1889. *The Principles of Arithmetic.* In van Heijenoort 1967, pp. 85–97.

Péter, R., 1934. Über den zusammenhang der verschiedenen Begriffe der rekursiven Funktionen. *Mathematische Annalen*, 110: 612–632.

Péter, R., 1935. Konstruktion nichtrekursiver Funktionen. *Mathematische Annalen*, pp. 42–60.

Péter, R., 1951. *Rekursive Funktionen*. Budapest: Akadémiai Kiadó.

Péter, R., 1967. *Recursive Functions*. New York: Academic Press.

Robinson, R., 1952. An essentially undecidable axiom system. In *Proceedings of the International Congress of Mathematicians, Cambridge, Mass., 1950, Vol. 1*, pp. 729–730. Providence, R.I.

Rosser, J. B., 1936. Extensions of some theorems of Gödel and Church. *Journal of Symbolic Logic*, pp. 230–235.

Russell, B., 1902. Letter to Frege. In van Heijenoort 1967, pp. 124–125.

Russell, B. and Whitehead, A. N., 1910–13. *Principia Mathematica*. Cambridge: Cambridge University Press.

Shepherdson, J. C. and Sturgis, H. C., 1963. Computability of recursive functions. *Journal of the Association for Computing Machinery*, 10: 217–255.

Shoenfield, J. R., 1967. *Mathematical Logic*. Reading, MA: Addison-Wesley.

Skolem, T., 1923. The foundations of elementary arithmetic established by means of the recursive mode of thought, without the use of apparent variables ranging over infinite domains. In van Heijenoort 1967, pp. 303–333.

Tarski, A., 1933. *Pojecie Prawdy w Jezykach Nauk Dedukcyjnych*. Warsaw. Translated into English in Tarksi 1956, pp. 152–278.

Tarski, A., 1956. *Logic, Semantics, Metamathematics*. Oxford: Clarendon Press.

Tarski, A., Mostowski, A., and Robinson, R., 1953. *Undecidable Theories*. Amsterdam: North-Holland Publishing Co.

Tourlakis, G., 2002. A programming formalism for PR. www.cs.yorku.ca/∼gt/papers/loop-programs.pdf.

Tourlakis, G., 2003. *Lectures in Logic and Set Theory*. Cambridge: Cambridge University Press.

Turing, A., 1936. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42: 230–65.

van Heijenoort, J. (ed.), 1967. *From Frege to Gödel*. Cambridge, MA: Harvard University Press.

von Neumann, J., 1927. Zur Hilbertschen Beweistheorie. *Mathematische Zeitschrift*, 26: 1–46.

Wang, H., 1974. *From Mathematics to Philosophy*. Routledge and Kegan Paul.

Wright, C., 1983. *Frege's Conception of Number as Objects*. Aberdeen: Aberdeen University Press.