15
Arithmetization

In this chapter we begin to bring together our work on logic from the past few chapters with our work on computability from earlier chapters (specifically, our work on recursive functions from Chapters 6 and 7). In section 15.1 we show how we can 'talk about' such syntactic notions as those of sentence and deduction in terms of recursive functions, and draw among others the conclusion that, once code numbers are assigned to sentences in a reasonable way, the set of valid sentences is semirecursive. Some proofs are deferred to sections 15.2 and 15.3. The proofs consist entirely of showing that certain effectively computable functions are recursive. Thus what is being done in the two sections mentioned is to present still more evidence, beyond that accumulated in earlier chapters, in favor of Church's thesis that all *effectively computable functions are recursive. Readers who feel they have seen enough evidence for Church's thesis for the moment may regard these sections as optional.*

15.1 Arithmetization of Syntax

A necessary preliminary to applying our work on computability, which pertained to functions on natural numbers, to logic, where the objects of study are expressions of a formal language, is to code expressions by numbers. Such a coding of expressions is called a *Gödel numbering*. One can then go on to code finite sequences of expressions and still more complicated objects.

A set of symbols, or expressions, or more complicated objects may be called recursive in a transferred or derivative sense if and only if the set of code numbers of elements of the set in question is recursive. Similarly for functions. Officially, a language is just a set of nonlogical symbols, so a language may be called recursive if and only if the set of code numbers of symbols in the language is recursive. In what follows we tacitly assume throughout that the languages we are dealing with are recursive: in practice we are going to be almost exclusively concerned with *finite* languages, which are trivially so.

There are many reasonable ways to code finite sequences, and it does not really matter which one we choose. Almost all that matters is that, for any reasonable choice, the following *concatenation* function will be recursive: $s * t =$ the code number for the sequence consisting of the sequence with code number s followed by the sequence with code number t. This is all that is needed for the proof of the next proposition, in which, as elsewhere in this section, 'recursive' could actually be strengthened to 'primitive recursive'.

So that the reader may have something definite in mind, let us offer one example of a coding scheme. It begins by assigning code numbers to symbols as in Table 15-1.

Table 15-1. *Gödel numbers of symbols (first scheme)*

Symbol	(	$\sim$	$\exists$	$=$	v_0	A^0_0	A^1_0	A^2_0	$\ldots$	f^0_0	f^1_0	f^2_0	$\ldots$
	)	$\vee$			v_1	A^0_1	A^1_1	A^2_1	$\ldots$	f^0_1	f^1_1	f^2_1	$\ldots$
	,				v_2	A^0_2	A^1_2	A^2_2	$\ldots$	f^0_2	f^1_2	f^2_2	$\ldots$
					$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$	$\vdots$	
Code	1	2	3	4	5	6	68	688	$\ldots$	7	78	788	$\ldots$
	19	29			59	69	689	6889	$\ldots$	79	789	7889	$\ldots$
	199				599	699	6899	68899	$\ldots$	799	7899	78899	$\ldots$
					$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$	$\vdots$	

Thus for the language of arithmetic $<$ or A^2_0 has code number 688, $\mathbf{0}$ or f^0_0 has code number $7'$, or f^1_0 has code number 78, $+$ or f^2_0 has code number 788, and $\cdot$ or f^2_1 has code number 7889. We then extend the code numbering to all finite sequences of symbols. The principle is that if the expression E has code number e and the expression D has code number d, then the expression ED obtained by concatenating them is to have the code number whose decimal numeral is obtained by concatenating the decimal numeral for e and the decimal numeral for d. Thus $(\mathbf{0} = \mathbf{0} \vee \sim\mathbf{0} = \mathbf{0})$, the sequence of symbols with code numbers

$$1, 7, 4, 7, 29, 2, 7, 4, 7, 19$$

has code number 174 729 274 719.

In general the code number for the concatenation of the expressions with code numbers e and d can be obtained from e and d as $e * d = e \cdot 10^{\lg(d,10)+1} + d$, where lg is the logarithm function of Example 7.11. For $\lg(d, 10) + 1$ will be the least power z such that $d < 10^z$, or in other words, the number of digits in the decimal numeral for d, and thus the decimal numeral for $e \cdot 10^{\lg(d,10)+1}$ will be that for e followed by as many 0 s as there are digits in that for d, and the decimal numeral for $e \cdot 10^{\lg(d,10)+1} + d$ will be that for e followed by that for d.

15.1 Proposition. The logical operations of negation, disjunction, existential quantification, substitution of a term for free occurrences of a variable, and so on, are recursive.

Proof: Let n be the code number for the tilde, and let neg be the recursive function defined by letting $\text{neg}(x) = n * x$. Then if x is the code number for a formula, $\text{neg}(x)$ will be the code number for its negation. (We do not care what the function does with numbers that are *not* code numbers of formulas.) This is what is meant by saying that the operation of negation is recursive. Similarly, if l and d and r are the code numbers for the left parenthesis and wedge and right parenthesis, $\text{disj}(x, y) = l * x * d * y * r$ will be the code number for the disjunction of the formulas coded by

x and y. If e is the code number for the backwards E, then exquant$(v, x) = e * v * x$ will be the code number for the existential quantification with respect to the variable with code number v of the formula with code number x. And similarly for as many other logical operations as one cares to consider. For instance, if officially the conjunction $(X \& Y)$ is an abbreviation for $\sim(\sim X \vee \sim Y)$, the conjunction function is then the composition conj $(x, y) =$ neg(disj(neg (x), neg(y))). The case of substitution is more complicated, but as we have no immediate need for this operation, we defer the proof.

Among sets of expressions, the most important for us will be simply the sets of formulas and of sentences. Among more complicated objects, the only important ones for us will be deductions, on whatever reasonable proof procedure one prefers, whether ours from the preceding chapter, or some other from some introductory textbook. Now intuitively, one can effectively decide whether or not a given sequence of symbols is a formula, and if so, whether it is a sentence. Likewise, as we mentioned when introducing our own proof procedure, one can effectively decide whether a given object D is a deduction of a given sentence from a given finite set of sentences Γ_0. If Γ is an *infinite* set of sentences, then a deduction of D from Γ is simply a deduction of D from some finite subset of Γ_0, and therefore, *so long as one can effectively decide whether a given sentence C belongs to* Γ, and hence can effectively decide whether a given finite set Γ_0 is a subset of Γ, one can also effectively decide whether a given object is a deduction of D from Γ. Church's thesis then implies the following.

15.2 Proposition. The sets of formulas and of sentences are recursive.

15.3 Proposition. If Γ is a recursive set of sentences, then the relation 'Σ is a deduction of sentence D from Γ' is recursive.

Collectively, Propositions 15.1–15.3 (and their various attendant lemmas and corollaries) are referred to by the imposing title at the head of this section. Before concerning ourselves with the proofs of these propositions, let us note a couple of implications.

15.4 Corollary. The set of sentences deducible from a given recursive set of sentences is semirecursive.

Proof: What is meant is that the set of *code numbers* of sentences deducible from a given recursive set is semirecursive. To prove this we apply Proposition 15.3. What is meant by the statement of that proposition is that if Γ is recursive, then the relation

$$\begin{aligned} Rsd \leftrightarrow\ & d \text{ is the code number of a sentence and} \\ & s \text{ is the code number of a deduction of it from } \Gamma \end{aligned}$$

is recursive. And then the set S of code numbers of sentences deducible from Γ, being given by $Sd \leftrightarrow \exists s\ Rsd$, will be semirecursive.

15.5 Corollary (Gödel completeness theorem, abstract form). The set of valid sentences is semirecursive.

Proof: By the Gödel completeness theorem, the set of valid sentences is the same as the set of demonstrable sentences, that is, as the set of sentences deducible from $\Gamma = \varnothing$. Since the empty set $\varnothing$ is certainly recursive, it follows from the preceding corollary that the set of valid sentences is semirecursive.

The preceding corollary states as much of the content of the Gödel completeness theorem as it is possible to state without mentioning any particular proof procedure. The next corollary is more technical, but will be useful later.

15.6 Corollary. Let Γ be a recursive set of sentences in the language of arithmetic, and $D(x)$ a formula of that language. Then:

(a) The set of natural numbers n such that $D(\mathbf{n})$ is deducible from Γ is semirecursive.
(b) The set of natural numbers n such that $\sim D(\mathbf{n})$ is deducible from Γ is semirecursive.
(c) If for every n either $D(\mathbf{n})$ or $\sim D(\mathbf{n})$ is deducible from Γ, then the set of n such that $D(\mathbf{n})$ is deducible from Γ is recursive.

Proof: For (a), we actually show that the set R of pairs (d, n) such that d is the code number for a formula $D(x)$ and $D(\mathbf{n})$ is deducible from Γ is semirecursive. It immediately follows that for any one fixed $D(x)$, with code number d, the set of n such that $D(\mathbf{n})$ is deducible from Γ will be semirecursive, since it will simply be the set of n such that Rdn. To avoid the need to consider substituting a term for the free occurrences of a variable (the one operation mentioned in Proposition 15.1 the proof of whose recursiveness we deferred), first note that for any n, $D(\mathbf{n})$ and $\exists x(x = \mathbf{n} \,\&\, D(x))$ are logically equivalent, and one will be a consequence of, or equivalently, deducible from, Γ if and only if the other is. Now note that the function taking a number n to the code number $\mathrm{num}(n)$ for the numeral $\mathbf{n}$ is (primitive) recursive, for recalling that officially s' is $'(s)$ we have

$$\mathrm{num}(0) = z \qquad \mathrm{num}(n') = a * b * \mathrm{num}(n) * c$$

where z is the code number for the cipher $\mathbf{0}$ and a, b, and c are the code numbers for the accent and the left and right parentheses. The function f taking the code number d for a formula $D(x)$ and a number n to the code number for $\exists x(x = \mathbf{n} \,\&\, D(x))$ is recursive in consequence of Proposition 15.1, since we have

$$f(d, n) = \mathrm{exquant}(v, \mathrm{conj}(i * b * v * k * \mathrm{num}(n) * c),\ d)$$

where v is the code number for the variable, i for the equals sign, k for the comma. The set S of code numbers of sentences that are deducible from Γ is semirecursive by Corollary 15.4. The set R of pairs is then given by

$$R(d, n) \leftrightarrow S(f(d, n)).$$

In other words, R is obtained from the semirecursive set S by substituting the recursive total function f, which implies that R is itself semirecursive.

As for (b), we actually show that the set Q of pairs (d, n) such that d is the code number for a formula $D(x)$ and $\sim D(\mathbf{n})$ is deducible from Γ is semirecursive. Indeed, with R as in part (a) we have

$$Q(d, n) \leftrightarrow R(\mathrm{neg}(d), n).$$

So Q is obtained from the semirecursive R by substitution of the recursive total function neg, which implies that Q is itself semirecursive.

Obviously there is nothing special about negation as opposed to other logical constructions here. For instance, in the language of arithmetic, we could consider the operation taking $D(x)$ not to $\sim D(x)$ but to, say,

$$D(x) \,\&\, \sim\exists y < x\; D(y)$$

and since the relevant function on code numbers would still, like neg, be recursive in consequence of Proposition 15.1, so the set of pairs (d, n) such that

$$D(\mathbf{n}) \,\&\, \sim\exists y < \mathbf{n} D(y)$$

is deducible from Γ is also semirecursive. We are not going to stop, however, to try to find the most general formulation of the corollary.

As for (c), if for any n both $D(\mathbf{n})$ and $\sim D(\mathbf{n})$ are deducible from Γ, then *every* formula is deducible from Γ, and the set of n such that $D(\mathbf{n})$ is deducible from Γ is simply the set of *all* natural numbers, which is certainly recursive. Otherwise, on the assumption that for every n either $D(\mathbf{n})$ or $\sim D(\mathbf{n})$ is deducible from Γ, the set of n for which $D(\mathbf{n})$ is deducible and the set of n for which $\sim D(\mathbf{n})$ is deducible are complements of each other. Then (c) follows from (a) and (b) by Kleene's theorem (Proposition 7.16).

There is one more corollary worth setting down, but before stating it we introduce some traditional terminology. We use 'Γ *proves* D', written $\Gamma \vdash D$ or $\vdash_\Gamma D$, interchangeably with 'D is deducible from Γ'. The sentences proved by Γ we call the *theorems* of Γ. We conscript the word *theory* to mean a set of sentences *that contains all the sentences of its language that are provable from it*. Thus the theorems of a theory T are just the sentences in T, and $\vdash_T B$ and $B \in T$ are two ways of writing the same thing.

Note that we do not require that any subset of a theory T be singled out as 'axioms'. If there *is* a recursive set Γ of sentences such that T consists of all and only the sentences provable from Γ, we say T is *axiomatizable*. If the set Γ is finite, we say T is *finitely axiomatizable*. We have already defined a set Γ of sentences to be *complete* if for every sentence B of its language, either B or $\sim B$ is a consequence of Γ, or equivalently, is provable from Γ. Note that for a *theory* T, T is complete if and only if for every sentence B of its language, either B or $\sim B$ is in T. Similarly, a set Γ is *consistent* if not every sentence is a consequence of Γ, so a *theory* T is consistent if not every sentence of its language is in T. A set Γ of sentences is *decidable* if the set of sentences of its language that are consequences of Γ, or equivalently, are proved by Γ, is recursive. Note that for a *theory* T, T is decidable if and only if T is recursive. This terminology is used in stating our next result.

15.7 Corollary. Let T be an axiomatizable theory. If T is complete, then T is decidable.

Proof: Throughout, 'sentence' will mean 'sentence of the language of T'. The assumption that T is an axiomatizable theory means that T is the set of sentences provable from some recursive set of sentences Γ. We write T^* for the set of *code numbers of* theorems of T. By Corollary 15.4, T^* is semirecursive. To show that T is decidable we need to show that T^* is in fact recursive. By Proposition 15.2, T^* will be so if it is simply the set of *all* code numbers of sentences, so let us consider the case where this is not so, that is, where not every sentence is a theorem of T. Since every sentence *would be* a theorem of T if for any sentence D it happened that both D and $\sim D$ were theorems of T, for no sentence D can this happen. On the other hand, the hypothesis that T is complete means that for every sentence D, at least one of D and $\sim D$ is a theorem of T. It follows that D is not a theorem of T if and only if $\sim D$ *is* a theorem of T. Hence the complement of T^* is the union of the set X of those numbers n that are not code numbers of sentences at all, and the set Y of code numbers of sentences whose negations are theorems of T, or in other words, the set of n such that neg(n) is in T^*. X is recursive by Proposition 15.2. Y is semirecursive, since it is obtainable by substituting the recursive function neg in the semirecursive set T^*. So the complement of T^* is semirecursive, as was T^* itself. That T^* is recursive follows by Kleene's theorem (Proposition 7.16).

It 'only' remains to prove Propositions 15.2 and 15.3. In proving them we are once again going to be presenting evidence for Church's thesis: we are one more time going to be showing that certain sets and functions that must be recursive if Church's thesis is correct are indeed recursive. Many readers may well feel that by this point they have seen enough evidence, and such readers may be prepared simply to take Church's thesis on trust in future. There is much to be said for such an attitude, especially since giving the proofs of these propositions requires going into details about the Gödel numbering, the scheme of coding sequences, and the like, that we have so far largely avoided; and it is very easy to get bogged down in such details and lose sight of larger themes. (There is serious potential for a woods–trees problem, so to speak.) Readers who share the attitude described are therefore welcome to postpone *sine die* reading the proofs that fill the rest of this chapter. Section 15.2 concerns (the deferred clause of Proposition 15.1 as well as) Proposition 15.2, while section 15.3 concerns Proposition 15.3.

15.2* Gödel Numbers

We next want to indicate the proof of Proposition 15.2 (also indicating, less fully, the proof of the one deferred clause of Proposition 15.1, on the operation of substituting a term for the free occurrences of a variable in a formula). The Gödel numbering we gave by way of illustration near the beginning of this chapter is not, in fact, an especially convenient one to work with here, mainly because it is not so easy to show that such functions as the one that gives the the length (that is, number of symbols) in the expression with a given code number are primitive recursive. An alternative way of assigning code numbers to expressions begins by assigning code numbers to symbols as in Table 15-2.

Table 15-2. *Gödel numbers of symbols (second scheme)*

Symbol	(	)	,	$\sim$	$\vee$	$\exists$	$=$	v_i	A^n_i	f^n_i
Code	1	3	5	7	9	11	13	$2 \cdot 5^i$	$2^2 \cdot 3^n \cdot 5^i$	$2^3 \cdot 3^n \cdot 5^i$

Thus for the language of arithmetic $<$ or A^2_0 has code number $2^2 \cdot 3^2 \cdot 5^0 = 4 \cdot 9 = 36$, $\mathbf{0}$ or f^0_0 has code number $2^3 \cdot 3^0 \cdot 5^0 = 8$, $\prime$ or f^1_0 has code number $2^3 \cdot 3^1 \cdot 5^0 = 8 \cdot 3 = 24$, $+$ or f^2_0 has code number $2^3 \cdot 3^2 \cdot 5^0 = 8 \cdot 9 = 72$, and similarly $\cdot$ has code number 360. We then extend the code numbering to all finite sequences of symbols by assigning to an expression E consisting of a sequence of symbols $S_1 S_2 \cdots S_n$ the code number $\#(E)$ for the sequence $(|S_1|, |S_2|, \ldots, |S_n|)$ according to the scheme for coding finite sequences of numbers by single numbers based on prime decomposition. [In contrast to the earlier scheme, we need to distinguish, in the case of the expression consisting of a single symbol S, the code number $\#(S)$ of S *qua* expression from the code number $|S|$ of S *qua* symbol. In general the code number for a single-term sequence (n) is $2 \cdot 3^n$, so we get $\#(S) = 2 \cdot 3^{|S|}$.] Thus the code number for the sentence we have been writing $\mathbf{0} = \mathbf{0}$, which is officially $=(\mathbf{0}, \mathbf{0})$, is that for (13, 1, 36, 5, 36, 3), which is $2^6 \cdot 3^{13} \cdot 5 \cdot 7^{36} \cdot 11^5 \cdot 13^{36} \cdot 17^3$. This is a number of 89 digits. Fortunately, our concern will only be with what kinds of calculations could in principle be performed with such large numbers, not with performing such calculations in practice.

The calculation of the length $\mathrm{lh}(e)$ of the expression with code number e is especially simple on this scheme, since $\mathrm{lh}(e) = \mathrm{lo}(e, 2)$, where lo is the logarithm function in Example 7.11, or in other words, the exponent on the prime 2 in the prime decomposition of e. What are not so easy to express as primitive recursive functions on this coding scheme are such functions as the one that gives the code number for the concatenation of the expressions with two given code numbers. But while such functions may not have been so easy to prove primitive recursive, they *have* been proved to be so in Chapter 7. We know from our work there that in addition to the concatenation function $*$, several further *cryptographic* or code-related functions are primitive recursive. Writing $\#(\sigma)$ for the code number of sequence σ, and $\S(s)$ for the sequence with code number s, we list these functions in Table 15-3.

Table 15-3. *Cryptographic functions*

$\mathrm{lh}\,(s)$	$=$ the length of $\S(s)$
$\mathrm{ent}(s, i)$	$=$ the ith entry of $\S(s)$
$\mathrm{last}(s)$	$=$ the last entry of $\S(s)$
$\mathrm{ext}(s, a)$	$= \#(\S(s)$ with a added at the end$)$
$\mathrm{pre}(a, s)$	$= \#(\S(s)$ with a added at the beginning$)$
$\mathrm{sub}(s, c, d)$	$= \#(\S(s)$ with c replaced by d throughout$)$

More complicated objects, such as finite sequences or finite sets of expressions, can also be assigned code numbers. A code number for a finite sequence of expressions is simply a code number for a finite sequence of natural numbers, whose entries are themselves in turn code numbers for expressions. As a code number for a finite *set* of expressions, we may take the code number for the finite sequence of expressions that list the elements of the set (without repetitions) *in order of increasing code number*.

This means that a code number of a finite set of expressions will be a code number for a finite sequence of expressions *whose entries are increasing*, with later entries larger than earlier ones. A virtue of this coding is that such relations as 'the expression with code number i belongs to the set with code number s' and 'the set with code number t is a subset of the set with code number s' will all be simply definable in terms of the cryptographic functions, and hence recursive. (The first amounts to 'i is an entry of the sequence coded by s', and the second amounts to 'every entry of the sequence coded by t is an entry of the sequence coded by s'.) Similarly the coding can be extended to finite sequences or finite sets of finite sequences or finite sets, and so on.

Towards proving Proposition 15.2, the first thing to note is that one- and two-place relations like those given by 'a is the code number of a predicate' and 'a is the code number of an n-place predicate' are primitive recursive. For the former is equivalent to the existence of n and i such that $a = 2^2 \cdot 3^n \cdot 5^i$, and the latter is equivalent to the existence of i such that $a = 2^2 \cdot 3^n \cdot 5^i$. The function f given by $f(n, i) = 2^2 \cdot 3^n \cdot 5^i$ is primitive recursive, being a composition of exponentiation, multiplication, and the constant functions with values 2^2, 3, and 5. So the relation '$a = 2^2 \cdot 3^n \cdot 5^i$' is primitive recursive, being the graph relation '$a = f(n, i)$'. The two relations of interest are obtained from the relation '$a = 2^2 \cdot 3^n \cdot 5^i$' by existential quantification, and in each case the quantifiers can be taken to be *bounded*, since if $a = 2^2 \cdot 3^n \cdot 5^i$, then certainly n and i are less than a. So the first condition amounts to $\exists n < a$ $\exists i < a(a = 2^2 \cdot 3^n \cdot 5^i)$ and the second to $\exists i < a(a = 2^2 \cdot 3^n \cdot 5^i)$.

Similar remarks apply to 'a codes a variable', 'a codes a function symbol', and 'a codes a constant (that is, a zero-place function symbol)', 'a codes an n-place function symbol', and 'a codes an atomic term (that is, a variable or constant)'. These all give primitive recursive relations. If we are interested only in formulas and sentences of some language L less than the full language containing all nonlogical symbols, we must add clauses 'and a is in L' to our various definitions of the items just listed. So long as L is still primitive recursive, and in particular if L is finite, the relations just listed will still be primitive recursive. (If L is only recursive and not primitive recursive, we have to change 'primitive recursive' to 'recursive' both here and below.)

Considering only the case without identity and function symbols, the relation given by 's codes an atomic formula' is also primitive recursive, being obtainable by simple operations (namely, substitution, conjunction, and bounded universal quantifications) from the relations mentioned in the preceding paragraph and the graph relations of the primitive recursive functions of some of the cryptographic functions listed earlier. Specifically, s codes an atomic formula if and only if there is an n less than lh(s) such that the following holds:

> lh $(s) = 2n + 2$, and
>
> ent $(s, 0)$ is the code number for an n-place predicate, and
>
> ent $(s, 1) = 1$ (the code number for a left parenthesis), and
>
> for every i with $1 < i <$ lh $(s) - 1$:
>
> > if i is odd then ent$(s, i) = 5$ (the code number for a comma), and
> >
> > if i is even then ent(s, i) is the code number for an atomic term, and
>
> last $(s) = 3$ (the code number for a right parenthesis).

Now s is the code number of a formula S if and only if there is some r that is the code number for a formation sequence for S. In general, the relation given by 'r is the code number of a formation sequence for a formula with code number s' is primitive recursive, since this relation holds if and only if the following does:

```
For all $j < \mathrm{lh}(r)$ either:
    $\mathrm{ent}\,(r, j)$ is the code number for an atomic sentence, or
    for some $k < j$,
        $\mathrm{ent}\,(r, j) = \mathrm{neg}(\mathrm{ent}\,(r, k))$, or
    for some $k_1 < j$ and some $k_2 < j$,
        $\mathrm{ent}\,(r, j) = \mathrm{disj}(\mathrm{ent}\,(r, k_1),\ \mathrm{ent}\,(r, k_2))$, or
    for some $k < j$ and some $i < \mathrm{ent}\,(r, j)$,
    $\mathrm{ent}\,(r, j) = \mathrm{exquant}\,(2 \cdot 5^i,\ \mathrm{ent}\,(r, k))$
and $\mathrm{last}\,(r) = s$.
```

Here neg, disj, and exquant are as in the proof of Proposition 15.1.

We can give a rough upper bound on the code number for a formation sequence, since we know (from the problems at the end of Chapter 9) that if S is a formula—that is, if S has any formation sequence at all—then S has a formation sequence in which every line is a substring of S, and the number of lines is less than the length of S. Thus, if there is any formation sequence at all for s, letting $n = \mathrm{lh}(s)$, there will be a formation sequence for s of length no greater than n with each entry of size no greater than s. The code number for such a formation sequence will therefore be less than the code number for a sequence of length n all of whose entries are s, which would be $2^n \cdot 3^s \cdot \cdots \cdot \pi(n)^s$, where $\pi(n)$ is the nth prime, and this is less that $\pi(n)^{s(n+1)}$. So there is a primitive recursive function g, namely the one given by $g(x) = \pi(\mathrm{lh}(x))^{x[\mathrm{lh}(x)+1]}$, such that if s is the code number for a formula at all, then there will be an $r < g(s)$ such that r is a code number for a formation sequence for that formula. In other words, the relation given by 's is the code number for a formula' is obtainable by bounded quantification from a relation we showed in the preceding paragraph to be primitive recursive: $\exists r < g(s)$ (r codes a formation sequence for s). Thus the relation 's is the code number for a formula' is itself primitive recursive.

In order to define sentencehood, we need to be able to check which occurrences of variables in a formula are bound and which free. This is also what is needed in order to define the one operation in Lemma 15.1 whose proof we deferred, substitution of a term for the *free* occurrences of a variable in a formula. It is not the substitution itself that is the problem here, so much as recognizing which occurrences of the variable are to be substituted for and which not. The relation 's codes a formula and the eth symbol therein is a free occurrence of the dth variable' holds if and only if

```
$s$ codes a formula and $\mathrm{ent}\,(s, e) = 2 \cdot 5^d$ and
for no $t, u, v, w < s$ is it the case that
    $s = t * v * w$ and $\mathrm{lh}\,(t) < e$ and $e < \mathrm{lh}(t) + \mathrm{lh}(v)$ and
    $u$ codes a formula and $v = \mathrm{exquant}\,(2 \cdot 5^d, u)$.
```

For the first clause says that s codes a formula and the eth symbol therein is the dth variable, while the second clause says that the eth symbol does not fall within any subsequence v of the formula that is itself a formula beginning with a quantification of the dth variable. This relation is primitive recursive. Since the relation 's codes a sentence' is then simply

> s codes a formula and
>
> for no $d, e < s$ is the eth symbol therein a free occurrence of the dth variable

it is primitive recursive, too, as asserted.

So much for the proof in the case where identity and function symbols are absent. If identity is present, but not function symbols, the definition of atomic formula will be the disjunction of the clause above covering atomic formulas involving a nonlogical predicate with a second, similar but simpler, clause covering atomic formulas involving the logical predicate of identity. If function symbols are present, it will be necessary to give a preliminary definitions of *term formation sequence* and *term*. The definition for term formation sequence will have much the same gross form as the definition above of formation sequence; the definition for term will be obtained from it by a bounded existential quantification. We suppress the details.

15.3* More Gödel Numbers

We indicate the proof of Proposition 15.3, for the proof procedure used in the preceding chapter, only in gross outline. Something similar can be done for any reasonable proof procedure, though the details will be different.

We have already indicated how sets of sentences are to be coded: s is a code for a set of sentences if and only if s is a code for a sequence and for all $i < \mathrm{lh}(s)$, $\mathrm{ent}(s, i)$ is a code for a sentence, and in addition for all $j < i$, $\mathrm{ent}\,(s, j) < \mathrm{ent}(s, i)$. It follows that the set of such codes is primitive recursive. A derivation, on the approach we took in the last chapter, is a sequence of sequents $\Gamma_1 \Rightarrow \Delta_1$, $\Gamma_2 \Rightarrow \Delta_2$, and so on, subject to certain conditions. Leaving aside the conditions for the moment, a sequence of sequents is most conveniently coded by a code for $(c_1, d_1, c_2, d_2, \ldots)$, where c_i codes Γ_i and d_i codes Δ_i. The set of such codes is again primitive recursive. The sequence of sequents coded by the code for $(c_1, d_1, \ldots, c_n, d_n)$ will be a deduction of sentence D from set Γ if and only if: first, the sequence of sequents coded is a derivation; and second, c_n codes a sequences whose entries are all codes for sentences in Γ, and d_n codes the sequence of length 1 whose sole entry is the code for D. Assuming Γ is recursive, the second condition here defines a recursive relation.

The first condition defines a primitive recursive set, and the whole matter boils down to proving as much. Now the sequence of sequents coded by a code for $(c_1, d_1, \ldots, c_n, d_n)$ will be derivation if for each $i \leq n$, the presence of c_i and d_i is justified by the presence of zero, one, or more earlier pairs, such that the sequent $\Gamma_i \Rightarrow \Delta_i$ coded by c_i and d_i follows from the sequents $\Gamma_j \Rightarrow \Delta_j$ coded by these earlier $c_j$and d_j according to one or another rule. In gross form, then, the definition of coding a derivation will resemble the definition of coding a formation sequence, where the presence of any code for an expression must be justified by the presence of zero, one, or more earlier codes for expressions from which the given expression 'follows' by or another 'rule' of formation. The rules of formation are just the rules the zero-'premiss' rule allowing atomic formulas to appear, the one-'premiss' rule allowing a negation to be 'inferred' from the expression it negates, the two-'premiss' rule allowing a disjunction to be 'inferred' from the two expressions it disjoins—and so on. Definitions of this gross form define primitive recursive relations, provided the individual rules in them do.

So, going back to derivations, let us look at a typical one-premiss rule. (The zero-premiss rule would be a bit simpler, a two-premiss rule a bit more complicated.) Take

$$\text{(R2}a\text{)} \qquad \frac{\Gamma \cup \{A\} \Rightarrow \Delta}{\Gamma \Rightarrow \{\sim A\} \cup \Delta}.$$

The relation we need to show to be primitive recursive is the relation 'e and f code a sequent that follows from the sequent coded by c and d according to (R2a)'. But this can be defined as follows:

$$\begin{aligned}
&c, d, e, f \text{ code sets of formulas, and } \exists a < \text{lh}(c) \quad \exists b < \text{lh}(f)\\
&\quad \text{ent}(f, b) = \text{neg}(\text{ent}(c, a)), \text{ and}\\
&\quad \forall i < \text{lh}(c)\ (i = a \text{ or } \exists j < \text{lh}(e)\ \text{ent}(c, i) = \text{ent}(e, j)), \text{ and}\\
&\quad \forall i < \text{lh}(e) \quad \exists j < \text{lh}(c)\ \text{ent}(e, i) = \text{ent}(c, j), \text{ and}\\
&\quad \forall i < \text{lh}(d) \quad \exists j < \text{lh}(f)\ \text{ent}(d, i) = \text{ent}(f, j), \text{ and}\\
&\quad \forall i < \text{lh}(f)(i = b \text{ or } \exists j < \text{lh}(d)\ \text{ent}(f, i) = \text{ent}(d, j)).
\end{aligned}$$

Here the last four clauses just say that the only difference between the sets coded by c and e is the presence of the sentence A coded by ent(c, a) in the former, and the only difference between the sets coded by d and f is the presence of the sentence B coded by ent(f, b) in the latter. The second clause tells us that $B = \sim A$. This is a primitive recursive relation, since we known neg is a primitive recursive function.

To supply a full proof, each of the rules would have to be analyzed in this way. In general, the analyses would be very similar, the main difference being in the second clauses, stating how the 'exiting' and 'entering' sentences are related. In the case we just looked at, the relationship was very simple: one sentence was the negation of the other. In the case of some other rules, we would need to know that the function taking a formula $B(x)$ and a closed term t to the result $B(t)$ of substituting t for all the free occurrences of x is recursive, or rather, that the corresponding function on codes is. We suppress the details.

Problems

15.1 On the first scheme of coding considered in this chapter, show that the *length* of, or number of symbols in, the expression with code number e is obtainable by a primitive recursive function from e.

15.2 Let Γ be a set of sentences, and T the set of sentences in the language of Γ that are deducible from Γ. Show that T is a theory.

15.3 Suppose an axiomatizable theory T has only infinite models. If T has only one isomorphism type of denumerable models, we know that it will be complete by Corollary 12.17, and decidable by Corollary 15.7. But suppose T is *not* complete, though it has only *two* isomorphism types of denumerable models. Show that T is still decidable.

15.4 Give examples of theories that are decidable though not complete.

15.5 Suppose $A_1, A_2, A_3, \ldots$ are sentences such that no A_n is provable from the conjunction of the A_m for $m < n$. Let T be the theory consisting of all sentences provable from the A_i. Show that T is not finitely axiomatizable, or in other words, that there are not some other, finitely many, sentences $B_1, B_2, \ldots, B_m$ such that T is the set of consequences of the B_j.

15.6 For a language with, say, just two nonlogical symbols, both two-place relation symbols, consider interpretations where the domain consists of the positive integers from 1 to n. How many such interpretations are there?

15.7 A sentence D is *finitely* valid if every finite interpretation is a model of D. Outline an argument *assuming Church's thesis* for the conclusion that the set of sentences that are *not* finitely valid is semirecursive. (It follows from Trakhtenbrot's theorem, as in the problems at the end of chapter 11, that the set of such sentences is *not* recursive.)

15.8 Show that the function taking a pair consisting of a code number a of a sentence A and a natural number n to the code number for the conjunction $A \,\&\, A \,\&\, \cdots \,\&\, A$ of n copies of A is recursive.

15.9 *The Craig reaxiomatization lemma* states that any theory T whose set of theorems is semirecursive is axiomatizable. Prove this result.

15.10 Let T be an axiomatizable theory in the language of arithmetic. Let f be a one-place total or partial function f of natural numbers, and suppose there is a formula $\phi(x, y)$ such that for any a and b, $\phi(\mathbf{a}, \mathbf{b})$ is a theorem of T if and only if $f(a) = b$. Show that f is a recursive total or partial function.