Math Logic Notes

Ted Sider

Notes for Boolos & Jeffrey, *Computability and Logic*

CHAPTER 1: Enumerability

I. Lists (the basic idea)

I'm going to assume that people are basically familiar with talk of sets. Sets are things that have members; there is such a thing as the empty set, etc.

The main concept of the first chapter is that of an "enumerable set". The rough idea is that a set is enumerable if you can "list" its members:

Definition of a list (informal)

i) a list has a beginning
ii) list may be infinite or finite, BUT
iii) every member of the list must be some finite distance from the beginning -- i.e. every entry must be the *n*th entry, for some finite *n*.

For example, ...-2, -1, 0, 1, 2, ... is no good; neither is 0,2,4,6,....1,3,5,...

Notice a couple things about this last non-list. Even though this won't do, we *can* come up with a genuine list of the very same numbers -- simply rearrange as follows: 0,1,2,3,4.... Therefore this set (the set of natural numbers) comes out enumerable, since there is *a* way it can be listed; it doesn't matter if there exist some failed attempts at listing the set. Likewise, the first non-list of the set of all integers, both negative and positive, can be converted into a list thus: 0, 1, -1, 2, -2, ...

Notice also that there can be many ways to list the same set:

0,3,2,1,6,5,4,9,8,7,....
0,1,2,3,4,5,6,7,8,9,....

These listings, or *sequences*, are different things from sets. A set doesn't have an order, there's no such thing as the "third" member of a set, and so on. Whereas sequences *do* have order.

A couple other qualifications of our concept of a list:

iv) We allow repeats. So 0,1,2,2,3,3,3,4,4,4,4,... counts as a list of the non-negative integers, just as legit as 0,1,2,....

v) We allow gaps. So 0,-,1,-,2,- ... counts.

II. Examples of enumerable sets

In a bit I'll introduce a more rigorous definition of "enumerable", but our intuitive definition we've got suffices to show a couple (somewhat) surprising things.

A. The union of two enumerable sets is enumerable

Suppose we've got two enumerable sets, A and B. We can show easily that their union is enumerable. Since A is enumerable, we can list its members; let a1, a2,... be a list of its members. Likewise for b1, b2, ... OK, now construct the list for the union of A and B as follows: a1, b1, a2, b2, ... Notice that it's no good to first list all the members of A, then all the members of B.

This is the first example of the kinds of proofs we'll be doing in this course. This gives a sense of what counts as a proof. This is a usual mathematical proof — *not* a derivation in a formalized system. Rather, it is a perfectly rigorous, though non-formalized, *argument* establishing a certain conclusion. Note its structure. We want to show that the union of *any* two enumerable sets is enumerable. So we proceeded by saying: let A and B be any enumerable sets. The definition of an enumerable set is that there's some list of its members. So then we produce an enumeration of the union $A \cup B$ — thus satisfying the definition of $A \cup B$'s being enumerable.

B. The union of an enumerable set of enumerable sets is enumerable

Now an even trickier example: suppose we have an enumerable set of enumerable sets. How do we approach this?

The first step in doing this sort of proof is to get clear what information is provided in the assumptions of the theorem, and to get clear on what the theorem is saying.

So, we are given an arbitrary enumerable set of enumerable sets. What is that?

Well, first of all, we have a set — call it A — that is enumerable. That means that its members can be listed thus: a_1, a_2, ... Secondly, the members of A — i.e., the

a_i's — are themselves enumerable sets. I.e., the members of each a_i can be listed thus: a_{i1}, a_{i2}, ... So that is what we are given in the "set-up" of the theorem.

We now need to get clear on what we are trying to establish. We're trying to establish that the union of A is enumerable. What is the union of A?

The union of a set of sets is the set of things that are in the members of that set — i.e., $\cup A = \{x : x \in$ some member of A$\}$.

What are we supposed to show about $\cup A$? That it is enumerable. What does this require? Producing an enumeration of this set.

This, now, requires ingenuity. Here is the standard way of doing it. We represent the members of $\cup A$ on a two-dimensional grid thus:

a_1:	a_{11}	a_{12}	a_{13}	.	.	.
a_2:	a_{21}	a_{22}	a_{23}	.	.	.
a_3:	a_{31}	a_{32}	a_{33}	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.

Given this grid, we can then specify a method for enumerating $\cup A$. We simply sweep through the array in such a way that it is clear that we eventually get to each point in the array after some finite amount of time. For example, the enumeration could work like this:

a_{11}, a_{12},a_{21},a_{13},a_{22},a_{31},...

We could give an algebraic account of what we've done. (For each i=2,3,..., we placed onto our list all those entries whose indices add up to i, in lexicographic order. This gives a sequential procedure in which at each stage, only finitely many entries are added to the list.) Or we could stick with the geometrical description of a procedure for generating the list. Either is fine.

C. The set of positive rational numbers is enumerable

Finally, show that the set of positive rationals is enumerable. The positive rational numbers are the fractions -- anything that has the form n/m, where n and m are positive integers. It seems a bit surprising that this set is enumerable, because the rationals are dense -- between any two there are infinitely many --, but it is. For consider the following method:

<the grid>

notice that we'll get repeats, but that's ok.

III. A more rigorous definition of 'enumerable'; functions

We'll need to introduce some concepts involving *functions*. A function is a rule that associates *values* with *arguments*, in such a way that no two values are assigned to the same argument. Let f be a function; when the value of f for argument x is y, we write f(x)=y. The defining condition of a function is that we can never have f(x)=y and f(x)=z unless z=y.

Note, though, that it can happen that f(x)=f(y) even though x≠y. For example, consider the function f(n)=1. In that case, the function is not called "one-to-one"; if this *never* happens — i.e., if whenever f(x)=f(y) then x=y — then the function is called one-to-one.

A function needn't give a value for every argument -- a function can be undefined for a given argument. For example, the reciprocal function r (r(6)=1/6; r(1/6)=6, etc.) is undefined for zero, since zero doesn't have a reciprocal. And it certainly isn't defined for me -- people don't have reciprocals.

The set of all arguments for which a function is defined ({x: there is some y such that f(x)=y}) is called its *domain*. The *range* of a function is the set of all its values ({x: there is some y such that f(y)=x}). When the range of a function is set A, we say that the function is *onto* A. When the range is some subset of A (perhaps all of A; perhaps not), we say that the function is *into* A.

Functions don't need to be only numeric. The father-of function is defined on people, in that its domain is all persons, and its range is a subset of the set of persons consisting of all males that have offspring. But we'll often be talking about functions defined on positive integers. When we do so, we'll have some additional terminology: if the domain of such a function is P, the entire set of positive integers, then we'll say the function is *total*; otherwise it is *partial* (we'll count the function whose domain is the empty set as a partial function on integers, even though it isn't defined for any integer.) We're now in a position to restate our **definition**:

> Set A is enumerable iff there is some function from positive integers (whether total or partial; whether or not one-to-one) onto A

This, then, is the official definition of enumerability. But we have the question: what counts (for the purposes of this class) as a demonstration that a given set is enumerable? That is, what counts as an acceptable demonstration that a function from positive integers onto the set in question exists? The demonstration may be given in various ways. One

way is to give a "mathematical definition". This is easy in some cases, such as showing that the set of perfect squares is enumerable: $f(n)=n^2$. But in other cases it's harder, such as the proof that the set of positive rationals was enumerable; in that case, we used a picture to show our function (list). That's ok too.

IV. Examples

A. Any set of positive integers is enumerable

i.e., Show that any subset of P is enumerable. Can anybody think of a simple way to enumerate this set? (f(n)=n if n is in S; f(n) is undefined otherwise).

B. Any subset of an enumerable set is enumerable

This is a generalization of the previous example. Here we just work through the definitions. If A is enumerable, then it has an enumeration: a_1, a_2, ... Now just use the same enumeration, but leave gaps whenever a_1 is not in the subset of A.

We can think of this in terms of functions. Suppose that f enumerates A, and $B \subseteq A$. Then define f′ as follows

$$\begin{aligned} f'(i) &= f(i) \text{ if } f(i) \in B \\ &= \text{undefined otherwise} \end{aligned}$$

This new function, f′, enumerates B.

C. Is the successor function (on positive integers): into P? Onto P? one-one? Total?

D. The set of all grammatical sentences of English is enumerable

Begin with a finite alphabet. Enumerate all finite strings of this alphabet, beginning with length 1 strings, then length 2 strings... In each case, use alphabetical order. Then delete all the ungrammatical ones.

V. Cardinality

Now we begin to make precise the idea that two sets have the "same size":

A and B are *equinumerous*, or *have the same cardinality*, or are *the same size*, iff there is

a one-one function whose range is A and whose domain is B.

A has *greater cardinality than* , or is *larger than*, B, iff B is not equinumerous with A, but is equinumerous with some subset of A.

This seems intuitive correct in the case of finite sets -- e.g., the sets {a,b,c} and {1,2,3} are equinumerous in virtue of the function f(a)=1, f(b)=2, f(c)=3. But we get some odd results in the case of infinite sets, for we can show that an infinite set is equinumorous with one of its proper subsets. For example, we can show that P is equinumorous with the set of even positive integers. And that seems weird, because in one sense it seems like there are fewer even integers than integers.

(What I think is really going on here is that our ordinary concept of "fewer than" is based on thinking about finite sets. in the case of finite sets, we have several tests for whether one set is smaller than another, and the tests are equivalent for finite sets. For example, for finite sets, a set is always larger than its proper subsets; and sets are the same size if they can be put in 1-1 correspondence. But these tests come apart in the infinite case. Moreover, no one of them is clearly the one and only thing we meant by "fewer", "same size as", etc.. so what we have is a number of ways one could generalize our concept of fewer to the infinite case. mathematicians use the 1-1 correspondence method.)

The concept of equinumerosity gives us another way to show that a set is enumerable, for if a set is equinumerous with any set of positive integers, then it must be enumerable.

If S is enumerable, then so is any set equinumerous with S. (Use function composition.)

VI. Recursion and induction

This is a small mathematical aside, because it may help with some of the homework problems.

A. Recursive definition

A very useful method for defining a function is a *recursive* definition. Here is an example:

$0! = 1$
$(n+1)! = (n+1) \cdot n!$

Note that these definitions are in a sense circular: one defines factorial in terms of itself. But one always defines the factorial of a number in terms of the factorial of a smaller number, and this ultimately "bottoms out" with the base case: one

defines the factorial of 0 non-circularly.

A non-mathematical example of a recursive definition is the familiar definition of truth in an interpretation in propositional logic. Let an *interpretation* be any function that assigns a truth value (T or F) to each sentence letter. Then we say:

> Recursive definition of truth relative to an interpretation, I:
>
> - if ϕ is atomic, ϕ is true if $I(\phi)=T$; otherwise ϕ is false
> - $\phi\&\psi$ is true iff ϕ is true and ψ is true; otherwise $\phi\&\psi$ is false
> - $\phi\vee\psi$ is true iff either ϕ is true or ψ is true; otherwise $\phi\vee\psi$ is false
> - $\phi\rightarrow\psi$ is false iff ϕ is true and ψ is false; otherwise $\phi\rightarrow\psi$ is true
> - $\phi\leftrightarrow\psi$ is true iff ϕ and ψ have the same truth value; otherwise $\phi\leftrightarrow\psi$ is false

Here the first clause is the base case; the other clauses define the truth value of a complex formula in terms of the truth values of its parts.

Here's another example. Suppose we are given a total function of positive integers, f. Suppose we want to define a new function, g, with the following properties:

> g is an *increasing* function — i.e., for any i, j; if i>j then g(i)>g(j)
> g *dominates* f — i.e., for any i, g(i)>f(i)

How do we do it? It's easy to get an increasing function — the identity function g(i)=i is an increasing function. And it's easy to get a function that dominates f — g(i)=f(i)+1. But how can we get a function that satisfies both conditions?

One way is to use recursion:

> g(1) = f(1)+1
> g(i+1) = f(i) + g(i)

B. Inductive Proof

Recursion is a method for *defining* something. There is a related method for *proving* a statement: induction.

Here is the basic idea. Suppose we want to prove something of this form:

Every number, n, has a certain property F (i.e., "for all n, F(n)")

The inductive proof requires proving two separate things. First, we prove:

F(1)

i.e., the number 1 has the property. This is the "base step". Second, we prove the following conditional statement is true:

For any n, if F(n) then F(n+1)

This is the "induction step". If we can prove these two things, then the principle of mathematical induction says that we may conclude that for all n, F(n). The idea is that the conditional shows that the property F is *inherited*: if it holds for a number, it must hold for the next number. But if 1 has the property, then all the rest of the numbers must have it too.

Example: prove that every number is either even or odd, assuming that an even number is defined as one that = 2n, for some positive integer n, and that an odd number is defined as one that is 2n-1, for some positive integer n. Well, the base is easy. We must prove that 1 is either even or odd. 1 is odd, since it is $2 \cdot 1 - 1$. Now for the induction step. To prove this conditional, we must make the *inductive hypothesis*:

(ih) n is either odd or even

We must now prove that n+1 is either odd or even. If we succeed, then we're done. OK, here we separate cases:

Case 1: n is odd. That means that n=2m-1, for some m. But then, n+1=2m. So n+1 is even, and hence is either odd or even.

Case 2: n is even. That means that n=2m, for some m. But then n+1=2m+1=2m+2-1=2(m+1)-1, and so n+1 is odd, and so even or odd

In either case, n+1 is either even or odd, given (ih). QED

Inductive proof is often a particularly powerful form of proof when combined with recursive definition. When the statement to be proved contains terms that were defined recursively, often the proof of the base case is based on the base case of the recursive definition, and the proof of the induction step is based on the recursive part of the definition.

VII. One other comment for the homework

It's important to distinguish these statements:

> The set of all sets of a certain kind K is enumerable
> Every set of kind K is enumerable

These are very different statements. To establish the first, you need to produce an enumeration of a certain set. Its members are sets of kind K, but what you are listing is the sets of kind K, not the members of sets of kind K. Whereas to establish the second statement, you must show that every set of kind K can be enumerated — there you must list the members of sets of kind K.

CHAPTER 2: Diagonalization

I. Non-enumerable sets

Every finite set is enumerable. But, as we'll now show, there are infinite sets that aren't enumerable -- this then implies that there are infinite sets that aren't the same size as the set of positive integers. We'll also show that these sets are larger than the set of positive integers.

II. The power set of P (the set of positive integers) is *not* enumerable

Call the power set of P, P^*. We first need to get clear what P is. It is the set of all P's subsets — i.e., $\{x: x \subseteq P\}$.

What are we trying to show? That this set is not enumerable. How could we possibly show that? We know how to show that a set *is* enumerable: we produce an enumeration. But how could we show that a set is *not* enumerable? By showing that a contradiction would follow from the supposition that there *was* an enumeration of the set.

So, suppose for reductio that P^* is enumerable. Then it has an enumeration — a list, L. What are the members of L like? Each one is a set of positive integers. Now, we can represent these members of L on a 2D array as follows:

	1	2	3	.	.
L_1:	L_{11}	L_{12}	L_{13}	.	.
L_2:	L_{21}	L_{22}	L_{23}	.	.
L_3:	L_{31}	L_{32}	L_{33}	.	.
.	.	.	.	.	.

. . . . . .

What I have in mind here is this. In each case, L_{ij} is to be the number j, if j is a member of L_i; otherwise L_{ij} is blank. Thus, the members of each L_i are listed in order, with blanks whenever a number is missing from L_i.

Now we use a neat trick. Consider the sequence of members along the *diagonal* of this array: L_{11}, L_{22}, L_{33}, ... Now consider coming up with a *new* sequence, L^-, the antidiagonal sequence, as follows:

L^-_i = i, if L_{ii} is blank
= blank otherwise

In other words, we run through the diagonal sequence L_{11}, L_{22}, L_{33}, ..., switching blanks to numbers and vice versa.

Let A be the set of numbers listed in L^- — i.e., the range of the function L^-. A is a set of numbers (possibly empty, of course). It is therefore a subset of P. So if the original enumeration L really was an enumeration of P^*, then A must show up on L somewhere. Now, each member of a list is at some finite distance from the beginning; thus, for some i, $A=L_i$.

But given the way we defined A, this can't be. Look at how we represented each L_i on the array: we listed it in order. But A *differs* from L_i at its i^{th} place — i.e., the place where the listing of L_i intersects with the diagonal. If the integer i was a member of L_i, then the anti-diagonal sequence got a blank in its i^{th} spot. Moreover, the number i never could have gotten into the anti-diagonal sequence at any other spot, because each L_j was listed in order. So if $i \in L_i$, $i \notin A$. But now, on the other hand, suppose $i \notin L_i$. Then L_i had a blank at its i^{th} spot. In that case, i went into the anti-diagonal sequence, and so into A. So, if $i \notin L_i$, $i \in A$. Either way, A and L_i have different members, and so are different sets.

This means that A cannot be L_i, for *any* i (since i was arbitrarily chosen). But A is a set of integers, and so *must* show up on the enumeration of P^* somewhere, if there really were such as thing as this enumeration. We have arrived at a contradiction, based on the assumption that there is such a thing as an enumeration of P^*. Conclusion: P^* cannot be enumerated. It is "too big".

III. The set of real numbers between 0 and 1 (non-inclusive) is not enumerable

The same sort of proof is commonly used to give the proof that the set of real numbers between 0 and 1 is not enumerable. Thus, though the set of rational numbers is enumerable (and so is no "bigger" than the set of integers), the set of reals is "bigger"

than the set of integers.

Suppose for reductio that there is an enumeration of (0,1): r_1, r_2, ... Each of these real numbers can be represented as a possibly infinite decimal expansion:

r_1:	a_{11}	a_{12}	...
r_2:	a_{21}	a_{22}	...
.	.	.	.

We can now construct a real number not on the list: $0.d_1d_2...$, where the digits are gotten by "reversing" 7s and 8s (say) on the diagonal:

d_i = 7 if $a_{ii} \neq 7$
= 8 if $a_{ii}=7$

Thus, the set of real numbers between 0 and 1 cannot be placed upon any list. *Note*: this does *not* mean that there is some "unlistable" real number: some real number that is missing from any list. That would be absurd. What it means is that every list is missing some real number.

IV. The set of (total or partial) functions of positive integers is not enumerable

Suppose otherwise; suppose we have an enumeration f_i of these functions. These functions can be arranged on a 2D array in the usual way:

f_1:	$f_1(1)$	$f_1(2)$	.	.
f_2:	$f_2(1)$	$f_2(2)$	.	.
.	.	.	.	.
.	.	.	.	.

Note that the entries here are the *values* of the functions. Now let us construct a new function that is not on the list:

d(i) = 1, if $f_i(i)$ is undefined
= undefined if $f_i(i)$ is defined

This function d cannot be on the list, because it differs from each function f_i on the list in (at least) what it assigns to i.

CHAPTER 3: Turing Machines

I. Introduction

Our goal is to show that some tasks are not accomplishable by a mechanical procedure. More carefully, we will show that some *functions* (on integers) are not "effectively computable" — i.e., that there is no mechanical procedure that can be used to compute these functions.

Now, this statement is not capable of mathematical proof, because of the imprecision in the notion of a "mechanical procedure". What we will do is this. We will give a precise definition of a Turing Machine — a certain kind of simple computer — and we will then be able to prove that some functions are not computable by any Turing machine.

But what if the reason some functions are not computable by any Turing machines is just that we've defined Turing machines too weakly? Perhaps more powerful computers could do better where Turing machines fail. In fact, for any other sort of machine anyone has come up that intuitively computes in a mechanical way, Turing machines are at least as powerful.

Still, we can't mathematically prove that no more powerful sort of computing device will be devised. But it seems a reasonable belief that none ever will. This belief (or conjecture) is *Church's Thesis*: a function is intuitively computable iff it is Turing-computable.

II. Definition of a Turing Machine

A. Capabilities of Turing Machines

We have a *tape* and a *processor*. The processor can move around on the tape, and read and write things. Here are the constraints:

1. The *tape* extends infinitely to the right and to the left, and is marked into squares
2. At any point of time, there are only a finite number of squares that are non-blank
3. A non-blank square has a symbol on it from the list S_1-S_n (we can think of a blank square as having the symbol S_0 inscribed on it.)
4. The processor is always scanning exactly one square at any given time.
5. The processor can do the following things:
    a. Tell what is on the square it is scanning
    b. Erase whatever is on the square
    c. Write one of the symbols S_1-S_n
    d. Move 1 square to the left or right
    e. Halt

6. Each action of the processor takes the same amount of time (say, 1 second).

B. Turing Machine programs

When we speak of a "Turing machine", we usually have in mind a *program*. A program is a series of instructions for the processor, which tell it what to do depending on what it sees on the tape.

A program consists of a finite set of *states* $q_1 \ldots q_m$, which are the instructions. Each state is complex, and has the following two parts, which tell the machine what to do while it is *in* that state:

1. A specification of different actions dependent on what is on the tape
2. A specification of what state to enter next, also dependent on what is on the tape

Thus, when the machine is in a state, it first scans the tape, then depending on what it sees on the tape, it carries through whatever action the current state says should occur conditional on what is on the tape; then it enters whatever new state the old state says should follow, dependent on what it saw on the tape.

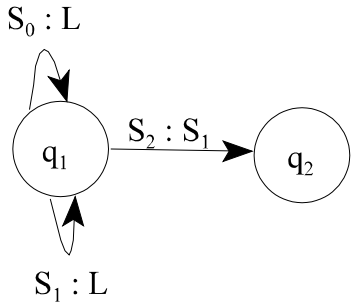
To be concrete, a certain state q_1 might look like this:

State q_1:
If the tape is blank then move once to the left and goto state q_1
If S_1 is on the tape, then move once to the left, and goto state q_1
If S_2 is on the tape then write S_1, and halt

Suppose we have a machine that only allows S_1s and S_2s. This state will move to the left until it sees an S_2, in which case it will erase the S_2 and write a S_1, and then halt. Note that if there is no S_2 to the left of the square the machine was initially scanning, it will move off to the left forever; it will "hang".

C. Flow Diagrams

This can be made more intuitive by means of a diagram.



Note the lack of arrows coming from state q_2. That's how we represent halting: a state with no specification of what to do.

D. Quadruples representation

Another way of representing programs, less intuitive but more theoretically useful (for some purposes) is using quadruples. A quadruple has this form:

$<q_i,S_j,a,q_k>$

Which means: "when in state q_i, if scanning symbol S_j, do action a, and then goto state q_k"

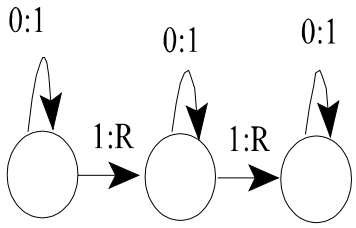
Our example program, represented in quadruples format, would look like this

$<q_1,S_0,L,q_1>$
$<q_1,S_1,L,q_1>$
$<q_1,S_2,S_1,q_2>$

E. Turing machine examples

Let's let the symbols just be the digits 1, 2, 3, ...; and let the blank be represented by 0. Let's assume the tape is empty other than the "input", that we start the machine on the left of the input, and let's make the machine halt after finishing its task.

1. Write three 1s on a blank tape, then halt



2. **Double the 1s**

This one is a bit trickier, and begins introducing some of the programming tactics required for Turing machines.

The reason this exercise is harder than the last is that the program is supposed to double the number of 1s, however many are there initially. That means that we can't "hard-wire" writing a certain number of 1s, as we did with the previous exercise. That machine was only supposed to write a fixed number of 1s on the tape. But here we are supposed to write a variable number of 1s, depending on how many were there initially.

Obviously, this will require a kind of *loop*: we will need to run through a certain series of actions, one for each 1 that is there on the tape in the beginning. How will we keep track of how many times we've been through the loop? By using the original block of 1s as a *counter*. We'll erase one of those 1s each time, and when they're all gone the program will know that it's time to quit.

What we'll do, then, is this. We'll repeatedly run through the following sequences of actions:

i) erase one of the counter block of 1s
ii) write two new 1s in a new location
iii) if the counter block is now gone, then we're done!
iv) otherwise we need to repeat the loop starting with item #1

It isn't quite that simple, because there are some little annoying complications. Suppose we do it this way:

ia) erase the left-hand 1
ib) move to the left until you hit a block of 1s

ii) write two 1s to the right of this block

iii) then move back to where the original block of 1s was.
iiia) if there's a blank there, HALT

iv) If not, back to step ia)

This looks good at first glance, but there are loads of problems. *First*, step ib) won't work the first time through. There won't be any 1s to the left of the original block of 1s. *Second*, if we always add to the growing block of ones on the right, as in ii), then we'll run out of room between our two blocks. *Third*, iii) won't work the final time through. How will we know where the original block of 1s was? The only thing we can do is move to the right until we hit a 1; but the final time through we will have erased the final 1 at step 1a, and then the program will never halt; it will continue to the right forever.

So we need a better strategy. We will erase 1s from the *right* of the original block (so the original position of the left-most 1 of this block will remain fixed); and we will add 1s to the *left* of the growing block:

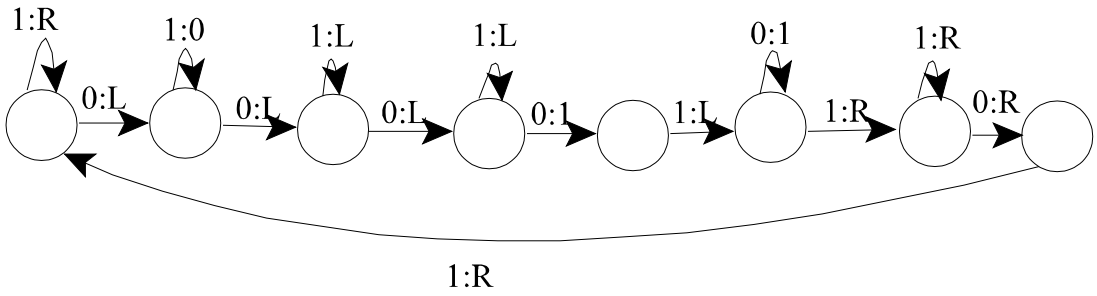
ia) move to right of the counter block of 1s
ib) erase it
ic) move back through any remaining 1s
id) move 1 more to the left. Now we're at the start of the growing block (if there's anything there at all)

iia) move to the left of any 1s of the growing block
iib) add two 1s to the left of this block
iic) move back to the right of all 1s
iid) move one more square to the right. Now we're at the left of the counter block

iii) If we're scanning a blank, HALT

iv) If we're scanning a 1, go back to step ia)

This may be implemented as follows:



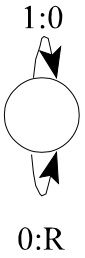
3. **Erase all the 1s to the right**

Here we must be very careful about how we word the task. We could mean one of two things:

i) For each square to the right of the initial position, at some point in time that square is erased and remains erased thereafter

ii) The machine must erase all the squares on the right and then halt when it is done.

The first task is easy to do:



But the second task is intuitively impossible, because the machine will never "know" when to quit.

That the second task is impossible can actually be proved if we take a bit more care stating just what the alleged machine should be like:

> We want a machine, M, such that for any tape, T, when started out on T, M eventually halts, at which point T is empty of all 1s to the right of the point M was initially scanning

Suppose, now, for reductio, that there is such a machine, and let T be any tape. Start M on T at some square S. By hypothesis, M will halt after some finite amount of time, after which the tape is all blank to the right of S. But now, consider another tape T′, which is just like T except that it has some 1s on squares to the right of S that M never scanned during its run on tape T. (There must be some such squares because M halted after a finite amount of time, and there's no Zenoing.) Now run M on this new tape, T, starting at square S. It must do the same thing as it did initially: it must scan exactly the same squares — no more and no less — since only what a Turing machine "sees" can affect its behavior. Therefore, M will halt without erasing the new 1s. But that violates the requirement that M have the desired effect on *every* possible tape: it failed to erase squares to the right on the new tape T′. QED

> (Note: this proof used a vaguely worded principle: "only what a Turing machine "sees" can affect its behavior". This principle could be stated more rigorously: "if a Turing machine executes a certain sequence of actions when started on a certain tape, then it will execute the same sequence of actions when started on any other tape that is like the first tape with respect to every square that the machine visited". Moreover, this principle could be proved inductively. I won't bother since the principle is pretty obviously true.)

Of course, there's no contradiction in supposing a machine that will do the trick for *some* tapes. For example, we could design a machine that would erase 1,000,000 squares to the right, then halt. This would work on any tape with no 1s more than 1,000,000 squares over. Moreover, we could design another machine that would erase 1,000,000,000 squares to the right. This would work on any tape with no 1s more than 1,000,000,000 squares over. And so on. What we can't have is *one* machine that works on *any* tape.

4. **Find the 1 to the right; then halt**

This is a related problem. There is something we *can* do: design a machine such that

> *if* there is a 1 to the right of the initial square, the machine will eventually find it, and then halt

But what we *can't* do is design a machine such that

> *if* there is a 1 to the right of the initial square, the machine will eventually find it, and then halt scanning it; but *if* there is *no* 1 to the right of the initial square, the machine will eventually halt scanning a 0

This is obvious: there's no way the machine would know when to quit looking. This illustrates an important distinction. We can have a machine that will find a positive answer, if one exists (first machine), without necessarily having a machine that tells us *whether* an answer exists.

It also shows us how careful we need to be in specifying what our machines are to satisfy, before we go around saying what machines can or can't do! The first task *is* something that can be accomplished. Note that there will never be any time at which we can be sure that that machine will find the 1. All we can say is that *if* there is a 1 out there to be found, the machine will find it eventually.

CHAPTER 5: Uncomputability via Diagonalization

I. Cardinality argument for uncomputability

It's fairly easy to give an non-rigorous proof that some functions of integers are not computable by any Turing machine. The reason is that there are too many functions of integers. As we proved earlier, the set of functions of integers is not enumerable. But as we will show below, the set of Turing machines *is* enumerable.

This is non-rigorous since we don't have a definition, yet, of what it is for a Turing machine to compute a function of integers. Intuitively, the idea is that the Turing machine must, for any input to the function, give the correct answer as to what the value of the function is. But we must define this more carefully, in particular paying attention to how we "feed inputs" to the Turing machine and to how we interpret its answers. After all, Turing machines manipulate symbols, not numbers.

II. Computing functions in monadic notation

We seek to define "Turing Machine M computes function f (from positive integers to positive integers)"

A. Input conventions: "standard input"

We first introduce conventions governing how we feed input to the Turing machine

1. Arguments in monadic notation ("1" is the only symbol), separated by single blanks (if the function is more than 1-place), on an otherwise blank tape.
2. Initially, the machine is scanning the leftmost 1 on the tape.

B. **Output convention: "standard configuration"**

The machine must halt scanning the leftmost 1 in a string of 1s on an otherwise empty tape

C. **The definition**

Machine M computes n-place function f iff:

For any n integers a_1, ..., a_n, when the machine is fed blocks of 1s representing a_1, ..., a_n in standard input form,

i) *if* $f(a_1...a_n)$ is defined, then the machine eventually halts in standard configuration with $f(a_1...a_n)$ 1s on the tape

ii) *if* $f(a_1...a_n)$ is undefined, the machine never halts in standard configuration

D. **Notes on the definition**

Note that there are two ways a machine that computes f can behave, if it is set to run on arguments for which f is not defined. It can never halt; or, it can halt, but in non-standard configuration.

Note also that we do not require that, when a Turing machine computes a partial function, it tells us when the function is undefined. It just must be the case that the machine will never halt in standard configuration; it need not send a message "I am never going to halt!". When we run the machine, we may not know in advance whether it will ever halt (more on this later).

Note also that we're talking about functions of *positive* integers. Thus, there will never be a case where the input to a function is zero.

Note also that *every* Turing Machine computes some n-place function, for every n. That's because, given any tape as input, a Turing machine does *something*. It will either halt in standard configuration or it won't. Halting in standard configuration indicates a certain value; not halting in standard configuration indicates

undefined.

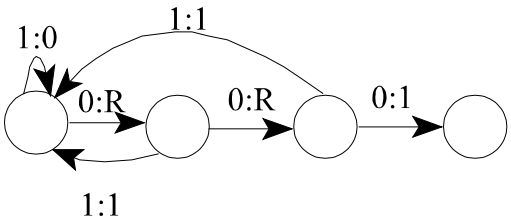
E. **Examples**

1. **What does this compute?**

Answer: it computes the one place identity function, f(i)=i; the two place function f(i,j)= undefined, the three-lace function f(i,j,k)=undefined, etc.

2. **Construct a Turing machine that, for every n, computes $f(x_1...x_n)=1$**

This is easy — erase all blocks of 1s until you get to two blanks in a row; then write a 1 and quit.

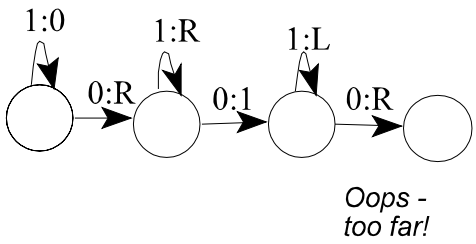


3. **Construct a Turing machine that computes f(x,y)=x+y**

This also is easy. Erase one 1, and fill in the blank between the two blocks with a 1.

Note: all we're asked to do is find a Turing machine that computes a certain 2 place function. So we're entitled to assume that our input consists of two blocks of ones. We don't need to worry that our machine will mess up if it is fed other things, for example one argument, or three arguments. The machine might well mess up in those cases, but that doesn't matter, for all the definition of 'computes' requires of the machine, if it is to compute *this* function (the plus function), is that it does the right

thing *if* it is fed *two* inputs in standard input form.



III. **Definition of Turing-comput*ability***

We have given a definition of what it is for a Turing machine to compute a given function. It is now easy to define Turing-computability: a function is T-computable iff *some* Turing machine computes it.

We now move a step closer to our goal by producing an enumeration of the Turing machines.

IV. **An Enumeration of the Turing Machines**

A. **First step: enumerate the "alphabet"**

Recall that the Turing machines may be represented as quadruples: $<q_i,Sj,a,q_k>$. Individual entries in these quadruples may be thought of as letters in an alphabet. Let's associate these letters with numbers as follows:

R	L	S_0	q_1	S_1	q_2	S_2	. . .
1	2	3	4	5	6	7	. . .

B. **Second step: enumerate the "words"**

The book uses a slick "12" method. Each Turing machine can be thought of as a word in this alphabet. Just put its quadruples one right after another, starting with the initial state of the machine, e.g., :

$q_1S_0Rq_2q_1S_1Lq_2q_2S_0S_1q_5$. . .

Now each of these words may be assigned a place in a list as follows. Replace each letter with the digit 1 followed by n 2s, where n is the place of the letter in the enumeration of the alphabet. Concatenate the digits thus defined for the entire

word. The result is a long numeral — a string of the digits 1 and 2. Interpreted as a base 10 numeral, it stands for a number: the position in a (very gappy!) list of all the words in this alphabet.

For example, the Turing machine $q_1S_0Rq_2$ gets assigned a spot in the list as follows:

q_1	S_0	R	q_2
12222	1222	12	1222222

= 122,221,222,121,222,222

C. **Eliminate the junk, close the gaps**

We now have a gappy list in which every Turing machine shows up. But many spots in this list have sequences of letters of the alphabet that don't specify Turing machines (e.g., $q_1q_1q_1$). So consider a new list with all these deleted. (We could give conditions defining when a sequence of letters of the alphabet doesn't count as a Turing machine; see p. 45 of the book.)

Now consider net another list with all the gaps closed: M_1, M_2, ... This is a non-gappy complete list of the Turing machines.

V. An Enumeration of the 1-place Turing-Computable functions

Each Turing machine, as noted, computes some n-place function, for each n. So corresponding to the list M_1, M_2, ..., there exists a list of 1-place functions f_1, f_2, ... — namely, the list of 1-place functions f_i computed by the Turing machines M_i. In fact, this list f_i is a *complete* enumeration of all the 1-place Turing-computable functions (of positive integers), since M_i was a complete list of all the Turing-machines.

For example, let's figure out what f_1 is. Well, what's the first Turing machine in the sequence? It is: $q_1S_0Rq_1$. (Why? This one generates fewer digits in the 12s method than any other Turing Machine.) This machine computes the 1-place identity function f(x)=x (since it always halts when started on standard input).

VI. An uncomputable function, by diagonalization

We now have our enumeration of the Turing-computable functions. It has no gaps, though it has repetition. It lacks gaps because we deleted all the gaps (and because there

are infinitely many Turing machines — you can always add more states.) It has repetition because more than 1 Turing machine can compute the same function.

We can now easily construct a function that not in the enumeration f_i, by our usual method:

f_1:	$f_1(1)$	$f_1(2)$	.	.	.
f_2:	$f_2(1)$	$f_2(2)$	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.

Define a new function u(n): = 1 if $f_n(n)$ is undefined
= $f_n(n)+1$ if $f_n(n)$ is defined

This function is not on the list, but it is a perfectly good function of positive integers.

VII. The function u and halting

Let's think a bit about this function, u. We could actually compute some of its values. For example, we know that f_1 is the identity function, and so $f_1(1)=1$, and hence we know that u(1)=2. We could figure out the second Turing machine on the list, and pretty easily figure out u(2). We could see from inspecting the Machine whether it will halt, and if so, figure out what its value will be in that case.

Why then isn't this function, u, Turing-computable? We seem to have a pretty straightforward procedure for computing any given value of u. Why couldn't we implement this informal procedure on a Turing machine?

Let's look more closely at what we're supposed to do to calculate u(n):

i) examine the flow diagram of M_n
ii) decide whether M_n will eventually halt in standard configuration
iii) If it will not, the answer is 1
iv) If it will eventually halt, then *run* machine M_n, as applied to the input n, and then add 1 to its answer

In fact, steps i), iii), and iv) are unproblematic. Step iv) might look difficult to implement on a Turing machine. What we're supposed to do, recall, is to implement steps i) - iv) on a single Turing machine. So our Turing machine must have the ability to simulate what other Turing machines do (including itself!!): we give it a number, n, representing machine M_n, plus a number for machine M_n to "work" on (in this case, n again); and the machine must spit back whatever output M_n would give, when applied to input n. But in fact this *can* be done. Such a machine is called a universal Turing machine, and has been constructed.

However, as of yet we have no way to implement step ii). How are we to develop a mechanical procedure for deciding when a given Turing machine will halt?

Here is one procedure that *won't* work: *run* machine n (with our universal Turing machine, say) and wait. The problem is that we don't know how long to wait. However long we wait, machine n may still halt at some time later, or it may not. So we can't say in step ii) "run M_n for 40 hours, and if it hasn't stopped, then go to step iii) and say 1". Nor can we say "just run M_n and wait for the answer". For suppose M_n never halts. Then we will never get to step iii) and say "1".

Obviously, the difficulty of deciding when a given Turing machine will halt is intimately related to the fact that not all functions are Turing-computable. For as we just saw (informally, and taking it on faith that Universal Turing Machines exist), if we *did* have a way for deciding whether a Turing Machine would eventually halt, it looks like we *could* compute function u after all.

VIII. **The Halting Problem**

Let's investigate this question further: that of deciding mechanically whether a given Turing Machine will ever halt. This is called "the halting problem".

The halting problem may be translated into the question of whether a certain function is computable:

$h(m,n) = 2$ if machine M_m will eventually halt, when fed n 1s in standard input form

$= 1$ otherwise

So, this function h is (intuitively) computable iff the halting problem is solveable.

Now, as it turns out we can fairly directly show that this function h is *not* Turing-computable. (This gives us, incidentally, another example of a function that isn't Turing-computable.)

Suppose, for reductio, that h *is* computed by some Turing machine, call it H. We will now use H to construct an *impossible machine*. This machine, call it M_m (it must be in the list somewhere; call its place m) will be designed so as to have the following feature:

> For any number, n, M_m will halt given n 1s in standard input iff M_n does *not* halt, given n 1s in standard input.

In other words:

> For any other machine, M_m will halt given that machine's code iff that machine will *not* halt when given its own code.

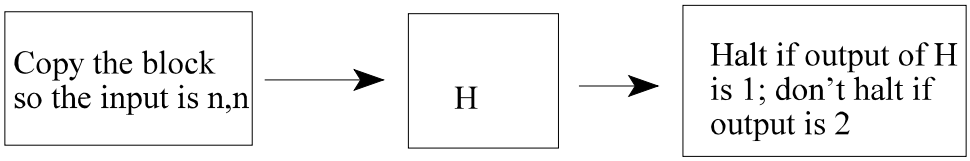
The way our machine, M_m will work is this. When M_m is given n ones on the tape:

..., _, _, 1, 1, 1, _, _, ...

it makes a copy of the block of 1s separated from the original block by a single blank:

..., _, _, 1, 1, 1, _, 1, 1, 1, _, _, ...

It then returns to the very leftmost 1 of the left hand block of 1s. At this point it has an input of n, n set up to feed to a machine that computes a two-place function. To machine H, this represents the question whether machine n will halt if given input n. We can now set H to work on this question. That is, we include H as a part of M_m. H computes numerical function h, and h is a total function, so H will definitely halt, telling us whether M_n would have halted if given n. We now simply need to make M_m halt if the answer was no, and make it not halt otherwise. Here is a picture of what M_m looks like:



OK, so now we have our machine M_m. What's the problem? Well, set M_m to work on its *own* code, m. We showed that M obeys this statement:

> M_m halts, given n, iff M_n does not halt, given n

But when we let n=m, we get:

> M_m halts, given m, iff M_m does not halt, given m

Which is impossible. M_m, accordingly, cannot exist; so neither can H. The function, h, is not Turing-computable.

If Church's thesis is true, therefore, there is no mechanical procedure at all that would allow one to decide whether a given Turing machine will ever halt for a given input.

IX. **Time-sharing, universal Turing machines, Church's thesis, etc.**

CHAPTER 6: Abacus computable functions are Turing computable

Our overall goal is to investigate the limits of mechanical procedures, understood intuitively. We've considered one stipulative definition of computability: Turing-computability. We will now consider another definition, in which the methods of computation are more like actual computers.

I. Abaci

The main addition in moving from Turing Machines to Abacus machines is adding *random access*: we have a number of different locations in which we can store information, and we can access those locations directly.

A. Registers

The storage locations are called registers. There are unlimited registers R_0, R_1, ... Each register has some number of "stones" in it, from 0 (empty) up (there is no upper limit).

B. Operations

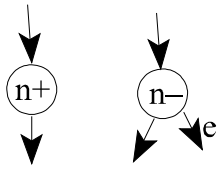
The abacus machine can do the following things:

1. Add 1 to register R_n
2. Determine whether a box is empty; remove 1 if not
3. Halt

In each case the machine executes a new instruction after performing the action.

C. Flow Diagrams

The two main types of action are diagramed like this:



The number n indicates which register is being incremented or decremented. The arrows point to the next instruction (instructions = what we called "states" for Turing machines). The 'e' indicates the arrow pointing to the state that comes next if register n is empty.

Further conventions: we'll indicate the *first* node by an arrow coming out of nowhere, and a halting node by an arrow going nowhere.

II. Examples of abacus machines

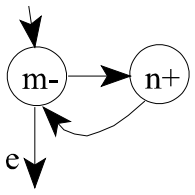
A. Notation

Let us introduce the following notation for describing abacus machine tasks.

- To indicate the number of stones in box m, we write: [m].
- To indicate that the machine is to put some number, x, of stones in box m, we write: $x \rightarrow m$

Thus, if we want the machine to put [m]+[n] stones in box n, we would write $[m]+[n] \rightarrow n$

B. Empty box m into box n (i.e., [m]+[n]→n; 0→m)

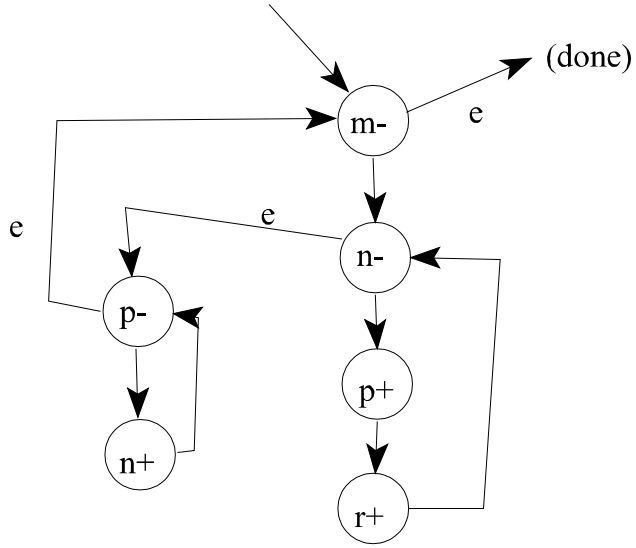


C. Multiplication

$[m]\cdot[n]\rightarrow r$ (assuming p, r initially empty)

Basically, we need to do what we did before, empty n into r, but do it repeatedly

— do it [m] times. But note that we can't trash n when we do this. So we need to empty n into r while preserving n, m times:



III. A series of functions

Consider the following series of functions:

i) The successor function, $'$

ii) Addition, which may be recursively defined in terms of successor as follows:

$$m+0 = m$$
$$m+n' = (m+n)'$$

iii) Multiplication:
$$m\cdot 0 = 0$$
$$m\cdot n' = m\cdot n + m$$

iv) Exponentiation:
$$m^0 = 1$$
$$m^{n'} = m^n \cdot m$$

v) Superexponentiation:
$$\sup(m,0) = 1$$
$$\sup(m,n') = m^{\sup(m,n)}$$

Two points about these functions. First, we can make abacus machines to do them. The multiplication machine was just repeated cumulative addition; exponentiation is repeated cumulative multiplication, etc. Secondly, these are reasonably powerful mathematical

operations here. So we have more support for Church's thesis, if we can show that all Abacus-computable functions are Turing-computable.

IV. Abacus-computable functions are Turing-computable

To prove this, we first need a rigorous definition of 'abacus-computable function'.

A. Conventions for Abaci

INPUT: for an n-place function, arguments in first n registers; all other registers empty.

OUTPUT: in some specified register (could be any register)

DEFINITION: Abacus A computes $f^n =_{df}$ for any $x_1, ..., x_n$, if $f^n(x_1, ..., x_n)$ is defined then A halts with $f^n(x_1, ..., x_n)$ in the designated output spot; otherwise A never halts.

B. Notes on the definition

1. this is really a definition of 'A computes f relative to output spot o'
2. It is a consequence of the definition that for any n and any register o, any abacus machine computes exactly one n-place function relative to o. This includes the case in which n is zero; in that case there is no input, and the result is a number in the output register.

C. Representing abacus registers on a Turing machine tape

We need to have a way to represent the registers of an abacus machine on the tape of a Turing machine. That's not too hard:

- Registers get represented by blocks of 1s on the tape, separated by single blanks
- Represent the registers in order. Thus, the leftmost block of 1s represents register 1, the next block represents register 2, and so on.
- Registers can be empty, so let n 1s represent n-1 stones. Thus, 1 block represents an empty register
- There are infinitely many registers, so we can't represent *all* the empty registers with 1s. So, there will be two ways to represent an empty register: i) a single 1; ii) a blank. But we will need some way of marking

the right-hand side of all the registers we're representing. We'll let 2 blanks in a row do this. Therefore, if to the right there are any more non-empty registers we are representing, then all intervening empty registers must be represented with 1s

- When we're designing a Turing machine to compute an n-place function, the function's argument places (i.e., the first n registers) must be represented with 1s, not blanks.

D. The idea of the proof

OK, let's get clear what we're trying to do. We're trying to prove that each abacus-computable function is Turing computable. To do that, we will show that for any abacus machine, A that computes a function f^n, relative to some output register, p, its flow diagram can be converted into the flow diagram of a Turing machine, M, that computes f^n (in the official sense defined earlier).

It is important to be clear that we only need to show that each abacus machine has its corresponding Turing machine. We don't need to come up with *one* Turing machine that will do the work of all the abacus machines. Rather, we need to specify a procedure that will yield *different* Turing machine flow graphs, depending on what the given abacus machine flow graph looks like.

This procedure involves going through the abacus machine flow graph, state by state, and for each state, adding in a bunch of states to a growing Turing machine flow diagram. There are two kinds of states in abacus machines, (s+) and (s-); in each case we'll show how to construct a part of a Turing machine to do the work of that state of the abacus machine. Of course, the Turing machine will be operating on the tape representation of the registers, since Turing machines don't know how to access registers directly.

For each state, we'll assume that the Turing machine begins in standard position, scanning the leftmost 1 on the tape. The Turing machine will update the tape appropriately, and then will return to the left-most 1.

E. Replacing (s+)

Here we specify a procedure for converting a (s+) state to a Turing machine program that has the analogous effect on the tape-representation of registers. The basic idea is this:

- Move out to the s^{th} block of 1s, replacing blanks with 1s

representation if necessary.

- Add a 1 to the right of this block
- Move all blocks of 1s to the right of the s^{th} block over a space to make room
- Return to the very left of the 1s
- Pass on control to the Turing-instructions corresponding to whatever abacus state this one was pointing to

Here is how B&J represent this (p. 63):

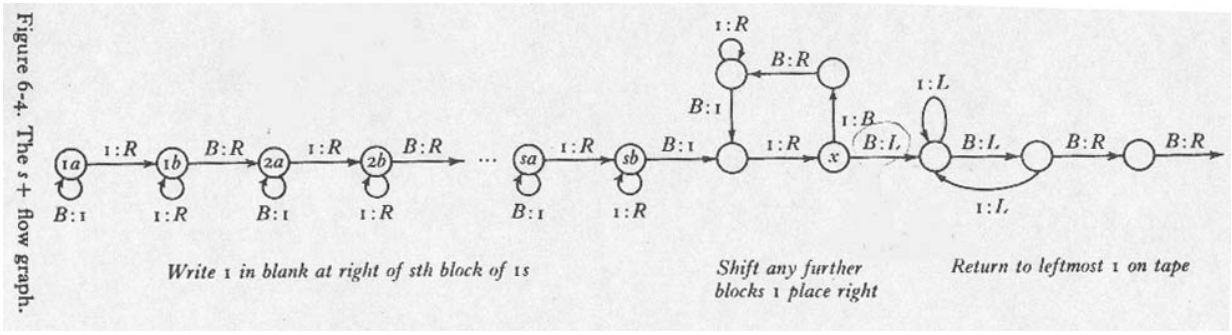


Figure 6-4. The $s+$ flow graph.

Note the ". . ." in this flow diagram. That's fine. This is not supposed to be a particular flow diagram. Rather, it is a picture of what the flow diagrams look like in general. Each one will be different, depending on what s was. Remember: all we're doing is showing that any abacus machine can be replaced by some Turing machine or other, not that a single Turing machine can replace all of the abacus machines at once.

F. **Replacing** (s-) (B&J, p. 65)

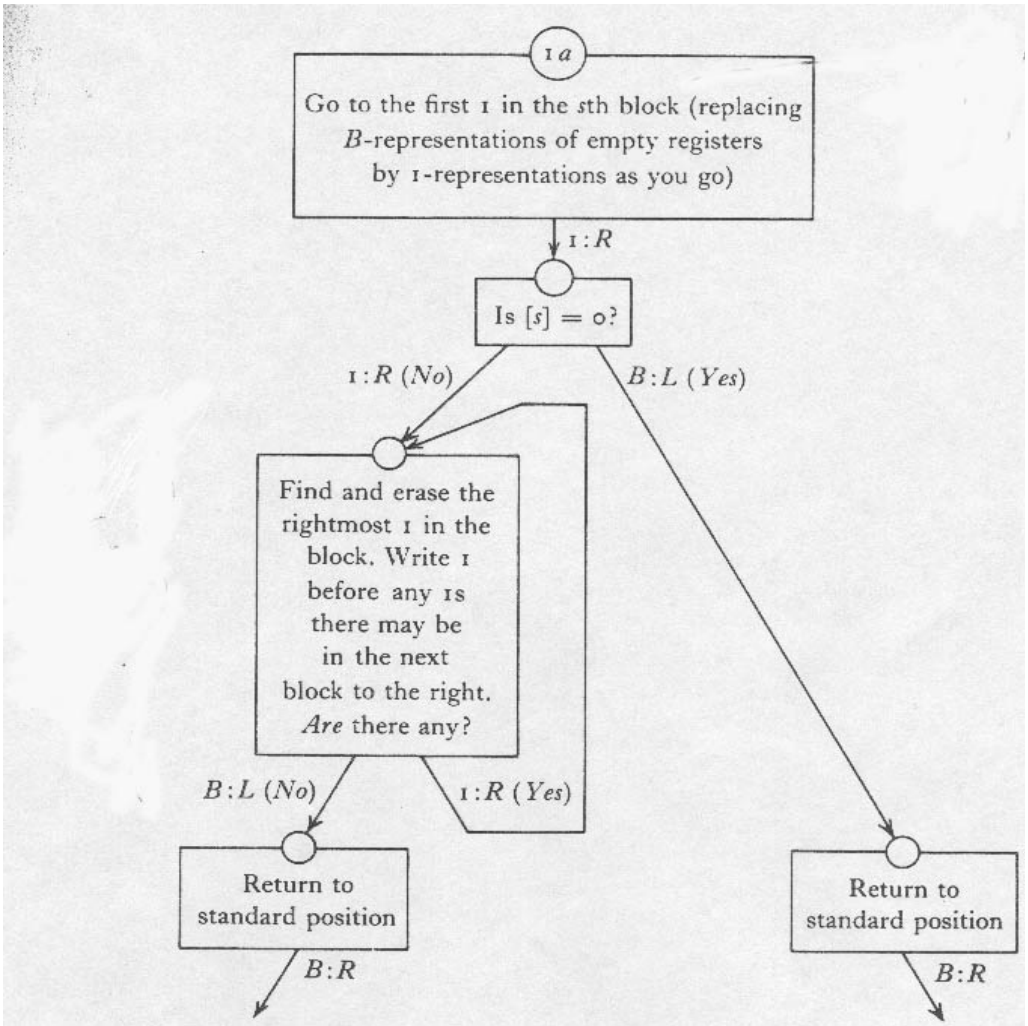


Figure 6-6. Block diagram for *s*-flow graph.

G. **Mop-up graph**

There are two ways in which we need to augment our machine. First, an abacus machine indicates "HALT" by an arrow pointing nowhere; that needs to be done Turing-machine-style. Second, at the end, the output of the machine needs to be in Turing-machine standard output form. So what we do is make every one of the Abacus machine's halting arrows point to a Turing machine mop-up-graph. This:

- Moves the output register's information to the same square that was initially the left-most 1.
- Erases the rest of the tape.

Note that we are "given" which register is the output register. So if it happens to be register 1, then we must just erase all of the tape other than the first block of 1s. Otherwise, we move that register's block of 1s to the appropriate location, and then erase the rest of the tape. These two possibilities do not *each* have to be done by our mop-up graph; and the mop-up-graph does not have to be able to "figure out" what the output register, p, is. What p is is "given" to us, and can be hard-wired into the mop-up graph.

This is done as an exercise in the book, p. 67.

CHAPTER 7: Recursive functions are abacus computable

As before, we are in search of more confirmation of Church's thesis, by discussing yet another notion of computability and showing that computable functions, thus defined, are all abacus computable (and so, given the result of the last chapter, Turing computable).

I. Recursive functions

Return to our series, $'$, +, ·, exp, sup, ... These are all intuitively computable, because *repeated* application of a simple function, successor, lets us compute the value of any of the more complicated functions.

Moral: if a function is defined *recursively* in terms of *initial functions* that are in an intuitive sense computable, then that function should be computable. This guides our third definition of computability, which applies to functions directly, rather than indirectly through a characterization of a sort of machine. Here's the basic idea:

- define the *initial functions*
- define *admissible operations* which yield new functions from old
- The *recursive functions* will then be those one can derive from the initial functions by a finite sequence of applications of admissible operations

If we choose for initial functions only functions that are intuitively computable, and if we choose for admissible operations only operations that are computability-preserving, then the recursive functions ought to be computable, in the intuitive sense.

II. Initial functions

$z(x) = 0$	zero
$s(x) = x+1$	successor

$id(x) = x$
$id^2_1(x,y)=x$, $id^2_2(x,y)=y$, etc. projection functions

III. Admissible operations

A. Composition

1. Definition

Let f be a function of m arguments, and let g_1, ..., g_m be each functions of n arguments. Then the function h, defined by:

$$h(x_1, ..., x_n) = f(g_1(x_1...x_n), ..., g_m(x_1...x_n))$$

is said to "come from f, g_1, ..., g_m *by composition*". For short, "$h=Cn(f,g_1...g_m)$"

Think of Cn as being a function which takes functions as arguments and yields a function as a value. The idea here is that you *first* apply the g's to the arguments of $Cn(f,g_1...g_m)$, and then apply f to these values. Note that $Cn(f,g_1...g_m)$ will be a n-place function.

2. Example: h(x)=x+3 — show that h is recursive

This is easy, since h(x) = s(s(s(x))). So h is arrived at by function composition from the successor function.

To get absolutely clear about this, it is good to construct a name for h in "official notation", as follows:

- s, the successor function, is a recursive function, since it is an initial function
- the function s(s(x)) is obtained by composition from s and s by composition. Its name in official notation is Cn(s,s)
- Thus, h, in primitive notation, is Cn(s,Cn(s,s)).

If we can provide a name for a function in official notation, that function is clearly recursive.

3. Example: f(x,y)=3 — express in official notation

well, f(x,y)=s(s(s(z(x)))). Thus, f = Cn(s,Cn(s, Cn(s,id^2_1)))

Note that since f(x,y) also is the third successor of z(y), we could just as well have used id^2_2.

B. Primitive Recursion

1. Definition

This is the second admissible operation. Abbreviation: let's abbreviate "$x_1...x_n$" by "**x**". Here is the definition:

- Let f be an n-place function, g an n+2 place function, and h an n+1 place function.
- Suppose that for any integers **x**, y, h(**x**,0)=f(**x**), and h(**x**,s(y))=g(**x**,y,h(**x**,y))
- Then we say "h is definable by primitive recursion from f and g"; i.e., "h=Pr(f,g)"

The idea is just the same idea of recursion we've dealt with all along, only formalized and generalized. The recursion takes place "on" the final argument place of h. The base case — i.e., h(**x**,0) — is given by f. And then, the value of h for a successor — i.e., h(**x**,s(y)) — is given by a certain function, g. That is the recursion clause. That function g gets to "look at" not only the previous value of h — i.e., h(**x**,y) — but also y, and also the other arguments of h — i.e., **x**.

2. Example: addition

Let's write this out in official notation, to see clearly how the informal recursive definitions we've given so far fit into the official mold. First, the informal definition:

+(x,0) = x
+(x,s(y)) = s(+(x,y))

Now we need to see what f and g are, so that we can say + = Pr(f,g).

Well, f is the function such that f(x)=+(x,0)=x. So f = id.

As for g, since + is two-place, g must be three-place. It must be that $g(x,y,+(x,y)) = +(x,s(y)) = s(+(x,y))$. Thus, $g=Cn(s,id^3_3)$.

In words, g is a 3-place function; given 3 inputs, it applies id^3_3 to select the final argument; it then takes the successor of that argument.

So, finally, $+ = Pr(id, Cn(s,id^3_3))$.

3. **Example: multiplication**

$x\cdot 0 = \cdot(x,0) = 0$
$x\cdot s(y) = \cdot(x,s(y)) = x\cdot y+x = +(\cdot(x,y),x)$

So, f=z.

As for g, $g(x,y,\cdot(x,y))$ is to be $+(\cdot(x,y),x)$. That means $g=Cn(+,id^3_3,id^3_1)$; i.e., $g=Cn(Pr(id, Cn(s,id^3_3)),id^3_3,id^3_1))$.

Therefore, $\cdot = Pr(z,Cn(Pr(id, Cn(s,id^3_3)),id^3_3,id^3_1)))$

4. **Example: factorial**

$!(0) = 1$; $!(s(y)) = s(y)\cdot y! = \cdot(s(y),!(y))$

f must now be a function of zero arguments, since ! is a one-place function. Let's allow such functions — i.e., constants — as new initial functions:

0, 1, 2, ...

(We don't really need to, actually; the last exercise of the chapter shows a tricky method for avoiding their use.)

As for g, $g(y,!(y)) = \cdot(s(y),!(y))$. So $g = Cn(\cdot,Cn(s,id^2_1),id^2_2)$. We could substitute in the official notation for · if we wished.

IV. More functions

Part of the homework involves showing certain functions to be primitive recursive. Notice that this is a sort of cumulative procedure. The definition of a primitive recursive

function allows that any "chain" of definitions beginning in the initial functions, moving via admissible operations, results in a primitive recursive function. So if a function has already shown to be primitive recursive, that can be very useful in showing other functions to be primitive recursive.

The exercises in chapter 7 run through a large class of functions, showing them to be primitive recursive. The point is to give a sense of how wide the class of primitive recursive functions is. It will help you very much in the homework if you utilize some of these functions.

In general, you don't need to work out the official notation of a function unless I ask you. It's enough to define the functions informally, for example:

> !: 0! = 1; x′! = x′· x! (We've already shown · to be primitive recursive; this utilizes in addition successor, function composition, and primitive recursion).

I'll highlight a few other useful functions:

- *predecessor*: (like the usual predecessor function, except that pred(0)=0)

  pred(0)=0; pred(x′)=x (Slick recursive definition — we never even looked at the predecessor function.)

- *dot-minus* (I'll just use the "-" sign; the dot indicates that x-y=0 if y>x.)

  x-0 = x; x-y′ = pred(x-y)

- *signum* and *anti-signum* (these are "boolean functions", which tell you whether or not their arguments are zero. 1 means true; 0 means false.)

  sg(x) = 1-(1-x) "x is non-zero"
  ~~sg~~(x) = 1-x "x is zero"

- *general summation*: $g(x_1...x_n, y) = \sum_{i=0}^{y} f(x_1...x_n,i)$

  f here is some n+1 place function; and the resulting function g is also n+1 place. If f is primitive recursive then so will be g, since g can be defined recursively as follows:

  $g(x_1...x_n,0) = f(x_1...x_n,0)$
  $g(x_1...x_n,y') = g(x_1...x_n,y)+f(x_1...x_n,y')$

general product can also be defined.

- *definition by cases*

First we need to introduce the notion of a *condition*. A condition, $C(x_1..x_n)$ on n numbers is something that is either *true* or *false* of those numbers. Thus, "x>y" is a condition on x and y. The *characteristic function* of a condition, C(**x**), is the function c(**x**)=1 if C holds of **x**; c(**x**)=0 otherwise. A characteristic function says "yes" or "no".

Now suppose we have a function, f, defined by cases as follows:

$$\begin{aligned} f(\mathbf{x}) &= g_1(\mathbf{x}), \text{ if } C_1(\mathbf{x}) \text{ holds} \\ &= . \\ &= . \\ &= g_m(\mathbf{x}), \text{ if } C_m(\mathbf{x}) \text{ holds} \end{aligned}$$

where the conditions $C_1...C_m$ are *exclusive* (no two of them can hold of a given **x**), and *exhaustive* (at least one must hold, for a given **x**), their characteristic functions $c_1...c_m$ are primitive recursive, and the functions $g_1 ... g_m$ are primitive recursive. In that case, f must be primitive recursive, for

$$f(\mathbf{x}) = g_1(\mathbf{x})\cdot c_1(\mathbf{x}) + ... + g_m(\mathbf{x})\cdot c_m(\mathbf{x})$$

Note that the "..." here do not indicate a need for general summation. We're defining a function that is the sum of a *particular* set of other functions (i.e., the number m and the functions $g_1...g_m$ are fixed). So we just need plain old binary summation and function composition.

- *bounded quantification*

Suppose we have an n+1 place condition, C(**x**,y), whose characteristic function c(**x**,y) is primitive recursive. We can then form another condition whose characteristic function is primitive recursive:

$$\forall i(i \leq y \rightarrow C(\mathbf{x},i))$$

This is an n+1 place condition on **x**, y saying that C(**x**,i) holds for every $i \leq y$. Why is this new condition primitive recursive? Because its characteristic function, call it f, may be defined thus:

$f(\mathbf{x},y)=\prod_{i=0}^{y}c(\mathbf{x},i)$

- *bounded minimization*

  Let $f(\mathbf{x},y)$ be an n+1 place primitive recursive function, and consider the n+1 place function $Mn_w(f)$ defined as follows:

  $Mn_w(f)(\mathbf{x})$ = the smallest y between 0 and w, inclusive, such that $f(\mathbf{x},y)=0$, if there is such a y

  0 otherwise

  This can be seen to be primitive recursive because it can be defined by cases:

  $Mn_w(f)(\mathbf{x})$ = 0 if $\forall y(y\leq w \rightarrow f(\mathbf{x})\neq 0)$
  $\sum_{i=0}^{w} sg(\prod_{j=0}^{i} f(\mathbf{x},j))$ otherwise

V. Proof that all recursive functions are abacus-computable

Now that we've convinced ourselves that a whole lot of functions are primitive recursive, let's show that all primitive recursive functions are abacus-computable. A primitive recursive function is one that is definable from the initial functions by admissible operations. So if we can show that i) the initial functions are abacus-computable, and we can show that ii) admissible functions preserve abacus-computability, we will have established the desired result.

A bit more carefully, we'll really be establishing the existence of abacus machines that compute these functions with two special features: i) they put the output in the register immediately after the input, and ii) they preserve what's in the input registers. So we'll show that the initial functions are computed by abacus machines *of this sort*, and we'll show that computability-by-abacus-machines-of-this-sort is preserved under the admissible operations.

A. Conventions

Let's introduce some conventions for our abacus machines. These aren't required by the definition of 'abacus-computable'; but they will facilitate the proof.

1. Assumption of blank registers

These machines assume that certain other registers, other than the input registers, are blank. It will be specified, in each case, which registers are assumed to be blank.

2. **Standard location for output**

When designing an abacus machine that computes an n-place function, let's always place the output in register n+1. That way we don't have to keep mentioning the output register. Thus, what we'll do is show that each primitive recursive function can be computed by an abacus machine of this sort.

3. **Preserving inputs**

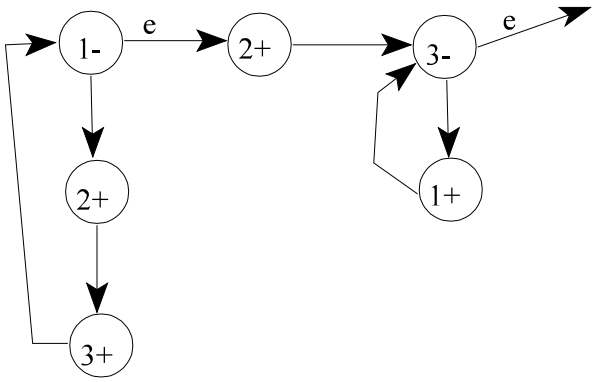
We'll also preserve the inputs, in the sense that the Abacus machines will restore the input registers as it found them.

B. The zero function is abacus-computable

A vacuous machine that does nothing computes this function (since register n+1 is initially empty).

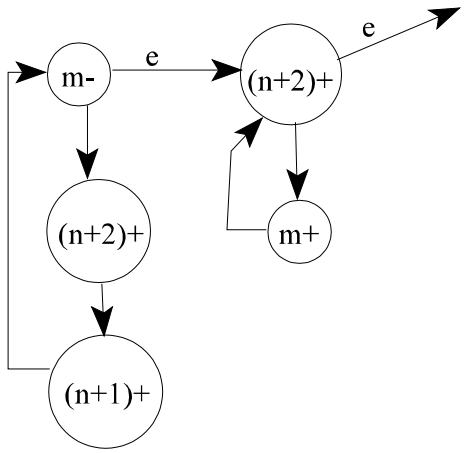
C. The successor function is abacus-computable

(Assumes 3 initially blank).



D. **The projection functions are abacus-computable**

Here is a machine to compute id^n_m. The output is in register n+1 (since the function is n-place); the program assumes that register n+2 is blank:



E. **Function composition preserves abacus-computability**

Suppose we have m n-place functions: g_1, ..., g_m, and an m-place function f, each of which are abacus-computable. We will show that $Cn(f,g_1, ..., g_m)$ is also abacus computable.

So suppose we have m+1 abacus machines that compute f, g_1, ..., g_m. We'll suppose that these machines place their outputs in the registers directly after the input registers, and that they don't trash their inputs, in line with our conventions. What we need to do is construct an abacus machine that computes $Cn(f,g_1, ..., g_m)$.

$Cn(f,g_1, ..., g_m)$ is an n-place function. So basically, we need to run the machines that compute g_1, ..., g_m on the n inputs, each time storing the result in some register. Then we need to run f on these results.

There is how it will go for m=2, n=3:

1. Look through the programs for f, g_1 and g_2 and find (m+n=) 5 unused registers
2. Run g_1 (the n inputs for g_1 are already set up)
3. Store result in first free register

4. Run g_2 (no need to restore original input, since we are assuming the machines for the g_i's preserve their inputs)
5. Store result in second free register
6. Move original input into the last three free registers
7. Move what's in the first two free registers into registers 1 and 2
8. Run f machine
9. Store result in register 4
10. Move last three free registers into registers 1-3 (we're not to trash input registers)

It should be reasonably obvious that we can do this for any n, m (simply replace the numerals in this proof with appropriate variables).

F. **Primitive recursion preserves abacus-computability**

Suppose n-place function f, and n+2 place function, g, are abacus computable. We must show that Pr(f,g) is also abacus computable.

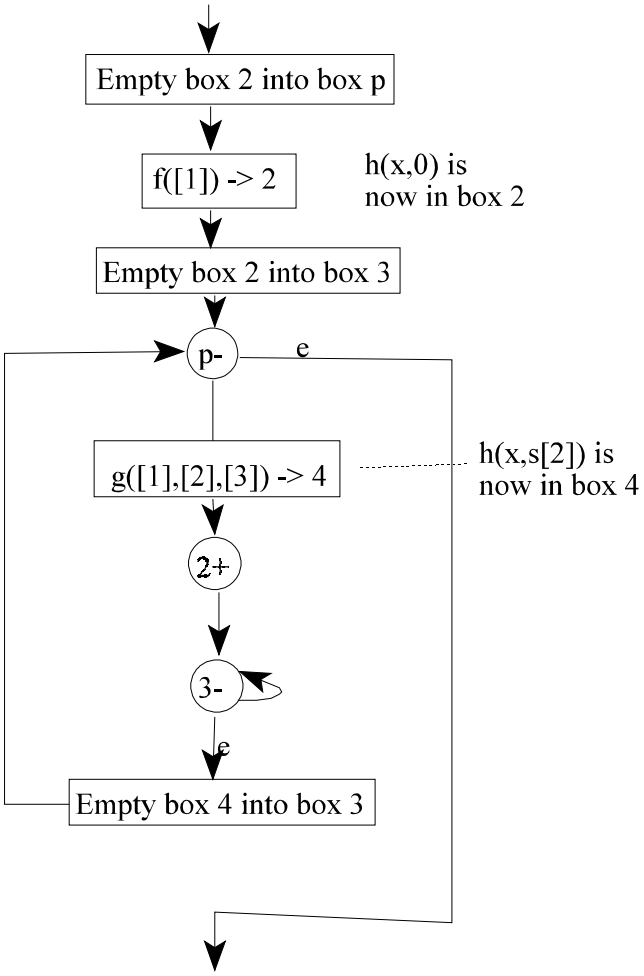
Recall that the function h= Pr(f,g) is an n+1 place function defined by these recursion equations:

$h(\mathbf{x},0) = f(\mathbf{x})$
$h(\mathbf{x},y') = g(\mathbf{x},y,h(y))$

So the idea is this.

- The machine we're to build must first compute $h(\mathbf{x},0)$, by running the f machine on the first n inputs.
- It must then compute $h(\mathbf{x},1)$ by running the g machine, feeding the g-machine the first n inputs, the number 0, and the number just spit out by the f machine. (I.e., $h(\mathbf{x},0)$).
- It must then compute $h(\mathbf{x},2)$ by running the g machine, feeding the g-machine the first n inputs, the number 1, and the number just spit out by the g-machine (i.e., $h(\mathbf{x},1)$).
- etc.

Here's how Boolos and Jeffrey do it, for the case where n=1. As before, it's easy to generalize this. We must first find an unused register, call it p, that we will use as a counter. We then design the machine as in B&J, p. 82:



VI. Recursive functions

At this point we've proved that abacus machines can compute all primitive recursive functions. Actually, they can do a bit more. We'll add a new admissible operation, and show that it, too, preserves abacus-computability.

A. Minimization

This new operation must be distinguished from *bounded minimization*, the operation we dealt with above. Bounded minimization of a primitive recursive function is primitive recursive — we don't need a new operation. But minimization simpliciter is an operation that genuinely extends the class of

primitive recursive functions.

The idea is that the minimization of f is a function that returns the smallest number y such that f(y) is zero. More carefully:

Where f is an n+1-place function, Mn(f) is an n-place function, h, such that:

h(**x**) = y, if f(**x**,y)=0 and f(**x**,t) is defined and non-zero for each t<y
= undefined if there is no such y

Note that while the primitive recursive functions were all total, the recursive functions can be partial.

Note: It was never made clear in the earlier definitions of composition and primitive recursion how they were supposed to work in cases where one of the component functions yielded an undefined value. I'm fairly sure those definitions should be understood as applying in the following way: a function, h, arrived at by composition and primitive recursion is defined for a given set of arguments only when all the functions leading up to the calculation of h's value for that set of arguments are defined. If any one of these is undefined, then h is undefined for the given set of arguments.

We should then look back and verify that the proofs we constructed to establish that composition and primitive recursion preserve abacus-computability still go through, now that the functions in question might be partial. They do: suppose that one of the functions involved in the calculation of h(**x**) is undefined. Then the machine that calculates that function will never halt. But then the machine calculating h won't halt either, since it contains the other machine as a component part.

B. Minimization preserves abacus-computability

Suppose that f^{n+1} is abacus-computable. Then it should be pretty clear why Mn(f) is as well. Basically, we just construct a machine that successively computes f(**x**,0), f(**x**,1), ..., and looks to see each time if f(**x**,i)=0. The first time f(**x**,i) is zero, the machine returns the answer i. If in one of these cases f(**x**,i) isn't defined then the f-machine would never halt, and so the machine we're designing would never halt — but that's what we want, since in this case Mn(f) is undefined. Also, if f is always defined, but is never zero, then again the machine will never halt, and again, this is the appropriate reaction since in this case Mn(f) is undefined.

The book has a diagram of a machine to implement this, but it's pretty obvious

how this would go, so I won't go through it (see figure 7-4 on p. 84).

CHAPTER 9: First-order Logic revisited

I. Overall goal

We turn now (finally!) to logic. We're going to prove various results *about* predicate logic with identity and function symbols. This is in contrast to working *within* logic (for example doing derivations or constructing truth tables).

Some of the notions we will be dealing with are familiar from intro logic. In our study of propositional logic we introduced the notion of a tautology. We will generalize this to predicate logic, and introduce the notion of a *valid* formula — a formula that is, in a certain sense, "true no matter what". (A tautology is, in essence, a formula that is true no matter what — it is true relative to every "case", i.e., every assignment of truth values to the sentence letters). Relatedly, we will introduce the notion of one formula's *implying* another.

Our first goal, after introducing these notions, will be to show that there is a mechanical test for validity, and second to show that there is *no* mechanical test for *in*validity.

The formal development of a given logic is usually broken into two tasks: syntax and semantics. Syntax usually includes grammar — i.e., the definition of a well formed formula, as well as the definition of a *theorem* — i.e. a formula that is in some sense *provable*. Semantics involves meaning — there we define the notion of an *interpretation* of a language, in which the terms are given meanings, and use interpretations to define such notions as validity and implication.

II. Syntax of first-order logic

We begin by being a lot more careful than we were in introductory logic about grammar. It will become apparent as we proceed that it is crucial to do this if we are to be able to give the kinds of rigorous meta-proofs that we wish to give.

A. Definition of a language

The languages we will discuss have two sorts of symbols, *logical* and *non-logical* symbols. The logical symbols will be common to all our (first-order) languages:

Logical symbols:	Variables (an enumerably infinite class), ~, $\rightarrow$, &, $\vee$, $\leftrightarrow$, $\forall$, $\exists$, (,), , (the comma), = (a two-place predicate)
Non-logical symbols:	names, function symbols (with fixed numbers of places), predicate letters other than = (with fixed number of places), sentence letters

All our languages will share the same logical vocabulary, so we can identify a language with the set of its non-logical symbols. E.g., the language of arithmetic: {**0**, ′, **+**, ·}.

B. Definition of a term

Our goal is to define the notion of a formula, but we must first define the terms:

i) names are terms
ii) variables are terms
iii) if f is an n-place function symbol and α_1, ..., α_n are n terms, then $\ulcorner f(\alpha_1, ..., \alpha_n)\urcorner$ is a term.
iv) only strings that can be shown to be terms on the basis of a finite series of applications of i)-iii) are terms

C. Definition of a formula

Now we can give our desired definition:

i) Sentence letters are formulas
ii) if α_1, ..., α_n are terms, and Π^n is an n-place predicate, then $\ulcorner \Pi^n(\alpha_1, ..., \alpha_n)\urcorner$ is a formula

(The formulas from i) and ii) are called atomic formulas.)

iii) If ϕ and ψ are formulas, and α is a variable, then the following are formulas:

$(\phi\&\psi)$	$(\phi\rightarrow\psi)$	$\exists\alpha\,\phi$
$(\phi\vee\psi)$	$\sim\phi$	
$(\phi\leftrightarrow\psi)$	$\forall\alpha\,\phi$	

iv) Nothing else is a formula

D. Relaxing the rules

We do this. E.g., we drop outer parentheses, put the = sign between its terms, we write ∀xFx instead of ∀xF(x); we write **0′′+0′** instead of **+(′(′(0)),′(0))**.

E. Definition of open and closed formulas

Intuitively, the idea of a *free variable* is that of a variable that doesn't "belong to" any quantifier, for example x in: ∀yRxy. Note that it is *occurrences* of variables that are free; note moreover that freedom is a relative matter: an occurrence of a variable is free or bound *in* a given formula. The following seems to be an acceptable definition:

- Any occurrence of any variable in an atomic formula is free in that formula
- If an occurrence of α is free in ϕ then it is free in $\phi\&\psi$, $\phi\vee\psi$, $\phi\rightarrow\psi$, $\phi\leftrightarrow\psi$, and $\sim\phi$
- If an occurrence of α is free in ϕ, then it is free in $\forall\beta\ \phi$ and $\exists\beta\ \phi$, where $\beta\neq\alpha$.
- No other occurrences of any variables are free in any other formulas
- If an occurrence of α is not free in ϕ, it is *bound* in ϕ.
- ϕ is *open* iff it contains some occurrences of variables that are free in ϕ; otherwise ϕ is *closed*. A closed formula is often called a *sentence*.

F. Theoremhood

"Syntax" often includes a definition of a *theorem* — a provable formula. One way to define theoremhood proceeds roughly as follows. One first identifies some small group of formulas as axioms. One then identifies *rules of inference*, by which some formulas follow from others. (A rule of inference is a relation over formulas; when $R(\phi_1, ..., \phi_n, \psi)$, then we say that ψ follows from $\phi_1, ..., \phi_n$ via R.) One then defines a *proof* as any finite sequence of formulas, each member of which is either an axiom, or follows from earlier members of the sequence via some rule of inference. Finally, a theorem is defined as the last line of any proof.

Another method for defining theoremhood is that of natural deduction. Systems

of proof from books such as Hardegree's book, Kalish and Montague's book, etc., which contain methods of proof like conditional proof and indirect proof, are natural deduction systems. Constructing proofs in these systems is much much easier than constructing axiomatic proofs. However, proofs about natural deduction systems are harder.

At any rate, we simply skip over the definition of theoremhood in this book.

III. Semantics for first-order logic

The ultimate goal, as we said, is to provide a definition of validity — "truth-no-matter-what", or "truth in all *cases*, or all *interpretations*". An interpretation is a sort of possible situation, together with the meanings of the terms in that situation. We will give a definition of what it is for a (closed) formula to be *true*, *relative to* a given interpretation.

This is relatively easy for propositional logic. An interpretation is an assignment of truth values to the sentence letters. Any possibility that can be represented by the language of propositional logic can be represented as a distribution of truth values over sentence letters. Then, one needs to define what truth values of complex formulas are *inherited*, once the truth values for the sentence letters are fixed. That's easy: that's what the truth tables are for.

Things are harder for predicate logic because of the presence of variables; moreover, we want to do things much more rigorously. But the overall picture will be the same: provide semantic values for the non-logical symbols of the language, and then give a definition of what truth values are inherited by complex formulas.

A. Definition of an interpretation (or model), I, of a language L

I must consist of:

1. A domain — i.e., a non-empty set
2. For each name in L, a *denotation* — some member of the domain
3. For each n-place function symbol of L, a (total) n-place *function* defined on the domain
4. For each sentence letter of L, a *truth value* (i.e., 0 or 1)
5. For each predicate letter, Π^n, of L, an n-place *characteristic function* over the domain — i.e., an n-place function that assigns to any n objects of the domain a truth value.
6. I does not assign values other than those specified in 1-5

We can write the semantic value I assigns to symbol S thus: I(S).

An *interpretation of a sentence* is an interpretation of a language of which that sentence is a sentence.

B. Example: the standard model of arithmetic

This interpretation, N, must supply meanings for the non-logical symbols of the language of arithmetic: **0**, ′, **+**, ·. As follows:

N(**0**) = 0
N(′)= s, the successor function
N(**+**) = sum, the addition function
N(·) = prod, the multiplication function

It's important not to get confused between symbols and what they stand for here. To avoid this, I've made my symbols for symbols **boldface**. Thus:

0

stands for a name in the language in the language of arithmetic, whereas

0

stands for the real, live number zero.

So, in the standard model, the sentence **0′′+0′′′=0′′′′′** should be true. Note that other interpretations of the language of arithmetic are possible. For example, there is an interpretation just like the standard one but in which the interpretations of · and **+** are reversed. And there are interpretations that have nothing to do with mathematics, ones in which the domain consists of people, say.

C. Another example

Consider a language with the predicate L, the name a, and the function symbol f. The following is a sentence of the language: aLf(f(a))

Suppose we let the domain of an interpretation of this language be the set of people, and let a denote me, let the interpretation of f be the father-of function, and let the interpretation of L be the characteristic function of the loving relation

(i.e., I(L)(x,y)=1 iff x loves y, 0 otherwise.) Then this sentence would be true in this interpretation iff I love my paternal grandfather.

D. Definition of truth in an interpretation

We want to introduce interpretations for entire sentences now; $I(\phi)=1$ if ϕ is true in I; $I(\phi)=0$ if ϕ is false in I. We first need to define the denotation of terms other than names:

- The denotation for names in I is already defined (hard-wired into I)
- where $\alpha_1, ..., \alpha_n$ are terms, and f^n is an n-place function symbol, the denotation of $f^n(\alpha_1, ..., \alpha_n)$ in I is defined as $I(f^n)(I(\alpha_1), ..., I(\alpha_n))$
- nothing else has a denotation

Note the recursive definition; the base case is the assignment of denotations to the "atomic terms" — i.e., names

Now we can give our definition of $I(\phi)$, for arbitrary sentences ϕ:

1. If ϕ is a sentence letter, $I(\phi)$ is already assigned by the model
2. If $\phi = \ulcorner \Pi^n(\alpha_1, ..., \alpha_n) \urcorner$, then $I(\phi) = I(\Pi^n)(I(\alpha_1), ..., I(\alpha_n))$
3. If $\phi = \ulcorner \alpha_1 = \alpha_2 \urcorner$, then $I(\phi)=1$ iff $I(\alpha_1)$ is identical to $I(\alpha_2)$; $I(\phi)=0$ otherwise.
4. For any sentences ϕ, ψ:
   a. $I(\phi \& \psi) = 1$ iff $I(\phi)=1$ and $I(\psi)=1$; $I(\phi \& \psi)=0$ otherwise
   b. $I(\phi \vee \psi)=1$ iff $I(\phi)=1$ or $I(\psi)=1$ or both; $I(\phi \vee \psi)=0$ otherwise
   c. $I(\phi \rightarrow \psi) = 0$ iff $I(\phi)=1$ and $I(\psi)=0$; $I(\phi \rightarrow \psi)=1$ otherwise
   d. $I(\phi \leftrightarrow \psi) = 1$ iff $I(\phi)$ and $I(\psi)$ are identical; $I(\phi \leftrightarrow \psi)=0$ otherwise
   e. $I(\sim\phi)=1$ iff $I(\phi)=0$; $I(\sim\phi)=0$ otherwise
5. For any formula ϕ and any variable α:
   a. $I(\forall \alpha\ \phi)=1$ iff $I^a_o(\phi_a a)=1$ for every o in the domain of I; $I(\forall \alpha\ \phi)=0$ otherwise
   b. $I(\exists \alpha\ \phi)=1$ iff $I^a_o(\phi_a \alpha)=1$ for at least one o in the domain of I; $I(\exists \alpha\ \phi)=0$ otherwise

   WHERE: $\phi_a a$ is the sentence obtained by writing a in place of every free occurrence of α in ϕ; a is some name that doesn't occur in ϕ; I^a_o is the interpretation that is just like I with the possible exception that the denotation of a in I^a_o is object o.

The reason the definition of truth for the quantifier needs to be so complicated is that we are trying to define the truth of a formula in terms of the truth values of

smaller formulas. But for a quantified sentence such as ∀xFx, the subformula Fx has no truth value since it contains a free variable. So what we do is let some name, a, be a "temporary" name of all the members of the domain in turn, and see whether the formula Fa is true in each case.

E. Examples

We can now show more rigorously why aLf(f(a)) gets the truth value true in the model considered above, where a denotes me, L means the characteristic function of the loving relation, and f means the father-of function. (Work through it...)

Other examples: the following formulas turn out true in the standard model of arithmetic: $\mathbf{0''+0'''=0'''''}$, ~∃x x′=**0**

IV. Validity, equivalence, and other definitions

A. Validity

- Some synonyms: I *satisfies* S, S is *true in* L; I is a *model of* S; I(S)=1
- S is *satisfiable* iff S is true in some interpretation
- S is *valid* ("⊢S") iff S is true in any interpretation of S

B. Example: show: ⊢ ∀x(xLx→∃y xLy)

Let's proceed by reductio. We'll suppose that this formula is not valid. So it is not true in some interpretation.

Now, it is very easy to show that every sentence is either true or false but not both in every interpretation of that sentence. (Mini proof by induction: This can be seen to be true of atomics by the definition of truth for atomics; and by the other clauses of the truth definition the property of being either true or false but not both is inherited by complex formulas.) Thus, if the formula in question is not true in some interpretation in I, then it is false in I:

I(∀x(xLx→∃y xLy))=0

From the clause in the definition of truth for ∃, we then infer that:

For some o in the domain of I, I^{a}_{o}(aLa→∃yaLy)=0

Call this o "o". From the clause for → we infer:

$I^a_o(aLa)=1$ and $I^a_o(\exists y\ aLy)=0$

From the second thing, by the clause for ∃ we infer:

For every p in the domain of I^a_o, $I^{a\ b}_{o\ p}(aLb)=0$

Now, our object o is in the domain of I, and so in the domain of I^a_o (since I and I^a_o are supposed to be exactly alike except regarding the denotation of the name a); hence, we know:

$I^{a\ b}_{o\ o}(aLb)=0$

By the definition of truth for atomics, we have:

$I^{a\ b}_{o\ o}(L)(I^{a\ b}_{o\ o}(a),I^{a\ b}_{o\ o}(b))=0$

Now let's think about what $I^{a\ b}_{o\ o}(L)$ is: it is the characteristic function assigned to predicate L by model $I^{a\ b}_{o\ o}$. But this interpretation is just like I except regarding the denotations of the names a and b. So $I^{a\ b}_{o\ o}(L)$ is just I(L). Moreover, $I^{a\ b}_{o\ o}(a)$ and $I^{a\ b}_{o\ o}(b)$ are each obviously the object o. So, we have:

I(L)(o,o)=0.

But this is impossible. I(L) is a function; but above we learned that $I^a_o(aLa)=1$, from which it follows that I(L)(o,o)=1. So our reductio assumption is false; the sentence in question is valid.

C. **Implication and equivalence**

We start with the simplest notion of implication, that of a single sentence implying another, and then generalize to the notion of a set implying a sentence:

- $S_1 \vdash S_2$ iff for every interpretation I of both S_1 and S_2, if $I(S_1)=1$ then $I(S_2)=1$
- $S_1, ..., S_n \vdash S$ iff for every interpretation I of $S_1, ..., S_n$, S, if $S_1, ..., S_n$ are all true in I, then so is S
- Where Γ is a (perhaps infinite) set of formulas, $\Gamma \vdash S$ iff for every interpretation I of S and all the members of Γ, if every member of Γ is true in I then so is S
- $S_1 \cong S_2$ ("S_1 and S_2 are logically equivalent") iff for every interpretation, I, of both S_1 and S_2, $I(S_1)$ is identical to $I(S_2)$

NOTE: the symbol $\vdash$ here stands for implication. The relevant sort of implication is *semantic*, since it is defined via models. This symbol is usually used for a different sort of implication, syntactic implication; S_1 syntactically implies S_2 iff S_2 is *provable* from S_1. Similarly, $\vdash S$ is often used to mean that S is a *theorem* (i.e., provable from no premises) rather than that S is valid.

FACTS:
If $S_1 \vdash S_2$ then $\vdash (S_1 \rightarrow S_2)$
If $S_1, ..., S_n \vdash S$ then $\vdash [(S_1 \& ... \& S_n) \rightarrow S]$
$\vdash S$ iff $\emptyset \vdash S$ ($\emptyset$ is the null set)
$S_1 \cong S_2$ iff $S_1 \vdash S_2$ and $S_2 \vdash S_1$
$\cong$ is an equivalence relation

EXAMPLE: Show that $a=b \vdash f(a)=f(b)$

D. Model-of, satisfiability

I is a *model of* S iff I(S)=1
I is a *model of* Γ iff I is a model of each member of Γ
Γ (or S) is *satisfiable* iff some I is a model of Γ (or S)

FACTS:
The null set, $\emptyset$, is satisfiable (vacuously)
$\Gamma \vdash S$ iff $\Gamma \cup \{\sim S\}$ is unsatisfiable (prove it)

E. Equivalence and implication for open formulas

What we want is the following. We want the following two formulas to be equivalent:

$\sim(Fx \& Gy)$
$\sim Fx \vee \sim Gy$

And we want to be able to infer from their equivalence that the following sentences are equivalent:

$\forall x \forall y \sim(Fx \& Gy)$
$\forall x \forall y (\sim Fx \vee \sim Gy)$

I.e., we want to be able to freely substitute logical equivalents within other formulas.

Here are the definitions. Where F_1 and F_2 are any formulas:

- $F_1 \vdash F_2$ iff $F_1^* \vdash F_2^*$ (in the old sense of implication)
- $F_1 \cong F_2$ iff $F_1^* \cong F_2^*$ (in the old sense of equivalence)

WHERE: F_1^* and F_2^* are the results of substituting names for free variables in both F_1 and F_2 such that: i) all the names used are new to both F_1 and F_2, ii) distinct variables get replaced by distinct names, and iii) distinct names are only used for distinct variables

F. **Substitution of equivalents**

As mentioned above, the idea is to vindicate the principle of substitution of equivalents:

Substitution of equivalents (principle 9.14, p. 108):
If $F_1 \cong F_2$, then if G_1 and G_2 are otherwise alike except that some occurrences of F_1 in G_1 have been replaced in G_2 with occurrences of F_2, then $G_1 \cong G_2$

Proof:

This proof will proceed by *strong induction*. The principle of strong induction is the following:

IF: for all m, if P(n) holds for all natural numbers n < m then P(m)
THEN: for all m, P(m)

It should be distinguished from the principle of weak induction:

IF: P(0) and for all m, if P(m) then P(m+1)
THEN: for all m, P(m)

How can induction be used to prove facts about formulas, when induction concerns numbers? Every formula has some finite number of connectives. So to show that every formula has a certain feature, it suffices to show that for any n, every formula with n connectives has the feature. Here we go:

Let's define the "size" of a formula as the number of connectives in that formula other than those in F_1 or F_2. And let's prove our desired assertion

by "induction on the size of the formulas G_1 and G_2 — i.e., we want to show that for all n, any two formulas G_1 and G_2 of size n are equivalent if they differ only by substitution of occurrences of F_2 for occurrences of F_1.

So we assume: (ih): For every n<m, any two formulas of size n are equivalent if they differ only by substitution of occurrences of F_2 for occurrences of F_1

We must now establish that any two formulas of size m are equivalent if they differ only by substitution of occurrences of F_2 for occurrences of F_1. So let G_1 and G_2 be any two such formulas. We must show $G_1 \cong G_2$.

One possibility is that m is 0. In that case (ih) tells us nothing. But in that case, it must be that *either* $G_1=F_1$ and $G_2=F_2$, *or* $G_1=G_2$. Either way, clearly $G_1 \cong G_2$.

Suppose, on the other hand, that m>0. In that case, G_1 and G_2 must take one of the following forms:

i) $G_1 = G_{1a} \& G_{1b}$, $G_2 = G_{2a} \& G_{2b}$
ii) $G_1 = G_{1a} \vee G_{1b}$, $G_2 = G_{2a} \vee G_{2b}$
iii) $G_1 = G_{1a} \rightarrow G_{1b}$, $G_2 = G_{2a} \rightarrow G_{2b}$
iv) $G_1 = G_{1a} \leftrightarrow G_{1b}$, $G_2 = G_{2a} \leftrightarrow G_{2b}$
v) $G_1 = \sim G_{1a}$, $G_2 = \sim G_{2a}$
vi) $G_1 = \forall \alpha G_{1a}$, $G_2 = \forall \alpha G_{2a}$
vii) $G_1 = \exists \alpha G_{1a}$, $G_2 = \exists \alpha G_{2a}$

where G_{1a}, G_{1b}, G_{2a}, and G_{2b} each have size < m, and are such G_{2a} differs from G_{1a} only by substitution of occurrences of F_2 for occurrences of F_1, and G_{2b} differs from G_{1b} only by substitution of occurrences of F_2 for occurrences of F_1. By the inductive hypothesis, we know that $G_{1a} \cong G_{2a}$ and $G_{1b} \cong G_{2b}$. We must, in each case, argue that $G_1 \cong G_2$.

Cases i)-v) are similar. To show that $G_1 \cong G_2$, we must show that for any interpretation, I, $I(G_1^*)=I(G_2^*)$. In cases i)-v), G_1 and G_2 are boolean combinations of formulas G_{1a} and G_{1b}, and G_{2a} and G_{2b}, respectively; hence G_1^* and G_2^* are boolean combinations of what we can call G_{1a}^* and G_{1b}^*, and G_{2a}^* and G_{2b}^*, respectively. But since we know that $G_{1a} \cong G_{2a}$ and $G_{1b} \cong G_{2b}$, we know that in any interpretation, I, $I(G_{1a}^*)=I(G_{2a}^*)$, and $I(G_{1b}^*)=I(G_{2b}^*)$. But notice that the truth value of a boolean combination of formulas, in any interpretation, is a function of the truth values of its parts. Hence, since the parts of G_1^* and G_2^* have the same truth values in I, G_1^* and G_2^* themselves have the same truth value in I.

Cases vi) and vii) remain. Take vi). We must show $\forall\alpha G_{1a} \cong \forall\alpha G_{2a}$, and thus we must show that for any I, $I((\forall\alpha G_{1a})^*)=I((\forall\alpha G_{2a})^*)$. By my placement of the parentheses here I mean to indicate that the *-ing operation applies to the entire formulas including the quantifier $\forall\alpha$. Thus, in $(\forall\alpha G_{1a})^*$ and $(\forall\alpha G_{2a})^*$, occurrences of the variable α have *not* been changed to a new name, since those occurrences are not free in these formulas. Consider now G_{1a} and G_{2a}. These have size < m, and also differ only by substitution of occurrences of F_2 for occurrences of F_1. So by the inductive hypothesis, $G_{1a} \cong G_{2a}$. That is, **for any I, $I(G_{1a}^*)=I(G_{2a}^*)$**. In *-ing G_{1a} and G_{2a}, let us choose the very same names to replace free variables as we did in *-ing $\forall\alpha G_{1a}$ and $\forall\alpha G_{2a}$. Note that some occurrences of the variable α may well be free in G_{1a} and G_{2a}, in which case these occurrences will be replaced uniformly by some name a that was not used in *-ing $\forall\alpha G_{1a}$ and $\forall\alpha G_{2a}$. OK. Let us return to what we need to show, that for any I, $I((\forall\alpha G_{1a})^*)=I((\forall\alpha G_{2a})^*)$. What is $I((\forall\alpha G_{1a})^*)$? Call the part of $(\forall\alpha G_{1a})^*$ that occurs after $\forall\alpha$, "G". Then, $I((\forall\alpha G_{1a})^*)$ is 1 or 0 depending on whether for all o in the domain of I, $I^a_o(G\neg a)=1$. Note that here I chose a as my name with which to replace free occurrences of α — the very same name I used to replace α while *-ing G_{1a} and G_{2a}. This is allowed, because the name a does not occur in G. But that means that $G\neg a$ — i.e., the result of replacing all free occurrences of α in G with the name a — is simply G_{1a}^*. Therefore, $I((\forall\alpha G_{1a})^*)$ is 1 or 0 depending on whether for all o in the domain of I, $I^a_o(G_{1a}^*)=1$. By similar reasoning, $I(I((\forall\alpha G_{2a})^*)$ is 1 or 0 depending on whether for all o in the domain of I, $I^a_o(G_{2a}^*)=1$. But given the boldface statement above, one of these last two statements holds iff the other does. Therefore, we have what we wanted to prove: for any I, $I((\forall\alpha G_{1a})^*)=I((\forall\alpha G_{2a})^*)$.

Case vii) is basically the same. QED.

V. **Prenex**

Say that a formula is in *prenex normal form* if all the quantifiers in that formula are "out front" — i.e., if it looks like this: $Q_1\alpha_1 \ldots Q_n\alpha_n \phi$, where each Q_i is either $\exists$ or $\forall$, $\alpha_1, \ldots, \alpha_n$ are variables, and ϕ has no quantifiers in it.

It is a fact that every formula is equivalent to some formula in prenex. The basic idea of how to prove this is as follows. (Homework assignment 4 is to give a rigorous proof of this fact.)

Here are some equivalences (verifying that these really are equivalences is part of the homework assignment). Let G be a formula containing no free occurrences of variable α;

where Q is any quantifier, let Q′ be the other quantifier (i.e., ∀ if Q is ∃; ∃ is Q is ∀):

(9.16)	$\sim QvF \cong Q'v\sim F$
(9.17):	$QvF \,\&\, G \cong Qv(F\&G)$
	$G \,\&\, QvF \cong Qv(G\&F)$
	$QvF \vee G \cong Qv(F\vee G)$
	$G \vee QvF \cong Qv(G\vee F)$
	$QvF \rightarrow G \cong Q'v(F\rightarrow G)$
	$G \rightarrow QvF \cong Qv(G\rightarrow F)$
(9.18)	$QvF \cong QwF_v w$

You can use these equivalences (plus the principle of substitution of equivalents) to transform any formula lacking the ↔ into an equivalent formula in prenex normal form. As for the ↔, any formula $\phi\leftrightarrow\psi$ is equivalent to $(\phi\rightarrow\psi)\&(\psi\rightarrow\phi)$.

Here's an example of how the transformations are done:

• $\exists xFx \rightarrow \sim\forall x(\exists yRxy \vee Gx)$	
• $\forall x(Fx\rightarrow\sim\forall x(\exists yRxy \vee Gx))$	(9.17)
• $\forall x(Fx\rightarrow\exists x\sim(\exists yRxy \vee Gx))$	(9.16), plus substitution of equivalents (stress the reliance on S of E here)
• $\forall z(Fz\rightarrow\exists x\sim(\exists yRxy\vee Gx))$	(9.18)
• $\forall z\exists x(Fz\rightarrow\sim(\exists yRxy\vee Gx))$	(9.17), S of E
• $\forall z\exists x(Fz\rightarrow\sim\exists y(Rxy\vee Gx))$	″
• $\forall z\exists x(Fz\rightarrow\forall y\sim(Rxy\vee Gx))$	″
• $\forall z\exists x\forall y(Fz\rightarrow\sim(Rxy\vee Gx))$	″

Each formula is equivalent to the last for the reasons cited; by transitivity of ≅, the final formula is equivalent to the first.

CHAPTER 10: Undecideability

I. Positive vs. negative tests

The goal of this chapter is to show that there can be no mechanical procedure for establishing whether a given formula is valid or invalid. But first this assertion must be clarified.

A *positive* mechanical test for a certain property is a procedure such that i) if a given thing has the property, the procedure eventually says *yes*, and ii) if the thing does not have the property, the procedure will never say *yes*. This is not to say that the procedure will

ever say *no*, if the object does not have the property. That's the job of a negative test.

A *negative* mechanical test for a property is a procedure that is such that i) if a given thing lacks the property, the procedure eventually says so (i.e., it says *no*); and ii) if a given thing has the property, the procedure never says that it doesn't have it (i.e., it never says *no*). A negative test for a property need not say *yes* if the thing has the property; it just must refrain from saying *no*.

A *decision procedure* for a given property combines both of these features: i) if a given thing has the property, it eventually says *yes*, and will never say *no*; and ii) if a given thing lacks the property, the procedure eventually says *no*, and will never say *yes*. A decision procedure (eventually) tells you *whether or not* the property holds of a given object.

Note that if you have a positive and negative test for a property, it is easy to construct a decision procedure. Simply use the "time-sharing" technique to alternate between the positive and negative tests: run the positive test for one minute, then the negative test for one minute, then the positive test, then the negative test .... Eventually one of those two tests will give you an answer.

II. Outline of the proof that first-order logic is undecidable

What we will prove is that there is no decision procedure for validity for first-order logic. We will do this by "reducing" the problem of finding a decision procedure for validity to the problem of solving the halting problem — i.e., we will show that if you *could* find a decision procedure for validity then you could solve the halting problem. But we know you can't do that.

Here's the outline of the proof:

- Given any Turing machine M and integer n, we'll show how to effectively construct a certain finite set of sentences Δ and another sentence, H
- These will be such that **$\Delta \vdash$H iff M will eventually halt given input n**
- But $\Delta \vdash H$ iff $\vdash \text{conj}(\Delta) \rightarrow H$.
- So if we had a decision procedure for validity, we would have a decision procedure for halting: we could tell whether M will eventually halt by constructing $\text{conj}(\Delta) \rightarrow H$ and then running the validity-machine.

Note the liberal use of Church's thesis. What we showed rigorously in chapter 5 was that there is no *Turing-machine* that computes a certain function, h, corresponding to the halting problem. But the proof we will give does not show how one could construct a Turing machine that would compute h; rather, it sketches an intuitively mechanical procedure one could follow. Church's thesis then insures us that this could be

implemented on a Turing machine.

Informally, Δ and H will be constructed as follows. Δ will describe how the Turing machine flow graph works, and it will describe the initial set-up of the tape. H will say that the machine will eventually halt. Thus, conj(Δ)→H will say that *if* you have such and such a Turing machine and you run it on such and such input, then it will eventually halt.

III. A language and an interpretation for Δ and H

Now, sentences in and of themselves don't *say* anything; they only say things relative to interpretations. We will introduce a language for the sentences in Δ and H, and an interpretation of that language in which those sentences concern Turing machines.

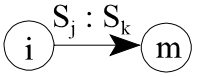
- The domain of the interpretation is the set of integers, both negative and positive
- These can be thought of as representing times (0, 1, ...) and also squares on the tape (...-2, -1, 0, 1, 2, ...).
- The machine is started at t=0, on square 0.

Our language includes:

- for each state q_i of the Turing machine, a two-place predicate Q_i. $Q_i$tx will mean, in the interpretation, that at time t, the machine is in state q_i and is scanning square x.
- for each symbol S_j that the machine can write, a two-place predicate S_j; $S_j$tx means that at time t the symbol S_j is on square x.
- a two-place predicate < that means the less-than relation in the interpretation we're discussing)
- a name, **0**, that denotes 0 (in our interpretation)
- a one-place function symbol ′, that means the successor function (in our interpretation)

IV. Constructing Δ

Let us suppose that the machine writes symbols S_0 ..., S_r. We go through the flow graph of the Turing machine, and for each arrow in the flow graph we include an appropriate sentence in Δ:

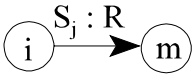
For each arrow 

we include a sentence of the form:

$$\forall t \forall x \{ [tQ_i x \& tS_j x] \rightarrow [t' Q_m x \ \& \ t' S_k x \ \& \ \forall y (y \neq x \rightarrow ((tS_0 y \rightarrow t' S_0 y) \ \& \ \ldots \ \& \ (tS_r y \rightarrow t' S_r y)))] \}$$

which means:

> If at some time the machine is scanning square x, is in state q_i, and symbol S_j is on square x, then at the next moment it will still be scanning that same square, will be in state q_m, but symbol S_k will then be in square x; moreover, at the next moment all the squares other than x will have the same symbols as they had previously.

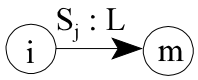
For each arrow 

we include a sentence of the form:

$$\forall t \forall x \{ [tQ_i x \& tS_j x] \rightarrow [t' Q_m x' \ \& \ \forall y ((tS_0 y \rightarrow t' S_0 y) \ \& \ \ldots \ \& \ (tS_r y \rightarrow t' S_r y))] \}$$

which means:

> If at some time t the machine is scanning square x and is in state q_i, the next moment it will be in state q_m scanning the next square; all the symbols on the tape will be the same afterwards as before.

For each arrow 

we include a sentence of the form:

$$\forall t \forall x \{ [tQ_i x' \& tS_j x'] \rightarrow [t' Q_m x \ \& \ \forall y ((tS_0 y \rightarrow t' S_0 y) \ \& \ \ldots \ \& \ (tS_r y \rightarrow t' S_r y))] \}$$

which means:

> If at some time t the machine is scanning a square and is in state q_i, then the next moment it will be scanning the preceding square, will be in state q_m, and all the symbols on the tape will remain the same.

We also need to include a description of how the machine starts. The following sentence says that the machine is initially in state q_1 scanning the leftmost of an unbroken sequence of n 1s, on an otherwise empty tape. (Equation 10.4, p. 115)

$$\mathbf{0} Q_1 \mathbf{0} \ \& \ \mathbf{0} S_1 \mathbf{0} \ \& \ \mathbf{0} S_1 \mathbf{0}' \ \& \ \ldots \ \mathbf{0} S_1 \mathbf{0}^{(n-1)} \ \& \ \forall y [(y^1 \mathbf{0} \ \& \ y^1 \mathbf{0}' \ \& \ \ldots \ \& \ y^1 \mathbf{0}^{(n-1)}) \rightarrow \mathbf{0} S_0 y]$$

Note the notation: $\mathbf{0}^{(n)}$ means **0** with n successor signs.

For reasons that will become clear below, we will need sentences in Δ that will have certain implications about the successor sign. Here are two. One says (in the intended model) that each integer is the successor of exactly one integer:

$$\forall z \exists x\ z=x' \ \&\ \forall z \forall x \forall y(\ [z=x' \ \&\ z=y'] \rightarrow x=y) \qquad (10.5)$$

The other has the sentence $\forall x\ x^{(p)} \neq x^{(q)}$ as a logical consequence whenever p and q are different natural numbers:

$$\forall x \forall y \forall z[(x<y \ \&\ y<z) \rightarrow x<z] \ \&\ \forall x \forall y(x'=y \rightarrow x<y) \ \&\ \forall x \forall y(x<y \rightarrow x \neq y) \qquad (10.6)$$

(You could do an inductive proof that $\forall x\ x^{(p)} \neq x^{(q)}$ is a consequence of (10.6). But here's an example. Let's show that $\forall x\ x' \neq x'''$. To show this is implied, we'll proceed by "universal proof": we'll show that an arbitrary instance, namely $a' \neq a'''$, is a logical consequence of 10.6. Well, $a''=a''$ is a logical truth; and the middle conjunct implies $a''=a'' \rightarrow a'<a''$, and so implies: $a'<a''$. Similarly, the middle conjunct implies $a''<a'''$. But then the first conjunct implies: $a'<a'''$; and then the final conjunct implies $a' \neq a'''$. By universal proof, 10.6 implies: $\forall x\ x' \neq x'''$.)

V. Constructing H

H, recall, is to say in the intended interpretation that the machine eventually halts. Now, a Turing machine halts if it is in a state, q_i, which has no arrow for some symbol S_j, but the machine is at that time scanning a square that has symbol S_j. So we just let H be the following:

$$\ldots \vee \exists t \exists x(tQ_i x \& tS_j x) \vee \ldots$$

I.e., the disjunction of all sentences saying that at some time the machine is in one of these potential halting states, scanning some square that contains the halting symbol.

VI. If Δ⊢H then M will eventually halt given input n

We have now shown how to effectively construct Δ and H. Recall that the goal was to have Δ and H be such that **Δ⊢H iff M will eventually halt given input n**. So now this biconditional must be proved. One direction is easy:

> Suppose **Δ⊢H**. Clearly, every sentence in Δ is true in the interpretation we defined above. So H is true in that interpretation as well. But then the machine eventually halts, because H says in that interpretation that the machine eventually hits a state and is looking at a symbol for which there's no arrow.

VII. **If M will eventually halt given input n, then Δ⊢H**

This is harder. Here is our overall strategy.

i) We will develop the notion of a *description of time s*. This is a sentence that completely describes the state of the tape at time s. Here is the general structure of a description of time s: [(10.8), p. 117]:

$$\mathbf{0}^{(s)}Q_i\mathbf{0}^{(p)} \,\&\, \mathbf{0}^{(s)}S_{j1}\mathbf{0}^{(p1)} \,\&\, \ldots \,\&\, \mathbf{0}^{(s)}S_j\mathbf{0}^{(p)} \,\&\, \ldots \,\&\, \mathbf{0}^{(s)}S_{jk}\mathbf{0}^{(pk)} \,\&$$
$$\forall y[(y\neq\mathbf{0}^{(p1)} \,\&\, \ldots \,\&\, y\neq\mathbf{0}^{(p)} \,\&\, \ldots \,\&\, y\neq\mathbf{0}^{(pk)}) \rightarrow \mathbf{0}^{(s)}S_0y]$$

Here p1 . . . p . . . pk must be an increasing sequence of integers. In other words, relative to the interpretation we have specified, a description of time s says what state the machine is in at time s, what square it is scanning then, and what symbols are on the tape then.

ii) We will prove by induction that <u>if the machine hasn't halted before time s, then Δ implies some description of s</u>

iii) Given this, we can reason as follows. Suppose the machine eventually halts at some time; consider the moment, s, before then. It must have been that at moment s, the machine was in a state q_i, at a square p, which contained some symbol S_j, such that q_i had no arrow for symbol S_j. By ii), Δ implies some description, G, of s. Since Δ is true in the interpretation we introduced, so is G. But that means that the G must contain these two conjuncts: $\mathbf{0}^{(s)}Q_i\mathbf{0}^{(p)}$ and $\mathbf{0}^{(s)}S_j\mathbf{0}^{(p)}$ (because the symbols, states, and squares that G represents must "match up" with s, q_i, S_j and p.) Therefore, G — and so Δ — logically implies $\exists t\exists x(tQ_ix\&tS_jx)$, and so logically implies H.

VIII. <u>Representing negative numbers</u>

These definitions require that we have a way of representing negative numbers on the tape. But we don't have any function term for the predecessor function. So how do we represent, e.g., $xQ_i\mathbf{0}^{(-2)}$? As follows: $\exists z(xQ_iz \,\&\, z^{(2)}=\mathbf{0})$.

In what follows I will assume that statements of the form $xQ_i\mathbf{0}^{(p)}$ can be manipulated in the same way, whether p is positive or negative. Here is a sketch of why this is OK. Suppose, for example, that we want to move from $\mathbf{0}Q_i\mathbf{0}^{(p)}$ and $\forall x\forall y(xQ_iy\rightarrow x'Q_jy)$ to $\mathbf{0}'Q_i\mathbf{0}^{(p)}$ As will be argued below, "universal instantiation" is a valid rule, so if $\mathbf{0}^{(p)}$ really were a term, then the inference would be unproblematic. But in fact $\mathbf{0}^{(p)}$ is not a term if p=-q, for some positive integer q. But we can still get the effect. The first premise is just

an abbreviation for:

$$\exists z(\mathbf{0}Q_i z \;\&\; z^{(q)}=\mathbf{0})$$

It would be easy to show that this and the second premise logically entail:

$$\exists z(\mathbf{0}'Q_j z \;\&\; z^{(q)}=\mathbf{0})$$

But this is an abbreviation for:

$$\mathbf{0}'Q_j\mathbf{0}^{(p)}$$

IX. Proof that if the machine hasn't halted before time s, then Δ implies some description of s

The proof is by induction on the time, s.

Base case: s=0. The set Δ contains, and so implies, a description of 0 — the sentence that was put in to represent the initial configuration of the tape (equation 10.4)

Induction case

We begin by assuming (ih) that if the machine hasn't halted before time s, then Δ implies some description of s, and try to show that if the machine hasn't halted before s+1, Δ implies some description of s+1.

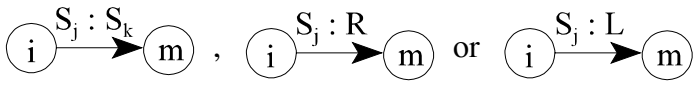
So, suppose the machine hasn't halted by s+1. Then it hadn't halted by s either, and so by (ih), Δ implies some description of s:

$$\mathbf{0}^{(s)}Q_i\mathbf{0}^{(p)} \;\&\; \mathbf{0}^{(s)}S_{j1}\mathbf{0}^{(p1)} \;\&\; \ldots \;\&\; \mathbf{0}^{(s)}S_j\mathbf{0}^{(p)} \;\&\; \ldots \;\&\; \mathbf{0}^{(s)}S_{jk}\mathbf{0}^{(pk)} \;\&$$
$$\forall y[(y\neq\mathbf{0}^{(p1)} \;\&\; \ldots \;\&\; y\neq\mathbf{0}^{(p)} \;\&\; \ldots \;\&\; y\neq\mathbf{0}^{(pk)}) \rightarrow \mathbf{0}^{(s)}S_0 y]$$

Moreover, since every sentence in Δ is true in the intended model, this sentence is as well; therefore at time s, the machine is in state q_i and is scanning square p, which contains symbol S_j; the other squares on the tape also contain the other symbols represented by this sentence.

We must now show that Δ implies a description of s+1.

Well, since the machine didn't halt at s, it must have encountered an arrow — something of one of the following forms:



But in each case, Δ must then have contained sentences sufficient to allow us to show that Δ implies a description of s+1. In the first case, Δ contains the following sentence:

$$\forall t \forall x \{ [tQ_i x \& tS_j x] \rightarrow [t' Q_m x \ \& \ t' S_k x \ \& \ \forall y (y \neq x \rightarrow ((tS_0 y \rightarrow t' S_0 y) \ \& \ \ldots \ \& \ (tS_r y \rightarrow t' S_r y)))] \}$$

But this, together with the description of time s above, implies the following sentence:

$$\mathbf{0}^{(s+1)} Q_m \mathbf{0}^{(p)} \ \& \ \mathbf{0}^{(s+1)} S_{j1} \mathbf{0}^{(p1)} \ \& \ \ldots \ \mathbf{0}^{(s+1)} S_k \mathbf{0}^{(p)} \ \& \ \ldots \ \mathbf{0}^{(s+1)} S_{jk} \mathbf{0}^{(pk)} \ \& \ \forall y [(y \neq \mathbf{0}^{(p1)} \ \& \ \ldots \ \& \ y \neq \mathbf{0}^{(p)} \ \& \ \ldots \ \& \ y \neq \mathbf{0}^{(pk)}) \rightarrow \mathbf{0}^{(s+1)} S_0 y]$$

This is a description of s+1.

Now, why does this implication hold? At this point it would take forever if we tried to rigorously prove every implication that we're going to need to assert to hold. Basically we'll need to assume that our definition of validity is such that the ordinary arguments we customarily take to be valid in introductory logic are indeed valid. These arguments could be shown to be valid, if we wanted to bother. For example, universal instantiation is this argument:

$$\forall v \phi \vdash \phi_v t$$

where t is any closed term (i.e., any term without variables). Here's why this is valid. Let I be any interpretation of these two sentences, such that $I(\forall v \phi)=1$. Then, by the truth definition of ∀, where o is the denotation of t in I, $I^b_o(\phi_v b)=1$, where b does not occur in either ϕ or t. We can then conclude that $I(\phi_v t)=1$ using two principles:

Continuity: If I and I′ are interpretations that differ *only* in what they assign to symbols *not* occurring in sentence S, then I(S)=I(S′)

Extensionality: If t and t′ are terms, and if I(t)=I(t′), then I(ϕ)=I(ϕ′), where ϕ and ϕ′ are sentences that are exactly alike except that some or all instances of t in ϕ have been replaced by t′ in ϕ′.

(Each of these principles is intuitively true, and could be established by a quick induction.) For by extensionality, $I^b_o(\phi_v b) = I^b_o(\phi_v t)$; and by continuity, $I^b_o(\phi_v t) = I(\phi_v t)$ (b does not occur in $\phi_v t$ since b did not occur in either ϕ or t).

At any rate, we will be assuming that universal instantiation and other familiar forms of inference are indeed valid within our logical language. So, back to the task. We're trying to show that the sentences implied by Δ, which include:

(1) $\mathbf{0}^{(s)}Q_i\mathbf{0}^{(p)} \,\&\, \mathbf{0}^{(s)}S_{j1}\mathbf{0}^{(p1)} \,\&\, \ldots \,\&\, \mathbf{0}^{(s)}S_j\mathbf{0}^{(p)} \,\&\, \ldots \,\&\, \mathbf{0}^{(s)}S_{jk}\mathbf{0}^{(pk)} \,\&$
$\forall y[(y\neq\mathbf{0}^{(p1)} \,\&\, \ldots \,\&\, y\neq\mathbf{0}^{(p)} \,\&\, \ldots \,\&\, y\neq\mathbf{0}^{(pk)}) \rightarrow \mathbf{0}^{(s)}S_0 y]$

and

(2) $\forall t\forall x\{[tQ_i x \& tS_j x] \rightarrow [t'Q_m x \,\&\, t'S_k x \,\&$
$\forall y(y\neq x \rightarrow ((tS_0 y \rightarrow t'S_0 y) \,\&\, \ldots \,\&\, (tS_r y \rightarrow t'S_r y)))]\}$

imply the following description of s+1:

(3) $\mathbf{0}^{(s+1)}Q_m\mathbf{0}^{(p)} \,\&\, \mathbf{0}^{(s+1)}S_{j1}\mathbf{0}^{(p1)} \,\&\, \ldots\, \mathbf{0}^{(s+1)}S_k\mathbf{0}^{(p)} \,\&\, \ldots\, \mathbf{0}^{(s+1)}S_{jk}\mathbf{0}^{(pk)} \,\&$
$\forall y[(y\neq\mathbf{0}^{(p1)} \,\&\, \ldots \,\&\, y\neq\mathbf{0}^{(p)} \,\&\, \ldots \,\&\, y\neq\mathbf{0}^{(pk)}) \rightarrow \mathbf{0}^{(s+1)}S_0 y]$

Well, from (1) we get:

$\mathbf{0}^{(s)}Q_i\mathbf{0}^{(p)} \,\&\, \mathbf{0}^{(s)}S_j\mathbf{0}^{(p)}$

Then by UI on (2) on the terms $\mathbf{0}^{(s)}$ and $\mathbf{0}^{(p)}$ and modus ponens we get:

(+) $\mathbf{0}^{(s+1)}Q_m\mathbf{0}^{(p)}) \,\&\, \mathbf{0}^{(s+1)}S_k\mathbf{0}^{(p)} \,\&$
$\forall y(y\neq\mathbf{0}^{(p)} \rightarrow ((\mathbf{0}^{(s)}S_0 y \rightarrow \mathbf{0}^{(s+1)}S_0 y) \,\&\, \ldots \,\&\, (\mathbf{0}^{(s)}S_r y \rightarrow \mathbf{0}^{(s+1)}S_r y))$

The first conjunct of this is the first conjunct of (3): $\mathbf{0}^{(s+1)}Q_m\mathbf{0}^{(p)}$. Moreover, the second conjunct of this is another of the conjuncts in (3): $\mathbf{0}^{(s+1)}S_k\mathbf{0}^{(p)}$.

We now need the other conjuncts of the first half of (3) — we need to derive $\mathbf{0}^{(s+1)}S_{ji}\mathbf{0}^{(pi)}$, where pi≠p. Well, for each such i, we have in (1):

$\mathbf{0}^{(s)}S_{ji}\mathbf{0}^{(pi)}$

Moreover, (10.6) is a member of D, and it entails:

$\forall x\ x^{(pi)}\neq x^{(p)}$

and so

$$\mathbf{0}^{(pi)} \neq \mathbf{0}^{(p)}$$

So from this and the second half of (+) we get

$$\mathbf{0}^{(s)}S_{ji}\mathbf{0}^{(pi)} \rightarrow \mathbf{0}^{(s+1)}S_{ji}\mathbf{0}^{(pi)}$$

and so

$$\mathbf{0}^{(s+1)}S_{ji}\mathbf{0}^{(pi)}$$

Finally we need to establish the final bit of (3):

$$\forall y[(y \neq \mathbf{0}^{(p1)} \,\&\, .\, .\, \&\, y \neq \mathbf{0}^{(p)} \,\&\, \ldots \,\&\, y \neq \mathbf{0}^{(pk)}) \rightarrow \mathbf{0}^{(s+1)}S_0 y]$$

We proceed by universal derivation and conditional derivation (which could easily be shown to be sound methods for establishing validities); we will try to establish that this formula is a consequence of D by showing that

$$a \neq \mathbf{0}^{(p1)} \,\&\, .\, .\, \&\, a \neq \mathbf{0}^{(p)} \,\&\, \ldots \,\&\, a \neq \mathbf{0}^{(pk)} \rightarrow \mathbf{0}^{(s+1)}S_0 a$$

(where a does not occur anywhere in Δ) is a consequence of the sentences in Δ; and we show this by showing that the consequent of this conditional is a consequence of the sentences in Δ plus its antecedent. Well, the last part of (1) is

$$\forall y[(y \neq \mathbf{0}^{(p1)} \,\&\, .\, .\, \&\, y \neq \mathbf{0}^{(p)} \,\&\, \ldots \,\&\, y \neq \mathbf{0}^{(pk)}) \rightarrow \mathbf{0}^{(s)}S_0 y]$$

From this and the antecedent of the conditional, we get:

(*) $\mathbf{0}^{(s)}S_0 a$

Moreover, the antecedent of the conditional implies

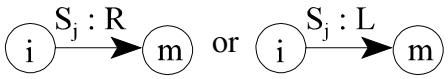
$$a^{1}\mathbf{0}^{(p)}$$

Which, together with the last part of (+), implies

$$\mathbf{0}^{(s)}S_0 a \rightarrow \mathbf{0}^{(s+1)}S_0 a$$

which, together with (*), implies the consequent of the conditional.

We still have yet to establish that D implies a description of s+1 in the remaining two cases, in which the machine encounters an arrow of the form



or

. But these cases differ from the one we just considered only in details. In each case, we added a sentence to D corresponding to the arrow in question in the flow graph, and this sentence, together with the others in D, can be used to derive a description of s+1. (Those descriptions are listed in the book, p. 118.)

CHAPTER 11: Derivations and Soundness

I. Soundness and completeness

We have shown that there can be no decision procedure for validity — no procedure that is both a negative test and a positive test for validity. Our goal in the next two chapters is to show that there *is* a *positive* test for validity.

Recall that a positive test must say "yes" eventually for a given formula if and only if that formula is valid. The "if" and the "only if" part are usually given separate names:

Soundness: if the test says "yes", then the formula is valid

Completeness: if the formula is valid, the test says "yes"

These names, in fact, are usually used when the test is some sort of proof procedure. They then mean:

Soundness: if a formula is provable then it is valid
Completeness: if a formula is valid then it is provable

II. Validity and satisfaction

We will actually work with the concept of unsatisfiability of finite sets of sentences, rather than validity, because:

Finite Δ is unsatisfiable iff $\vdash \sim conj(\Delta)$
S is valid iff $\{\sim S\}$ is unsatisfiable

Aside: There cannot be any positive test for satisfiability. For suppose there were. Consider the formula $conj(\Delta) \rightarrow H$ from the last chapter. It is valid iff its negation is unsatisfiable. But it is valid iff the machine halts. Therefore, the machine halts

iff its negation is unsatisfiable, and so the machine does not halt iff it *is* satisfiable. So, if we had a positive test for satisfiability, we would have a negative test for halting — i.e., a procedure such that it will eventually tell us if a machine will never halt, and it will never incorrectly say a machine will never halt when the machine in fact will halt. But we *do* have a positive test for halting: just build the machine and run it. So we would have a decision procedure for halting, which is impossible.

III. **Derivations**

Our positive test for validity will be a sort of derivation system. However, it will be an odd match of syntactic and semantic notions. Here's the idea. We want to apply our test to finite sets of sentences Δ, with the goal of showing Δ to be unsatisfiable. The method will be to produce *derivations* from Δ in a familiar way, which ultimately result in lines in the derivation that are i) quantifier free, and ii) are unsatisfiable. Such a derivation will be a "refutation" of Δ.

More carefully, we define a *derivation from Δ* as follows.

- The sentences in Δ are assumed to be in prenex normal form. (If they're not, then just replace them with prenex equivalents. This can be done in a mechanical fashion.)

- A derivation from Δ is defined as a list of formulas and annotations, each member of which is either i) a member of Δ, and annotated Δ, or ii) follows from an earlier line n via a rule of inference, and is annotated n.

- The rules of inference are:

UI		EI	
$\forall v\phi$		$\exists v\phi$	
-------		------	
$\phi_v t$	(where t is *any* term)	$\phi_v a$	(where a is a name *new*, both to Δ and the derivation)

- A *refutation* of Δ is a derivation from Δ that contains a finite set of quantifier free lines that is unsatisfiable.

IV. **Example**

$\{\exists x \forall y\ xLy, \sim\forall y \exists x\ xLy\}$,

Prenex equivalent: $\Delta = \{\exists x \forall y\ xLy, \exists y \forall x\ \sim xLy\}$

Here is a refutation of Δ:

1.	$\exists x \forall y\ xLy$	Δ
2.	$\exists y \forall x\ \sim xLy$	Δ
3.	$\forall y\ aLy$	1
4.	$\forall x\ \sim xLb$	2
5.	aLb	3
6.	$\sim aLb$	4

Lines 5 and 6 are quantifier-free and jointly unsatisfiable. In this case, the quantifier-free lines were a contradiction — i.e., a formula and its negation. But this isn't required in general. How does one tell that a finite set of quantifier-free sentences is unsatisfiable? It will be shown later that there is a mechanical procedure for doing this.

V. **Basic property of EI**

The rule EI can be given the familiar justification: it isn't really a sound rule, but we can treat the instantial name *a* as a temporary name of something of which ϕ is true. The following principle summarizes this, and also will be used in proofs below:

Basic property of EI: Suppose the sentence $\exists v\phi$ is a member of Γ, suppose name t does not occur in Γ, and let I be a model of Γ. Then there is some object o in the domain of I such that I^t_o is a model of $\Gamma \cup \{\phi_v t\}$

Proof: Since $I(\exists v\phi)=1$, for some o in the domain of I, $I^t_o(\phi_v t)=1$. Moreover, since every member of Γ was true in I, by continuity, every member of Γ is true in I^t_o, since t doesn't occur in Γ.

VI. **Soundness theorem**

ST: If there is a refutation of Δ then Δ is unsatisfiable, where Δ is a set of sentences in prenex with no vacuous quantifiers

We prove the stronger strong soundness theorem, from which ST follows:

SST: If I is a model of Δ, and D is a (possibly infinite) derivation from Δ, then there exists a model, L, of the set of all sentences occurring in D. Moreover, L can be chosen so that it differs from I (if at all) only over the new names and function symbols that are in D but not Δ.

ST follows from the first sentence in SST. For let Δ have a refutation, D, and suppose for reductio that Δ is satisfiable. Then the set of all the sentences in D has a model; but D was supposed to be a refutation, and so it has an unsatisfiable subset.

VII. **Proof of SST**

Let I be a model, and let the following be a derivation from I:

1.	S_1	A_1
2.	S_2	A_2
3.	S_3	A_3
.	.	.
.	.	.
.	.	.

Let us now define a sequence of sets, Δ_i, as follows:

$\Delta_0 = \Delta$
$\Delta_n = \Delta \cup \{S_1 \ldots S_n\}$, so long as D has at least n lines

Let us also define a sequence of interpretations. Our goal is to have it be the case that for each k, I_k is a model of Δ_k:

$I_0 = I$
I_{k+1} is defined by cases, as a function of I_k, assuming S_{k+1} exists. In each case, I_{k+1} will be defined in such a way that I_{k+1} is guaranteed to be a model of Δ_{k+1} *if* I_k was a model of Δ_k. Given that I_0 is indeed a model of Δ_0, then every I_k will indeed be a model of Δ_k:

Case 1: $A_{k+1} = \Delta$. Then let I_{k+1} be I_k (since $\Delta_{k+1} = \Delta_k$)

Case 2: A_{k+1} refers to a line with a universally quantified formula, and the instantial terms contains no new names or function symbols beyond those in Δ_k. Here again, we let I_{k+1} be I_k. Why? Because i) the designation of the term is in the domain of I, and ii) as we mentioned above, universal instantiation is a valid rule — if its premise is true in a model, and the model is an interpretation of the conclusion, then the conclusion is true in that model as well.

Case 3: Same as the last case, but new names or function symbols are present in the instantial term. Here, we suppose that at the beginning of the proof we have chosen a certain object, d, for use

in these situations. (Why avoid using the axiom of choice? We need it later anyway...). We now let I_{k+1} be just like I_k, except that all the new names refer to d, and all the new function symbols get assigned constant functions whose values are always d. Thus constructed, I_{k+1} is a model of Δ_{k+1}, since i) I_k and I_{k+1} differ only over names that don't appear in Δ_k, and so I_{k+1}, by continuity, is a model of Δ_k, and ii) UI's validity then ensures that I_{k+1} is also a model of Δ_{k+1}.

Case 4: A_{k+1} refers to a line with an existentially quantified sentence. Here, the instantial term, t, is a name that is new to the derivation and to Δ. So, we can use the basic property of EI to conclude that for some o in the domain of I_k, $I_k{}^t{}_o$ is a model of $\{S_{k+1}\} \cup \Delta_k$ — i.e., is a model of Δ_{k+1}. We let $I_{k+1} = I_k{}^t{}_o$, for some such o. (Note the use of the axiom of choice here.)

We have now defined a sequence of sets, Δ_k, and a sequence of interpretations, I_k, such that each I_k is a model of Δ_k. Our goal is to locate a model, L, of the set of all the sentences in the derivation D. We cannot set L equal to the last I_k, because the derivation might be infinite. Instead we do this:

L = the model that is just like I, except that: for each name or function symbol in D but not in Δ, L assigns to it whatever I_k assigns it, where S_k is the first place in D in which that name or function symbol occurs.

We now show that L is a model of all the sentences in the derivation. Let S_k be any such sentence. S_k is true in I_k. But I_k and L differ, if at all, only what they assign to names and function symbols that occur in the derivation only later than spot k, and thus names and function symbols that don't occur in S_k (since the denotation in L of a name not in Δ is fixed by its interpretation in the first I_k that interprets it **-- be more explicit here**); so, by continuity, $I_k(S_k)=L(S_k)$. QED.

VIII. The significance of soundness

The soundness theorem thus provides us a method for establishing that sentences are valid and that implications hold. For a sentence S is valid iff $\{\sim S\}$ is unsatisfiable, so we can show S to be valid by producing a refutation of $\{\sim S\}$. Moreover, $\Gamma \vdash S$ iff $\Gamma \cup \{\sim S\}$ is unsatisfiable; therefore we can show that $\Gamma \vdash S$ by producing a refutation of $\Gamma \cup \{\sim S\}$.

This is not to say that we have yet established the existence of a mechanical positive test for validity:

- we have not described a mechanical procedure for constructing refutations — we just have our ingenuity.

- we have no guarantee that a refutation will always exist if the set is unsatisfiable.

- we have not produced a mechanical procedure to decide whether a finite set of quantifier-free lines is unsatisfiable. So we may be producing refutations but not be in a position to recognize them

All three of these lacunae will be addressed in the next chapter.

IX. Examples

A. Use the soundness theorem to establish $\forall xFx \vdash \exists xFx$

Well, $\forall xFx \vdash \exists xFx$ iff $\{\forall xFx, \sim\exists xFx\}$ is unsatisfiable. Put this set in prenex, and try for a refutation:

$D = \{\forall xFx, \forall x\sim Fx\}$

1.	$\forall xFx$	Δ
2.	$\forall x\sim Fx$	Δ
3.	Fa	1
4.	$\sim Fa$	2

This is a refutation since lines 3 and 4 are jointly unsatisfiable.

B. Use the soundness theorem to establish $\vdash\exists x\forall yRxy \rightarrow \forall y\exists xRxy$

First, note that $\vdash\exists x\forall yRxy \rightarrow \forall y\exists xRxy$ iff $\{\sim(\exists x\forall yRxy \rightarrow \forall y\exists xRxy)\}$ is unsatisfiable. To show this set is unsatisfiable, we must first put the formula in prenex:

$\sim(\exists x\forall yRxy \rightarrow \forall y\exists xRxy)$
$\sim\forall y(\exists x\forall yRxy \rightarrow \exists xRxy)$
$\sim\forall y\exists x(\exists x\forall yRxy \rightarrow Rxy)$
$\sim\forall z\exists w(\exists x\forall yRxy \rightarrow Rwz)$
$\sim\forall z\exists w\forall x\exists y(Rxy\rightarrow Rwz)$
$\exists z\forall w\exists x\forall y\sim(Rxy\rightarrow Rwz)$

Next, let's try for a refutation of $\Delta=\{\exists z\forall w\exists x\forall y\sim(Rxy\rightarrow Rwz)\}$

1.	$\exists z\forall w\exists x\forall y\sim(Rxy\rightarrow Rwz)$	Δ
2.	$\forall w\exists x\forall y\sim(Rxy\rightarrow Rwa)$	1
3.	$\exists x\forall y\sim(Rxy\rightarrow Raa)$	2
4.	$\forall y\sim(Rby\rightarrow Raa)$	3
5.	$\exists x\forall y\sim(Rxy\rightarrow Rba)$	2
6.	$\forall y\sim(Rcy\rightarrow Rba)$	5
7.	$\sim(Rba\rightarrow Raa)$	4
8.	$\sim(Rca\rightarrow Rba)$	7

But lines 7 and 8 are jointly unsatisfiable.

CHAPTER 12: Completeness and Compactness

I. Completeness

As mentioned, we have a long way to go in establishing the existence of a positive test for validity. Part of what we need to establish is:

Completeness: If a set, Δ, is unsatisfiable, then there exists a refutation of Δ

We will also establish some other results, namely the compactness theorem and the Lowenheim-Skolem theorem. (Note: B&J say that the sets in question are assumed to be enumerable from now on. But note that the set of all sentences in any of the languages we've introduced so far is always guaranteed to be enumerable.)

II. Canonical Derivations

We will prove our theorems by defining a certain kind of derivation. A *canonical derivation from Δ* is a derivation, D, such that:

1) Every sentence in Δ appears in D
2) If a sentence $\exists vF$ appears in D, then for some term, t, the sentence $F_v t$ appears in D
3) If a sentence $\forall vF$ appears in D, then for some term, t, the sentence $F_v t$ appears in D
4) If a sentence $\forall vF$ appears in D, then for every term t that can be formed

from names and function symbols appearing in D, the sentence $F_v t$ appears in D

5) All function symbols appearing in D appear in Δ

III. **Lemma 1: For any set, Δ, there exists a canonical derivation D from Δ**

We now prove this lemma. The proof of this lemma will in fact exhibit a mechanical procedure for constructing a canonical derivation D from Δ. Thus, another part of the task of constructing a positive test for validity will be accomplished.

Let $\Delta = \{F_1, F_2, ...\}$. If Δ is empty, then let D be the vacuous derivation; otherwise we will construct D in stages.

The construction of D in stages will need to be accomplished carefully. Why? Well, the definition of a canonical derivation requires that if ∀vF is in the derivation, then *every* instance $F_v t$ must be in the derivation, where t is any term that can be formed from names and function symbols occurring in the derivation. But if f and a are in the derivation, then f(a), f(f(a)), f(f(f(a))), . . . are all such terms; therefore, infinitely many instances of ∀vF must be included. Similarly, if there is another sentence ∀vG in the derivation, it, too, must have infinitely many instances. Therefore, it will not do to give a construction of D in which we say: OK, now include all the instances of ∀vF. Then include . . . For the first part, including all the instances of ∀vF, requires adding infinitely many sentences to the derivation; if we *then* go on to add more, the derivation is no longer a list.

Stage	Part	Instructions
N	Na	Add sentence F_N (i.e., the N^{th} sentence of Δ)
	Nb	EI stage: do EI on all lines occurring in the derivation so far, EXCEPT: i) never duplicate an earlier sentence of D, and ii) never use an existential more than once, but iii) if the result of one of these applications of EI is an existential, do EI on it, too
	Nc	UI stage: do all possible applications of UI to lines occurring so far in the derivation, EXCEPT: i) again, never duplicate an earlier sentence of D ii) Only do UI with instantial terms that a) can be formed from the names and function symbols earlier in D, and b) contain fewer than N occurrences of function symbol iii) exception to clause ii): if F_N is a universal, and no names yet occur in D, then apply UI to F_N once, with any chosen name iv) If the result of an application of UI is a universal, do UI on it too

One can then verify that at each stage we add in only finitely many sentences to the

derivation. This is due to two facts. First, at the EI and UI stages, at each case we begin with only finitely many existentials or universals that occur in the derivation so far. Doing EI or UI to these do generate further existentials or universals, but the resulting formulas are smaller, so eventually this process ends. Secondly, in the UI stage, there are only a finite number of names or function symbols that have occurred in the derivation up until a certain point, and there are only a finite number of terms with less than N function symbols that can be formed from these, and so there only a finite number of applications of UI that will be performed.

It must also be verified that the resulting derivation is canonical. (It is obviously a derivation, because every line we ever add is either a member of Δ or follows from an earlier line by either EI or UI.) So we run through the requirements in the definition:

1) Each F_N in Δ goes into D at stage N
2) Suppose $\exists vF$ goes into D at stage N. If it was added in at stage Na or Nb, then an instance goes into D at stage Nb. If it was added in at stage Nc, an instance goes into D at stage N+1b.
3) If $\forall vF$ goes into D at stage N, then an instance goes in at stage Nc. *Note*: we now see the importance of adding in some instance or other, even in the case where no names yet occur in the derivation.
4) Suppose $\forall vF$ went into D at stage N, and let t be any term formed from names and function symbols occurring in D. Let i be the number of function symbols in t; let j be the first stage in the derivation where all the names and function symbols in t have appeared in the derivation. Then the instance $F_v t$ will occur in the derivation at stage max(i+1,j,N).
5) In constructing D, we never picked function symbols in doing UI other than those already in the derivation. EI never produces new function symbols. Thus, the only source of new function symbols is stage Na. But these sentences are sentences from Δ.

IV. **Matching**

We now need a new concept. Let Γ be a set of *quantifier-free* sentences.

I *matches* Γ iff I is a model of Γ
and
(if any terms at all appear in Γ) every object in the domain of I is denoted by some term in Γ

In the case where Γ has no terms at all, then Γ consists exclusively of sentence letters, in which case any model of Γ matches Γ. The usual case is where Γ does have terms. Now,

it's always the case that any term has a denotation in any interpretation, but it is not in general true that any object in the domain of an interpretation is denoted by some term. 'Matching' is a word for when this does occur.

NOTE: the terms that are "in" Γ in this definition must be terms that actually occur in Γ; they can't merely be terms that can be built up from terms in Γ. However, subterms of terms in Γ are also in Γ.

V. **Lemma 2: Let D be a canonical derivation from Δ, and let Γ be the set of quantifier-free sentences in D. Suppose that I matches Γ. Then I is a model of the set of *all* sentences in D, and therefore is a model of Δ**

Proof. Every non-logical symbol of D appears in Γ. (Why? The only sentences in D but not Γ are universals and existentials (since we begin in Δ with formulas in prenex). All non-logical symbols in these formulas eventually trickle down via UI and EI to the quantifier-free sentences in Γ.) So I assigns a truth-value to every member of D. We now show by reductio that these truth values must always be 1.

Suppose I assigns 0 to some sentence in D. Then there is some such sentence with a minimum *length*, where length is understood as the number of symbols, counting terms as single symbols. Let M be some such sentence. M is either a universal or an existential. I(M)=0.

If M is an existential, then by clause 2 in the definition of a canonical derivation, M has an instance in the derivation. This instance is shorter than M (it has the same number of terms because the variable has been replaced by a name; it is otherwise untouched except that it has one less quantifier.) So it is true in M. But EI is a "reverse-valid" rule — the premise follows from the conclusion. Therefore I(M) would have to be 1. Contradiction.

If M is a universal, then we have $I(\forall vF)=0$, and so for some name a not in F and some o in I's domain, $I^a_o(F_va)=0$. But this o is denoted by some term, t, occurring in Γ, since I matches Γ. By extensionality and continuity, $I(F_vt)=0$. By clause 4 in the definition of a canonical derivation, F_vt occurs in D. But F_vt is shorter than M, and so cannot be false in I. Contradiction.

VI. OK sets of sentences

Another definition:

A set, θ, is OK iff every finite subset of θ is satisfiable

So this definition doesn't require that θ itself has a model. For all the definition is

concerned, θ could be unsatisfiable. It just says that *if* θ contains a contradiction, you can only "see" that contradiction by looking at infinite subsets of θ.

We'll need a certain fact about OK sets:

If θ is OK, then either $\theta \cup \{S\}$ is OK or $\theta \cup \{\sim S\}$ is OK

I.e., given any sentence, you can always add either it or its negation to an OK set of sentences, preserving OK-ness.

Proof: Suppose for reductio that both $\theta \cup \{S\}$ and $\theta \cup \{\sim S\}$ are not OK. Then we know there are two sets, $\{A_1, \ldots A_n, S\}$ and $\{B_1, \ldots B_m, \sim S\}$ that are unsatisfiable, where the A's and the B's are all members of θ. (How do we know that S is in the first set and ~S is in the second set? Because if, e.g., $\{A_1, \ldots A_n\}$ were a finite unsatisfiable subset of θ then θ wouldn't have been OK in the first place.) But now, consider the set $\{A_1, \ldots A_n, B_1, \ldots B_m\}$. It is a finite subset of θ, and so is satisfiable, and so has some model, I. Now let J be some model like I except that it assigns a truth value to S. J is just I if I assigned a truth value to S; otherwise J just chooses any old meanings for non-logical terms in S that do not occur in θ. By continuity, J is a model of $\{A_1, \ldots A_n, B_1, \ldots B_m\}$. But either S or ~S is true in J. Therefore, either $\{A_1, \ldots A_n, S\}$ or $\{B_1, \ldots B_m, \sim S\}$ must be satisfiable after all.

VII. Equivalence relations

Digression: we need a quick fact from elementary mathematics/logic. Say that a relation, ~, is an *equivalence relation* iff it is transitive, symmetric and reflexive (with respect to some understood set). Here is the fact: an equivalence relation over a set A *partitions* A — i.e., generates a set of non-overlapping subsets of A whose union is A. Each member, x, of A is in exactly one of these subsets, which we call [x]. Each of these subsets is such that each of its members bears ~ to each of its other members. Moreover, [x]=[y] iff x~y.

VIII. Lemma 3: If Γ is an enumerable, OK set of quantifier-free sentences, then some I matches Γ

Lemma 3 is the last lemma we need for the completeness proof. Here's why:

Proof of completeness theorem: Let Δ be any unsatisfiable set. By Lemma 1 there exists a canonical derivation D from Δ. Let Γ be the set of quantifier-free lines of D. Suppose for reductio that D is not a refutation. Then Γ is OK. So by Lemma 3, some interpretation I matches Γ. By Lemma 2, I must be a model of Δ,

contradicting the fact that Δ was stipulated to be unsatisfiable.

We now prove Lemma 3.

Step 1. Let the sequence A_i be any enumeration of all the atomic sentences "formable" from Γ. By this I mean all atomics that are either i) sentence letters in Γ, or formed from predicate letters in Γ or '=' (even if '=' doesn't occur in Γ) and terms in Γ.

Step 2. Define a sequence Γ_i of OK sets, recursively.

$\Gamma_1 = \Gamma$
$\Gamma_{n+1} =$ $\Gamma_n \cup \{A_n\}$ if that is OK;
otherwise $\Gamma_n \cup \{\sim A_n\}$.

Given this definition, each Γ_i is OK, because Γ_1 is OK, and on the assumption that Γ_n is OK, given the fact about OK sentences proved above, Γ_{n+1} must also be OK.

Step 3. Let B_i be the sequence of sentences defined as follows. Note that exactly one of A_i, $\sim A_i$ is in Γ_{i+1} (at least one goes in by definition; exactly one goes in because Γ_{i+1} is OK). Let B_i be whichever of these is in Γ_{i+1}.

Step 4. We define an equivalence relation over the set of all terms in Γ:

r~s iff $\ulcorner$r=s$\urcorner$ is one of the Bs

Note that:

- ~ is reflexive: Note that exactly one of A_i, $\sim A_i$ is among the Bs. (Because exactly one goes into Γ_{i+1}; and the Γs are cumulative and OK.) So if r=r is not among the Bs, ~r=r is. But this can't be, because then one of the Γs would not be OK.

- ~ is transitive: Suppose r~s and s~t. That means r=s and s=t are among the Bs. Suppose that it's not the case that r~t. Then r=t is not among the Bs. So ~r=t is. But that means that eventually (after r=s, s=t and ~r=t have all occurred in the sequence of Bs) one of the Γs is not OK

- ~ is symmetric: proof is similar.

Step 5. We define an interpretation, I, that matches Γ.

(A) The domain of I is the set of equivalence classes [t]. (Unless Γ had no terms; in that case let the domain be any old set.)

Now we specify the meanings of things in I. The idea is to specify the denotations so that each term, t, refers to [t]; and we want to assign meanings to the predicate letters to insure that each B_i is true in I.

(B) I(t)=[t] if t is a name

(C) I(f) = the function **f** such that for all $[t_1] \ldots [t_n]$ in the domain, $\mathbf{f}([t_1],...,[t_n]) = [f(s_1,...s_n)]$ where $s_1...s_n$ are terms such that $s_i \in [t_i]$ and $f(s_1,...s_n)$ is a term in Γ, *if* there is some such sequence of terms s_i. If not, then $\mathbf{f}([t_1],...,[t_n])$ is set to any [t] we like.

(D) I(S) = 1 iff S is one of the Bs

(E) $I(R^n)$ is the characteristic function **r** such that $\mathbf{r}([t_1],...,[t_n])=1$ iff $R^n(t_1,...,t_n)$ is one of the Bs.

Note: There is a potential problem with clauses (C) and (E). What if there are terms s_i such that $s_i \in [t_i]$ and $f(s_1,...s_n)$ is a term in Γ, and there are also other terms s'_i such that $s'_i \in [t_i]$ and $f(s'_1,...s'_n)$ is a term in Γ, but is such that $[f(s'_1,...s'_n)] \neq [f(s_1,...s_n)]$. In that case condition (C) would not uniquely specify a function. But this can't occur. (And a similar problem with (E) can't occur.) For since $s_i \in [t_i]$ then each $s_i \sim t_i$, likewise each $s'_i \sim t_i$. So $s_i=t_i$ and $s'_i=t'_i$ are each among the Bs, for each i. But since $[f(s'_1,...s'_n)] \neq [f(s_1,...s_n)]$, it is not the case that $f(s'_1,...s'_n) \sim f(s_1,...s_n)$, and so $\ulcorner f(s'_1,...s'_n)=f(s_1,...s_n) \urcorner$ is not one of the Bs, and hence $\sim f(s'_1,...s'_n)=f(s_1,...s_n)$ *is* one of the Bs. But any set containing $f(s'_1,...s'_n)=f(s_1,...s_n)$ and $s_i=t_i$ and $s'_i=t'_i$ for each i is not OK.

Step 6: We now verify that I matches Γ.

- First it may easily be shown by induction that for every term, t, that appears in Γ, I(t)=[t]. Simply look at clauses (B) and (C).
- Second, let us show that for any A_i, $I(A_i)=1$ iff A_i is one of the Bs.
  - if A_i is a sentence letter, then this follows from (D)
  - If A_i is $\ulcorner s=t \urcorner$ then since I(s)=[s] and I(t)=[t], $I(A_i)=1$ iff [s]=[t] iff $s \sim t$ iff $\ulcorner s=t \urcorner$ is one of the Bs
  - If $A_i = \ulcorner R(t_1, ..., t_n) \urcorner$ then since $I(t_i)=[t_i]$, $I(A_i)=1$ iff $I(R)([t_1], ..., [t_n])=1$ iff (by (E)) $R(t_1, ..., t_n)$ is one of the Bs
- From the underlined sentence it follows that **each of the Bs is true in I**. For B_i is either A_i or $\sim A_i$. If it is the former then A_i is one of the Bs, and so by the underlined sentence $I(A_i)=I(B_i)=1$. If $B_i = \sim A_i$, then A_i is not one of the Bs (since exactly one of A_i and $\sim A_i$ is one of the Bs), and so by the underlined sentence

$I(A_i)\neq 1$, so $I(A_i)=0$, so $I(\sim A_i)=I(B_i)=1$.

- We now use the **boldface** sentence to show that I matches Γ.
  - First, we'll show that every $S\in\Gamma$ is true in I. Let S be any member of Γ. S is a truth-functional compound of finitely many A_is. So suppose all these A_is are among the following initial segment of the A_is: A_1, ..., A_k. Consider the corresponding Bs, B_1, ..., B_k. All are in Γ_{k+1}, as is S (since $S\in\Gamma$). Since Γ_{k+1} is OK, some model J makes all these sentences true. Since the Bs are also all true in I, A_1, ..., A_k have the same truth values in J as they do in I. Therefore, S has the same truth value in I as it has in J, and so is true in I. (The basic idea is this. By the OK-ness of Γ_{k+1}, B_1, ..., B_k are consistent with S. But the truth values of B_1, ..., B_k *settle* the truth value of S. So, given that they are consistent with S, they must *entail* S. Since they are all true in I, so is S.)
  - We finally need to show that every object in the domain of I is denoted by some term in Γ (assuming Γ has terms). If Γ has terms, then any member of the domain is an equivalence class [t], and that member is denoted by any of its members.

IX. Consequences of completeness

A. **Compactness**: θ is unsatisfiable iff some finite subset of θ is unsatisfiable

This is one immediate consequence of completeness. It's also very startling. It basically says that there cannot be any essentially infinite contradictions — i.e., there can't be any sets that contain contradictions, but none that can be "seen" by looking only at finite subsets.

Pf: It's trivial that if some finite subset of θ is unsatisfiable then θ is unsatisfiable.

For the other direct, let θ be unsatisfiable, and let θ consist of formulas in prenex. By the completeness theorem θ has a refutation, D. Moreover, this refutation becomes a refutation at some point D_n — the first point at which it contains a finite unsatisfiable set of sentences. But then D- — the derivation consisting of D_1 through D_n — is a refutation of some finite subset of θ — the set of formulas in θ that are used as premises at or before D_n. By the soundness theorem that subset is unsatisfiable.

B. **A consequence of compactness**: There is no way to symbolize "there are finitely many Fs"

This is one very striking consequence of compactness. First, let's clarify the assertion:

> There is no sentence, S, such that for any interpretation, I, of S, S is true in I iff there are finitely many objects of which a certain predicate, F, is true.

For suppose for reductio that there exists some such S. Now consider the infinite set that contains S, plus each of the following sentences:

F_1:	$\exists xFx$	"There is at least 1 F"
F_2:	$\exists x\exists y[Fx \ \& \ Fy \ \& \ x\neq y$	"There are at least 2 Fs"
F_3:	$\exists x\exists y\exists z[Fx\&Fy\&Fz\&x\neq y\&x\neq z\&y\neq z]$	"There are at least 3 Fs"

etc.

We could easily show that each F_i is true in a model iff there are at least i objects in that model of which the predicate F is true. Therefore, our set is unsatisfiable, for if it were true in a model, I, then each of the F_is is true in I; but that means that there must be infinitely many objects in I's domain of which F is true, since if there were some finite number, n, objects of which F is true, then F_{n+1} would be false. By compactness, our set has a finite subset that is unsatisfiable. But that is impossible. Let n be the maximum n such that F_n is in the set (let n be 1 if there is no such F_n) and let I be any model with n objects of which F is true. Each of the F_is where $i\leq n$, is true in i, as is S.

C. A sentence is implied by a set of sentences iff it is implied by some finite subset of that set

It's trivial that if S is implied by a finite subset of Γ then S is implied by Γ (let I be any model that makes every member of Γ true; I makes every member of the finite subset true and so must make I true).

For the other direction, suppose $\Gamma\vdash S$. Then $\Gamma\cup\{\sim S\}$ is unsatisfiable, so has a finite unsatisfiable subset θ. If $\sim S\notin\theta$ then $\theta\subseteq\Gamma$; but since θ is unsatisfiable it entails anything and so $\theta\vdash S$. Otherwise, let Γ_0 be $\theta-\{\sim S\}$. Γ_0 is a finite subset of Γ. But also, $\Gamma_0\vdash S$, since $\Gamma_0\cup\{\sim S\}$ (i.e., θ) is unsatisfiable.

D. **Skolem - Löwenheim theorem**: If Δ has a model then Δ has an enumerable model

B&J have a whole chapter on theorems like this one. This is just the most basic.

Let Δ be satisfiable. By Lemma 1 there is a canonical derivation from Δ, which by soundness is not a refutation of Δ. So no finite subset of Γ, the set of quantifier-free lines in the derivation, is unsatisfiable. So, by Lemma III there exists some interpretation I that matches Γ; by lemma II, this interpretation I is a model of Δ. We now construct an enumerable model of Δ.

If Δ has no predicate letters then simply let the domain be any enumerable set; let the sentence letters have the same truth values as in I.

Otherwise, I is our desired model. For everything in I's domain is denoted by a term in Γ. So the domain of I is enumerable.

E. **There is an effective positive test for the unsatisfiability of sentences**

i) find a prenex equivalent of the sentence, call it S
ii) construct longer and longer initial segments of a canonical derivation from {S}
iii) at each stage, check to see whether the derivation is a refutation yet, as follows:

Any finite set, θ, of quantifier-free sentences contains some finite number, n, of distinct terms. Now, if θ is satisfiable then it is OK (since it is finite). So by Lemma 3 it has a model with no more than n things in the domain. And of course if θ is *not* satisfiable then it has no such model. Therefore, θ is satisfiable iff it has a model with $\leq$ n things in the domain. So, we can simply search through all the models with domains {1...m}, where $1 \leq m \leq n$, and where the models only assign meanings to the non-logical symbols in θ. Any finite model with domain of size $\leq$n is isomorphic to one of these, and so θ will have a model of size $\leq$n iff one of these is a model of θ. It is mechanical whether all the members of θ are true in a given one of these. For we can write down a *diagram* of any such model: a list of the predicate meanings, function meanings and so on, and then compute the denotations of terms, compute the truth values of atomics, and finally use the truth tables to compute the truth values of propositional combinations of atomics.

iv) If the derivation is a refutation, answer "yes, unsatisfiable". Otherwise

keep constructing the canonical derivation

This is a positive test for unsatisfiability of sentences. By the proof of the completeness theorem, it is clear that the canonical derivation from {S} will eventually be a refutation, if S is unsatisfiable, and so the test will eventually say "yes". Conversely, by the soundness theorem, if the test ever says "yes" then S must be unsatisfiable.

CHAPTER 14: Representability in Q

We now begin discussion of topics that will eventually lead to several important results about the metatheory of arithmetic, such as Gödel's incompleteness theorem, the undecidability of arithmetic, and Tarski's indefinability theorem. We begin with a bunch of definitions.

I. **Theory**: A set of sentences in some language closed under $\vdash$

When sentence $A \in T$, we write $\vdash_T A$, and say "A is a theorem of T". Note that this usage is non-standard, given Boolos and Jeffrey's nonstandard conception of logical consequence. Usually "$\vdash$" means "syntactically derivable from", and so "theory" means "set of sentences closed under syntactic consequence, and "theorem of T" means "provable from T".

We will be primarily interested in *numerical* theories — theories in languages that include the name **0** and the functor $'$.

II. **Numerals**

Where n is a number, the *numeral for n*, or **n**, is a certain term, namely, **0** followed by n successor signs. So, for example, **4** is $\mathbf{0}''''$. Also, note that **n+1** is $\mathbf{n}'$.

It is vitally important to be clear from the start about the distinction between numerals and numbers. Numbers are, well, numbers. They are abstract entities that we cannot presumably see or touch; they are the subject matter of mathematics. Numerals are linguistic entities — they are certain terms in the language of arithmetic.

Another bit of notation. Suppose we have a formula $A(x_1, ..., x_n)$, where $x_1, ..., x_n$ are free in A. Then $A(\mathbf{p_1}, ..., \mathbf{p_n})$ is the formula that results from replacing free occurrences of x_i in A with the numeral $\mathbf{p_i}$. Similarly, where $y_1 ... y_n$ are variables, $A(y_1 ... y_n)$ is the formula you get by beginning with $A(x_1 ... x_n)$, replacing any bound occurrences of any of $y_1 ... y_n$ in A with new variables, and then replacing x_i with y_i.

(Why the complicated definition of "$A(y_1...y_n)$"? The basic idea is that we want to take out the free x_is and replace them with free y_is. But the formula $A(x_1...x_n)$ might have quantifiers binding y_is, so if you just replaced free x_is with y_is, those y_is would become bound. Solution: first re-letter any bound y_is, then do the replacement.)

It will often be convenient to abbreviate $x_1...x_n$ with $\underline{x}$. Similarly, we replace talk of the n numerals $\mathbf{p_1}...\mathbf{p_n}$ with $\mathbf{\underline{p}}$, and we replace talk of the n numbers $p_1 ... p_n$ with $\underline{p}$.

III. Representability

n-place function, f, is *representable in T* iff there is some formula $A(\underline{x}, x_{n+1})$ such that for any natural numbers $\underline{p}$, j, if $f(\underline{p})=j$ then $\vdash_T \forall x_{n+1}(A(\mathbf{\underline{p}},x_{n+1})\leftrightarrow x_{n+1}=\mathbf{j})$. In this case we say that *A represents f in T*.

Note that $\forall x_{n+1}(A(\mathbf{\underline{p}},x_{n+1})\leftrightarrow x_{n+1}=\mathbf{j})$ is equivalent to $A(\mathbf{\underline{p}},\mathbf{j}) \,\&\, \forall x_{n+1}(A(\mathbf{\underline{p}},x_{n+1})\rightarrow x_{n+1}=\mathbf{j})$ — this says "**j** is the one and only thing such that A($\mathbf{\underline{p}}$, it)". Thus, the definition of representability is basically saying that a function is representable in T if, whenever the function holds of some numbers, the theory "says" this.

Note that representation requires that *all particular cases* of the function have corresponding theorems in the theory. It doesn't require that a single general sentence be a theorem and cover all cases of the function. Thus, looking ahead, the formula $x_1+x_2=x_3$ represents addition in Q, since certain sentences involving numerals are theorems of Q, e.g., for the case +(1,1)=2, the following must be a theorem of Q: $\forall x_3(\mathbf{0'+0'=}x_3)\leftrightarrow x_3=\mathbf{0''})$. Here we have particular numerals, $\mathbf{0'}$ and $\mathbf{0''}$, rather than a general sentence with variables.

IV. Robinson's arithmetic, Q

The goal of this chapter is to prove that the recursive functions are representable in a certain theory, Q. We know what recursive functions are, and now we know what 'representable' means. It remains to describe the particular theory Q in which we're saying the recursive functions are representable.

The language of this theory is the language of arithmetic: $\{\mathbf{0}, ', \mathbf{+}, \cdot\}$. Q is the set of sentences that are consequences of the following seven sentences ("axioms of Q"):

Q1: $\forall x\forall y(x'=y'\rightarrow x=y)$
Q2: $\forall x\ \mathbf{0}\neq x'$
Q3: $\forall x(x\neq\mathbf{0}\rightarrow\exists y x=y')$
Q4: $\forall x\ x\mathbf{+0}=x$

Q5: $\forall x \forall y\ x+y' = (x+y)'$
Q6: $\forall x\ x \cdot \mathbf{0} = \mathbf{0}$
Q7: $\forall x \forall y\ x \cdot y' = (x \cdot y)+x$

In this case, we have a theory with a finite list of axioms, but this isn't true for theories in general; theories just have to be sets of sentences closed under $\vdash$.

Note that Q is satisfiable. For each of its axioms is true in the standard model of arithmetic; since every other theorem of Q is a consequence of these axioms, every other theorem of Q is true in this model as well. However, it is not the case that every sentence true in the standard model is true in Q. See exercise 14.2. (One example is $\forall x \forall x(x+y=y+x)$.) Note, though, that *quantifier-free* sentences in the language of arithmetic that are true in the standard model are theorems of Q — this is one of the things you will be proving in homework 7a.

Our goal for the remainder of the chapter will be to prove that all recursive functions are representable in Q. Note that by 'recursive' here we now mean only to include the total recursive functions. Hence, we will replace the operation of minimization with minimization of *regular* functions. Remember that n+1 place function f is regular iff for every $\underline{p}$, there exists a j such that $f(\underline{p},j)=0$. Min(f) is the n-place function g such that $g(\underline{p})$ = the least j such that $f(\underline{p},j)=0$. We abbreviate this: $g(\underline{p}) = \mu j\ f(\underline{p},j)=0$.

V. recursive vs. Recursive (capital R) functions

It will be convenient to work with a slightly different definition of the recursive functions:

f is *Recursive* iff: f can be obtained by (a finite number of applications of) composition or minimization of regular functions from +, ·, $f_=$ (the characteristic function of identity), and the projection functions.

The idea is that all the "power" of definitions by primitive recursion is to be provided by including + and · among our initial functions.

Clearly all Recursive functions are recursive, because +, ·, and $f_=$ are recursive, and the admissible operations for Recursive functions are a subset of those for recursive functions. ($f_=$ is recursive because $f_=(i,j)=\underline{sg}[(i-j)+(j-i)]$.) We will now show that all (total - omitted henceforth) recursive functions are Recursive. We will then show that all Recursive functions are representable in Q.

VI. All recursive functions are Recursive

A. The initial functions are Recursive

$z(i) = \text{Min } j: i \cdot j = 0$ (Officially, $z = Mn(\cdot)$)

$s(i) = i+1 = Cn(+, id, Cn(f_{=}, id, id))$ (Note the trick with $f_{=}$ to get the constant 1.)

The projection functions are already defined as Recursive

B. Admissible operations preserve Recursiveness

Composition and minimization are admissible operations for Recursiveness, and thus preserve Recursiveness. The other admissible operations for recursiveness is primitive recursion. We must prove that if h comes from f and g by primitive recursion and f and g are Recursive, then so is h. This is somewhat hard and tedious.

C. Some recursive functions and relations

A relation is Recursive iff its characteristic function is Recursive. Let's investigate some Recursive relations

1. Boolean combinations of Recursive relations are Recursive

Consider n-place recursive relations R and S, with characteristic functions r and s. The characteristic function of R&S is

$$f_{R\&S}(\underline{i}) = r(\underline{i}) \cdot s(\underline{i})$$

The characteristic function of ~R is:

$f_{\sim R}(\underline{i}) = f_{=}(r(\underline{i}), z(r(\underline{i})))$ (i.e., $f_{=}(r(\underline{i}), 0)$)

All other boolean operations are definable in terms of ~ and &.

2. Minimization of Recursive relations

Next let's extend the definition of regularity and minimization to recursive relations.

- Say that an n+1 place relation, R, is *regular* iff for every $\underline{p}$, there exists an i such that $R\underline{p},i$.

- Say that a function, e, comes from R by minimization if for any $\underline{p}$, $e(\underline{p})=\mu i\ R\underline{p},i$.

- $e(\underline{p}) = \mu i\ f_{\sim R}(\underline{p},i)=0$. Therefore e is Recursive.

3. **Bounded quantifications of Recursive relations are Recursive**

If we have an n+1 place relation, R, then we define the *bounded universal quantification* of R as the relation S such that for all $\underline{p}$, j, $S\underline{p},j$ iff $\forall i<j$, $R\underline{p},i$. I.e., $S\underline{p},j$ says that $R(\underline{p},i)$ holds for all numbers i smaller than j.

Similarly, the bounded existential quantification of R is the relation S such that $S\underline{p},j$ iff $\exists i<j\ R(\underline{p},j)$

Suppose n+1 place relation R is recursive. Let us define a new n+2 place relation, T, as follows: $T\underline{p},j,i$ iff either not $R\underline{p},i$ or j=i. Clearly T is Recursive (it is a boolean combination of Recursive relations). Moreover, T is regular, since we can always set i to equal j. Therefore, as we just proved, the function d that comes by minimization from T is Recursive: $d(\underline{p},j)= \mu i\ T\underline{p},j,i$. Now let $e(\underline{p},j)= f_{=}(j,d(\underline{p},j))$. e is Recursive. But e is the characteristic function of S, the bounded universal quantification of R. For the only time when $d(\underline{p},j) =j$ is when the least i such that $T\underline{p},j,i$ is j, and this occurs only when $R\underline{p},i$ is true for all $i<j$.

Now, for bounded existential quantification of R. ~R is Recursive, so T, the bounded universal quantification of ~R, is Recursive, so ~T is Recursive. ~T holds between $\underline{p}$, j iff it is not the case that for every i<j, R does not hold between $\underline{p}$ and i. Thus, ~T is the bounded existential quantification of R.

D. Pairing function

We need a way to represent ordered pairs of numbers with numbers, and we need to do this with Recursive functions. Here is our method for encoding pairs:

$J(a,b) = ½ (a+b)(a+b+1) + a$

Lemma 14.1: J is a one-one function whose domain is the set of all ordered pairs <a,b> of natural numbers and whose range is the set of all natural numbers.

Pf: We will define an ordering (no repeats, no gaps) of the ordered pairs <a,b>, and we will show that J(a,b) = n iff <a,b> is the n^{th} pair in the ordering.

Say that <c,d> precedes <a,b> iff either (c+d<a+b) or (c+d=a+b and c<a). (We use c<a to break ties in the case where the ordered pairs have the same sum.)

This is clearly gives a list of the ordered pairs. So now consider any <a,b>. How many pairs are there in the ordering before <a,b>?

Well, first there are all the pairs whose sums are less than a+b. For each m, there are m+1 pairs that sum to m: <0,m>, <1,m-1>, ..., <m,0>. So for m=0 ... (a+b)-1, there are 1 + 2 + ... + (a+b) pairs that sum to less than a+b — i.e., ½ (a+b+1)(a+b) pairs.

And then there are all the pairs <c,d> that sum to a+b, but where c<a. There are a of these (where c = 0, 1, ..., a-1).

Thus, J gives the place in the ordering of <a,b>.

Note that J is Recursive, for it can be defined as follows:

$J(a,b) = \mu i\ i+i = (a+b+1)(a+b) + a + a$

(since $J(a,b)=1/2(a+b)(a+b+1) +a$, $2J(a,b)=(a+b)(a+b+1) + a + a$))

We can also define *inverse* pairing functions, that take in a number, and spit back the left/right members of the ordered pair corresponding to that number.

$K(i) = \mu a\ \exists b \leq i\ J(a,b)=i$ (Officially minimization involves <, not ≤, but we can convert by gerrymandering the function using =)

$L(i) = \mu b\ \exists a \leq i\ J(a,b)=i$

These functions are also Recursive. For i) a and b are obviously both ≤ J(a,b), so i

is a fine bound for the quantification, and ii) Lemma 14.1 ensures that for any i, there exists a (unique) <a,b> such that J(a,b)=i.

E. Some more Recursive functions and relations

- *m divides n* $\leftrightarrow \exists i \leq n\ i \cdot m = n$
- *p is prime* $\leftrightarrow p \neq 0 \ \& \ p \neq 1 \ \& \ \forall m \leq p[m \text{ divides } p \rightarrow (m=1 \vee m=p)]$
- $m<n \leftrightarrow \exists i<n\ i=m$
- $m-n = \mu i([n<m \rightarrow n+i=m] \& [\sim n<m \rightarrow i=0])$
- *n is a power of the prime p* $\leftrightarrow n \neq 0$ & p is prime & $\forall m \leq n[m \text{ divides } n \rightarrow (m=1 \vee p \text{ divides } m)]$

  We can see why this last definition works as follows. Consider n's prime factorization. If it has any primes in it other than p, then let one of those be m — the second condition fails. But if it doesn't, then only p and 1 divide it, in which case the condition holds. (Note that this strategy only works to define powers when the base is prime. What is neat about the general approach here is it introduces coding without the full power of exponentiation, which isn't so easily shown to be Recursive.[1])

F. $*_p$

We are going to want to use individual numbers to represent strings of numbers. E.g., suppose we have the string of numbers, 54, 76, 89, and we want to represent this string by a single number. The natural thing to do is string the digits together: 547689. Now, this isn't a unique representation yet, since this could just as well represent 5, 4768, 9; but it's a start.

But let's think about the process again more carefully. We "strung" the digits together — i.e., we took the base 10 representation of the numbers, and then concatenated those representations. Imagine doing this in another base, say, base 7. The base 7 representations of 54, 76 and 89 are: 105, 136, 155. String these together and we get 105136155, which represents some number in base 7.

Let's give a name to this concatenating operation. Where $a \neq 0$, let $a *_p b$ be the number that is denoted in base p by writing the base p numeral for a followed by the base p numeral for b.

---

[1] Thanks to Alex Jackson for spotting this.

So, for example, let's compute $4*_7 25$. Well, 4 in base 7 is "4"; 25 in base 7 is "34"; concatenate these and you get "434", which is the base 7 numeral for 221.

Note that $*_p$ is a function returns a *number*, though that number is picked out by what is denoted by a certain numeral.

Let's show that this $*_p$ function is Recursive. First we define the following:

$\eta(p,b) = \mu i[(p \text{ is prime } \& \text{ i is a power of the prime p } \& \text{ i}>b \ \& \ i>1) \vee (p \text{ is not prime } \& \ i=0)$

The case we want here is where p is a prime. (If p isn't prime then the result just comes back 0.) In that case, $\eta(p,b)$ returns the least power of p that is greater than b. So, e.g., $\eta(7,35)=49$.

We can now use η to define $*_p$:

$$a*_p b = a \cdot \eta(p,b) + b$$

This can be seen to work with examples. Recall $4*_7 25$. In base 7, 4 and 25 become "4" and "34". Now, in the base 7 numeral "434" , 4 is in the 7^2 spot — this numeral denotes $4 \cdot 7^2 + 3 \cdot 7^1 + 4 \cdot 7^0$. So we must simply add to 25, $4 \cdot 7^2$ — i.e., we must add $4 \cdot \eta(7,25)$.

G. The β function

We now want a relation that holds between numbers a and b iff a's base p numeral is a part of b's:

$$a \text{ part}_p \ b \leftrightarrow \exists c \leq b \ \exists d \leq b \ [c*_p a*_p d=b \vee c*_p a=b \vee a*_p d=b \vee a=b]$$

This is Recursive. Now let's define a function:

$$\alpha(p,q,j) = \mu i \ [(p-1)*_p j*_p i \text{ part}_p \ q \vee i=q]$$

I.e., the least i such that you can find $(p-1)*_p j*_p i$ within q. (Let i be q itself if nothing smaller than q has this feature.) This too is Recursive.

Here's the idea. Think of α as a "information search" function. p is some prime number; q represents a long base-p numeral. Think of q as a filing cabinet, since it contains a long sequence of digits. But which sequence of digits q represents depends on what base we use to interpret it — i.e., it depends on the choice of p.

Thus, think of p as being the key to the cabinet. We can call the pair <p,q> a "filing system" — i.e. a cabinet plus a key. The number j represents the something we want to search for in q. But we don't search precisely for j itself in q; we search for a sequence that looks like this: $(p-1)*_p*j*i$. So think of j as being the index entry for what we're looking for. If we can find such a sequence in q (as decoded by p), we return the smallest such i. What does this i mean? We can think of i as being the information that is stored in q under the index entry j. When we actually use this function below, we will use it on q's that have been set up in this sort of way, to have index entries j, and corresponding information i.

Now we define a second Recursive function:

$$\beta(i,j) = \alpha(K(i),L(i),j)$$

Think of i as a coded ordered pair, and it represents a filing system: its left hand member is the key to a cabinet, and the right hand member is the cabinet. The second input to β, j, is the index entry of the information we're searching for. **So $\beta(i,j)$ retrieves the j^{th} entry in the filing system i.**

H. **The β-function lemma**: For any k and any finite sequence of natural numbers $i_0, ..., i_k$, there exists a natural number i such that for every $j \leq k$, $\beta(i,j)=i_j$.

This lemma basically guarantees that for any finite sequence of natural numbers, there exists a filing system (cabinet+key, represented by i) that encodes this sequence; the j^{th} entry, i_j, is indexed as entry j in the filing system.

Proof: Let p be a prime such that p-1 is greater than all of the i's, and also greater than k. Let s=p-1. Now define q:

$$q = s*_p0*_p*i_0*_ps*_p1*i_1*_p...*_ps*_pk*_pi_k$$

Here we concatenate all the triples $s*_pj*_pi_j$ for $j \leq k$. Note that each of s, 0, ..., k, and $i_0, ..., i_k$ are represented by single digits in base p. Thus, there won't be any "ambiguity" in the representation by q of the string of triples $<s,j,i_j>$ – the base p numeral for a given triple will only show up once in the base p numeral for q, and hence for any $j \leq k$, $\alpha(p,q,j)=i_j$.

Let i=J(p,q).

I. **The proof**

We now need to show that definitions by primitive recursion preserve Recursiveness. Let f be an n-place function and g be an n+2-place function, and suppose that h = Pr(f,g).

First, for any numbers $\underline{p}$, k, let us consider the initial values of the h function for $j \leq k$ — i.e., $h(\underline{p},0)$, $h(\underline{p},1)$, ..., $h(\underline{p},k)$. We can define a relation (with n+2 places), $R\underline{p},k,i$, which holds when i is a filing system that encodes the values of $h(\underline{p},j)$ for all $j \leq k$:

$$R(\underline{p},k,i) \leftrightarrow \forall j \leq k\ [h(\underline{p},j)=\beta(i,j)]$$

Given the recursion equations for h, this can be rewritten:

$$R(\underline{p},k,i) \leftrightarrow f(\underline{p})=\beta(i,0)\ \&\ \forall j<k\ [\beta(i,j')=g(\underline{p},j,\beta(i,j))]$$

This is a Recursive relation (since β is Recursive, as are the other elements of the definition; note the assumption that f and g are Recursive.) Moreover, it is a regular relation, for by the β-function lemma, for any $\underline{p}$, and k, an i exists such that R(p,k,i) — i.e., an i exists that is a filing system encoding these initial values of h. So we can define an n+1-place Recursive function d as follows:

$$d(\underline{p},k) = \mu i\ R(\underline{p},k,i)$$

So d gives us the minimum filing system that encodes the first k+1 values of h (relative to $\underline{p}$). But now we can define h using d:

$$h(\underline{p},k) = \beta(d(\underline{p},k),k)$$

So h is Recursive. <say more about the intuitive idea??>

VII. **All Recursive functions are representable in Q**

Once we've shown this, then given the result of the preceding section, we will be able to make the main conclusion of the chapter: that all recursive functions are representable in Q. We'll do this by i) showing that all the initial Recursive functions are representable in Q, and then showing that representability in Q is preserved under composition and minimization.

A. **The projection functions are representable in Q**

For id^m_n, we use the formula: $x_1=x_1 \,\&\, \ldots \,\&\, x_m=x_m \,\&\, x_{m+1}=x_n$, since for any i_1, ..., i_m, the following sentence:

$$\forall x_{m+1}((\mathbf{i_1}=\mathbf{i_1} \,\&\, \ldots \,\&\, \mathbf{i_m}=\mathbf{i_m} \,\&\, x_{m+1}=\mathbf{i_n})\leftrightarrow x_{m+1}=\mathbf{i_n})$$

is a valid sentence, and so is a theorem of Q.

B. **Lemma 14.3**: if i+j=k, then $\vdash_Q \mathbf{i+j=k}$

First make sure we're clear about what we're saying here. Where saying that anytime there is a certain mathematical fact, then Q has a certain kind of theorem. Also notice that the theorems Q is asserted to contain don't involve quantifiers; they state particular mathematical facts, not general mathematical facts.

We prove this by induction. Base case: j=0. In that case, i=k, and so **i=k** — i.e., **i** and **k** are the same term — and hence what we are trying to show is $\vdash_Q \mathbf{i+0=i}$. But this follows from Q4.

Induction case: suppose assertion true for m; show true for j, where j=m+1. I.e., assuming that i+j=k, show that $\vdash_Q \mathbf{i+j=k}$. First note that **j** is $\mathbf{m'}$. *Secondly, let n be k's predecessor. (It must have one since j>0.)* Then, **k** is $\mathbf{n'}$. Thus, what we are trying to show is this: $\vdash_Q \mathbf{i+m'=n'}$.

Note that i+m=n. So, by the inductive hypothesis, $\vdash_Q \mathbf{i+m=n}$. Therefore, $\vdash_Q (\mathbf{i+m})'=\mathbf{n'}$ (Q is closed under entailment; function symbols express functions in any model). But given Q5 ($\forall x \forall y\ x+y' = (x+y)'$) we know that $\vdash_Q \mathbf{i+m'=(i+m)'}$. Therefore, we have $\vdash_Q \mathbf{i+m'=n'}$.

It is very important here to not make illicit assumptions about what is a theorem of Q. Q, remember, is merely defined as the consequences of axioms Q1-Q7. We cannot make any assumptions about what is or is not a theorem of Q simply because the sentence in question "looks true" — i.e., expresses a truth in the standard model. To show that something is a theorem of Q we must show that it is a logical consequence of Q1-Q7. On the other hand, when we are in the course of doing this, we may freely make use of our knowledge of mathematics. So when we are making a statement that concerns mathematics, rather than what is a theorem of Q, we can freely make use of acceptable mathematical inferences. The italicized sentences in the above proof are examples of such inferences.

C. **Lemma 14.4:** $x_1+x_2=x_3$ represents addition in Q

Whenever i+j=k, we know that $\vdash_Q \mathbf{i+j=k}$. By virtue of closure under logical

consequence, $\vdash_Q \forall x_3(\mathbf{i+j=}x_3 \leftrightarrow x_3=\mathbf{k})$.

D. **Lemma 14.5:** if $i \cdot j=k$ then $\vdash_Q \mathbf{i \cdot j=k}$

Base: suppose $i \cdot 0=k$. In that case $k=0$, so what we're trying to show is $\vdash_Q \mathbf{i \cdot 0=0}$, which holds in virtue of Q6.

Induction: suppose $i \cdot j=k$, where $j=m'$. Then since **j** is $\mathbf{m'}$, what we are trying to show is $\vdash_Q \mathbf{i \cdot m'=k}$. Here, $k=i \cdot m'=i \cdot m+i$. Let $n=i \cdot m$. We know from the inductive hypothesis that $\vdash_Q \mathbf{i \cdot m=n}$. We also know from Q7 that $\vdash_Q \mathbf{i \cdot m'=(i \cdot m)+i}$. Therefore (by closure), $\vdash_Q \mathbf{i \cdot m'=n+i}$. (Leibniz's Law is a valid rule.) Now, since $k=n+i$, given Lemma 14.3 we know that $\vdash_Q \mathbf{n+i=k}$. So we know that $\vdash_Q \mathbf{i \cdot m'=k}$.

E. **Lemma 14.6:** $x_1 \cdot x_2=x_3$ represents multiplication in Q

This follows from 14.5 just as 14.4 followed from 14.3.

F. **Lemma 14.7**: if $i \neq j$ then $\vdash_Q \mathbf{i \neq j}$

My proof of this is slightly different from the book's. Let's proceed by induction on i, regarding the theorem as having the form: for any i: (for all j, if $i \neq j$, then $\vdash_Q \mathbf{i \neq j}$.)

Base: Let $i=0$. Assume $i \neq j$. Then $j=m'$ for some m, and $\mathbf{j=m'}$. Then we have $\vdash_Q \mathbf{0 \neq m'}$ — i.e., $\vdash_Q \mathbf{i \neq j}$ — from Q2.

Inductive step. Assume the inductive hypothesis: for any k, if $k \neq m$ then $\vdash_Q \mathbf{k \neq m}$; and show that, where $i=m'$, for any j, if $i \neq j$ then $\vdash_Q \mathbf{i \neq j}$. Note that $\mathbf{i = m'}$.

> Case 1: $j=0$. Then we have $\vdash_Q \mathbf{m' \neq j}$ from Q2.
>
> Case 2: $j>0$. Then $j=n'$ for some n. Since $i \neq j$, $n \neq m$. So by the inductive hypothesis, we know that $\vdash_Q \mathbf{m \neq n}$. By Q1 we have $\vdash_Q \mathbf{m' \neq n'}$; i.e., $\vdash_Q \mathbf{i \neq j}$.

G. **Lemma 14.8:** $(x_1=x_2 \ \& \ x_3=\mathbf{1}) \vee (x_1 \neq x_2 \ \& \ x_3=\mathbf{0})$ represents $f_=$ in Q

Call this formula $A(x_1,x_2,x_3)$. We must show that whenever $f_=(i,j)=k$ then $\vdash_Q \forall x_3(A(\mathbf{i},\mathbf{j},x_3) \leftrightarrow x_3=\mathbf{k})$.

Case 1: i=j. Then **i=j** and k=1, so what we need to show is:

$\vdash_Q \forall x_3([(\mathbf{i=i} \ \& \ x_3\mathbf{=1}) \vee (\mathbf{i \neq i} \ \& \ x_3\mathbf{=0})] \leftrightarrow x_3\mathbf{=1})$.

But this formula is valid, and so is a theorem of Q.

Case 2: i≠j. Then k=0, and what we need to show is:

$\vdash_Q \forall x_3([(\mathbf{i=j} \ \& \ x_3\mathbf{=1}) \vee (\mathbf{i \neq j} \ \& \ x_3\mathbf{=0})] \leftrightarrow x_3\mathbf{=0})$.

But this formula is a consequence of **i≠j**, which is a theorem of Q by Lemma 14.7.

H. Function composition preserves representability in Q

Suppose that m-place function f, and g_1, ..., g_m (each with n places) are representable in Q, and in particular represented by $A(x_1, ..., x_m, x_{m+1})$ and $B_1(\underline{x}, x_{n+1})$, ..., $B_m(\underline{x}, x_{n+1})$, respectively. Our formula to represent $h=Cn(f,g_1, ..., g_m)$ (an n-place function) will be:

$$C(\underline{x},x) = \exists y_1 ... \exists y_m (B_1(\underline{x},y_1) \& ... \& B_m(\underline{x},y_m) \ \& \ A(y_1,...,y_m,x))$$

For suppose $h(\underline{p})=j$, and call $g_i(\underline{p})$ i_i. So $f(i_1,...,i_m)=j$. We must show $\vdash_Q \forall x(C(\underline{\mathbf{p}},x) \leftrightarrow x=\mathbf{j})$.

Since the B_is represent the g_is in Q we have:

(a) $\vdash_Q B_i(\underline{\mathbf{p}},\mathbf{i_i})$ and (b) $\vdash_Q \forall x_{n+1}(B_i(\underline{\mathbf{p}},x_{n+1}) \rightarrow x_{n+1}=\mathbf{i_i})$

And since A represents f we have:

(c) $\vdash_Q A(\mathbf{i_1},...,\mathbf{i_m},\mathbf{j})$ and (d) $\vdash_Q \forall x(A(\mathbf{i_1},...,\mathbf{i_m},x) \rightarrow x=\mathbf{j})$

Now, putting the (a)s and (c) together we get

$$\vdash_Q B_1(\underline{\mathbf{p}},\mathbf{i_1}) \& ... \& B_m(\underline{\mathbf{p}},\mathbf{i_m}) \& A(\mathbf{i_1},...,\mathbf{i_m},\mathbf{j})$$

So by existential generalization we get:

$$\vdash_Q C(\underline{\mathbf{p}},\mathbf{j})$$

So all we need further is $\vdash_Q \forall x(C(\underline{\mathbf{p}},x) \rightarrow x=\mathbf{j})$; i.e.:

$$\vdash_Q \forall x \ [\exists y_1 ... \exists y_m (B_1(\underline{\mathbf{p}},y_1) \& ... \& B_m(\underline{\mathbf{p}},y_m) \ \& \ A(y_1,...,y_m,x)) \rightarrow x=\mathbf{j}]$$

But this follows from the (b)s and (d). This can be seen by deriving this formula, using the (b)s and (d) as premises, in one of the usual natural deduction proof

systems (which we may assume to be sound). I will do this in an informal way; if we were to be careful about this we would either need to consider truth values of formulas in interpretations directly, or to prove soundness for some particular natural deduction system in which the proofs could be formalized.

Proceed by using universal and then conditional proof: assume the antecedent when we think of x as being an arbitrary variable. Then do existential instantiation to the antecedent to arbitrary variables y_1, ..., y_m. We then have (e) $B_i(\underline{\mathbf{p}},y_i)$ and (f) $A(y_1...y_m,x)$. From the (e)s and the (b)s, we infer $y_i=\mathbf{i}_i$; from these and (f) we infer $A(\mathbf{i_1}...\mathbf{i_m},x)$; from this and (d) we infer $x=\mathbf{j}$.

We now need to show that minimization preserves representability in Q.

I. **Lemma 14.9**: For each i, $\vdash_Q \forall x\ x'+\mathbf{i}=x+\mathbf{i}'$

Induction. Base: then we're showing that $\vdash_Q \forall x\ x'+\mathbf{0}=x+\mathbf{0}'$. This follows from Q4 ($\forall x\ x+\mathbf{0}=x$) and Q5 ($\forall x \forall y\ x+y'=(x+y)'$). (This can be seen by doing universal proof; $x'+\mathbf{0}=x'$ by Q4; and $x+\mathbf{0}'=(x+\mathbf{0})'$ (by Q5) = x' (by Q4); so $x'+\mathbf{0}=x+\mathbf{0}'$.

Induction step: suppose $\vdash_Q \forall x\ x'+\mathbf{i}=x+\mathbf{i}'$; show $\vdash_Q \forall x\ x'+\mathbf{i}'=x+\mathbf{i}''$. Again, I'll establish that this is a theorem of Q by deriving it in an unformalized natural deduction system of a familiar kind. Universal proof of the latter from the former plus axioms of Q. So we need to establish $x'+\mathbf{i}'=x+\mathbf{i}''$:

1.	$x'+\mathbf{i}=x+\mathbf{i}'$	ih
2.	$(x'+\mathbf{i})' = (x+\mathbf{i}')'$	1; ′ is a function symbol
3.	$(x+\mathbf{i}')' = x+\mathbf{i}''$	Q5
4.	$(x'+\mathbf{i})' = x'+\mathbf{i}'$	Q5
5.	$x'+\mathbf{i}'=x+\mathbf{i}''$	3, 4

J. **Definition:** $x_1<x_2$ is defined as the formula $\exists x_3\ x_3'+x_1=x_2$

K. **Lemma 14.10**: if i<j then $\vdash_Q \mathbf{i<j}$

If i<j then there exists some m (namely, j-i′) such that m′+i=j. By Lemma 14.3, $\vdash_Q \mathbf{m}'+\mathbf{i}=\mathbf{j}$, so by existential generalization, $\vdash_Q \exists x_3\ x_3'+\mathbf{i}=\mathbf{j}$ — i.e., $\vdash_Q \mathbf{i<j}$.

L. **Lemma 14.11**: For each i, $\vdash_Q \forall x[x<\mathbf{i} \rightarrow (x=\mathbf{0} \vee x=\mathbf{1} \vee ... \vee x=\mathbf{i-1})$ (If i=0

then regard the consequent as just **0≠0**.)

Induction. Base: i=0. Then the antecedent is x**<0** — i.e., $\exists x_3\ x_3'$**+**x**=0**. If we can establish the negation of this (thinking of x as an arbitrary variable in a universal proof) then what we want will have been established:

1.	$\exists x_3\ x_3'$**+**x**=0**	Suppose for reductio
2.	b′**+**x**=0**	existential elimination
3.	x**=0** ∨ ∃y x**=**y′	Q3
4.	x**=0** → b′**=0**	2, Q4
5.	b′**=0** → X	Q2
6.	∃y x=y′	assumption for CP
7.	x=y′	EI
8.	b′**+**y′**=0**	2,7
9.	(b′**+**y)′**=0**	8, Q5
10.	X	9, Q2
11.	∃y x**=**y′ → X	6-10, CP
12.	X	3, 4, 5, 11

Induction step. Assume $\vdash_Q \forall x[x$**<i** → (x**=0**∨x**=1**∨...∨x**=i-1**); show $\vdash_Q \forall x[x$**<i′** → (x**=0**∨x**=1**∨...∨x**=i-1**∨x**=i**)]. Proceed by universal derivation and conditional proof:

1.	x**<i**′	assume
2.	∃y y′**+**x**=i**′	Definition of 1
3.	b′**+**x**=i**′	2, EI
4.	x**=0** ∨ ∃y x**=**y′	Q3
5.	x**=0**→ (x**=0**∨x**=1**∨...∨x**=i-1**∨x**=i**)	logic
6.	∃y x**=**y′	Assume for CP
7.	x=c′	6, EI
8.	b′**+**c′**=i**′	7,3
9.	(b′**+**c)′**=i**′	8, Q5
10.	b′**+**c**=i**	9, Q1
11.	c**<i**	10, def
12.	c**=0**∨ c**=1** ∨ ... ∨ c**=i-1**	11, ih
13.	c′**=1** ∨ c′**=2** ∨ ... ∨ c′**=i**	12, ′ is a fx symbol
14.	x**=0** ∨ x**=1** ∨ x**=2** ∨ ...∨ x**=i**	13, addition; 7 LL
15.	∃y x**=**y′→ (x**=0**∨x**=1**∨...∨x**=i-1**∨x**=i**)	6-14, CP
16.	x**=0**∨x**=1**∨...∨x**=i-1**∨x**=i**	4, 5, 15

M. **Lemma 14.12:** for each i, $\vdash_Q \forall x(\mathbf{i}<x \rightarrow x=\mathbf{i}' \vee \mathbf{i}'<x)$

Again, use our quasi-natural deduction:

1.	$\mathbf{i}<x$	Assume
2.	$\exists y\ y'+\mathbf{i}=x$	1, def
3.	$b'+\mathbf{i}=x$	2, EI
4.	$b+\mathbf{i}'=x$	3, 14.9
5.	$b=\mathbf{0} \vee \exists y\ y'=b$	Q3
6.	$b=\mathbf{0}$	As for CP
7.	$\mathbf{0}'+\mathbf{i}=\mathbf{i}'$	14.3
8.	$x=\mathbf{i}'$	7, 3, 6
9.	$x=\mathbf{i}' \vee \mathbf{i}'<x$	8
10.	$b=\mathbf{0} \rightarrow x=\mathbf{i}' \vee \mathbf{i}'<x$	6-9
11.	$\exists y\ y'=b$	assume for CP
12.	$c'=b$	EI
13.	$c'+\mathbf{i}'=x$	12, 4
14.	$\exists y\ y'+\mathbf{i}'=x$	13
15.	$\mathbf{i}'<x$	14, def
16.	$x=\mathbf{i}' \vee \mathbf{i}'<x$	15
17.	$\exists y\ y'=b \rightarrow x=\mathbf{i}' \vee \mathbf{i}'<x$	11-16
18.	$x=\mathbf{i}' \vee \mathbf{i}'<x$	17, 10, 5

N. **Lemma 14.13:** for each i, $\vdash_Q \forall x(\mathbf{i}<x \vee x=\mathbf{i} \vee x<\mathbf{i})$

This is what the last few lemmas have been gearing up to prove.

Induction on i. Base: i=0. Then the desired assertion follows from Q3 and Q4: Q3 implies $x=\mathbf{0} \vee \exists y\ y'=x$. The left disjunct just is $x=\mathbf{i}$. And the right disjunct, in virtue of Q4, implies $\exists y\ y'+\mathbf{0}=x$ — i.e., $\mathbf{0}<x$. So $\vdash_Q(\mathbf{i}<x \vee x=\mathbf{i})$, and so $\vdash_Q \forall x(\mathbf{i}<x \vee x=\mathbf{i} \vee x<\mathbf{i})$.

Inductive hypothesis: $\vdash_Q \forall x(\mathbf{i}<x \vee x=\mathbf{i} \vee x<\mathbf{i})$. We must show $\vdash_Q \forall x(\mathbf{i}'<x \vee x=\mathbf{i}' \vee x<\mathbf{i}')$; let's show how to establish an arbitrary instance: $\mathbf{i}'<x \vee x=\mathbf{i}' \vee x<\mathbf{i}'$:

1.	$\mathbf{i}<x \vee x=\mathbf{i} \vee x<\mathbf{i}$	ih
2.	$\mathbf{i}<x \rightarrow x=\mathbf{i}' \vee \mathbf{i}'<x$	Lemma 14.12
3.	$\mathbf{i}<x \rightarrow x=\mathbf{i}' \vee \mathbf{i}'<x \vee x<\mathbf{i}'$	2
4.	$\mathbf{i}<\mathbf{i}'$	Lemma 14.10
5.	$x=\mathbf{i} \rightarrow x=\mathbf{i}' \vee \mathbf{i}'<x \vee x<\mathbf{i}'$	4 (LL to the third disjunct; then addition)

6.	$x<\mathbf{i} \rightarrow (x=\mathbf{0} \vee x=\mathbf{1} \vee ... \vee x=\mathbf{i-1})$	Lemma 14.11
7.	$\mathbf{0}<\mathbf{i}'$	
8.	$\mathbf{1}<\mathbf{i}'$	these lines come from
	.	Lemma 14.10
	.	
8+i	$\mathbf{i-1}<\mathbf{i}'$	
9+i	$x<\mathbf{i} \rightarrow x<\mathbf{i}'$	6, 7-8+i
10+i	$x<\mathbf{i} \rightarrow x=\mathbf{i}' \vee \mathbf{i}'<x \vee x<\mathbf{i}'$	9+i
11+i	$x=\mathbf{i}' \vee \mathbf{i}'<x \vee x<\mathbf{i}'$	10+i, 5, 3, 1

O. **Minimization preserves representability in Q**

Let f be a regular n+1 place function that is representable in Q; let g = Mn(f). Suppose that $A(\underline{x}, x_{n+1}, x_{n+2})$ represents f in Q. Our formula that will represent g in Q will be $B(\underline{x}, x_{n+1})$ =

$$A(\underline{x},x_{n+1},\mathbf{0}) \,\&\, \forall w\, (w<x_{n+1} \rightarrow \sim A(\underline{x},w,\mathbf{0}))$$

To get a grip on what this formula is saying, think of what it means in the standard model of arithmetic. It then says that the function $f(\underline{x}, x_{n+1})=0$, but that $f(\underline{x},w)\neq 0$ for all $w<x_{n+1}$.) This formula will only be true of $\underline{x}$ and x_{n+1} when x_{n+1} is the least number such that $f(\underline{x},x_{n+1})=0$.

We must show that B represents g in Q. So suppose $g(\underline{p})=i$. We must establish two things: $\vdash_Q B(\underline{\mathbf{p}},\mathbf{i})$ and $\vdash_Q \forall x_{n+1}(B(\underline{\mathbf{p}},x_{n+1}) \rightarrow x_{n+1}=\mathbf{i})$; i.e.:

(1) $\vdash_Q A(\underline{\mathbf{p}},\mathbf{i},\mathbf{0}) \,\&\, \forall w\, (w<\mathbf{i} \rightarrow \sim A(\underline{\mathbf{p}},w,\mathbf{0}))$

(2) $\vdash_Q \forall x_{n+1}([A(\underline{\mathbf{p}},x_{n+1},\mathbf{0}) \,\&\, \forall w\, (w<x_{n+1} \rightarrow \sim A(\underline{\mathbf{p}},w,\mathbf{0}))] \rightarrow x_{n+1}=\mathbf{i})$

Since $g(\underline{p})=i$, we know that $f(\underline{p},i)=0$, and we know that for all $j<i$, $f(\underline{p},j)\neq 0$. Since A represents f in Q, from the fact that $f(\underline{p},i)=0$ we know:

(a) $\vdash_Q A(\underline{\mathbf{p}},\mathbf{i},\mathbf{0})$

(b) $\vdash_Q \forall x_{n+2}(A(\underline{\mathbf{p}},\mathbf{i},x_{n+2}) \rightarrow x_{n+2}=\mathbf{0})$

For each $j<i$, call $f(\underline{p},j)$ just k_j. Since A represents f we know that in each of these cases:

$\vdash_Q A(\underline{\mathbf{p}},\mathbf{j},\mathbf{k_j})$

$\vdash_Q \forall x_{n+2}(A(\underline{\mathbf{p}},\mathbf{j},x_{n+2}) \rightarrow x_{n+2}=\mathbf{k_j})$

Since each $k_j\neq 0$, given Lemma 14.7 we have $\vdash_Q \mathbf{0}\neq\mathbf{k_j}$ for each $j<i$; from the second statement above we then have:

(c) $\vdash_Q \sim A(\mathbf{p},\mathbf{j},\mathbf{0})$

for each j<i. But now given Lemma 14.11 and these instances of (c), we have:

$\vdash_Q \forall w(w<\mathbf{i} \rightarrow \sim A(\mathbf{p},w,\mathbf{0}))$

This, put together with (a), gives us (1). It remains to establish (2). We do this by one of our informal derivations; doing universal proof then conditional proof. We need to establish $x_{n+1}=\mathbf{i}$:

1.	$A(\mathbf{p},x_{n+1},\mathbf{0}) \& \forall w\ (w<x_{n+1} \rightarrow \sim A(\mathbf{p},w,\mathbf{0}))$	As.
2.	$A(\mathbf{p},\mathbf{i},\mathbf{0})$	(a)
3.	$\forall w(w<\mathbf{i} \rightarrow \sim A(\mathbf{p},w,\mathbf{0}))$	Shown above
4.	$\sim \mathbf{i}<x_{n+1}$	1,2
5.	$\sim x_{n+1}<\mathbf{i}$	1,3
6.	$\forall x(\mathbf{i}<x \vee x=\mathbf{i} \vee x<\mathbf{i})$	Lemma 14.13
7.	$x_{n+1}=\mathbf{i}$	

CHAPTER 15: Undecidability, Indefinability and Incompleteness

I. The idea and importance of Gödel-numbering

Gödel-numbering is a method for assigning numbers to expressions such that:

i) we have a one-one assignment; i.e., every expression gets a unique g.n.
ii) it is effectively calculable what the g.n. of an expression is
iii) for any number, n, it is effectively calculable whether n is the g.n. of some expression, and if so what that expression is

What's the point? Consider any sentence in the language of arithmetic, say:

$\forall x \exists y\ y=x'+x$

Let's think about what this sentence means, but in two different ways. First, if we think about what it means as a statement about numbers, say as interpreted in the standard model of arithmetic, then it makes a statement about numbers. But we can also interpret this statement in a different way: we can think of it as talking about *expressions* via their Gödel numbers. In the first way of thinking about the sentence the '**+**' expression stands for addition; on the second way, it stands for some operation on expressions — namely, the operation defined this way:

a + b = the expression whose Gödel number is the sum of the Gödel numbers of a and b

Similarly, ′ can be thought of as expressing a function of expressions as well. Now, the sentence might well not express anything very interesting about expressions, given this second interpretation. Indeed, it may not express anything more interesting than: "for any expression whose g.n. number is x, there exists an expression whose g.n. is x′+x." But it *might* express something more interesting. There might be a property of numbers that holds iff the number is a theorem of Q. And there might be a relation between numbers that holds between x and y iff x and y are the g.n.s of expressions such that one is the negation of the other. And now, suppose we have predicates T and R that express this property and this relation in the standard model of arithmetic. Consider the sentence ∃x∃y (Rxy & ~Tx & ~Ty). One can think of this as meaning *both* a statement about numbers, and the following statement about expressions: "there exists a sentence that is neither a theorem of Q nor is its negation a theorem of Q". What I mean by saying this sentence *means* this statement about expressions is that the sentence is true in the standard model iff the statement about expressions is true.

Finally, and this is the most important fact, it might be that a sentence

(e): . . . n . . .

which is making an assertion about an expression f, with g.n. n, is talking about *itself*, if expression f is one and the same as expression e — i.e., if n is the Gödel number of e. This raises the possibility of modeling mathematical proofs on the famous paradoxes of self-reference — e.g., "this sentence is not true".

II. A particular Gödel numbering

We first want to assign a number to each individual symbol in the language of arithmetic; assuming that x_0 is x; x_1 is y; f^0_0 is **0**, f^1_0 is **′**, f^2_0 is **+**, f^2_1 is ·, and A^2_0 is **=**. We do this via the following two tables: to find the code of a symbol, find the symbol in the first table, then find the number at the corresponding position in the second table:

(	)	&	∃	x_0	f^0_0	f^1_0	f^2_0	. . .	A^0_0	A^1_0	A^2_0	. . .
	,	∨	∀	x_1	f^0_1	f^1_1	f^2_1	. . .	A^0_1	A^1_1	A^2_1	. . .
		~		x_2	f^0_2	f^1_1	f^2_2	. . .	A^0_2	A^1_2	A^2_2	. . .
		↔		.	.	.	.		.	.	.	
		→		.	.	.	.		.	.	.	
				.	.	.	.		.	.	.	
1	2	3	4	5	6	68	688	. . .	7	78	788	. . .
	29	39	49	59	69	689	6889	. . .	79	789	7889	. . .
		399		599	699	6899	68899	. . .	799	7899	78899	. . .

3999 . . . .
39999 . . . .
. . . .

We now need a method for getting a Gödel number for any expression — i.e., any finite sequence of symbols. Here we simply concatenate the base 10 numerals for the g.n.s of the symbols in the expression (in order), and then take the resulting number named.

Example: The Gödel number for **0+1=1** is what? Well, write it out officially:

=(+(0,′(0)),′(0))

Remember that **=** is A^2_0, **+** is f^2_0, **′** is f^1_0, and **0** is f^0_0. Thus the g.n. for the whole expression is:

```
 = ( + (0,  ′( 0)), ′ (0))
78816881629681622968162
```

But from now on, let's ignore the need for parentheses in function symbols.

This assignment of numbers to expressions satisfies the definition of a Gödel numbering above as follows. ii) obviously holds — every expression has a g.n., and it is effectively calculable what that is. i) and iii) hold as well; showing this is a homework assignment.

III. Diagonalization

Here we introduce the basic method for self-reference (the analog of "this sentence" in the liar sentence "this sentence is false".)

Let A be an expression with Gödel number n. Then let us use $\ulcorner A \urcorner$ to refer to **n** — the numeral for A's Gödel number.

And let us define *the diagonalization of A* to be the expression:

$\exists x(x=\ulcorner A \urcorner \ \& \ A)$

Now, consider a case where A is a formula that has just one free variable, namely x. In that case the diagonalization of A is a sentence; let us think about what this sentence means. Interpreted in the standard model, *N*, it means that *A is true of its own Gödel number*, for in that model $\ulcorner A \urcorner$ denotes n, the Gödel number of A, and so the sentence is true iff $A_x a$ is true in N^a_n.

IV. **Lemma 1**: There is a recursive function, diag, such that if n is the Gödel number of A, diag(n) = the Gödel number of A's diagonalization

Let's think about what this lemma is saying. What Gödel numbering does is allow us to "mathematicize" logic: it lets us represent facts about logic as mathematical facts. The construction of the diagonalization of a formula is a kind of logical operation: it is a function from expressions to expressions: diagonalization(A) = $\exists x(x=\ulcorner A \urcorner \ \& \ A)$. Diag is the mathematical version of this; n represents A, and diag(n) represents diagonalization(A).

The proof basically relies on some of the same tricks we used in chapter 14, using individual numbers to represent sequences of numbers by concatenating numerals. However we won't need a variable base; we'll just use base 10.

We first define a recursive function, ln(n), which returns the length of n's base 10 numeral:

$$ln(n) = \mu m(0<m \ \& \ n<10^m).$$

This is a recursive function.

Now let's define concatenation: m*n. This is the same as $m*_{10}n$ from chapter 14; but we'll only use base 10 so we'll omit the subscript.

$$m*n = m \cdot 10^{ln(n)} + n$$

Let's finally define a function num(n) that gives us the Gödel number of the numeral for n:

num(0) = 6
num(n+1) = num(n)*68 (I.e., the Gödel number for the numeral for n concatenated with the Gödel number of the successor sign.)

(We'll forget about the parentheses that are officially generated by the function symbol ′.)

Now we define diag:

diag(n) = 4515788 * num(n) * 3 * n * 2
$\exists x(x =$ **n** $\&$ A)

V. **Lemma 2 (the diagonal lemma)**: Let T be a theory in which diag is representable. Then for any formula B(y) (in the language of T, containing just the variable y free), there is a sentence G such that $\vdash_T G \leftrightarrow B(\ulcorner G \urcorner)$

This is the fundamental Lemma on which everything else in this chapter will be based. Its significance will be easier to explain later; but the basic idea is this. This lemma is what intuitively lets us achieve self-reference – it does what "I" does in the liar sentence "I am not true". English has the word 'I', whereas we want to show that there exists within the language of arithmetic the means for doing the same sort of thing. Where B is a predicate, we want to show how to construct sentences in the language of arithmetic that say "I am B". We will later then let B be all kinds of funky things; perhaps "is not true", or "is not provable".

So let's get clear about what we mean by achieving self-reference in arithmetic. Remember that talk about the language of arithmetic "talking about" expressions is really talk about godel numbers – an expression in the language of arithmetic is "saying something about expressions" when its interpretation in the standard model – which concerns numbers – is interpreted as a statement about the expressions represented by those numbers via the godel numbering.

So now consider B(y). It is a predicate; and can be thought of as expressing a property of expressions (those expressions whose Gödel numbers B(y) is true of in the standard model). What we want to do is find a statement G, that "says" "**I am B**". What would it be for a sentence, G, to "say": "I am B"? Answer: G must be true in the standard model if and only if the predicate B is true of G's Godel number – i.e., $G \leftrightarrow B(\ulcorner G \urcorner)$ must be true in the standard model.

Actually what we want is a sentence G that says "I am B" *in theory T* - what this means is that it must be a theorem of T that "$G \leftrightarrow B(\ulcorner G \urcorner)$". OK, so how might we find such a G?

OK, here's the proof. Suppose A(x,y) is a formula that represents diag in T. Let

$$F = \exists y(A(x,y) \,\&\, B(y))$$

Let

n = the Gödel number of F

Let

$$G = \exists x(x=\mathbf{n} \,\&\, \exists y(A(x,y) \,\&\, B(y)))$$

So G is the diagonalization of F.

Let's think informally about what all this means. Since A(x,y) represents diag in T, one can think of it as meaning, informally, that "y is the (g.n. of the) diagonalization of (the formula whose g.n. is) x". One can think of F, then, as meaning "B is true of the diagonalization of x". And then finally, one can think of G as meaning "B is true of the diagonalization of n" – i.e., "B is true of the diagonalization of F". But since G itself *is* the diagonalization of F, G essentially means "B is true of *me*". What we need to do now is to turn all this talk of "G essentially means that.." into a proof that $G \leftrightarrow B(\ulcorner G \urcorner)$ is a theorem of T.

First, notice that:

$$G \cong \exists y(A(\mathbf{n},y) \,\&\, B(y))$$

Therefore, by definition of $\cong$ and the fact that T is a theory and therefore includes all logical truths, we have:

(1) $\vdash_T G \leftrightarrow \exists y(A(\mathbf{n},y) \,\&\, B(y))$

Now let k = the Gödel number of G. So diag(n)=k. So, since A represents diag in T,

(2) $\vdash_T \forall y(A(\mathbf{n},y) \leftrightarrow y=\mathbf{k})$

Given (1) and (2) we have:

$$\vdash_T G \leftrightarrow \exists y(y=\mathbf{k} \,\&\, B(y))$$

and so

$$\vdash_T G \leftrightarrow B(\mathbf{k})$$

which is what we're trying to prove since **k** is $\ulcorner G \urcorner$ (remember that k is the g.n. of G).

VI. **Consistency**

We now need a bunch more definitions. A theory is *consistent* iff there is no sentence, S, such that $\vdash_T S$ and $\vdash_T \sim S$. (A theory is consistent iff it is satisfiable iff some sentence in its language fails to be a theorem.)

VII. **Definability**

Let θ be a set of natural numbers and let T be a theory.

> θ is *definable in T* iff there is some formula, B(x), such that for any k, if k∈θ then $\vdash_T$B(**k**), but if k∉θ then $\vdash_T$~B(**k**)

So definability for sets is a lot like representability for functions. Similarly, we'll say that a relation R on natural numbers — i.e., a set of ordered pairs of natural numbers — is definable in T iff there exists some formula, B(x,y), such that for any <i,j>, if <i,j>∈R then $\vdash_T$B(**i**,**j**), and if <i,j>∉R then $\vdash_T$~B(**i**,**j**).

VIII. Extension

Finally, S is an extension of T iff T⊆S (i.e., iff S is a superset of T). So note that if a function is representable in S then it is representable in all the extensions of S; if a set or relation is definable in S then it is definable in all the extensions of S.

IX. Lemma 3: If T is a consistent extension of Q then the set of Gödel numbers of theorems of T is not definable in T

We can now start making use of the diagonal lemma to prove something interesting. This lemma says that as long as T is sufficiently powerful theory of mathematics, then it "cannot define theoremhood in itself".

Here is the proof. Suppose for reductio that C(y) defines θ, the set of Gödel numbers of theorems of T.

T is a consistent extension of Q; diag is recursive (lemma 1), and all recursive functions are representable in Q (chapter 14); therefore diag is representable in T. Therefore we can apply the diagonal lemma. Let's apply it to the formula ~C(y); we conclude that there exists some G such that:

$$\vdash_T G \leftrightarrow \sim C(\ulcorner G \urcorner)$$

Informally we can think of the sentence here, $G \leftrightarrow \sim C(\ulcorner G \urcorner)$, as saying that G is true iff G is not a theorem of T. For C(y) defines θ, the set of T's theorems, in T, and so we can think of C(y) as meaning that y is (the Gödel number of) a member of θ — i.e. y is a theorem of T. But then we can go through something like liar-sentence type reasoning.

Here's how it goes (back to the formal proof). Let k be the Gödel number of G. Thus, we have:

$\vdash_T G \leftrightarrow \sim C(\mathbf{k})$

Now, either $k \in \theta$ or $k \notin \theta$. Suppose the former. Then $\vdash_T G$. So $\vdash_T \sim C(\mathbf{k})$. But since $k \in \theta$ and C represents θ in T, $\vdash_T C(\mathbf{k})$. So T is inconsistent; but we are given that it is consistent. Contradiction.

So suppose the latter. Then since C represents θ in T, $\vdash_T \sim C(\mathbf{k})$. So $\vdash_T G$. So $k \in \theta$. Contradiction.

X. **Decidable sets**

One last definition. Let E be a set of expressions. We want to characterize the idea that there is an effective procedure for deciding whether a given expression is or is not a member of E.

> Let gn(E) be the set of Gödel numbers of expressions of E. E is *decidable* iff gn(E) is a recursive set.

This seems like an appropriate definition, given Church's thesis. For a set of natural numbers is recursive iff its characteristic function is recursive, and a function is recursive iff it is Turing-computable, and thus iff there is some mechanical procedure for determining its values. And given our Gödel-numbering, we can pass mechanically between Gödel numbers and expressions. Thus, a decidable set is one that we can mechanically tell what's in it and what's not.

XI. **Theorem 1**: No consistent extension of Q is decidable

For the next while we simply mine Lemma 3 for interesting consequences. Here is the proof of Theorem 1. Let T be a consistent extension of Q. Suppose for reductio that T is decidable. Then θ is a recursive set; i.e., $f_{\surd}$, the characteristic function of θ, is a recursive function. Since $f_{\surd}$ is a recursive function it is representable in Q and so in T, by, say, some formula A(x,y).

But then the formula A(x,**1**) defines θ in T. For if $k \in \theta$ then $f_{\surd}(k)=1$, so $\vdash_T \forall y(A(\mathbf{k},y) \leftrightarrow y=\mathbf{1})$ (since A represents $f_{\surd}$ in T) but $\vdash_T \mathbf{1}=\mathbf{1}$, so $\vdash_T A(\mathbf{k},\mathbf{1})$. And if $k \notin \theta$, then $f_{\surd}(k)=0$, so $\vdash_T \forall y(A(\mathbf{k},y) \leftrightarrow y=\mathbf{0})$; but since T is an extension of Q, by Lemma 14.7 we have $\vdash_T \mathbf{1} \neq \mathbf{0}$, whence we have $\sim A(\mathbf{k},\mathbf{1})$.

So θ turns out definable in T. But by Lemma 3, θ is not definable in T. Contradiction.

XII. **Lemma 4:** Q is not decidable

This is a trivial consequence of Theorem 1, since Q is consistent (a model is the standard model).

XIII. **Theorem 2, Church's undecidability theorem**: L is not decidable

Let L be the set of all sentences in the language of arithmetic that are *valid*. Theorem 2 says that this set is not decidable. Thus, we have here another proof that there's no decision procedure for first order logic. For if there were, then we would decide whether any sentence is valid, and so whether any sentence in the language of arithmetic is valid.

The proof of this depends on Q's having finitely many axioms.

Let C = the conjunction of all the axioms of Q. So $\vdash_Q A$ iff $\vdash_L (C \rightarrow A)$.

Intuitively, the proof is now done, since if we had a machine to test theoremhood in L we could use it to decide theoremhood in Q. But we want to make it rigorous, so as not to assume Church's thesis. So. . .

Let q be the Gödel number of C. We can define the following function:

$g(n) = 1 * q * 39999 * n * 2$ = the Gödel number of $(C \rightarrow A)$,
$(\quad C \quad \rightarrow \quad A \quad)$ where n is the Gödel number of A

This function g is recursive.

Now suppose for reductio that L is decidable. Then the set of Gödel numbers of theorems of L would be recursive — i.e., the characteristic function of this set, f_L, would be recursive. But then the function:

$f(n) = 1$ if $f_L(g(n))=1$ (i.e., iff $\vdash_L C \rightarrow A$)
$= 0$ otherwise

would be recursive as well. But f is the characteristic function of the set of Gödel numbers of theorems of Q, since $\vdash_Q A$ iff $\vdash_L (C \rightarrow A)$. So Q is decidable, whereas Lemma 4 says it isn't.

XIV. **Theorem 3:** Arithmetic is not decidable

Let us define *Arithmetic* as the set of all sentences in the language of arithmetic that are true in the standard model. Arithmetic is closed under $\vdash$ and so is a theory: let any sentence A in this theory have B as a logical consequence; then B is true in any model in which A is true; but A is true in the standard model, so B must be as well; so B is a member of Arithmetic.

Arithmetic is an extension of Q (since all the members of Q are true in the standard model) and it is consistent (since it has a model — the standard model). So, by Theorem 1, Arithmetic is not decidable.

This is a further, extremely important result. It implies, given Church's thesis, that there is no purely mechanical procedure one can use to decide whether a given sentence of mathematics is *true* (in the standard model).

XV. **Theorem 4, Tarski's indefinability theorem**: The set of Gödel numbers of sentences true in the standard model is not definable in Arithmetic

This follows directly from Lemma 3.

Let's think about what this means. Roughly, it means that "Arithmetical truth is not arithmetically definable". More carefully: consider a formula, B(x), with just x free. This formula would define the set A of sentences true in Arithmetic if: B(**k**) were a theorem of arithmetic (i.e., true in the standard model) iff k is the Gödel number of a sentence true in the standard model, and ~B(**k**) were a theorem of arithmetic (i.e., true in the standard model) whenever k is the Gödel number of a sentence not true in the standard model. So, in the standard model, B(x) would be a formula that says "x is true in the standard model". This theorem says that there is no such formula B.

XVI. **Lemma 5**: Any recursive set is definable in arithmetic

Pf.: If θ is a recursive set then its characteristic function is recursive; and so representable in Q, and so representable in arithmetic, say by some formula A(x,y). As with the proof of theorem 1, the formula A(x,**1**) then defines θ in arithmetic.

What is the significance of this lemma? It involves the relative strength of theorems 3 and 4. Each denies a certain property of Arithmetic: Theorem 3 says that Arithmetic is undecidable — i.e., that gn(Arithmetic) is not recursive — , whereas Theorem 4 says that Arithmetic is not definable within Arithmetic. But now Lemma 5 says that recursiveness implies definability in arithmetic, and hence indefinability implies undecidability. Hence, Theorem 4 (plus Lemma 5) implies Theorem 3. So Theorem 4 is at least as strong as

Theorem 3. In fact, it is shown in the exercises that definability does not imply recursiveness: some functions are definable in Arithmetic but are not recursive. So Theorem 4 is in a sense stronger, since for all Theorem 3 is concerned, Theorem 4 might be false — gn(Arithmetic) might be one of the non-recursive functions that is definable in Arithmetic.

XVII. **Completeness and axiomatizability**

More definitions. A theory is *complete* iff for every sentence, A, in its language, either A or ~A is a theorem. (If the theory is consistent then exactly one will be a theorem.)

A theory is *axiomatizable* iff it is the set of consequences of some decidable set (it is finitely axiomatizable iff it is the set of consequences of some finite decidable set).

Obviously, every decidable theory is axiomatizable — just take the axioms to be all the theorems. But Q isn't decidable, as was shown in Lemma 4, despite the fact that it is axiomatizable; indeed, finitely axiomatizable. (The reason a theory can be axiomatizable without being decidable is that there is no decision procedure to figure out logical consequences, since first-order logic is undecidable.)

But although not every axiomatizable theory is decidable, some of them are: namely *complete* axiomatizable theories. This is established in the next theorem:

XVIII. **Theorem 5: Any axiomatizable complete theory is decidable**

Let T be any axiomatizable complete theory. Either T is consistent or inconsistent. Take the second case first. In that case T is the set of all sentences in its language. But this set is clearly decidable. For (by Church's thesis) it would be easy to construct a Turing machine that would decide whether a given number is the Gödel number of a well-formed formula.

So suppose on the other hand that T is consistent; and let S be a decidable set of axioms for T. Where A is any sentence in the language, call a sentence *A-interesting* iff it has the form $S_1 \& S_2 \& ... S_n \rightarrow B$, where $S_i \in S$ and B is either A or ~A. By compactness (in the form $\Gamma \vdash A$ iff for some finite $\Gamma_0 \subseteq \Gamma$, $\Gamma_0 \vdash A$), we know that:

(1) $\vdash_T A$ iff some *valid* A-interesting sentence has A as a consequent

Moreover, since T is consistent and complete, $\vdash_T A$ iff it is not the case that $\vdash_T \sim A$, from which we can conclude:

(2) (not: $\vdash_T A$) iff some valid A-interesting sentence has ~A as a consequent

Given these two facts, we can now come up with a Turing machine that takes in the Gödel number of a sentence, A, as an input, and returns 1 iff $\vdash_T A$; 0 otherwise. Assuming we can do this, we will have established what we wanted. For this Turing machine computes the characteristic function of gn(T); and since this function is Turing-computable it is recursive as well.

Here is a sketch of the Turing machine. We take for granted the existence of a Turing machine that is a positive test for validity (this was shown earlier.)

Given the code for sentence, A, run a loop with counter, n. On the n^{th} time through the loop, do the following:

i) construct the sentences with Gödel numbers $\leq$ n that are A-interesting. (This can be done mechanically since A is a decidable set; only finitely many steps are required because the Gödel numbers are $\leq$n.)

ii) Run n steps of the validity machine on each of the sentences constructed in part i)

iii) If the validity machine ever says "yes" then determine whether the consequent of the A-interesting sentence was A or ~A. By our two biconditionals (1) and (2) above, if the consequent is A we can answer that "yes, $\vdash_T A$; if the consequent is ~A we can answer no, not $\vdash_T A$.

iv) otherwise increment the counter and keep going

Since either A or ~A is a theorem (completeness), there exists a valid A-interesting sentence (by the biconditionals (1) and (2)); that sentence has some Gödel number, i. Since the validity machine is indeed a positive test for validity, it will stop after some number, j, of steps, when given the sentence, and it will say "yes". Therefore, our machine will give an answer after max(i,j) times through the loop. And it will give the correct answer, given the accuracy of the validity machine and given the biconditionals (1) and (2).

XIX. **Theorem 6 (Gödel's first incompleteness theorem)**: There is no consistent, complete, axiomatizable extension of Q

Corollary: Arithmetic is not axiomatizable.

This theorem is an immediate consequence of Theorems 1 and 5. And the Corollary is an immediate consequence of the theorem, given that Arithmetic is a consistent, complete extension of Q.

Let's think about what this theorem means. Remember that soundness and completeness are often proven for "formal" or syntactic methods of derivation (e.g., a natural deduction system). Since these are sound and complete, formal consequence is equivalent to our notion of semantic consequence $\vdash$. A "formal theory" is often taken to be one whose theorems are syntactically deducible from a set of axioms; since syntactic deducibility is equivalent to $\vdash$, formal theories are equivalent to our axiomatizable theories. Thus, we could restate the theorem thus:

Any sufficiently strong formal theory of arithmetic is incomplete

Here "sufficiently strong" means that the theory at least contains Q. This theorem indicates an inherent limitation in the formal or axiomatic method. Euclid wanted to axiomatize geometry: supply a set of axioms from which all and only the truths of geometry would be deducible. We have already shown (in Theorem 3) that Arithmetic is not decidable, and so we already knew that this axiomatic method could never provide a decision procedure for deciding whether a given sentence of Arithmetic is true. But now we have shown that the axiomatic method can never succeed in even capturing the truths of arithmetic. For any (recursive) set of axioms you choose, either there will be leftover non-theorems that are true in the standard model, or the chosen axioms will be inconsistent. Arithmetical truth ranges beyond (first-order) provability.