Lecture I

First Order Languages

In a first order language L, all the primitive symbols are among the following:

Connectives: $\sim$, $\supset$.

Parentheses: (,).

Variables: $x_1, x_2, x_3, \ldots$.

Constants: $a_1, a_2, a_3, \ldots$.

Function letters:	$f^1_1, f^1_2, \ldots$	(one-place);
	$f^2_1, f^2_2, \ldots$	(two-place);
	:	
	:	
Predicate letters:	$P^1_1, P^1_2, \ldots$	(one-place);
	$P^2_1, P^2_2, \ldots$	(two-place);
	:	
	:	

Moreover, we place the following constraints on the set of primitive symbols of a first order language L. L must contain *all* of the variables, as well as the connectives and parentheses. The constants of L form an initial segment of $a_1, a_2, a_3, \ldots$, i.e., either L contains all the constants, or it contains all and only the constants $a_1, \ldots, a_n$ for some n, or L contains no constants. Similarly, for any n, the n-place predicate letters of L form an initial segment of $P^n_1, P^n_2, \ldots$ and the n-place function letters form an initial segment of $f^n_1, f^n_2, \ldots$ However, we require that L contain at least one predicate letter; otherwise, there would be no formulae of L.

(We could have relaxed these constraints, allowing, for example, the constants of a language L to be $a_1, a_3, a_5, \ldots$ However, doing so would not have increased the expressive power of first order languages, since by renumbering the constants and predicates of L, we could rewrite each formula of L as a formula of some language L' that meets our constraints. Moreover, it will be convenient later to have these constraints.)

A first order language L is determined by a set of primitive symbols (included in the set described above) together with definitions of the notions of a term of L and of a formula of L. We will define the notion of a *term* of a first order language L as follows:

(i) Variables and constants of L are terms of L.
(ii) If $t_1, ..., t_n$ are terms of L and f^n_i is a function letter of L, then $f^n_i t_1...t_n$ is a term of L.
(iii) The terms of L are only those things generated by clauses (i) and (ii).

Note that clause (iii) (the "extremal clause") needs to be made more rigorous; we shall make it so later on in the course.

An *atomic formula* of L is an expression of the form $P^n_i t_1...t_n$, where P^n_i is a predicate letter of L and $t_1, ..., t_n$ are terms of L. Finally, we define *formula* of L as follows:

(i) An atomic formula of L is a formula of L.
(ii) If A is a formula of L, then so is ~A.
(iii) If A and B are formulae of L, then $(A \supset B)$ is a formula of L.
(iv) If A is a formula of L, then for any i, (x_i) A is a formula of L.
(v) The formulae of L are only those things that are required to be so by clauses (i)-(iv).

Here, as elsewhere, we use 'A', 'B', etc. to range over formulae.

Let x_i be a variable and suppose that $(x_i)B$ is a formula which is a part of a formula A. Then B is called the *scope* of the particular occurrence of the quantifier (x_i) in A. An occurrence of a variable x_i in A is *bound* if it falls within the scope of an occurrence of the quantifier (x_i), or if it occurs inside the quantifier (x_i) itself; and otherwise it is *free*. A *sentence* (or *closed formula*) of L is a formula of L in which all the occurrences of variables are bound.

Note that our definition of formula allows a quantifier (x_i) to occur within the scope of another occurrence of the same quantifier (x_i), e.g. $(x_1)(P^1_1x_1 \supset (x_1)\ P^1_2x_1)$. This is a bit hard to read, but is equivalent to $(x_1)(P^1_1x_1 \supset (x_2)\ P^1_2x_2)$. Formulae of this kind could be excluded from first order languages; this could be done without loss of expressive power, for example, by changing our clause (iv) in the definition of formula to a clause like:

(iv') If A is a formula of L, then for any i, (x_i) A is a formula of L, provided that (x_i) does not occur in A.

(We may call the restriction in (iv') the "nested quantifier restriction"). Our definition of formula also allows a variable to occur both free and bound within a single formula; for example, $P^1_1x_1 \supset (x_1)\ P^1_2x_1$ is a well formed formula in a language containing P^1_1 and P^1_2. A restriction excluding this kind of formulae could also be put in, again without loss of expressive power in the resulting languages. The two restrictions mentioned were adopted by Hilbert and Ackermann, but it is now common usage not to impose them in the definition of formula of a first order language. We will follow established usage, not imposing the restrictions, although imposing them might have some advantages and no important disadvantadge.

We have described our official notation; however, we shall often use an unofficial notation. For example, we shall often use 'x', 'y', 'z', etc. for variables, while officially we should use 'x_1', 'x_2', etc. A similar remark applies to predicates, constants, and function letters. We shall also adopt the following unofficial abbreviations:

$(A \vee B)$ for $(\sim A \supset B)$;
$(A \wedge B)$ for $\sim(A \supset \sim B)$;
$(A \equiv B)$ for $((A \supset B) \wedge (B \supset A))$;
$(\exists x_i)$ A for $\sim(x_i) \sim A$.

Finally, we shall often omit parentheses when doing so will not cause confusion; in particular, outermost parentheses may usually be omitted (e.g. writing $A \supset B$ for $(A \supset B)$). It is important to have parentheses in our official notation, however, since they serve the important function of *disambiguating* formulae. For example, if we did not have parentheses (or some equivalent) we would have no way of distinguishing the two readings of $A \supset B \supset C$, viz. $(A \supset (B \supset C))$ and $((A \supset B) \supset C)$. Strictly speaking, we ought to prove that our official use of parentheses successfully disambiguates formulae. (Church proves this with respect to his own use of parentheses in his *Introduction to Mathematical Logic*.)

Eliminating Function Letters

In principle, we are allowing function letters to occur in our languages. In fact, in view of a famous discovery of Russell, this is unnecessary: if we had excluded function letters, we would not have decreased the expressive power of first order languages. This is because we can eliminate function letters from a formula by introducing a new n+1-place predicate letter for each n-place function letter in the formula. Let us start with the simplest case. Let f be an n-place function letter, and let F be a new n+1-place predicate letter. We can then rewrite

$$f(x_1, ..., x_n) = y$$

as

$$F(x_1, ..., x_n, y).$$

If P is a one-place predicate letter, we can then rewrite

$$P(f(x_1, ..., x_n))$$

as

$$(\exists y)\ (F(x_1, ..., x_n, y) \wedge P(y)).$$

The general situation is more complicated, because formulae can contain complex terms like f(g(x)); we must rewrite the formula f(g(x)) = y as $(\exists z)\ (G(x, z) \wedge F(z, y))$. By repeated applications of Russell's trick, we can rewrite all formulae of the form t = x, where t is a term. We can then rewrite all formulae, by first rewriting

$$A(t_1, ..., t_n)$$

as

$$(\exists x_1)...(\exists x_n)\ (x_1 = t_1 \wedge ... \wedge x_n = t_n \wedge A(x_1, ..., x_n)),$$

and finally eliminating the function letters from the formulae $x_i = t_i$.

Note that we have two different ways of rewriting the negation of a formula $A(t_1,...,t_n)$. We can either simply negate the rewritten version of $A(t_1, ..., t_n)$:

$$\sim(\exists x_1)...(\exists x_n)\ (x_1 = t_1 \wedge ... \wedge x_n = t_n \wedge A(x_1, ..., x_n));$$

or we can rewrite it as

$$(\exists x_1)...(\exists x_n)\ (x_1 = t_1 \wedge ... \wedge x_n = t_n \wedge \sim A(x_1, ..., x_n)).$$

Both versions are equivalent. Finally, we can eliminate constants in just the same way we eliminated function letters, since $x = a_i$ can be rewritten P(x) for a new unary predicate P.

Interpretations

By an *interpretation* of a first order language L (or a *model* of L, or a *structure appropriate for* L), we mean a pair <D, F>, where D (the *domain*) is a nonempty set, and F is a function that assigns appropriate objects to the constants, function letters and predicate letters of L. Specifically,

- F assigns to each constant of L an element of D;
- F assigns to each n-place function letter an n-place function with domain D^n and range included in D; and

- F assigns to each n-place predicate letter of L an n-place relation on D (i.e., a subset of D^n).

Let I = <D, F> be an interpretation of a first order language L. An *assignment* in I is a function whose domain is a subset of the set of variables of L and whose range is a subset of D (i.e., an assignment that maps some, possibly all, variables into elements of D). We now define, for given I, and for all terms t of L and assignments s in I, the function Den(t,s) (the *denotation* (in I) of a term t with respect to an assignment s (in I)), that (when defined) takes a term and an assignment into an element of D, as follows:

(i) if t is a constant, Den(t, s)=F(t);
(ii) if t is a variable and s(t) is defined, Den(t, s)=s(t); if s(t) is undefined, Den(t, s) is also undefined;
(iii) if t is a term of the form $f_i^n(t_1, ..., t_n)$ and $Den(t_j,s)=b_j$ (for j = 1, ..., n), then Den(t, s)=$F(f_i^n)(b_1, ..., b_n)$; if $Den(t_j,s)$ is undefined for some $j \leq n$, then Den(t,s) is also undefined.

Let us say that an assignment s is *sufficient* for a formula A if and only if it makes the denotations of all terms in A defined, if and only if it is defined for every variable occurring free in A (thus, note that all assignments, including the empty one, are sufficient for a sentence). We say that an assignment s in I *satisfies* (in I) a formula A of L just in case

(i) A is an atomic formula $P_i^n(t_1, ..., t_n)$, s is sufficient for A and $<Den(t_1,s),...,Den(t_n,s)> \in F(P_i^n)$; or
(ii) A is ~B, s is sufficient for B but s does not satisfy B; or
(iii) A is (B ⊃ C), s is sufficient for B and C and either s does not satisfy B or s satisfies C; or
(iv) A is (x_i)B, s is sufficient for A and for every s' that is sufficient for B and such that for all $j \neq i$, $s'(x_j)=s(x_j)$, s' satisfies B.

We also say that a formula A is *true* (in an interpretation I) *with respect to an assignment* s (in I) iff A is satisfied (in I) by s; if s is sufficient for A and A is not true with respect to s, we say that A is *false with respect to* s.

If A is a sentence, we say that A is *true in* I iff all assignments in I satisfy A (or, what is equivalent, iff at least one assignment in I satisfies A).

We say that a formula A of L is *valid* iff for every interpretation I and all assignments s in I, A is true (in I) with respect to s (we also say, for languages L containing P_1^2, that a formula A of L is *valid in the logic with identity* iff for every interpretation I=<D,F> where $F(P_1^2)$ is the identity relation on D, and all assignments s in I, A is true (in I) with respect to s). More generally, we say that A is a *consequence* of a set Γ of formulas of L iff for every interpretation I and every assignment s in I, if all the formulas of Γ are true (in I) with respect to s, then A is true (in I) with respect to s. Note that a sentence is valid iff it is true in all its interpretations iff it is a consequence of the empty set. We say that a formula A is *satisfiable* iff for some interpretation I, A is true (in I) with respect to some assignment in I. A sentence is satisfiable iff it is true in some interpretation.

For the following definitions, let an interpretation I=<D,F> be taken as fixed. If A is a formula whose only free variables are $x_1, ..., x_n$, then we say that the n-tuple $<a_1, ..., a_n>$ ($\in D^n$) *satisfies* A (in I) just in case A is satisfied by an assignment s (in I), where $s(x_i) = a_i$ for i = 1, ..., n. (In the case n = 1, we say that a satisfies A just in case the 1-tuple <a> does.) We say that A *defines* (in I) the relation R ($\subseteq D^n$) iff $R=\{<b_1, ..., b_n>: <b_1,...,b_n>$ satisfies A}. An n-place relation R ($\subseteq D^n$) is *definable* (in I) in L iff there is a formula A of L whose only free variables are $x_1, ..., x_n$, and such that A defines R (in I). Similarly, if t is a term whose free variables are $x_1, ..., x_n$, then we say that t defines the function h, where $h(a_1, ..., a_n) = b$ just in case Den(t,s)=b for some assignment s such that $s(x_i) = a_i$. (So officially formulae and terms only define relations and functions when their free variables are $x_1, ..., x_n$ for some n; in practice we shall ignore this, since any formula can be rewritten so that its free variables form an initial segment of all the variables.)

The Language of Arithmetic

We now give a specific example of a first order language, along with its *standard* or *intended* interpretation. The language of arithmetic contains one constant a_1, one function letter f^1_1, one 2-place predicate letter P^2_1, and two 3-place predicate letters P^3_1, and P^3_2. The standard interpretation of this language is <**N**, F> where **N** is the set {0, 1, 2, ...} of natural numbers, and where

$F(a_1) = 0$;
$F(f^1_1)$ = the successor function $s(x) = x+1$;
$F(P^2_1)$ = the identity relation $\{<x, y>: x = y\}$;
$F(P^3_1) = \{<x, y, z>: x + y = z\}$, the graph of the addition function;
$F(P^3_2) = \{<x, y, z>: x \cdot y = z\}$, the graph of the multiplication function.

We also have an unofficial notation: we write

0 for a_1;
x' for f^1_1x;
x = y for P^2_1xy;
A(x, y, z) for P^3_1xyz;

M(x, y, z) for P^3_2xyz.

This presentation of the language of arithmetic is rather atypical, since we use a function letter for successor but we use predicates for addition and multiplication. Note, however, that formulae of a language involving function letters for addition and multiplication instead of the corresponding predicate letters could be rewritten as formulae of the language of arithmetic via Russell’s trick.

A *numeral* is a term of the form $\mathbf{0}'^{\cdots}{}'$, i.e. the constant **0** followed by zero or more successor function signs. The numeral for a number n is zero followed by n successor function signs; we shall use the notation $\mathbf{0}^{(n)}$ for the numeral for n (note that ‘n’ is not a variable of our formal system, but a variable of our informal talk). It may be noted that the only terms of the language of arithmetic, as we have set it up, are the numerals and expressions of the form $x_i'^{\cdots}{}'$.

Finally, note that for the language of arithmetic, we can define satisfaction in terms of truth and substitution. This is because a k-tuple $<n_1, ..., n_k>$ of numbers satisfies $A(x_1, ..., x_k)$ just in case the sentence $A(\mathbf{0}^{(n_1)}, ..., \mathbf{0}^{(n_k)})$ is true (where $A(\mathbf{0}^{(n_1)}, ..., \mathbf{0}^{(n_k)})$ comes from A by substituting the numeral $\mathbf{0}^{(n_i)}$ for all of the free occurrences of the variable x_i).

Lecture II

The Language RE

We shall now introduce the language RE. This is not strictly speaking a first order language, in the sense just defined. However, it can be regarded as a fragment of the first order language of arithmetic.

In RE, the symbols $\wedge$ and $\vee$ are the primitive connectives rather than ~ and $\supset$. RE further contains the quantifier symbol $\exists$ and the symbol < as primitive. The terms and atomic formulae of RE are those of the language of arithmetic as presented above. Then the notion of formula of RE is defined as follows:

(i) An atomic formula of RE is a formula.
(ii) If A and B are formulae, so are $(A \wedge B)$ and $(A \vee B)$.
(iii) If t is a term not containing the variable x_i, and A is a formula, then $(\exists x_i)$ A and $(x_i < t)$ A are formulae.
(iv) Only those things generated by the previous clauses are formulae.

The intended interpretation of RE is the same as the intended interpretation of the first order language of arithmetic (it is the same pair <D,F>). Such notions as truth and satisfaction for formulae of RE and definability by formulae of RE are defined in a way similar to that in which they would be defined for the language of arithmetic using our general definitions of truth and satisfaction; in the appropriate clause, the quantifier $(x_i < t)$ is intuitively interpreted as "for all x_i less than t..." (it is a so called “bounded universal quantifier”).

Note that RE does not contain negation, the conditional or unbounded universal quantification. These are not definable in terms of the primitive symbols of RE. The restriction on the term t of $(x_i < t)$ in clause (iii) above is necessary if we are to exclude unbounded universal quantification from RE, because $(x_i < x_i')$ B is equivalent to (x_i) B.

The Intuitive Concept of Computability and its Formal Counterparts

The importance of the language RE lies in the fact that with its help we will offer a definition that will try to capture the intuitive concept of computability. We call an n-place relation on the set of natural numbers *computable* if there is an effective procedure which, when given an arbitrary n-tuple as input, will in a finite time yield as output 'yes' or 'no' as the n-tuple is or isn't in the relation. We call an n-place relation *semi-computable* if there is an effective procedure such that, when given an n-tuple which is in the relation as input, it eventually yields the output 'yes', and which when given an n-tuple which is not in the relation as input, does not eventually yield the output 'yes'. We do *not* require the procedure to eventually yield the output 'no' in this case. An n-place total function ϕ is called *computable* if there is an effective procedure that, given an n-tuple $<p_1,...,p_n>$ as input, eventually yields $\phi(p_1,...,p_n)$ as output (unless otherwise noted, an n-place function is defined for *all* n-tuples of natural numbers (or all natural numbers if n = 1) —this is what it means for it to be total; and only takes natural numbers as values.)

It is important to note that we place no time limit on the length of computation for a given input, as long as the computation takes place within a finite amount of time. If we required there to be a time limit which could be effectively determined from the input, then the notions of computability and semi-computability would collapse. For let S be a semi-computable set, and let P be a semi-computation procedure for S. Then we could find a computation procedure for S as follows. Set P running on input x, and determine a time limit L from x. If $x \in S$, then P will halt sometime before the limit L. If we reach the limit L and P has not halted, then we will know that $x \notin P$. So as soon as P halts or we reach L, we give an output 'yes' or 'no' as P has or hasn't halted. We will see later in the course, however, that the most important basic result of recursion theory is that the unrestricted notions of computability and semi-computability do not coincide: there are semi-computable sets and relations that are not computable.

The following, however, is true (the *complement* of an n-place relation R (-R) is the collection of n-tuples of natural numbers not in R):

Theorem: A set S (or relation R) is computable iff S (R) and its complement are semi-computable.

Proof: If a set S is computable, there is a computation procedure P for S. P will also be a semi-computation procedure for S. To semi-compute the complement of S, simply follow the procedure of changing a ‘no’ delivered by P to a ‘yes’. Now suppose we have semi-computation procedures for both S and its complement. To compute whether a number n is in S, run simultaneously the two semi-computation procedures on n. If the semi-computation procedure for S delivers a ‘yes’, the answer is yes; if the semi-computation procedure for -S delivers a ‘yes’, the answer is no.

We intend to give formal definitions of the intuitive notions of computable set and relation, semi-computable set and relation, and computable function. Formal definitions of these notions were offered for the first time in the thirties. The closest in spirit to the ones that will be developed here were based on the formal notion of λ-definable function presented by Church. He invented a formalism that he called ‘λ-calculus’, introduced the notion of a function definable in this calculus (a λ-definable function), and put forward the thesis that the computable functions are exactly the λ-definable functions. This is Church’s thesis in its original form. It states that a certain formal concept correctly captures a certain intuitive concept.

Our own approach to recursion theory will be based on the following form of Church's thesis:

Church's Thesis: A set S (or relation R) is semi-computable iff S (R) is definable in the language RE.

We also call the relations definable in RE *recursively enumerable* (or *r.e.*). Given our previous theorem, we can define a set or relation to be *recursive* if both it and its complement are r.e.

Our version of Church's Thesis implies that the recursive sets and relations are precisely the computable sets and relations. To see this, suppose that a set S is computable. Then, by the above theorem, S and its complement are semi-computable, and hence by Church's Thesis, both are r.e.; so S is recursive. Conversely, suppose S is recursive. Then S and -S are both r.e., and therefore by Church's Thesis both are semi-computable. Then by the above theorem, S is computable.

The following theorem will be of interest for giving a formal definition of the remaining intuitive notion of computable function:

Theorem: A total function $\phi(m_1,...,m_n)$ is computable iff the n+1 place relation $\phi(m_1,...,m_n)=p$ is semi-computable iff the n+1 place relation $\phi(m_1,...,m_n)=p$ is computable.
Proof: If $\phi(m_1,...,m_n)$ is computable, the following is a procedure that computes (and hence also semi-computes) the n+1 place relation $\phi(m_1,...,m_n)=p$. Given an input $<p_1,...,p_n,p>$, compute $\phi(p_1,...,p_n)$. If $\phi(p_1,...,p_n)=p$, the answer is yes; if $\phi(p_1,...,p_n)\neq p$, the answer is no. Now suppose that the n+1 place relation $\phi(m_1,...,m_n)=p$ is semi-computable (thus the following would still follow under the assumption that it is computable); then to compute $\phi(p_1,...,p_n)$, run the semi-computation procedure on sufficient n+1 tuples of the form $<p_1,...,p_n,m>$, via some time-sharing trick. For example, run five steps of the semi-computation procedure on $<p_1,...,p_n,0>$, then ten steps on $<p_1,...,p_n,0>$ and $<p_1,...,p_n,1>$, and so on, until you get the n+1 tuple $<p_1,...,p_n,p>$ for which the 'yes' answer comes up. And then give as output p.

A partial function is a function defined on a subset of the natural numbers which need not be the set of all natural numbers. We call an n-place partial function *partial computable* iff there is a procedure which delivers $\phi(p_1,...,p_n)$ as output when ϕ is defined for the argument tuple $<p_1,...,p_n>$, and that does not deliver any output if ϕ is undefined for the argument tuple $<p_1,...,p_n>$. The following result, partially analogous to the above, still holds:

Theorem: A function $\phi(m_1,...,m_n)$ is partial computable iff the n+1 relation $\phi(m_1,...,m_n)=p$

is semi-computable.

Proof: Suppose $\phi(m_1,...,m_n)$ is partial computable; then the following is a semi-computation procedure for the n+1 relation $\phi(m_1,...,m_n)=p$: given an argument tuple $<p_1,...,p_n,p>$, apply the partial computation procedure to $<p_1,...,p_n>$; if and only if it eventually delivers p as output, the answer is yes. Now suppose that the n+1 relation $\phi(m_1,...,m_n)=p$ is semi-computable. Then the following is a partial computation procedure for $\phi(m_1,...,m_n)$. Given an input $<p_1,...,p_n>$, run the semi-computation procedure on n+1 tuples of the form $<p_1,...,p_n,m>$, via some time-sharing trick. For example, run five steps of the semi-computation procedure on $<p_1,...,p_n,0>$, then ten steps on $<p_1,...,p_n,0>$ and $<p_1,...,p_n,1>$, and so on. If you get an n+1 tuple $<p_1,...,p_n,p>$ for which the 'yes' answer comes up, then give as output p.

But it is not the case anymore that a function $\phi(m_1,...,m_n)$ is partial computable iff the n+1 relation $\phi(m_1,...,m_n)=p$ is computable. There is no guarantee that a partial computation procedure will provide a computation procedure for the relation $\phi(m_1,...,m_n)=p$; if ϕ is undefined for $<p_1,...,p_n>$, the partial computation procedure will never deliver an output, but we may have no way of telling that it will not.

In view of these theorems, we now give formal definitions that intend to capture the intuitive notions of computable function and partial computable function. An n-place partial function is called *partial recursive* iff its graph is r.e. An n-place total function is called *total recursive* (or simply *recursive*) iff its graph is r.e. Sometimes the expression 'general recursive' is used instead of 'total recursive', but this is confusing, since the expression 'general recursive' was originally used not as opposed to 'partial recursive' but as opposed to 'primitive recursive'.

It might seem that we can avoid the use of partial functions entirely, say by replacing a partial function ϕ with a total function ψ which agrees with ϕ wherever ϕ is defined, and which takes the value 0 where ϕ is undefined. Such a ψ would be a total extension of ϕ, i.e. a total function which agrees with ϕ wherever ϕ is defined. However, this will not work, since there are some partial recursive functions which are not totally extendible, i.e. which do not have any total extensions which are recursive functions. (We shall prove this later on in the course.)

Our version of Church's Thesis implies that a function is computable iff it is recursive. To see this, suppose that ϕ is a computable function. Then, by one of the theorems above, its graph is semi-computable, and so by Church's Thesis, it is r.e., and so ϕ is recursive. Conversely, suppose that ϕ is recursive. Then ϕ's graph is r.e., and by Church's Thesis it is semi-computable; so by the same theorem, ϕ is computable.

Similarly, our version of Church's Thesis implies that a function is partial computable iff it is partial recursive.

We have the result that if a total function has a semi-computable graph, then it has a computable graph. That means that the complement of the graph is also semi-computable.

We should therefore be able to show that the graph of a recursive function is also recursive. In order to do this, suppose that ϕ is a recursive function, and let R be its graph. R is r.e., so it is defined by some RE formula $B(x_1, ..., x_n, x_{n+1})$. To show that R is recursive, we must show that -R is r.e., i.e. that there is a formula of RE which defines -R. A natural attempt is the formula

$$(\exists x_{n+2})(B(x_1, ..., x_n, x_{n+2}) \wedge x_{n+1} \neq x_{n+2}).$$

This does indeed define -R as is easily seen, but it is not a formula of RE, for its second conjunct uses negation, and RE does not have a negation sign. However, we can fix this problem if we can find a formula of RE that defines the nonidentity relation $\{<m,n>:m\neq n\}$.

Let us define the formula

$$\text{Less}(x, y) =_{df.} (\exists z)\, A(x, z', y).$$

Less (x, y) defines the less-than relation $\{<m, n>: m < n\}$. We can now define inequality as follows:

$$x \neq y =_{df.} \text{Less}(x, y) \vee \text{Less}(y, x).$$

This completes the proof that the graph of a total recursive function is a recursive relation, and also shows that the less-than and nonidentity relations are r.e., which will be useful in the future.

While we have not introduced bounded existential quantification as a primitive notation of RE, we can define it in RE, as follows:

$$(\exists x < t)\, B =_{df.} (\exists x)\, (\text{Less}(x, t) \wedge B).$$

In practice, we shall often write 'x < y' for 'Less (x, y)'. However, it is important to distinguish the defined symbol '<' from the primitive symbol '<' as it appears within the bounded universal quantifier. We also define

$$(\exists x \leq t)\, B(x) =_{df.} (\exists x < t)\, B(x) \vee B(t);$$
$$(x \leq t)\, B(x) =_{df.} (x < t)\, B(x) \wedge B(t).$$

The Status of Church's Thesis

Our form of Church's thesis is that the intuitive notion of semi-computability and the formal notion of recursive enumerability coincide. That is, a set or relation is semi-computable iff it is r.e. Schematically:

r.e. = semi-computable.

The usual form of Church's Thesis is: recursive = computable. But as we saw, our form of Church's Thesis implies the usual form.

In some introductory textbooks on recursion theory Church's Thesis is assumed in proofs, e.g. in proofs that a function is recursive that appeal to the existence of an effective procedure (in the intuitive sense) that computes it. (Hartley Rogers' *Theory of Recursive Functions and Effective Computability* is an example of this.) There are two advantages to this approach. The first is that the proofs are intuitive and easier to grasp than very “formal” proofs. The second is that it allows the student to cover relatively advanced material fairly early on. The disadvantage is that, since Church's Thesis has not actually been proved, the student never sees the proofs of certain fundamental theorems. We shall therefore not assume Church's Thesis in our proofs that certain sets or relations are recursive. (In practice, if a recursion theorist is given an informal effective procedure for computing a function, he or she will regard it as proved that that function is recursive. However, an experienced recursion theorist will easily be able to convert this proof into a rigorous proof which makes no appeal whatsoever to Church's Thesis. So working recursion theorists should not be regarded as appealing to Church's Thesis in the sense of assuming an unproved conjecture. The beginning student, however, will not in general have the wherewithal to convert informal procedures into rigorous proofs.)

Another usual standpoint in some presentations of recursion theory is that Church's Thesis is not susceptible of proof or disproof, because the notion of recursiveness is a precise *mathematical* notion and the notion of computability is an *intuitive* notion. Indeed, it has not in fact been proved (although there is a lot of evidence for it), but in the author's opinion, no one has shown that it is *not susceptible* of proof or disproof. Although the notion of computability is not taken as primitive in standard formulations of mathematics, say in set theory, it does have many intuitively obvious properties, some of which we have just used in the proofs of perfectly rigorous theorems. Also, $y = x!$ is evidently computable, and so is $z=x^y$ (although it is not immediately obvious that these functions are recursive, as we have defined these notions). So suppose it turned out that one of these functions was not recursive. That would be an absolute disproof of Church's Thesis. Years before the birth of recursion theory a certain very wide class of computable functions was isolated, that later would come to be referred to as the class of “primitive recursive” functions. In a famous paper, Ackermann presented a function which was evidently computable (and which is in fact recursive), but which was not primitive recursive. If someone had conjectured that the computable functions are the primitive recursive functions, Ackermann’s function would have provided an absolute disproof of that conjecture. (Later we will explain what is the class of primitive recursive functions and we will define Ackermann’s function.) For another example, note that the composition of two computable functions is intuitively computable; so, if it turned out that the formal notion of recursiveness was not closed under composition, this would show that Church's Thesis is wrong.

Perhaps some authors acknowledge that Church's Thesis is open to absolute disproof, as in the examples above, but claim that it is not open to proof. However, the conventional argument for this goes on to say that since computability and semi-computability are merely intuitive notions, not rigorous mathematical notions, a proof of Church's Thesis could not be given. This position, however, is not consistent if the intuitive notions in question cannot be used in rigorous mathematical arguments. Then a disproof of Church's Thesis would be impossible also, for the same reason as a proof. In fact, suppose for example that we could give a list of principles intuitively true of the computable functions and were able to prove that the only class of functions with these properties was exactly the class of the recursive functions. We would then have a proof of Church's Thesis. While this is in principle possible, it has not yet been done (and it seems to be a very difficult task).

In any event, we can give a perfectly rigorous proof of *one half* of Church's thesis, namely that every r.e relation (or set) is semi-computable.

Theorem: Every r.e. relation (or set) is semi-computable.

Proof: We show by induction on the complexity of formulae that for any formula B of RE, the relation that B defines is semi-computable, from which it follows that all r.e. relations are semi-computable. We give, for each formula B of RE, a procedure P_B which is a semi-computation of the relation defined by B.

If B is atomic, then it is easy to see that an appropriate P_B exists; for example, if B is the formula $x_1''' = x_2'$, then P_B is the following procedure: add 3 to the first input, then add 1 to the second input, and see if they are the same, and if they are, halt with output 'yes'.

If B is $(C \wedge D)$, then P_B is the following procedure: first run P_C, and if it halts with output 'yes', run P_D; if that also halts, then halt with output 'yes'.

If B is $(C \vee D)$, then P_B is as follows. Run P_C and P_D simultaneously via some time-sharing trick. (For example, run 10 steps of P_C, then 10 steps of P_D, then 10 more steps of P_C, ...) As soon as one answers 'yes', then let P_B halt with output 'yes'.

Suppose now that B is $(y < t)\ C(x_1, ..., x_n, y)$. If t is a numeral $\mathbf{0}^{(p)}$, then $<m_1, ..., m_n>$ satisfies B just in case all of $<m_1, ..., m_n, 0>$ through $<m_1, ..., m_n, p\text{-}1>$ satisfy C, so run P_C on input $<m_1, ..., m_n, 0>$; if P_C answers yes, run P_C on input $<m_1, ..., m_n, 1>$, ... If you reach p-1 and get an answer yes, then $<m_1, ..., m_n>$ satisfies B, so halt with output 'yes'. If t is a term $x_i'^{\cdots}'$, then the procedure is basically the same. Given an input which includes the values $m_1, ..., m_n$ of $x_1, ..., x_n$, as well as the value of x_i, first calculate the value p of the term t, and then run P_C on $<m_1, ..., m_n, 0>$ through $<m_1, ..., m_n, p\text{-}1>$, as above. So in either case, an appropriate P_B exists.

Finally, if $B = (\exists y)\ C(x_1, ..., x_n, y)$, then P_C is as follows: given input $<m_1, ..., m_n>$, run P_C on $<m_1, ..., m_n, k>$ simultaneously for all k and wait for P_C to deliver 'yes' for some k.

Again, we use a time-sharing trick; for example: first run P_C on $<m_1, ..., m_n, 0>$ for 10 steps, then run P_C on $<m_1, ..., m_n, 0>$ and $<m_1, ..., m_n, 1>$ for 20 steps each, then .... Thus, an appropriate P_B exists in this case as well, which completes the proof.

This proof cannot be formalized in set theory, so in that sense the famous thesis of the logicists that all mathematics can be done in set theory might be wrong. But a weaker thesis that every intuitive mathematical notion can always be replaced by one definable in set theory (and coextensive with it) might yet be right.

Kreisel's opinion—in a review—appears to be that computability is a legitimate primitive only for intuitionistic mathematics. In classical mathematics it is not a primitive, although (*pace* Kreisel) it could be taken to be one. In fact the above argument, that the recursive sets are all computable, is not intuitionistically valid, because it assumes that a number will be either in a set or in its complement. (If you don't know what intuitionism is, don't worry.)

It is important to notice that recursiveness (and recursive enumerability) is a property of a set, function or relation, not a *description* of a set, function or relation. In other words, recursiveness is a property of extensions, not intensions. To say that a set is r.e. is just to say that there exists a formula in RE which defines it, and to say that a set is recursive is to say that there exists a pair of formulae in RE which define it and its complement. But you don't necessarily have to know what these formulae are, contrary to the point of view that would be taken on this by intuitionistic or constructivist mathematicians. We might have a theory of recursive descriptions, but this would not be conventional recursive function theory. So for example, we know that any finite set is recursive; every finite set will be defined in RE by a formula of the form $x_1=\mathbf{0}^{(k1)}\vee...\vee x_n=\mathbf{0}^{(kn)}$, and its complement by a formula of the form $x_1\neq\mathbf{0}^{(k1)}\wedge...\wedge x_n\neq\mathbf{0}^{(kn)}$. But we may have no procedure for deciding whether something is in a certain finite set or not - finding such a procedure might even be a famous unsolved problem. Consider this example: let S = {n: at least n consecutive 7's appear in the decimal expansion of π}. Now it's hard to say what particular n's are in S (it's known that at least four consecutive 7's appear, but we certainly don't know the answer for numbers much greater than this), but nonetheless S is recursive. For, if $n \in S$ then any number less than n is also in S, so S will either be a finite initial segment of the natural numbers, or else it will contain all the natural numbers. Either way, S is recursive.

There is, however, an intensional version of Church's Thesis that, although hard to state in a rigorous fashion, seems to be true in practice: whenever we have an intuitive procedure for semi-computing a set or relation, it can be "translated" into an appropriate formula of the formalism RE, and this can be done in some sense effectively (the "translation" is intuitively computable). This version of Church's Thesis operates with the notion of arbitrary descriptions of sets or relations (in English, or in mathematical notation, say), which is somewhat vague. It would be good if a more rigorous statement of this version of Church's Thesis could be made.

The informal notion of computability we intend to study in this course is a notion different from a notion of analog computability that might be studied in physics, and for which there is no reason to believe that Church's Thesis holds. It is not at all clear that every function of natural numbers computable by a physical device, that can use analog properties of physical concepts, is computable by a digital algorithm. There have been some discussions of this matter in a few papers, although the ones known to the author are quite complicated. Here we will make a few rather unsophisticated remarks.

There are certain numbers in physics known as universal constants. Some of these numbers are given in terms of units of measure, an are different depending on the system of units of measures adopted. Some other of these numbers, however, are not given in terms of units of measure, for example, the electron-proton mass ratio; that is, the ratio of the mass of an electron to the mass of a proton. We know that the electron-proton mass ratio is a positive real number *r* less than 1 (the proton is heavier than the electron). Consider the following function ψ: $\psi(k)$ = the kth number in the decimal expansion of *r*. (There are two ways of expanding finite decimals, with nines at the end or with zeros at the end; in case *r* is finite, we arbitrarily stipulate that its expansion is with zeros at the end.) As far as I know, nothing known in physics allows us to ascribe to *r* any mathematical properties (e.g., being rational or irrational, being algebraic or transcendental, even being a finite or an infinite decimal). Also, as far as I know, it is not known whether this number is recursive, or Turing computable.

However, people do attempt to measure these constants. There might be problems in carrying out the measurement to an arbitrary degree of accuracy. It might take longer and longer to calculate each decimal place, it might take more and more energy, time might be finite, etc. Nevertheless, let us abstract from all these difficulties, assuming, e.g., that time is infinite. Then, as far as I can see, there is no reason to believe that there cannot be any physical device that would actually calculate each decimal place of *r*. But this is not an algorithm in the standard sense. ψ might even then be uncomputable in the standard sense.

Let us review another example. Consider some quantum mechanical process where we can ask, e.g., whether a particle will be emitted by a certain source in the next second, or hour, etc. According to current physics, this kind of thing is not a deterministic process, and only relevant probabilities can be given that a particle will be emitted in the next second, say. Suppose we set up the experiment in such a way that there is a probability of 1/2 for an emission to occur in the next second, starting at some second s_0. We can then define a function $\chi(k) = 1$ if an emission occurs in s_k, and $= 0$ if an emission does not occur in s_k. This is not a universally defined function like ψ, but if time goes on forever, this experiment is a physical device that gives a universally defined function. There are only a denumerable number of recursive functions (there are only countably many strings in RE, and hence only countably many formulae). In terms of probability theory, for any infinite sequence such as the one determined by χ there is a probability of 1 that it will lie outside any denumerable set (or set of measure zero). So in a way we can say with certainty that χ, even though "computable" by our physical device, is not recursive, or, equivalently, Turing computable. (Of course, χ may turn out to be recursive if there is an underlying deterministic structure to our experiment, but assuming quantum mechanics, there is not.) This example again illustrates the fact that the concept of physical computability involved is not the informal concept of computability referred to in Church's Thesis.

Lecture III

The Language Lim

In the language RE, we do not have a negation operator. However, sometimes, the complement of a relation definable by a formula of RE is definable in RE by means of some trick. We have already seen that the relation defined by $t_1 \neq t_2$ (where t_1, t_2 are two terms of RE) is definable in RE, and whenever B defines the graph of a total function, the complement of this graph is definable.

In RE we also do not have the conditional. However, if A is a formula whose negation is expressible in RE, say by a formula A* (notice that A need not be expressible in RE), then the conditional $(A \supset B)$ would be expressible by means of $(A^* \lor B)$ (provided B is a formula of RE); thus, for example, $(t_1=t_2 \supset B)$ is expressible in RE, since $t_1 \neq t_2$ is. So when we use the conditional in our proofs by appeal to formulae of RE, we'll have to make sure that if a formula appears in the antecedent of a conditional, its negation is expressible in the language. In fact, this requirement is too strong, since a formula appearing in the antecedent of a conditional may appear without a negation sign in front of it when written out only in terms of negation, conjunction and disjunction. Consider, for example, a formula

$$(A \supset B) \supset C,$$

in which the formula A appears as a part in the antecedent of a conditional. This conditional is equivalent to

$$(\sim A \lor B) \supset C,$$

and in turn to

$$\sim(\sim A \lor B) \lor C,$$

and to

$$(A \land \sim B) \lor C.$$

In the last formula, in which only negation, conjunction and disjunction are used, A appears purely positively, so it's not necessary that its negation be expressible in RE in order for $(A \supset B) \supset C$ to be expressible in RE.

A bit more rigorously, we give an inductive construction that determines when an occurrence of a formula A in a formula F whose only connectives are ~ and ⊃ is positive or negative: if A is F, A's occurrence in F is positive; if F is ~B, A's occurrence in F is negative if it is positive in B, and vice versa; if F is (B⊃C), an occurrence of A in B is negative if positive in B, and vice versa, and an occurrence of A in C is positive if positive in C, and negative if negative in C.

It follows from this that if an occurrence of a formula appears as a part in another formula in an even number of antecedents (e.g., A in the formula of the example above), the corresponding occurrence will be positive in an ultimately reduced formula employing only negation, conjunction and disjunction. If an occurrence of a formula appears as a part in another formula in an odd number of antecedents (e.g., B in the formula above), the corresponding occurrence will appear with a negation sign in front of it in the ultimately reduced formula (i.e., it will be negative) and we will have to make sure that the negated formula is expressible in RE.

In order to avoid some of these complications involved in working within RE, we will now define a language in which we have unrestricted use of negation, but such that all the relations definable in it will also be definable in RE. We will call this language *Lim.* Lim has the same primitive symbols as RE, plus a symbol for negation (~). The terms and atomic formulae of Lim are just those of RE. Then the notion of formula of Lim is defined as follows:

(i) An atomic formula of Lim is a formula of Lim;
(ii) If A and B are formulae of Lim, so are ~A, $(A \wedge B)$ and $(A \vee B)$;
(iii) If t is a term not containing the variable x_i, and A is a formula of Lim, then $(\exists x_i<t))$ A and $(x_i < t)$ A are formulae of Lim;
(iv) Only those things generated by the previous clauses are formulae.

Notice that in Lim we no longer have unbounded existential quantification, but only bounded existential quantification. This is the price of having negation in Lim.

Lim is weaker than RE in the sense that any set or relation definable in Lim is also definable in RE. This will mean that if we are careful to define a relation using only bounded quantifiers, its complement will be definable in Lim, and hence in RE, and this will show that the relation is recursive. Call two formulae with the same free variables *equivalent* just in case they define the same set or relation. (So closed formulae, i.e. sentences, are equivalent just in case they have the same truth value.) To show that Lim is weaker than RE, we prove the following

Theorem: Any formula of Lim is equivalent to some formula of RE.
Proof: We show by induction on the complexity of formulae that if B is a formula of Lim, then both B and ~B are equivalent to formulae of RE. First, suppose B is atomic. B is then a formula of RE, so obviously B is equivalent to some RE formula. Since inequality is an

r.e. relation and the complement of the graph of any recursive function is r.e., ~B is equivalent to an RE formula. If B is ~C, then by inductive hypothesis C is equivalent to an RE formula C* and ~C is equivalent to an RE formula C**; then B is equivalent to C** and ~B (i.e., ~~C) is equivalent to C*. If B is (C $\wedge$ D), then by the inductive hypothesis, C and D are equivalent to RE formulae C* and D*, respectively, and ~C, ~D are equivalent to RE formulae C** and D**, respectively. So B is equivalent to (C* $\wedge$ D*), and ~B is equivalent to (C** $\vee$ D**). Similarly, if B is (C $\vee$ D), then B and ~B are equivalent to (C* $\vee$ D*) and (C** $\wedge$ D**), respectively. If B is ($\exists x_i$ < t) C, then B is equivalent to ($\exists x_i$)(Less(x_i, t)$\wedge$C*), and ~B is equivalent to (x_i < t) ~C and therefore to (x_i < t) C**. Finally, the case of bounded universal quantification is similar.

A set or relation definable in Lim is recursive: if B defines a set or relation in Lim, then ~B is a formula of Lim that defines its complement, and so by the foregoing theorem both it and its complement are r.e. (Once we have shown that not all r.e. sets are recursive, it will follow that Lim is *strictly* weaker than RE, i.e. that not all sets and relations definable in RE are definable in Lim.) Since negation is available in Lim, the conditional is also available, as indeed are all truth-functional connectives. Because of this, showing that a set or relation is definable in Lim is a particularly convenient way of showing that it is recursive; in general, if you want to show that a set or relation is recursive, it is a good idea to show that it is definable in Lim (if you can).

We can expand the language Lim by adding extra predicate letters and function letters and interpreting them as recursive sets and relations and recursive functions. If we do so, the resulting language will still be weaker than RE:

Theorem: Let Lim' be an expansion of Lim in which the extra predicates and function letters are interpreted as recursive sets and relations and recursive functions. Then every formula of Lim' is equivalent to some formula of RE.
Proof: As before, we show by induction on the complexity of formulae that each formula of Lim' and its negation are equivalent to RE formulae. The proof is analogous to the proof of the previous theorem. Before we begin the proof, let us note that every term of Lim' stands for a recursive function; this is simply because the function letters of Lim' define recursive functions, and the recursive functions are closed under composition. So if t is a term of Lim', then both t = y and ~(t = y) define recursive relations and are therefore equivalent to formulae of RE.

Suppose B is the atomic formula P(t_1, ..., t_n), where t_1, ..., t_n are terms of Lim' and P is a predicate of Lim' defining the recursive relation R. Using Russell's trick, we see that B is equivalent to ($\exists x_1$)...($\exists x_n$)($t_1 = x_1 \wedge ... \wedge t_n = x_n \wedge$ P(x_1, ..., x_n)), where x_1, ..., x_n do not occur in any of the terms t_1, ..., t_n. Letting C_i be an RE formula which defines the relation defined by $t_i = x_i$, and letting D be an RE formula which defines the relation that P defines, we see that B is equivalent to the RE formula ($\exists x_1$)...($\exists x_n$)($C_1(x_1) \wedge ... C_n(x_n) \wedge$ D(x_1, ...,

x_n)). To see that ~B is also equivalent to an RE formula, note that R is a recursive relation, so its complement is definable in RE, and so the formula $(\exists x_1)...(\exists x_n)(t_1 = x_1 \wedge ... \wedge t_n = x_n \wedge \sim P(x_1, ..., x_n))$, which is equivalent to ~B, is also equivalent to an RE formula.

The proof is the same as the proof of the previous theorem in the cases of conjunction, disjunction, and negation. In the cases of bounded quantification, we have to make a slight adjustment, because the term t in $(x_i < t)$ B or $(\exists x_i < t)$ B might contain new function letters. Suppose B and ~B are equivalent to the RE formulae B* and B**, and let t = y be equivalent to the RE formula C(y). Then $(x_i < t)$ B is equivalent to the RE formula $(\exists y)(C(y) \wedge (x_i < y)\, B^*))$, and $\sim(x_i < t)$ B is equivalent to $(\exists x_i < t)$ ~B, which is in turn equivalent to the RE formula $(\exists y)\,(C(y) \wedge (\exists x_i < y)\, B^{**})$. The case of bounded existential quantification is similar.

This fact will be useful, since in RE and Lim the only bounds we have for the bounded quantifiers are terms of the forms $\mathbf{0}^{(n)}$ and $x_i'\cdots'$. In expanded languages containing function letters interpreted as recursive functions there will be other kinds of terms that can serve as bounds for quantifiers in formulae of the language, without these formulae failing to be expressible in RE.

There is a variant of Lim that should be mentioned because it will be useful in future proofs. *Lim*$^+$ is the language which is just like Lim except that it has function letters rather than predicates for addition and multiplication. (So in particular, quantifiers in Lim$^+$ can be bounded by terms containing + and ·.) It follows almost immediately from the previous theorem that every formula of Lim$^+$ is equivalent to some formula of RE. We call a set or relation *limited* if it is definable in the language Lim$^+$. We call it *strictly limited* if it is definable in Lim.

Pairing Functions

We will define a *pairing function* on the natural numbers to be a dominating total binary recursive function ϕ such that for all m_1, m_2, n_1, n_2, if $\phi(m_1, m_2) = \phi(n_1, n_2)$ then $m_1 = n_1$ and $m_2 = n_2$ (that a binary function ϕ is dominating means that for all m, n, $m \leq \phi(m, n)$ and $n \leq \phi(m, n)$). Pairing functions allow us to code pairs of numbers as individual numbers, since if p is in the range of a pairing function ϕ, then there is exactly one pair (m, n) such that $\phi(m, n) = p$, so the constituents m and n of the pair that p codes are uniquely determined by p alone.

We are interested in finding a pairing function. If we had one, that would show that the theory of recursive functions in two variables essentially reduces to the theory of recursive functions in one variable. This will be because it is easily proved that for all binary relations R, if ϕ is a pairing function, R is recursive (r.e.) iff the *set* $\{\phi(m, n): R(m, n)\}$ is recursive (r.e.). We are going to see that there are indeed pairing functions, so that there is no essential difference between the theories of recursive binary relations and of recursive sets.

This is in contrast to the situation in the topologies of the real line and the plane. Cantor discovered that there is a one-to-one function from the real line onto the plane. This result was found to be surprising by Cantor himself and by others, since the difference between the line and the plane seemed to lie in the fact that points in the plane could only be specified or uniquely determined by means of pairs of real numbers, and Cantor's result seemed to imply that every point in the plane could be identified by a single real number. But the real line and the plane are topologically distinct, that is, there is no homeomorphism of the real line onto the plane, which means that they are essentially different topological spaces. In fact, Brouwer proved a theorem from which the general result follows that there is no homeomorphism between m-dimensional Euclidean space and n-dimensional Euclidean space (for $m \neq n$).

The following will be our pairing function. Let us define $[x, y]$ to be $(x+y)^2+x$. This function is evidently recursive, since it is limited, as it is defined by the Lim^+ formula $z = (x + y)\cdot(x + y) + x$, and is clearly dominating. Let us show that it is a pairing function, that is, that for all z, if $z = [x, y]$ for some x and y, then x and y are uniquely determined. Let $z = (x+y)^2+x$. $(x+y)^2$ is uniquely determined, and it is the greatest perfect square $\leq z$: if it weren't, then we would have $(x + y + 1)^2 \leq z$, but $(x + y + 1)^2 = (x + y)^2 + 2x + 2y + 1 > (x + y)^2 + x = z$. Let $s=x+y$, so that $s^2=(x+y)^2$. Since $z>s^2$, we can put $x=z-s^2$, which is uniquely determined, and $y=s-x=s-(z-s^2)$, which is uniquely determined. This completes the proof that $[x,y]$ is a pairing function. Note that it is not onto, i.e. some numbers do not code pairs of numbers. For our purposes this will not matter.

(The earliest mention of this pairing function known to the author is in Goodstein's *Recursive Number Theory.* Several years later, the same function was used by Quine, who probably thought of it independently.)

Our pairing function can be extended to n-place relations. First, note that we can get a recursive tripling function by letting $[x, y, z] = [[x, y], z]$. We can similarly get a recursive n-tupling function, $[m_1, ..., m_n]$, and we can prove an analogous result to the above in the case of n-place relations: for all n-place relations R, if ϕ is a recursive n-tupling function, R is recursive (r.e.) iff the *set* $\{\phi(m_1,...,m_n): R(m_1,...,m_n)\}$ is recursive (r.e.).

Our pairing function has recursive inverses, i.e. there are recursive functions K_1 and K_2 such that $K_1([x, y]) = x$ and $K_2([x, y]) = y$ for all x and y. When z does not code any pair, we could let K_1 and K_2 be undefined on z; here, however, we let K_1 and K_2 have the value 0 on z. (So we can regard z as coding the pair <0, 0>, though in fact $z \neq [0, 0]$.) Intuitively, K_1 and K_2 are computable functions, and indeed they are recursive. To see this, note that K_1's graph is defined by the formula of Lim $(\exists y \leq z)\ (z = [x, y]) \lor (x = \mathbf{0} \land {\sim}(\exists y \leq z)\ (\exists w \leq z)\ z = [w, y])$; similarly, K_2's graph is defined by the formula of Lim $(\exists x \leq z)\ (z = [x, y]) \lor (y = \mathbf{0} \land {\sim}(\exists x \leq z)\ (\exists w \leq z)\ z = [x, w])$.

Coding Finite Sequences

We have seen that for any n, there is a recursive n-tupling function; or in other words, we have a way of coding finite sequences of fixed length. Furthermore, all these n-tupling functions have recursive inverses. This does not, however, give us a single, one-to-one function for coding finite sequences of arbitrary length. One of the things Cantor showed is that there is a one-to-one correspondence between the natural numbers and the set of finite sequences of natural numbers, so a function with the relevant property does exist. What we need to do, in addition, is to show that an effective way of assigning different numbers to different sequences exists, and such that the decoding of the sequences from their codes can be done also effectively.

A method of coding finite sequences of variable length, due to Gödel, consists in assigning to an n-tuple $<m_1, ..., m_n>$ the number $k=2^{m_1+1}\cdot 3^{m_2+1}\cdot ... \cdot p_n{}^{m_n+1}$ as code (where $p_1=2$ and p_{i+1}=the first prime greater than p_i). It is clear that k can be uniquely decoded, since every number has a unique prime factorization, and intuitively the decoding function is computable. If we had exponentiation as a primitive of RE, it would be quite easy to see that the decoding function is recursive; but we do not have it as a primitive. Although Gödel did not take exponentiation as primitive, he found a trick, using the Chinese Remainder Theorem, for carrying out the above coding with only addition, multiplication and successor as primitive. We could easily have taken exponentiation as a primitive — it is not essential to recursion theory that the language of RE have only successor, addition and multiplication as primitive and other operations as defined. If we had taken it as primitive, our proof of the easy half of Church's thesis, i.e. that all r.e. relations are semi-computable, would still have gone through, since exponentiation is clearly a computable function. Similarly, we could have added to RE new variables to range over finite sets of numbers, or over finite sequences. In fact, doing so might have saved us some time at the beginning of the course. However, it is traditional since Gödel's work to take quantification over numbers, and successor, addition, and multiplication as primitive and to show how to define the other operations in terms of them.

We will use a different procedure for coding finite sequences, the basic idea of which is due to Quine. If you want to code the sequence <5, 4, 7>, why not use the number 547? In general, a sequence of positive integers less than 10 can be coded by the number whose decimal expansion is the sequence. Unfortunately, if you want to code sequences containing numbers larger than or equal to 10, this won't quite work. (Also, if the first element of a sequence $<m_1, ..., m_n>$ is 0, its code will be the same as the code for the sequence $<m_2, ..., m_n>$; this problem is relatively minor compared to the other). Of course, it is always possible to use a larger base; if you use a number to code its base-100 expansion, for example, then you can code sequences of numbers as large as 99. Still, this doesn't provide a single method for coding sequences of arbitrary length.

To get around this, we shall use a modification of Quine's trick, due to the author. The main idea is to use a variable base, so that a number may code a different sequence to a different base. It also proves convenient in this treatment to use only prime bases. Another feature of our treatment is that we will code finite sets first, rather than finite sequences; this will mean that every finite set will have many different codes (thus, using base 10 only for purposes of motivation, 547 and 745 would code the same set {4, 5, 7}). We will not allow 0 as the first digit of a code (in a base p) of a set, because otherwise 0 would be classified as a member of the set, whether it was in it or not (of course, 0 will be allowed as an intermediate or final digit).

Our basic idea is to let a number n code the set of all the numbers that appear as digits in n's base-p expansion, for appropriate prime p. No single p will do for all sets, since for any prime p, there is a finite set containing numbers larger than p, and which therefore cannot be represented as a base-p numeral. However, in view of a famous theorem due to Euclid, we can get around this.

Theorem (Euclid): There are infinitely many primes.
Proof. Let n be any number, and let's show that there are primes greater than n . n! + 1 is either prime or composite. If it is prime, it is a prime greater than n. If it is composite, then it has some prime factor p; but then p must be greater than n, since n!+1 is not divisible by any prime less than or equal to n. Either way, there is a prime number greater than n; and since n was arbitrary, there are arbitrarily large primes.

So for any finite set S of numbers, we can find a prime p greater than any element of S, and a number n such that the digits of the base-p expansion of n are the elements of S. (To give an example, consider the finite set {1, 2}. This will have as “codes” in base 3 the numbers denoted by '12' and '21' in base 3 notation, that is, 5 and 7; it will have as “codes” in base 5 the numbers 7 and 11, etc.) We can then take [n, p] as a code of the set S (so, in the example, [5,3], [7,3], [7,5] and [11,5] are all codes of {1,2}). In this fashion different finite sets will never be assigned the same code. Further, from a code the numbers n and p are uniquely determined and effectively recoverable, and from n and p the set S is determined uniquely.

We will now show how to carry out our coding scheme in RE. To this effect, we will show that a number of relations are definable in Lim or Lim$^+$ (and hence not only r.e, but also recursive). Before we begin, let us note that the relation of nonidentity is definable in Lim and in Lim$^+$, for we can define a formula Less*(x,y) equivalent to the formula Less(x,y) of RE with only bounded quantification: Less*(x,y) $=_{df.}$ $(\exists z<y)(x+z'=y)$ (an even simpler formula defining the less than relation in Lim and Lim$^+$ would be $(\exists z<y)(x=z)$). Now, let's put

$$\mathrm{Pr}\,(x) =_{df.} x \neq \mathbf{0} \wedge x \neq \mathbf{0'} \wedge (y \leq x)(z \leq x)(M(y,z,x) \supset (y=x \vee z=x)).$$

Pr(x) defines the set of primes in Lim, as is easily seen. We want next to define the relation *w is a power of p*, for prime numbers p. This is done by

$$\text{Ppow}(p, w) =_{\text{df.}} \Pr(p) \wedge w \neq \mathbf{0} \wedge (x \leq w)(y \leq w)((M(x,y,w) \wedge \Pr(x)) \supset x = p).$$

Ppow (p, w) says that p is w's only prime factor, and that $w \neq 0$; this only holds if $w = p^k$ for some k and p is prime. Note that if p is not prime, then this trick won't work.

Next, we want to define a formula Digp (m, n, p), which holds iff m is a digit in the base-p expansion of n and p is prime. How might we go about this? Let's use base 10 again for purposes of illustration. Suppose $n > 0$, and let d be any number < 10. If d is the first digit of n's decimal expansion, then $n = d \cdot 10^k + y$, for some k and some $y < 10^k$, and moreover $d \neq 0$. (For example, $4587 = 4 \cdot 10^3 + 587$.) Conversely, if $n = d \cdot 10^k + y$ for some k and some $y < 10^k$ and if $d \neq 0$, then d is the initial digit of the decimal expansion of n. If d is an intermediate or final digit in n's decimal expansion, then $n = x \cdot 10^{k+1} + d \cdot 10^k + y$ for some k, x and y with $y < 10^k$ and $x \neq 0$, and conversely. (This works for final digits because we can always take $y = 0$.) So if $d < 10$ and $n \neq 0$, then d is a digit of n iff d is either an initial digit or an intermediate or final digit, iff there exist x, k, and y with $y < 10^k$ and such that either $d \neq 0$ and $n = d \cdot 10^k + y$, or $x \neq 0$ and $n = x \cdot 10^{k+1} + d \cdot 10^k + y$. If $10 \leq d$ then d is not a digit of n's decimal expansion, and we allow 0 to occur in its own decimal expansion. The restrictions $d \neq 0$ and $x \neq 0$ are necessary, since otherwise 0 would occur in the decimal expansion of every number: $457 = 0 \cdot 10^3 + 457 = 0 \cdot 10^4 + 0 \cdot 10^3 + 457$; and if we want to code any finite sets that do *not* have 0 as an element, we must prevent this. Noting that none of this depends on the fact that the base 10 was used, and finding bounds for our quantifiers, we can define a formula Digp*(m, n, p) in Lim$^+$, which is true of m,n,p iff m is a digit in the base-p expansion of n and p is prime:

$$\begin{aligned}\text{Digp}^*(m, n, p) =_{\text{df.}} \{ & n \neq \mathbf{0} \wedge m < p \wedge \\ & [[m \neq \mathbf{0} \wedge (\exists w \leq n)(\exists z < w)(n = m \cdot w + z \wedge \text{Ppow}(p, w))] \vee \\ & (\exists w \leq n)(\exists z_1 \leq n)(\exists z_2 < w)(z_1 \neq \mathbf{0} \wedge n = z_1 \cdot w \cdot p + m \cdot w + z_2 \\ & \quad \wedge \text{Ppow}(p, w))]\} \\ \vee & \\ & (m = \mathbf{0} \wedge n = \mathbf{0} \wedge \Pr(p)).\end{aligned}$$

This formula mirrors the justification given above. However, much of it turns out to be redundant. Specifically, the less complicated formula

$$\begin{aligned}\text{Digp}(m, n, p) =_{\text{df.}} (& n \neq \mathbf{0} \wedge m < p \wedge \\ & (\exists w \leq n)(\exists z_1 \leq n)(\exists z_2 < w)(n = z_1 \cdot w \cdot p + m \cdot w + z_2 \wedge \\ & \quad \text{Ppow}(p, w))])\end{aligned}$$

$$\lor\ (m = \mathbf{0} \land n = \mathbf{0} \land \Pr(p))$$

is equivalent to Digp* (this remark is due to John Barker). To see this, suppose first that Digp*(m,n,p), $m<p$ and $n\neq0$. Then $n = z_1 \cdot p^{k+1} + m \cdot p^k + z_2$ for some k, z_1 and some $z_2 < p^k$. This includes initial digits (let $z_1 = 0$) and final digits (let $z_2 = 0$). So Digp (m, n, p) holds. Conversely, suppose Digp (m, n, p) holds, and assume that $m<p$ and $n\neq0$. Then $n = z_1 \cdot p^{k+1} + m \cdot p^k + z_2$ for some k, z_1 and some $z_2 < p^k$, and moreover $p^k \leq n$. If $z_1 > 0$, then m must be an intermediate or final digit of n, so suppose $z_1 = 0$. Then $m > 0$: for if $m = 0$, then $n = 0 \cdot p^{k+1} + 0 \cdot p^k + z_2 = z_2$, but $z_2 < p^k$ and $p^k \leq n$, and so $n < n$. So m must be the first digit of n.

We can now define

$$x \in y =_{\text{df.}} (\exists n \leq y)(\exists p \leq y)(y = [n, p] \land \text{Digp}\ (x, n, p)).$$

$x \in y$ is true of two numbers a,b if b codes a finite set S and a is a member of S. Note that Digp(m,n,p) and $x \in y$ are formulae of Lim^+. We could have carried out the construction in Lim, but it would have been more tedious, and would not have had any particular advantage for the purposes of this course.

There are two special cases we should check to make sure our coding scheme works: namely, we should make sure that the sets {0} and Ø have codes. If y is not in the range of our pairing function, then $x \in y$ will be false for all x; so y will code Ø. And since Digp(0, 0, p) holds for any p, [0, p] codes the set {0}.

Lecture IV

Let us now note a few bounding tricks that will be useful in the future. The function z = [x, y] is monotone in both variables: i.e. if $x \leq x_1$ and $y \leq y_1$ then $[x, y] \leq [x_1, y_1]$. Moreover, x, y ≤ [x, y]. Finally, if n codes a set S, and x ∈ S, then x ≤ n: if n codes S, then n is [k, p] for some k and p, so k ≤ n; and x is a digit in k's base-p expansion, so x ≤ k. So we can introduce some new bounded quantifiers into Lim^+:

$$(x \in y)\ B =_{df.} (x \leq y)\ (x \in y \supset B);$$
$$(\exists x \in y)\ B =_{df.} (\exists x \leq y)\ (x \in y \wedge B).$$

Note also that if n codes a set S and S' ⊆ S, then there is an m ≤ n which codes S'. (This is because, if the elements of S are the digits of the base-p expansion of k, then there is a number j ≤ k such that the digits in j's base-p expansion are the elements of S'; since j ≤ k, [j, p] ≤ [k, p] and [j, p] codes S'.) We can therefore define

$$x \subseteq y =_{df.} (z \in x)\ z \in y;$$
$$(x \subseteq y)\ B =_{df.} (x \leq y)\ (x \subseteq y \supset B);$$
$$(\exists x \subseteq y)\ B =_{df.} (\exists x \leq y)\ (x \subseteq y \wedge B).$$

Now that we can code finite sets of numbers, it is easy to code finite sequences. For a sequence $<m_1, ..., m_n>$ is simply a function φ with domain {1, ..., n} and with $\phi(i) = m_i$; we can identify functions with their graphs, which are relations, i.e. sets of ordered pairs, which we can in turn identify with sets of numbers, since we can code up ordered pairs as numbers. (So, for example, we can identify the sequence <7, 5, 10> with the set {[1, 7], [2, 5], [3, 10]}.) Finally, those sets can themselves be coded up as numbers. We define a formula Seql (s, n) of Lim^+ which holds just in case s codes a sequence of length n:

$$\begin{aligned}\text{Seql}\ (s, n) =_{df.}\ & (x \in s)(\exists m_1 \leq s)(\exists m_2 \leq s)(x = [m_1, m_2] \wedge m_1 \neq \mathbf{0} \wedge m_1 \leq n) \wedge \\ & (m_1 \leq s)(m_2 \leq s)(m_3 \leq s)(([m_1, m_2] \in s \wedge [m_1, m_3] \in s) \supset m_2 = m_3) \\ & \wedge (m_1 \leq n)(\exists m_2 \leq s)(m_1 \neq \mathbf{0} \supset [m_1, m_2] \in s).\end{aligned}$$

The first conjunct simply says that every element of s is a pair whose first member is a positive integer ≤ n; the second says that s is single valued, i.e. is (the graph of) a function; and the third says that every positive integer ≤ n is in s's domain.

We can also define a formula Seq (s), which says that s codes a finite sequence of some length or other:

$$\mathrm{Seq}(s) =_{\mathrm{df.}} (\exists n \leq s)\ \mathrm{Seql}\ (s, n).$$

We can bound the initial quantifier, because if s codes a sequence of length n, then $[n, x] \in s$ for some x, and so $n \leq [n, x] \leq s$. Also, if x is the ith element of some sequence s, then $x \leq [i, x] \leq s$; we can use this fact to find bounds for quantifiers.

The following formula holds of two numbers if the second codes a sequence and the first occurs in that sequence:

$$x \text{ on } s =_{\mathrm{df.}} \mathrm{Seq}(s) \wedge (\exists y \leq s)\ ([y,x] \in s).$$

Gödel Numbering

We can use our method of coding up finite sequences of numbers to code up finite strings of symbols. As long as we have a countable alphabet, we will be able to find a 1-1 correspondence between our primitive symbols and the natural numbers; we can thus code up our primitive symbols as numbers. We can then identify strings of symbols with sequences of numbers, which we then identify with individual numbers. A scheme for coding strings of symbols numerically is called a *Gödel numbering*, and a numerical code for a symbol or expression is called a *Gödel number* for it.

Exactly how we do this is arbitrary. One way of doing it is this: if $S = s_1 \ldots s_n$ is a string of symbols, and $a_1, \ldots, a_n$ are the numerical codes for $s_1, \ldots, s_n$, then $<a_1, \ldots, a_n>$ is a sequence of numbers, and it therefore has a code number p; we can take p to be a Gödel number of S. (Note that, on our way of coding finite sequences, each sequence will have many different code numbers, so we must say "*a* Gödel number" rather than "*the* Gödel number.") Call this the *simple-minded* coding scheme.

We shall adopt a slightly more complicated coding scheme, which will make things easier later on. First, we code the terms of the language via the simple-minded scheme. Then, when coding formulae, we again use as a code for a string of symbols a code for the corresponding sequence of codes of symbols, except that now we treat terms as single symbols. So if a, b, c, d are the codes of the primitive symbols P^1_1, f^2_1, x_1, x_2, then any code p for <b, c, d> is a code for the term $f^2_1 x_1 x_2$, and any code for <a, p> codes $P^1_1 f^2_1 x_1 x_2$.

We want a single coding scheme for all the languages we shall consider, namely, the various first-order languages and the languages RE and Lim (and its variants). So we shall need to take all of the symbols (,), $\supset$, ~, $\wedge$, $\vee$, <, and $\exists$ as primitive, and provide code numbers for all of them. We also need code numbers for the constants, variables, predicates, and function letters. Our general scheme for doing this is to code a symbol s by a pair [x, y], where x represents s's grammatical category, and y represents additional information about s (e.g. its sub- and superscript). For definiteness, we make the following our official Gödel numbering:

Individual symbols:	(	)	∃	<	⊃	~	∧	∨
	[0, 0]	[0, 1]	[0, 2]	[0, 3]	[0, 4]	[0, 5]	[0, 6]	[0, 7]

Variables: [1, i] codes x_i

Constants: [2, i] codes a_i

(Special constants,
or “choice” constants: [3, i] codes b_i)

Function letters: [4, [n, i]] codes f_i^n

Predicate letters: [5, [n, i]] codes P_i^n

(We do not have special constants in the languages we have developed so far; but in case we need them, we have codes for them.) Note that this coding scheme is open-ended; we could add extra individual symbols, or even extra grammatical categories (e.g. new styles of variables), without disruption.

Identification

Strictly speaking, when we use an entity A to code an entity B, A and B are (in general) different entities. However, we often speak as though they were the same; for example, we say that the number 105 = [5, [1, 1]] *is* the symbol P_1^1, whereas strictly speaking we should say that it *codes* P_1^1. (Similarly, we will say, for example, that a certain predicate is true of exactly the formulae, or of exactly the terms, where we should say that it is true of the codes of formulae, or of the codes of terms). This has the problem that, since we have many different codes for a single expression, many different numbers are identified with the same expression. In order to avoid this talk of identification, we might modify our coding scheme so as to make the coding correspondence one-to-one, for example taking the least number among the codes to be the real code.

According to Geach's doctrine of relative identity, this talk of identification would be not only harmless, but absolutely legitimate. For Geach, it does not make sense to say simply that two objects are the same, this being only a disguised way of saying that they are the same F, for some property F. In this sense there is no such thing as absolute identity, according to Geach. His doctrine of relative identity would then allow us to say that although two objects are different numbers, they are the same formula. The author does not share Geach's views on this point, but it is useful to think in terms of relative identity in our context. Geach has applied his doctrine in other contexts.

The Generated Sets Theorem.

We shall now use our coding of finite sequences to show that some intuitively computable functions which are not obviously recursive are in fact recursive. Let's start with the factorial function y = x!. Note that 0! = 1 and (n+1)! = (n+1)·n! for all n, and that this is an inductive definition that specifies the function uniquely. The sequence <0!, ..., n!> is therefore the unique sequence $<x_1, ..., x_{n+1}>$ such that $x_1 = 0$ and for all $k \leq n$, $x_{k+1} = (k+1)\cdot x_k$. Thus, y = x! just in case y is the x+1st member of some such sequence. So the following formula of RE defines the graph of the factorial function:

$$(\exists s)(\mathrm{Seql}\ (s, x') \wedge [\mathbf{0'}, [\mathbf{0}, \mathbf{0'}]] \in s \wedge (z \leq s)(i \leq x')([i'', [i', z]] \in s \supset (\exists z_1 \leq s)\ ([i',[i,z_1]] \in s \wedge z = z_1\cdot i')) \wedge [x', [x, y]] \in s).$$

(Note that we could have written $\mathbf{0'} \in s$ instead of $[\mathbf{0'}, [\mathbf{0}, \mathbf{0'}]] \in s$, since $[1, [0, 1]] = (1+((0+1)^2+0))^2 + 1 = 5$. Note also that, while ⊃ is not definable in RE, its use in this formula is permissible, since its antecedent, $[i'', [i', z]] \in s$, expresses a relation whose complement is r.e. Also, the part of the formula following the initial unbounded quantifier (∃s) is a formula of Lim^+ (in which ⊃ is definable), and is therefore equivalent to a formula of RE, and so the entire formula is a formula of RE.)

The above definition of y = x! is an example of a definition by primitive recursion; we have a *base clause*

$$0! = 1$$

in which the function's value at zero is specified, and an *induction clause*

$$(n+1)! = (n+1)(n!)$$

in which the value at n+1 is specified in terms of its value at n. Another example of this kind of definition is that of the exponentiation function $z = x^y$: we stipulate that $x^0 = 1$ and $x^{y+1} = x^y\cdot x$. Here, the induction is carried out on the variable y; however the value of the function also depends on x, which is kept fixed while y varies. x is called a *parameter*; the primitive recursive definition of exponentiation is called a primitive recursive definition *with parameters*, and that of the factorial function is said to be *parameter free*. We can show that the exponentiation function is recursive, using a similar argument to the above.

In general, if h is an n-1-place function and g is an n+1-place function, then the n-place function f is said to come from g and h by *primitive recursion* if f is the unique function such that

$$f(0, x_2, ..., x_n) = h(x_2, ... x_n)$$

and

$$f(x_1+1, x_2, ..., x_n) = g(x_2, ..., x_n, x_1, f(x_1, x_2, ..., x_n))$$

for all $x_1, ..., x_n$. (Here we take 0-place functions to be constants, i.e. when n = 1, we let h be a number and let f(0) = h.) We define the class of primitive recursive functions inductively, as follows. (i) The basic primitive recursive functions are the zero function z(x) = 0, the successor function s(x) = x+1, and the identity functions $id^n_i(x_1, ..., x_n) = x_i$ (where $i \leq n$). (ii) The composition of primitive recursive functions is primitive recursive (that is, if $\psi(m_1,...,m_k)$ is a primitive recursive function in k variables, and $\phi_1(q_{1,1},...,q_{1,n_1}),..., \phi_k(q_{k,1},...,q_{k,n_k})$ are k primitive recursive functions in $n_1,...,n_k$ variables, respectively, then so is the function in $n_1+...+n_k$ variables $\psi(\phi_1(q_{1,1},...,q_{1,n_1}),..., \phi_k(q_{k,1},...,q_{k,n_k}))$). (iii) A function that comes from primitive recursive functions by primitive recursion is primitive recursive. (iv) And the primitive recursive functions are only those things required to be so by the preceding. Using the same sort of argument given in the case of the exponentiation function, we can show that all primitive recursive functions are recursive. (That the recursive functions are closed under primitive recursion is called the *primitive recursion theorem*.)

The converse, however, does not hold. Consider the sequence of functions

$\psi_1(x, y) = x + y$
$\psi_2(x, y) = x \cdot y$
$\psi_3(x, y) = x^y$

This sequence can be extended in a natural way. Just as multiplication is iterated addition and exponentiation is iterated multiplication, we can iterate exponentiation: let $\psi_4(x, 0) = x$, $\psi_4(x, 1) = x^x$, $\psi_4(x, 2) = x^{x^x}$, etc. This function is called *superexponentiation*. We can also iterate superexponentiation, giving us a super-superexponentiation function, and so on. In general, for n > 2, we define

$\psi_{n+1}(x, 0) = x$
$\psi_{n+1}(x, y+1) = \psi_n(x, \psi_{n+1}(x, y))$

We can turn this sequence of 2-place functions into a single 3-place function by letting $\chi(n, x, y) = \psi_n(x, y)$; χ is called the *Ackermann function.* Ackermann showed that this function is not primitive recursive, though it is clearly computable. (This is the function that we referred to earlier.) In fact, it can be shown that for any 1-place primitive recursive function ϕ, $\phi(x) < \chi(x, x, x)$ for all but finitely many x.

We shall next prove a theorem from which it follows that a wide range of functions, including both the primitive recursive functions and the Ackermann function, are recursive. This theorem will also be useful in showing that various interesting sets and relations are r.e. The theorem will further provide a way of making rigorous the extremal clauses in our earlier inductive definitions of term and formula of the different languages that we have introduced.

The basic idea that motivates the theorem is best illustrated by means of a definition formally similar to those of formula or term, that of a *theorem* of a *formal system.* In a formal system, certain strings of formulae are called axioms, and from them the theorems of the formal system are generated by means of certain rules of inference (for example, *modus ponens*, according to which if formulae A and $(A \supset B)$ are theorems, then B is a theorem). The notion of a theorem is defined inductively, specifying that all the axioms are theorems (basis clauses), that if a formula A follows from theorems $B_1, ..., B_n$ by one of the inference rules, then A is also a theorem (closure conditions, or generating clauses), and that the theorems are only those things generated in this way (extremal clause).

In a formal system a formula is a theorem if it has a proof. And a proof is a finite sequence of formulae each of which is either an axiom or a formula which comes from previous formulae in the sequence via one of the generating clauses (the inference rules). Sequences which are proofs are called proof sequences. We can generalize the notion of a proof sequence so as to apply it to the case of terms or formulae. Something is a formula if it occurs on a sequence each element of which is either an atomic formula or comes from previous formulae in the sequence via one of the generating clauses (the rules for the formation of complex formulae out of simpler ones). One such sequence can be seen as a proof that a string of symbols is a formula, which justifies using the phrase 'proof sequence' in this case as well. (Similar remarks could be made about the notion of a term).

Generalizing this, we introduce the following

Definition: A *proof sequence* for a set B, and relations $R_1, ..., R_k$ (n_1+1-place,..., n_k+1-place, respectively) is a finite sequence $<x_1, ..., x_p>$ such that, for all i = 1, ..., p, either $x_i \in B$ or there exist $j \leq k$ and $m_1, ..., m_{n_j} < i$ such that $R_j(x_{m_1}, ..., x_{m_{n_j}}, x_i)$.

Our extremal clauses will be understood as formulated with the help of the notion of a proof sequence determined by the appropriate sets and relations. And our proofs by induction on the complexity of terms or formulae would proceed rigorously speaking by induction on the length of the appropriate proof sequences.

If we have a set B and some relations $R_1, ..., R_k$, where each R_i is an n_i+1-place relation, the set *generated* by B and $R_1, ..., R_k$ is the set of those objects which occur in some proof sequence for B and $R_1, ..., R_k$. If S is the set generated by B and $R_1, ..., R_k$, we call B the *basis set* for S and $R_1, ..., R_k$ the *generating relations* for S.

Generated Sets Theorem: If B is an r.e. set and $R_1, ..., R_k$ are r.e. relations, then the set generated by B and $R_1, ..., R_k$ is itself r.e.
Proof. Let C be a formula of RE that defines the set B, and let $F_1, ..., F_k$ be formulae of RE that define $R_1, ..., R_k$. We first define

$$\begin{aligned}\mathrm{PfSeq}(s) =_{\mathrm{df.}} &\ \mathrm{Seq}(s) \land (m\leq s)(x<s)([m,x]\in s\supset C(x) \lor \\ &\quad (\text{clause } 1) \lor ... \lor (\text{clause } k)),\end{aligned}$$

where (clause j) is the formula

$$(\exists x_1 \leq s)...(\exists x_{n_j} \leq s)(\exists i_1 < i)...(\exists i_{n_j} < i)([i_1, x_1] \in s \land ... \land [i_{n_j}, x_{n_j}] \in s \land F_j(x_1, ..., x_{n_j}, y_1).$$

PfSeq(s) thus defines the set {s: s codes a proof sequence for B and $R_1, ..., R_k$}. We can therefore define the set G generated by B and $R_1, ..., R_k$ by means of the formula of RE

$$(\exists s)(\mathrm{PfSeq}(s) \land (\exists m \leq s)([m, x] \in s).$$

This completes the proof.

The generated sets theorem applies in the first instance to sets of numbers; but it also applies derivatively to things that can be coded up as sets of numbers, e.g. sets of formulae. Suppose some set G of formulae is the set generated by a basis set B of formulae and generating rules $R_1, ..., R_k$ among formulae. To show that the set G' of Gödel numbers of elements of G is r.e., simply show that the set B' of Gödel numbers of elements of B is r.e. and that the relations R_i' among Gödel numbers for formulae related by the relations R_i are r.e. (Of course, whether G' is in fact r.e. will depend on what the relations B and $R_1, ..., R_k$ are.) In this way, it is easy to show that the set of formulae of RE is itself r.e.

The Generated Sets Theorem is known to all logicians, although it is rarely stated explicitly. It provides a simpler method of proving that some sets or relations are r.e. (and hence that some total functions are recursive) than primitive recursion. Of course, it does not provide a general method of proving recursiveness, but it is infrequent in mathematical arguments to have the need to show that a set or relation is recursive besides being recursively enumerable. It is usually emphasized as a basic requirement of logic that the set of formulae of a given language must be decidable, but it is not clear what the theoretical importance of such a requirement is. Chomsky’s approach to natural language, for example, does not presuppose such a requirement. In Chomsky's view, a grammar for a language is specified by some set of rules for generating the grammatically correct sentences of a language, rather than by a decision procedure for grammatical correctness.

However, we will eventually state a theorem an application of which will be to show that the set of codes of formulae or terms of a language is recursive.

We can use the generated sets theorem to show that a function is recursive. For example, the function $y = x!$ is recursive iff the set $\{[x, x!] : x \in \mathbf{N}\}$ is r.e., and this set can be generated as follows: let the basis set be $\{[0, 1]\}$, and let the generating relation be $\{<[x, y], [x+1, y\cdot(x+1)]>: x, y \in \mathbf{N}\}$. It is easy to see that the basis set and generating relation are r.e (and indeed recursive), and that they generate the desired set. In fact, the result that all primitive recursive functions are recursive follows directly from the generated sets theorem in this way. Moreover, the generated sets theorem can be used to show that the Ackermann function is recursive. This is the virtue of the generated sets theorem: it is more powerful than the theorem about primitive recursiveness, and indeed it is easier to prove that theorem via the generated sets theorem than directly.

We may sometimes want to know that a set G is recursive, or even limited, in addition to being r.e. While the generated sets theorem only shows that G is r.e., in particular cases we can sometimes sharpen the result. For one thing, if the basis set and generating relations are recursive (or limited), then the formula PfSeq(s) defines a recursive (limited) relation. This does not itself show that G is recursive (limited), since the formula used to define G in the proof of the Generated Sets Theorem begins with the unbounded quantifier $(\exists s)$. If we can find some way of bounding this quantifier, then we can show that G is recursive (or limited). However, it is not always possible to bound this quantifier, for not all sets generated from a recursive basis set via recursive generating relations are recursive. For example, the set of Gödel numbers of valid sentences of the first-order language of arithmetic is r.e., but not recursive; and yet that set is clearly generated from a recursive basis set (the axioms) and recursive generating relations (the inference rules).

Exercises

1. a) Prove that every k-place constant function is recursive. Prove that the successor function is recursive.

b) Prove that if a function $\phi(m_1,...,m_k)$ in k variables is recursive (partial recursive), so is any k-1 place function obtained from ϕ by identifying two variables.

2. a) Prove that the composition of two 1-place total (partial) recursive functions is total (partial) recursive.

b) More generally, prove that if $\psi(m_1,...,m_k)$ is a total (partial) recursive function in k variables, and $\phi_1(q_{1,1},...,q_{1,n_1}),..., \phi_k(q_{k,1},...,q_{k,n_k})$ are k total (partial) recursive functions in $n_1,...,n_k$ variables, respectively, then so is the function in $n_1+...+n_k$ variables

$\psi(\phi_1(q_{1,1},...,q_{1,n_1}),..., \phi_k(q_{k,1},...,q_{k,n_k}))$.

3. Show that if ϕ is a recursive pairing function whose range is recursive, then a binary relation R is recursive iff the set $\{\phi(m,n): R(m,n)\}$ is recursive. Prove that a sufficient condition for the range of a recursive pairing function ϕ to be recursive is that $m,n\leq\phi(m,n)$. (This condition is satisfied by the pairing function we have been using and by nearly all the pairing functions used in practice). Where does the argument go wrong if we do not assume that the range is recursive? (a counterexample will be given later.)

4. For arbitrary $n > 1$, define an n-tupling function, verifying that it is indeed an n-tupling function. Generalize exercise 3 to arbitrary n-place relations accordingly.