II

The Logic Theory Machine

In the language we have constructed, we have *variables* (atomic sentences): p, q, r, A, B, C, ...: and *connectives*: - (not), v (or), → (implies). The connectives are used to combine the variables into *expressions* (molecular sentences). We have already considered one example of an expression:

1.7 $$-p \,.\!\rightarrow\!.\, q \vee -p$$

The task set for LT will be to prove that certain expressions are *theorems*—that is, that they can be derived by application of specified rules of inference from a set of primitive sentences or *axioms*.

The two connectives, - and v are taken as primitives. The third connective, → , is defined in terms of the other two, thus:

1.01 $$p \rightarrow q =_{def} -p \vee q$$

The five axioms that are postulated to be true are:

1.2 $$p \vee p \,.\!\rightarrow\!.\, p$$

1.3 $$p \,.\!\rightarrow\!.\, q \vee p$$

1.4 $$p \vee q \,.\!\rightarrow\!.\, q \vee p$$

1.5 $$p \,.\!\vee\!.\, q \vee r :\rightarrow: q \,.\!\vee\!.\, p \vee r$$

1.6 $$p \rightarrow q \,.\!\rightarrow\!: r \vee p \,.\!\rightarrow\!.\, r \vee q$$

Each of these acioms is stored as a list in the theorem, I, with all its variables marked *free* (F) in their respective elements.

From the axioms other true expressions can be derived as theorems. In the system of *Principia Mathematica*, there are two *rules of inference* by means of which new theorems can be derived from true expressions (theorems and axioms). These are:

Rule of Substitution: If $A(p)$ is any true expression containing the variable *p*, and *B* any expression, then $A(B)$ is also a true expression. increases length of an expression.

Rule of Detachment: If *A* is any true expression, and the expression $A \rightarrow B$ is also true, then *B* is a true expression. Decreases length of an expression

To these two rules of inference is added the *rule of replacement*, which states that an expression may be replaced by its definition. In the present context, the only definition is 1.01, hence the rule of replacement permits any occurrence of $(-p \vee q)$ in an expression to be replaced with $(p \rightarrow q)$, and any occurrence of $(p \rightarrow q)$ to be replaced with $(-p \vee q)$.[9] definitions are unnec.

In this system, then, a *proof* is a sequence of expressions, the first of which are accepted as axioms or as theorems, and each of the remainder of which is obtained from one or two of the preceding by the operations of substitution, detachment, or replacement.

[9] As we shall see, 1.01 is not held in storage memory, but is represented, instead, by two routines for actually performing the replacements.

Example; prove 2.01: $p \rightarrow -p \ .\!\rightarrow\!. \ -p$

2.01 Proof:[10] (1) $!\ p \vee p \ .\!\rightarrow\!. \ p$ (axiom 1.2)

(2) $!\ -p \vee p \ .\!\rightarrow\!. -p$ (from (1) by substitution of -p for p)

(3) $!\ p \rightarrow -p \ .\!\rightarrow\!. -p$ (from (2) by replacement on left)

The problem now is to specify a program for LT such that, when a problem is proposed in the form of a theorem to be proved (like 2.01 above), a proof will be discovered and constructed. First, it should be observed that there is a systematic algorithm for constructing such a proof, should one exist. Starting with the five axioms, we construct all the theorems that can be obtained from them by a single application of the rules of substitution, detachment, and replacement.[11] We thus obtain the set of all theorems that can be obtained from the axioms by proofs not more than one step in length. Repeating this process with the enlarged set of theorems, we obtain the set of all theorems that can be derived from the axioms by proofs not more than two steps in length. Continuing, we finally obtain the set of theorems that can be derived by proofs not more than n steps in length.

Now if the theorem in which we are interested possesses a proof k steps in length, we can, in principle, discover it by constructing all valid proof chains of length not more than k, and selecting any one of these that terminates in the theorem in question. This "in principle"

[10] The exclamation point in front of an expression indicates that the expression in question is asserted to be true. To designate an expression whose truth has not been demonstrated, we will use a question mark preceding the expression.

[11] A technical difficulty arises from the fact that there is an infinite number of valid substitutions. This difficulty can be removed rather easily, but the question is irrelevant for the purposes of this paper.

possibility may in fact be computationally infeasible because of the very large number of valid chains of length k that can be constructed, even when k is a number of moderate size. Under these circumstances, the rules of inference do not give us sufficient guidance to permit us to construct the proof we are seeking; and we need additional help from some system of heuristic.

The problem will be solved if we can devise a program for constructing chains of theorems, not at random but in response to cues in such a way as to make it probable that the desired proof will be discovered within a reasonable computing time. For example, suppose the rules of inference were such as to permit any given proof chain to be continued, on the average, in ten different ways. Then there would be ten thousand proof chains four steps in length (10^4). The expected number of proof chains that would have to be examined to find any particular one of these by random search is five thousand. Suppose, however, that LT responded to cues that permitted eight of the ten continuations at each step to be eliminated from consideration. Then the number of proof chains four steps in length that would have to be examined in full would be only sixteen (2^4), and the expected number that would have to be examined, only eight.

The Program of LT

We wish now to describe explicitly the program of LT. The program is given in full in Section III; hence, in thetext we shall refer frequently to Section III for detail. We shall refer to each routine by its name (e.g., LMc for the matching routine), but we shall need some additional notation to refer to the main segments of routines that do not themselves have names. The names of these segments are given in Section III in the column marked "Seg." In each of these segments there is generally one main operation to be performed; and this main operation, or sub-routine , is usually surrounded by a number of procedural and control operations that fit it into the larger routine. In ordinary language, we would say that the "function" of the segment is to perform the main operation that is contained in it. For example, the main operation in the third segment of LMc is LSby, a substitution-- the function of this segment in the matching program is to substitute one sub-expression for another in one of the expressions being matched. Hence it will sometimes be convenient to indicate the main operation in this segment by naming the segment LMc (Sby). Similar designation will be used for the other segments of routines. This notation, while not exact, emphasizes the fact that each routine consists in a sequence (or branching tree) of main operations that are connected by procedural and test operations. Thus, an abbreviated description of the matching routine might be given as:

<u>LMc</u>

T	Perform diagnostic tests
LMc	Recursion of matching with next elements in logic expression
Sby	Substitute the element $\underline{y}$ for the element $\underline{x}$
Sbx	Substitute the element $\underline{x}$ for the element $\underline{y}$
CN	Compare variablesin $\underline{x}$ only
Rp	Replace connectives, if required and possible

The Substitution Method

Let us take as our first example the very simple expression, 2.01, for which we have already given a proof. We suppose that when the problem is proposed, LT has in its theorem memory only the axioms, 1.2 to 1.6. We wish now to construct a proof (the one given above, or any other valid proof) for 2.01.

As the simplest possibility, let us consider proofs that involve only the rules of substitution and replacement. We may state the problem thus: how can we search for a proof of the theorem by substitution without considering all the valid substitutions in the five axioms? We will use two devices to focus the search. Both of these involve "working backward" from the theorem we wish to prove--for by taking account of the characteristics of that theorem, we can obtain cues as to the most promising lines to follow:

1. In attempting substitutions, we will limit ourselves to axioms (or other true theorems, if any have already been proved) that are in some sense "similar" in structure to the theorem to be proved. The routine that accomplishes this will be called the test similarity routine, CSm.

2. In selecting the particular substitutions to be made in a theorem that has been chosen for trial, we will attempt to match the variables in that theorem to the variables in the expression to be proved. Similarly, we will try to use the rule of replacement to match connectives. The routine in which these various operations occur is called the matching routine, LMc.

Using these devices, the proposed routine for proving theorems--the *method of substitution*, MSb--works as follows: (1) MSb(Sm). Search for an axiom or theorem that is similar to the expression to be proved. (2) MSb(Mc). When one is found, try to match it with the expression to be proved. If a match is successful, the expression is proved; if the list of axioms and theorems is exhausted without producing a match, the method has failed. (Reference to Section III will show that there is another segment of MSb--MSb(NAW)--that we have not mentioned. The function of this segment will be discussed later in connection with the executive routine, Ex.)

To see in detail how the method operates, we next examine, in turn, the main operations, CSm and LMc, of the two segments of the substitution method. For concreteness, we will carry out these operations explicitly for the proof of the expression 2.01.

2.01 ? p→ -p .→. -p

Test for Similarity, CSm. We must state what we mean by similarity. We start from a common-sense viewpoint and regard two propositions as similar if they "look" similar to the eye of a logician. But in Section I we have already defined three characteristics of an expression that can be used as criteria of similarity. These are: *K*, the number of *levels* in the expression; *J*, the number of distinct variables in the expression; and *H*, the number of variables in the expression.[12]

[12] The assertion is that two expressions having the same description "look alike" in some undefined sense; and hence if we are seeking to prove one of them as a theorem, while the other is an axiom or theorem

(cont'd - see next page)

Applying these definitions to 2.01 (routines NK, NJ, and NH, respectively), we find that K = 3, J = 1, and H = 3. That is, 2.01 has three levels, one distinct variable (*p*), and three variable places. We may write this:

$$D(2.01) = (3,1,3)$$

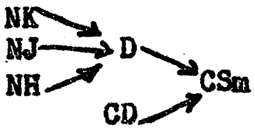
In the same way, we can write descriptions for the various sub-expressions contained in 2.01--in particular, the sub-expressions to the left and to the right of the main connective, respectively. We have for these:

$$DL(2.01) = (2,1,2); \quad DR(2.01) = (1,1,1)$$

Now, we say that two expressions, *x* and *y*, are similar if they have identical left and right descriptions, i.e., if DL(x) = DL(y) and DR(x) = DR(y). The routine for determining whether two theorems are similar, CSm, consists of two segments: (1) CSm(D), a description segment, and (2) CSm(CD), a comparison of descriptions. The description segment is made up of four description routines, D--one each to compute DL(x), DR(x), DL(y), and DR(y). The comparison segment is made up of two *compare description* routines, CD, one of which compares DL(x) with DL(y), the other DR(x) with DR(y).

A diagram of the hierarchy of principal sub-routines in testing similarity will look like this:

12 cont'd already proved, then the latter is likely construction material for the proof of the former. Empirically, it turns out that with the particular definition of similarity introduced here, in proving the theorems of Chapter 2 of *Principia Mathematica* about one theorem in five that is stored in the theorem memory turns out to be similar to the expression we are seeking to prove. It is easy to suggest a number of alternative, and quite different criteria that would be equally symptomatic of "similarity." One of these alternatives will be discussed in Section III. Uniqueness is of no account here; all we are concerned with is that we have *some* criteria that "work"--that select theorems suitable for matching.



In the case of 2.01, the segment MSb(Sm) will search the list of axioms and theorems and will find that axiom 1.2 is similar to 2.01:

1.2 : p v p .→. p

for it, too, has the descriptions: DL(1.2) = (2,1,2); DR(1.2) = (1,1,1). Moreover, 1.2 is the only axiom that has this description.

<u>Matching Expressions, LMc</u>. Next we carry out a point-by-point comparison between 2.01, the expression to be proved, and 1.2, the axiom that is similar to it. We start with the main connectives, and work systematically down the tree of the logic expressions—always as far as possible to the left. In the present case, the order in which we will match is: main connective (P = none), connective of left sub-expression (P=L), left variable of sub-expression, (P=LL), right variable of sub-expression (P=LR), right sub-expression (P=R).

The matching routine is fairly complicated, consisting of six segments, but not all segments are employed each time two elements are matched. The first segment, LMc(T), and the initial operations of most of the other segments, consists of tests that determine whether the two elements to be matched are already identical, or whether they can be made identical by substitution (if one is a free variable) or by replacement (if both are connectives), or—finally—whether matching is impossible. In Section III this network of decisions is laid out in graphic form. The second segment, LMc(LMc), is a recursion of the matching routine with each of the next lower pair of elements in the tree of the expression. This recursion segment operates only if the elements to be matched in LMc are identical connectives (or have been made so).

The third and fourth segments, LMc(Sby) and LMc(Sbx), apply the rule of substitution when the tests have shown this to be appropriate. LMc(Sby), which is executed whenever E(x) is a free variable,[13] simply substitutes the expression X(y) for E(x). LMc(Sbx), which is executed whenever E(y) is a free variable, substitutes the expression X(x) for E(y). In both cases, of course, substitution must take place throughout the whole expression in which the free variable occurs. This is taken care of automatically by the process LSb. Also, since LMc matches X(x) to X(y), LMc(Sby) has priority over LMc(Sbx), as a careful examination of the decision network will reveal.

The fifth segment, LMc(CN), reports the successful termination of the matching program if E(x) and E(y) are identical variables, its failure if they cannot be ma e identical by substitution.

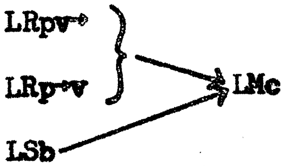
The sixth segment, LMc(Rp), operates when E(x) and E(y) have different connectives. The segment replaces the connective in x by the connective in y whenever this replacement is legitimate, and then

[13] Essentially, a variable is free when no substitution has yet been made for it. After any substitution it is bound and no longer available for subsequent substitutions. As previously noted, all variables in expressions stored in the theorem memory are free.

returns control to the recursion segment.

By virtue of the recursion segment, the matching routine will attempt to match each pair of elements; if successful, will proceed to the next pair; if unsuccessful, will report failure. Hence, the routine will continue until it makes the theorem that is being matched identical with the expression to be proved, or until the matching fails.

The hierarchy of principal routines looks like this:



Returning to our specific example of two similar expressions, 1.2 and 2.01, we carry out the matching routine as follows:

2.01 ? p→ -p .→. -p

1.2 ! A v A .→. A

(We use A instead of p in 1.2 to indicate that the variable is free (F).)

a. The main connectives agree: both are→ .

b. Proceeding downward to the left, the connective is → in 2.01, but v in 1.2. To change the v to →, we must have (because of the definition, 1.01), a - before the left-hand A in 1.2. This we can obtain by making the substitution of -B for A in 1.2. Having carried out this substitution, and having then replaced (-B v -B) with (B→ -B), we have the following situation:

2.01 ? p → -p .→. -p

1.2' ! B → -B .→. -B

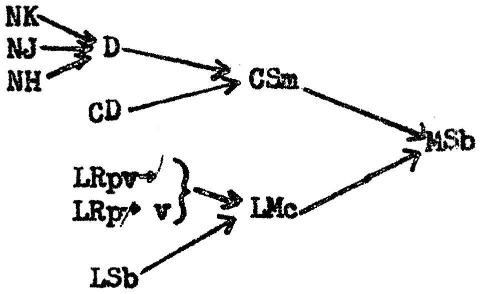
c. Proceeding again to the left, we find B in 1.2', but p in 2.01. We therefore substitute p for B in 1.2', and now find (after recursion through the remaining two elements) that we have a complete match:

2.01 ? p→ −p .→. −p

1.2" ! p→ −p .→. −p

Thus, we have discovered a proof of 2.01 (in fact, precisely the proof we gave before), which consists in substituting the variable −p for the variable in 1.2, and replacing the connective v in 1.2 with →. The reader who wishes to compare these operations in detail with the matching routine will observe that (in addition to the test sequence) in step (a) the sequence LMc(CN) was involved; in (b), the sequence LMc(Rp), in (c), the sequence LMc(Sby). All three steps were followed by the recursion sequence, LMc(LMc).

This completes our outline of the method of substitution as a routine for discovering proofs in symbolic logic. The method may be viewed as an information process that is composed of a considerable number of more elementary information processes arranged to operate in highly conditional sequences. Each of the main components--the test for similarity routine, and the matching routine--is made up, in turn, of sub-routines. The test conditions that control the branchings of the sequences depend in a number of instances upon the outcomes of searches through the theorem memory. Hence, the method of substitution represents a complex information process in the sense in which we have defined the term. Combining the two diagrams depicted above, we can illustrate the hierarchy of the main operations that enter into the substitution method:



The method is a heuristic one, for it employs cues, based on the characteristics of the theorem to be proved, to limit the range of its search; it does not systematically enumerate all proofs. This use of cues represents a great saving in search, but carries the penalty that a proof may not in fact be found. The test of a heuristic is empirical: does it work?

Moreover, the cues that are used in the method are not without cost. For example, in order to limit matching attempts to "similar" theorems, theorems must be described and compared. The net saving in computing time, as compared with random search, is measured by the reduction in the number of theorems that have to be matched less the cost of carrying out the search and compa e for similarity routines. Stated otherwise, cues are economical only if it is cheaper to obtain them than to obtain directly the information for which they serve as cues.

To be sure, we have found a proof for one proposition in Principia; but how general is the substitution method? On examination of the 67 propositions in Chapter 2 of Principia, it appears that some 22 can be proved by the method of substitution, including for example: 2.01, 2.02, 2.03, 2.04, 2.05, 2.07, 2.10, 2.12, 2.21, 2.26, 2.27. The remaining propositions evidently require more powerful techniques of discovery and proof. It is evident, for instance, that we must employ the rule of detachment.

The Method of Detachment

We will describe next the *method of detachment*, *MDt*, which, as its name implies, incorporates the rule of detachment. The method, of course, is not synonymous with the rule, but includes also heuristic devices that select particular theorems to which the rule is to be applied.

Let us review the principle of logic that underlies the method. Suppose that LT must prove that expression A is a theorem; and assume that there are already in the theorem memory two theorems, B and $B \rightarrow A$. Then, by application of the rule of detachment to B and $B \rightarrow A$, A is derivable immediately.

We can generalize this procedure by combining matching (substitution) and replacement) with detachment. Assume that the theorem memory contains B'' and $B' \rightarrow A'$; that A is obtainable from A' by substitution (and replacement); and that B is obtainable from B'' by substitution (and rereplacement). Then we can construct a proof of A as follows: (1) By substitution in B'', B' is a theorem. (2) Since $B' \rightarrow A'$ is also a theorem, it follows by detachment that A' is a theorem. (3) By substitution in A', A is a theorem.

This settles the problem of constructing a *valid* proof by the method of detachment. From the standpoint of the *discovery* of a proof employing this method, the trick lies again in narrowing down the search for $B' \rightarrow A'$ and B'', so that these do not have to be sought through a very large scale trial-and-error search and substitution program.

Structure of the Detachment Method. The basic structure of the detachment method is quite similar to that of the substitution method, for both methods utilize the same basic operations. The first two segments of the detachment method, MDt(SmV) and MDt(SmCt), carry out searches for similar expressions, in a way that will be indicated more precisely below. The next segment, MDt(Mc), carries out a matching of any expression so found with the theorem to be proved. If the matching is successful, a new problem is created by the segment MDt(F). This problem is then attacked, in the final segment, MDt(MSb), by the method of substitution.

Again, designate by *A* the expression to be proved. In MDt(SmV) we search the theorem memory for theorems whose *right sides* are similar (by the test CSm, described previously) to the *whole* expression *A*. If we find such a theorem (call it *T*), we go to segment MDt(Mc), and apply the matching operation to the right side of T and to A. If we are successful in the matching, we find the left side of T, MDt(P); and seek to prove by the method of substitution that it is a theorem, MDt(MSb). For if the left side of T is a theorem and T is a theorem, then by detachment, the right side of T is a theorem. But A can be obtained from the right side of T by substitution, hence is a theorem. (Note that a check is made to see that T has → for a connective.)

Contraction. If the detachment method fails to find a proof in the manner just described, a new attempt is made by means of the second segment, MDt(SmCt), which conducts a new search for theorems whose right sides are similar to A, but employing a different criterion of similarity from the one we have used thus far. If such a theorem is found, the method proceeds with the matching segment exactly as before.

To see what is involved in this generalized notion of similarity, let us consider two expressions, A and A', with different descriptions. If A has more levels and variable places than A', it is still possible that A is derivable from A' by substitution--specifically, by substituting appropriate molecular expressions for the variables of A. For example, take as A the expression:

2.06 ? p→q .→: q→r .→. p→r.

for which we have DL(2.06) = (2,2,2), DR(2.06) = 3,3,4); and take as A' the expression:

A' ? a .→. b→c

for which we have DL(A') = (1,1,1), DR(A') = (2,2,2).

If in A' we substitute p→q for a, q→r for b, and p→r for c, we obtain 2.06. Operating in the reverse direction, if we contract 2.06 by making the inverse substitutions, we obtain A'. We can therefore refer to A' as "2.06 viewed as contracted."

Since the purpose in searching for similar theorems is to find appropriate materials to which to apply the matching routine, there is no reason why we should not use this more general notion of similarity if it proves effective in finding materials that are useful.

In general, what parts of an expression should be considered as units in the search for proofs is not a "given" for the problem solver. LT makes an explicit decision each time it looks for similar expressions as to what subexpressions will be taken as units. In contracting 2.06, a decision has been made that the elements p, q, and r are too small, and that more aggregative elements, e.g., $(p \to q) = a$, should be perceived as units.

Examination of the routines for describing expressions (NH, NK, NJ) will reveal that these routines in fact count *units* rather than *variables*. Normally, the variables are the units used in description, for VV precedes CSm in every program except MDt. In the latter program, however, it is sometimes useful to view expressions as contracted, by means of VCt.

Example of Proof by Detachment. To illustrate the method of detachment, let us carry out explicitly the proof of 2.06:

2.06 ? $p \to q \ .\!\to\!: \ q \to r \ .\!\to\!. \ p \to r$

The reader may verify that this theorem cannot be proved by substitution in the axioms and earlier theorems. Moreover the detachment method without contraction will also fail, for there is no theorem whose right side is similar to 2.06. However, we have already seen that when we contract 2.06, we obtain:

A' ? $a \ .\!\to\!. \ b \to c$

where $p \to q$ has been contracted to *a*, $q \to r$ to b, and $p \to r$ to c. We now have $DL(A') = (1,1,1)$ and $DR(A') = (2,2,2)$, descriptions that are identical with the descriptions of the sub-expressions of the right side of 2.04.

2.04 ! A →. B → C :→: B →. A → C

A' a →. b → c

Having selected 2.04 by use of the routine MDt(SmCt), we now proceed to match its right side with 2.06 in segment MDt(Mc):

2.04		A	→.	B	→	C	:→:	B	→.	A	→	C
2.06	?							p→q	.→:	q→r	.→.	p→r
2.04'	!	q→r	.→:	p→q	.→.	p→r	.:→:.	p→q	.→:	q→r	.→.	p→r

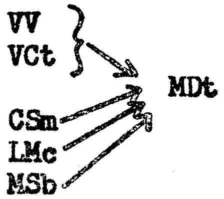
We have now created a new problem to replace the original one: to prove that the left side of 2.04' (the part underscored) is a theorem. We apply the method of substitution, MDt(MSb). The search of the theorem memory discloses 2.05 to be similar to the left side of 2.04', and we proceed to match them:

2.04'L ? q→r .→: p→q .→. p→r

2.05 ! A→B .→: C→A .→. C→B

It is easy to see that with the substitution of q for A, r for B, and p for C, the matching will be successful. Hence we have B (2.05 with the indicated substitution), and B→A (2.04'), from which A (2.06) follows by the rule of detachment.

The diagram below summarizes the principal routines incorporated in the method of detachment. A comparison of this diagram with the one for the substitution method shows clearly that both methods rest on the same component processes, with minor modifications and new combinations and conditions. The sole new process involved in detachment is the viewing of theorems as contracted.



The Chaining Method

A number of expressions that do not yield to the method of substitution can be proved by the method of detachment. We shall add an additional method, however, to the repertoire available to LT. We shall call this method chaining, MCh. Like the methods previously described, chaining involves heuristic procedures which we shall consider first.

Theorem 2.06, which we have just proved, embodies one form of the principle of the syllogism (2.05 is another form of this principle). Now suppose T_1, $(p \rightarrow q)$ is a true theorem, and T_2, $(q \rightarrow r)$ is another true theorem. Theorem 2.06 is of the form:

where E is $(p \rightarrow r)$, an expression not known to be true. By detachment, from ! T_1 and ! $(T_1 . \rightarrow . T_2 \rightarrow E$, we get ! $T_2 \rightarrow E$. By a second detachment, from ! T_2 and ! $T_2 \rightarrow E$, we get ! E. Hence, if we know $p \rightarrow q$ and $q \rightarrow r$ to be true, we can construct a proof of $p \rightarrow r$ by means of two detachments with the use of 2.06. Instead of carrying through this derivation explicitly in each instance, we simply construct a program that makes direct use of the transitivity of syllogism. This proof method is the basis for chaining.

Suppose that we wish to prove $A \rightarrow C$. We search for a theorem, T (with → for a connective) whose left side is similar to A, using the segment MCh(SmF). We match the left side of T with A, MCh(MgF), and if we are successful, we have then proved a theorem of the form $A \rightarrow B'$, for T, as modified by matching, is of this form. We now construct, by segment MCh(P), the expression $B \rightarrow C$, and attempt to prove this expression by substitution, MCh(MSb). If we are successful, we now have a chain: A→B, B→C. Then by syllogism, as indicated above, we obtain A→C, the expression we wished to prove.

The procedure just described is chaining forward. Alternatively, we may chain backward. That is, to prove A→C, we may search for a theorem of the form B→C; then try to prove A→B by substitutition.

Proof by the chaining method is illustrated by:

2.08 ? $p \to p$

A search for theorems that have left sides similar to 2.08 yields 1.3, 2.02, and 2.07. The latter is:

2.07 ! $p \mathrel{.{\to}.} p \vee p$

If we take 2.07 as the (A→B) of the schema given above, then B is (pvp). Two theorems have left sides similar to B: 1.2 and 2.01. An attempt to match the left side of 2.01 to the right side of 2.07 will be unsuccessful, but the matching is immediate with 1.2:

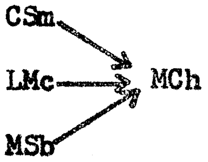
2.07 ! $p \mathrel{.{\to}.} p \vee p$

1.2 ! $p \vee p \mathrel{.{\to}.} p$

Hence we can take 1.2 as the (B→C) of the chaining method. We now form (A→C) by joining the left side of 2.07 to the right side of 1.2 by ⇒ . The result is 2.08:

2.08 ! $p \mathrel{.{\to}.} p$

The chaining method is summarized by the following diagram, which shows that the method again makes use of tests for similarity, matching, and substitution:



The Executive Routine

We have now introduced a set of methods that, jointly, are believed to be sufficiently powerful to prove in sequence the theorems of Chapter 2 of Principia. In fact, proofs have been constructed for these theorems by hand simulation of the procedures outlined here; and a hand simulation has also been carried out by Mr. Kalman Cohen, using a somewhat more elaborate program, of the proofs of the theorems of Chapter 3. Extension of these methods to the predicate calculus is, of course, another matter.

It remains to complete our specification of LT in two directions--first to assemble the three methods that have been described into a coherent program; second, to show how the information processes in terms of which LT has been described here can be specified precisely in terms of the elementary processes listed in the previous section of this paper. The latter task is carried out in detail in Section III. We will turn our attention here to the former, which is embodied in the executive routine, Ex.

In its first segment, Ex(R), the executive routine reads a new expression that is presented to it for proof, and places it in a working memory.[14] In the next three segments, Ex(MSb), Ex(MDt), and Ex(MCh), successive attempts are made to prove the expression by the methods of substitution, detachment, and chaining, respectively. If

[14] Certain segments of Ex, in particular Ex(R), Ex(WP), Ex(ST) and Ex(WNP), are not written formally in Section III in terms of the primitives but are simply indicated by parentheses. It would be rather simple to formalize them, but this would further lengthen the description of the program.

a proof is obtained by one of these methods, the executive routine writes the proof, Ex(WP); and stores the newly-proved theorem, after changing all its variables to free variables, as a theorem in the theorem memory, Ex(ST).

To explain what happens if the three methods are unsuccessful, we have to take up some details that were omitted above. These have to do with the creation of subsidiary problems and with stop rules.

Subsidiary problems. Both detac ment and chaining are two-step methods. Suppose we wish to prove A. In detachment, we try to find a theorem, $B \rightarrow A$, and if we are successful, we then try to prove B. The task of proving B we may call a subsidiary problem.

Suppose we wish to prove $a \rightarrow b$. In chaining, we try to find a theorem, $a \rightarrow c$, and if we are successful, we then try to prove $c \rightarrow b$. The task of proving $c \rightarrow b$ is also a subsidiary problem.

In both the detachment and chaining methods, only the method of substitution is applied to the subsidiary problem. If that method fails, failure is reported for the main problem. But before control is shifted back to the executive routine, the main element of the subsidiary problem is stored in the problem list in the storage memory. (The operation that stores the problem in the problem list is the operation SEN that can be found in segment MDt(P) and segment MCh(P).)

When the three methods have failed, the executive routine stores the expression in the inactive problem list, Q; goes to the problem list, P; and selects from that list the problem whose expression is, in a certain sense, the simplest-specifically an expression with the smallest possible number of levels, K, Ex(CK). It erases the new subsidiary problem from list P; checks to make certain it does not duplicate one previously attempted, Ex(CX); and then tries to solve this subsidiary problem by the methods of detachment and chaining.[15] This sequence is repeated until some subsidiary problem is solved (in which case the main problem is also solved), or until no problems remain on the problem list, or until the other stop rule, to be described, comes into operation. In the latter two cases,the routine reports that it is unable to prove the theorem, Ex(WMP).

The check to prevent duplication of subsidiary problems, Ex(CX), is handled as follows: For each problem that is selected from list P by Ex(CK), a check is made, by Ex(CX), against all expressions in the inactive problem list, Q, and if the new problem duplicates any expression found there, it is dropped. The main operation of this segment, CX, applies the same basic tests of identity of elements that are applied in the matching program, but does not modify the expressions to make them match.

<u>Stop Rules</u>. Since all proof methods may fail--even if the expression givento LT is a genuine theorem, and certainly if it is not--the executive routine needs a stop rule. One stop rule is provided by the exhaustion of list P, but there is no guarantee that the list will ever be exhausted. A second stop rule is provided by operations that measure the total amount of "work" that has been done in attempting

[15] There is no need to attempt to prove the subsidiary problem by substitution, since an unsuccessful substitution attempt was made immediately before the expression was stored in the subsidiary problem list.

to prove a theorem, and that terminate the program with a "no proof" report when the total work exceeds a specified amount. The first operation in the substitution routine, NAW, simply tallies one for each time the routine is used. This tally is kept in a special location in the storage memory. The executive routine, just before it seeks a new subsidiary problem, checks the cumulative tally in this register, Ex(CW), and if the tally exceeds a given limit, terminates the program. Since the substitution routine is used in each of the methods,the number of substitutions attempted seems to be one reasonable index of the amount of work that has been done.

This stop rule operates as a global constraint on the total work applied in trying to prove a single theorem. The rule does not govern the direction in which this effort is expended. The latter is determined by the priority rule previously described for selecting subsidiary problems from the problem memory and by the other elements of LT's program.

Learning Processes

The program we have described is primarily a performance program rather than a learning program. But, although the program of LT does not change as it accumulates experience in solving problems, learning does take place in one very important respect. The program stores the new theorems it proves, and these theorems are then available as building blocks for the proofs of subsequent theorems. Thus, in the theorems used as examples in this paper, 2.06 was proved with the aid of 2.05 and 2.04, and 2.08 was proved with the aid of 2.07. Without this form of learning it is doubtful whether the program would prove any but the first few theorems of Chapter 2 in a reasonable number of steps.