A Variant to Turing's Theory of Computing Machines*

HAO WANG

University of Oxford, Oxford, England

"Everyone should firmly persuade himself that none of the sciences, however abstruse, is to be deduced from lofty and obscure matters, but that they all proceed only from what is easy and more readily understood."—Descartes.

1. *Introduction*

The principal purpose of this paper is to offer a theory which is closely related to Turing's [1] but is more economical in the basic operations. It will be proved that a theoretically simple basic machine can be imagined and specified such that all partial recursive functions (and hence all solvable computation problems) can be computed by it and that only four basic types of instruction are employed for the programs: shift left one space, shift right one space, mark a blank space, conditional transfer. In particular, erasing is dispensable, one symbol for marking is sufficient, and one kind of transfer is enough. The reduction is somewhat similar to the realization of, for instance, the definability of conjunction and implication in terms of negation and disjunction, or of the definability of all these in terms of Sheffer's stroke function. As a result, it becomes less direct to prove that certain things can be done by the machines, but a little easier to prove that certain things cannot be done.

This self-contained theory will be presented, as far as possible, in a language which is familiar to those who are engaged in the use and construction of large-scale computers.

Turing's theory of computable functions antedated but has not much influenced the extensive actual construction of digital computers. These two aspects of theory and practice have been developed almost entirely independently of each other. The main reason is undoubtedly that logicians are interested in questions radically different from those with which the applied mathematicians and electrical engineers are primarily concerned. It cannot, however, fail to strike one as rather strange that often the same concepts are expressed by very different terms in the two developments. One is even inclined to ask whether a rapprochement might not produce some good effect. This paper will, it is believed, be of use to those who wish to compare and connect the two approaches.

Extensive use of Turing [1] and Kleene [2] (Chapter XIII) will be made, although the exposition is sufficient in itself.

* Presented at the meeting of the Association, June 23-25, 1954. The work for this paper was partly supported by the Burroughs Corporation, Research Center, Paoli, Pa. Thanks are due to Professors A. W. Burks and G. W. Patterson for useful suggestions and comments.

2. *The basic machine B*

No physically realizable general-purpose machine is truly general-purpose in the sense that all theoretically solvable computation problems can be solved by the machine. In each case, the storage unit is necessarily finite: the length of each instruction word is finite, and there are only a definite finite number of addresses in the storage. Given any actual machine, it is easy to find computation problems which require more than its storage capacity.

Indeed, it seems reasonable to agree that given any uniform method of representing positive integers and an arbitrary number n_0, there exist integers not representable by less than n_0 symbols. We shall refrain from philosophical discussions on this question, such as the possibility of infinitely many isolated atomic symbols, etc., but take this for granted. If we accept this reasonable assumption, it follows that, in order to discuss a machine which computes all computable problems, or even a machine which computes just all problems of a given kind, such as the addition of two positive integers, we have to use the fiction of an infinite or indefinitely expandable storage unit, which may consist of a (fictitious) tape, or a (fictitious) internal store, or both.

Consider first a fictitious machine B which has an indefinitely expandable internal (parallel) storage and which uses in addition a tape (a serial storage) that is divided into a sequence of squares (cells) and could be extended both ways as much as we wish. In addition, the machine has a control element and a reading-writing head which, at each moment, is supposed to be scanning one and only one square of the tape. The head can, under the direction of the control element and in conformity with the program step under attention, move one square to the left or right or mark the square under scan, or "decide" to follow one of two other preassigned program steps according as the square under scan is marked or blank. At each moment, the next operation of the machine is determined by the step of the program under attention of the control element, together with the content (blank or marked) of the square under scan. The machine can perform four kinds of operation in accordance with these two factors: (i) the reading-writing head moves one square to the left, (ii) it moves one square to the right, (iii) it marks the square under scan, (iv) the control element shifts its attention to some other program step.

Suppose that the addresses in the internal storage can be any positive integers. Corresponding to the four types of operation, there are four types of basic instructions:

(1) $\rightarrow$: shift the head one square to the right, i.e., shift to scan the next square to the right of the square originally under scan; the same purpose can be accomplished by shifting the tape one square to the left.

(2) $\leftarrow$: shift the head one square to the left.

(3) $*$: mark the square of the tape under scan.

(4) Cn: a conditional transfer.

Of these, (1)–(3) are three single instructions, while the conditional transfer really embodies an infinite bundle of instructions. A conditional transfer is of the

form $m.Cn$, according to which the m-th instruction word (i.e., the instruction word at the address m) is a conditional transfer such that if the square under scan is marked, then follow the n-th instruction; otherwise (i.e., if the square under scan is blank), follow the next instruction (i.e., the $(m + 1)$-th instruction). The numbers m and n can be any positive integers that are not greater than the total number of instructions of the program in which $m.Cn$ occurs. When $n = m + 1$, the instruction is wasteful; but we need not exclude such cases. There is nothing to prevent the occurrence of program steps which would instruct the reading-writing head to mark a square already marked, even though we can usually so construct the programs that when operating on inputs which interest us no such wasted actions will arise. To preclude the marking of already marked squares in the definition of programs would introduce unnecessary complications into the general considerations: e.g., replace every step $m.*$ by two steps $m.C(m + 2)$, $m + 1.*$ and renumber all steps $m + i$ in the original program by $m + i + 1$.

Since there is no separate instruction for halt (stop), it is understood that the machine will stop when it has arrived at a stage that the program contains no instruction telling the machine what to do next. For uniformity and explicitness, however, we shall agree that every program has as its two last lines $N - 1.\rightarrow$, $N.\leftarrow$. To illustrate, we give a simple program: $1.*$, $2.\rightarrow$, $3.C2$, $4.\rightarrow$, $5.\leftarrow$. This program enables the reading head to find and stop at the nearest blank to the right of the square initially under scan.

More exactly, a program or routine on the machine B can be defined as a set of ordered pairs such that there exists a positive integer $k (k > 2)$ for which (a) for every n, n occurs as the first member of exactly one pair in the set if and only if $1 \leqq n \leqq k$; (b) the second member of each pair is either $*$ or $\rightarrow$ or $\leftarrow$ or a number n, $1 \leqq n \leqq k - 1$; (c) there are the pairs $\langle k - 1, \rightarrow \rangle$, $\langle k, \leftarrow \rangle$. We can, according to this definition, represent the example above by: $\{\langle 1, * \rangle, \langle 2, \rightarrow \rangle, \langle 3, 2 \rangle, \langle 4, \rightarrow \rangle, \langle 5, \leftarrow \rangle\}$.

In contrast with Turing who uses a one-way infinite tape that has a beginning, we are following Post in the use of a 2-way infinite tape. This is considered a reduction because we are deprived of the privilege of appealing to the beginning of the tape. As it turns out, for the purpose of computation, there is little difference in the capacity of the machine whether we use a 1-way or a 2-way infinite tape. So far as the general theory is concerned, the 1-way tape tends to rule out certain programs as meaningless when they instruct the reading head to go beyond the beginning of the tape. This would introduce unnecessary complications.

We have defined the totality of all possible programs on the machine B. For each program, what the machine B will do is determined by the initial tape content and the initial position of the reading head relative to the tape, both of which can be given arbitrarily. To simplify our further considerations, we shall assume, once for all, that (i) the initial (input) tape contains only finitely many marked squares, and that (ii) at the beginning of each program, the reading head scans the sixth blank square to the right of the rightmost marked square. On the other hand, subroutines begin on any square within the minimum tape

portion containing all the initially marked squares and six blanks to the right, and end similarly. From (i), it follows that at each moment there are only finitely many marked squares on the tape.

We note that given any program II, its behaviour at each moment is completely determined by: the content of the tape at the moment, the position and content of the square under scan, the instruction of the program that is being attended to at the moment. The three factors plus the given program determine the *complete instantaneous state* of the machine B. To be precise, we can represent these states by numbers in the following manner. We represent $\rightarrow$, $\leftarrow$, $*$ by 1, 2, 3 respectively and n by $n + 3$, the i-th program step by a power of the i-th prime, so that, for example, $2^3.3^1.5^5.7^1.11^2$ represents the program

$$\{\langle 1,*\rangle,\ \langle 2,\rightarrow\rangle,\ \langle 3,2\rangle,\ \langle 4,\rightarrow\rangle,\ \langle 5,\leftarrow\rangle\}$$

given above. The length of a program II is represented by the number $lh(\pi)$ of distinct prime factors in its representing number π and the i-th line of II is represented by the number $(\pi)_i$ which is the exponent of the i-th prime number (2 being the first prime number) in the factorization of π. The functions $lh(\pi)$ and $(\pi)_i$ are well-known recursive functions (cf. e.g., Kleene [2]). The tape content and the square under scan can be represented by a number of the form $3^a.5^b.7^c$, where $b = 0$ or 1 according as the square under scan is blank or marked, a and c represent respectively the tape contents to the left and to the right of the square under scan, in such a way that the i-th digit (from right to left) of a (resp. b) is 0 or 1 according as whether the i-th square to the left (resp. right) is blank or marked. Since only finitely many squares are marked at each moment, a and c are at each time definite numbers. If, for instance, the only marked portion of the tape is

*		*	*				*	*

and the head scans the fourth (from left) of these squares, the number is $3^{101}.5.7^{1100}$. Using these numbers, we represent the complete instantaneous state at a moment by the number

$$2^{\pi}.3^a.5^b.7^c.11^i,$$

when the program has number π, the tape content with the square under scan has at the moment the number $3^a.5^b.7^c$, and the program step attended to is i. From this number, we can also recover π, a, b, c, i in a unique manner, on account of the unique factorization of a number into prime factors.

Since we assume that one program step is performed at each moment, the complete instantaneous state of the machine at each moment is determined by that at the preceding moment in a fairly simple manner. Thus, if the number at one moment is $2^{\pi}.3^a.5^b.7^c.11^i$ and $i < lh(\pi)$, then the number at the next moment is given as follows:

(i) if $(\pi)_i$ is 1, it is $2^{\pi}.3^{10a+b}.5^{c'}.7^{c/10}.11^{i+1}$ (c' being the remainder of dividing c by 10);

(ii) if $(\pi)_i$ is 2, it is $2^r.3^{a/10}.5^{a'}.7^{10c+b}.11^{i+1}$;

(iii) if $(\pi)_i$ is 3, it is $2^r.3^a.5.7^c.11^{i+1}$;

(iv) if $(\pi)_i$ is $3 + j$, it is $2^r.3^a.5^b.7^c.11^{i+1}$, if $b = 0$, and $2^r.3^a.5^b.7^c.11^j$, if $b = 1$.

Hence, given the complete instantaneous state at the moment when the machine begins to perform a program, all later states will be determined. The begining state is in turn entirely determined by the program and the input tape. It follows from these facts that for each given program π, there is a recursive function $\theta_\pi(\iota, t)$ which gives, for any given input tape content represented by ι, the number of the complete instantaneous state of the machine B at moment t. Then we can define a recursive function $\tau_\pi(\iota) = \mu_t[(\theta_\pi(\iota, t))_5 = lh(\pi)]$, which gives the smallest number t such that, for given ι, the machine will be attending the last line of the program Π at time t, provided the machine does get to that stage for the given ι. For values of ι which would make the machine run forever according to the program Π, $\tau_\pi(\iota)$ is not defined. It seems reasonable to call the function $\tau_\pi(\iota)$ the *speed function* of the program Π or of the function which Π determines. We may call the recursive function $\theta_\pi(\iota, t_\pi(\iota))/(2^r.11^{lh(\pi)})$ the function determined by the program Π, since it gives for each input ι, the corresponding output that results from a performance of the program Π. We omit the explicit definitions for these functions which are similar to those of Kleene [2], pp. 374–376.

It follows that the function determined by any program of the machine B is a recursive function. We can state the result by saying that all functions computable by the machine B are recursive. This is a slight generalization of the counterpart of the known theorem that all functions computable in Turing's sense are recursive (Kleene [2], p. 374).

The generalization consists in the waiving of the restriction that the initial input and final output tape contents must be of certain preassigned forms which are taken according to an ad hoc convention as representing positive integers (arguments or values of functions) or n-tuples of integers (arguments of non-singulary functions). For instance, in Kleene [2], the input tape is supposed to consist of a string of 1's, or a string of several strings of 1's any two of which are separated by a 0. What the restriction amounts to is to select effectively from all possible tape situations a suitable subset to represent all positive integers and their n-tuples. In terms of our numerical representation of tape situations, this means that ι is not to range over the set of all integers which represent tape situations but only over some simple recursive subsets of it. Moreover, in our representation of tape situations, we have included information about the square under scan; this is not needed when we are primarily interested only in interpreting initial and final tape contents as questions and answers for evaluating certain functions of positive integers.

When we wish to show that a program Π of the machine B represents a function (say $f(x)$) of positive integers, the natural thing to do would seem to be, (i) represent by a uniform method the argument of the function on the input tape,

(ii) read out by a uniform method the final value of the function from the final output tape for a given input by the program II, (iii) show that the values thus obtained always agree with the values of the function $f(x)$ for the given argument. What uniform methods of representation we choose is more or less arbitrary although they must be effective and indeed "simple" in some intuitive sense. While our considerations thus far are independent of how we choose these effective methods of representation, in order to prove our main theorem, we find it necessary to fix some simple specific method of putting questions on and reading answers from the tape. We shall not consider the general question as to all the possible methods of representing questions and answers by which all recursive functions are computable on the machine B.

We assume the particular method, to be described in the next section, of representing positive integers and strings of them and define computability relative to the representation. A function $f(x)$ of positive integers is said to be B-computable if, and only if, we can find a program such that when x is represented on the tape and when the machine is scanning the representation, the program leads the machine to print more marks on the tape so that when the machine has performed all the instructions of the program, the rightmost portion of the printed part of the tape represents the value of $f(x)$, provided $f(x)$ is defined for the given argument x. A similar definition is obtainable for functions of two or more variables. For example, it is possible to show that the functions

$$f(x) = 2x,$$
$$g(x, y) = xy$$

are B-computable by the above definition.

The main result, to be established in the next section, is that all (partial) recursive functions are B-computable. The proof depends essentially on the known fact (Kleene [2], p. 331) that all recursive functions can be obtained by a few (six, in fact) simple types of schemata. Since each schema, as we shall prove, can be handled by the machine B, the general result follows.

The result is a little stronger than the known fact that all recursive functions are computable in the sense of Turing, because Turing permits additional basic instructions (besides $*$, $\rightarrow$, $\leftarrow$, Cx) such as erasing a mark, marking a square by other symbols, unconditional transfer, and other types of conditional transfer. While it is obvious that not all these additional instructions are indispensable, it is by no means obvious that all of them are dispensable. Indeed, the actual proof of the adequacy of $*$, $\rightarrow$, $\leftarrow$ and Cx, which will be presented in the next section, is quite complex.

It is also known that all Turing computable functions are recursive. As all B-computable functions are obviously Turing computable, so all B-computable functions are recursive. This last conclusion also follows directly from the more general statement, previously established, that all functions computable by the machine B are recursive.

These results will establish the equivalence or coextensiveness of Turing computability and B-computability in the sense that a function is Turing computable

if and only if it is B-computable. The main result that all recursive functions are B-computable yields also, for example, a somewhat simpler proof of the recursive unsolvability of Thue's problem (the word problem for semi-groups) (Kleene [2], p. 382), since fewer types of basic operations need to be considered.

We interrupt the general discussion to give details of our proof of the main theorem. Readers not interested in technical matters may wish to skip the more formal parts of the next section.

3. *All recursive functions are B-computable*

We possess fairly simple inductive characterizations of the class of recursive functions: certain simple initial functions are recursive; given a class of recursive functions, certain simple schemata will yield new recursive functions; and these provide us with all recursive functions. To prove that all recursive functions are B-computable, we use induction accordingly: the initial recursive functions are B-computable; given a class of B-computable functions, functions got from them by the recursion schemata are again B-computable.

We shall follow Turing in making a distinction between principal and auxiliary squares on the tape. While this is merely a matter of dispensable convenient convention in Turing's approach, it is not clear that we can get what we want without some form of the distinction, in view of the fact that we do not permit erasing. We shall call one sequence of alternate squares the P-squares (principal squares) and the other sequence A-squares (auxiliary squares). Questions and answers on a tape are determined by the contents of the P-squares alone.

In general, between two marked consecutive P-squares, there is an A-square which may be either blank or marked. We shall represent each positive integer n by a string of n pairs of squares (called a "number expression") which begins with (i.e., at the left end) a P-square and ends with an A-square such that all P-squares in the string are marked and that the P-squares which immediately precede and follow the string are both blank. Incidentally, this leaves the number 0 unrepresented. We shall deliberately consider only positive integers. We could, if we wished to include the number 0, represent n by a string of $(n + 1)$ pairs of squares with all P-squares marked. But there is no need to do so. We shall call a number expression clean if all the A-squares in it are blank. In general, we always introduce at first clean number expressions. We begin to mark the A-squares in a number expression only when we operate on it (as we shall see, this means: copy it).

Our purpose is to prove that for each recursive function $f(x_1, \cdots, x_m)$, we can construct a program which, when fed into the machine B, will compute the values of the function. This task we interpret as follows. Given any constant argument values $x_1, \cdots, x_m$, we represent them anywhere on the tape from left to right, first the number x_1, followed by one blank P-square and one blank A-square, followed by the number x_2, etc. Once the argument values are given, it is determined which squares on the tape are the P-squares, since the beginning square of x_1 is a P-square. Our program for the function $f(x_1, \cdots, x_m)$ is to enable us so

to operate the machine B that for any given positive integers $x_1, \cdots, x_m$, the machine B will eventually stop and then there is a number expression on the tape to the right of which all squares are blank, and which represents the number that is the value of f for the given argument values $x_1, \cdots, x_m$, provided f is defined for these particular argument values. If $f(x_1, \cdots, x_m)$ is not defined, the machine may either never stop (circular) or stop at a stage when there is no number expression on the tape such that all squares to its right are blank (blocked). In other words, for each recursive function, we try to find a program such that for each given array of argument values (or for each argument value, when we have a function of one variable), the machine B makes a separate calculation on a separate tape. If $f(x_1, \cdots, x_m)$ is defined for all $x_1, \cdots, x_m$ (i.e., not only partial recursive but general recursive), we could use the same tape to calculate all values successively since each calculation uses up only a finite portion of the infinite tape. We shall, however, to simplify the picture, prefer to think that each time we use a new blank tape. We are not interested in what the machine does with the program when the initial tape content does not represent exactly a string of m integers.

We shall so construct our programs for recursive functions that each recursive function is not only B-computable but that each of the programs will satisfy several additional conditions. We assume that the initial tape contains a question of the right form. According to the definition of number expressions, any two number expressions must be separated by at least one blank P-square. The program will be such that in the process of operating the machine, (a) there are never more than two blank P-squares between two number expressions; (b) every marked P-square is always part of a number expression; (c) we shall never mark an A-square which lies outside the minimum tape portion that contains all the printed marks at the moment; (d) for any values $x_1, \cdots, x_m$, either the machine will never stop or the final tape will contain a number expression to the right of which there is no marked square (i.e., either the function is defined or the operation of the machine is circular, but never blocked). From (a) and (c) it follows that at every stage of operating the machine, there never appear more than 5 blank squares between two marked squares. We shall call a string of 5 (resp. 3) blank squares lying between two marked squares a big gap (resp. a gap). It follows from (a) and (c) that each gap or big gap always begins with an A-square and ends with an A-square. We shall eliminte a big gap by marking the A-square in the middle. The programs also satisfy the following conditions: (e) if the machine does stop for given arguments $x_1, \cdots, x_m$, the last number expression will be clean (i.e., no A-square in it is marked); (f) on stopping, all big gaps will be eliminated; (g) on stopping, the reading head will always end up scanning the 5-th blank square to the right of the last number expression, i.e., the 6-th blank square beyond the last marked square (we shall say that the reading head ends up scanning the open). Condition (e) assures that we can immediately use the value of one function in calculating the value of some other function. Condition (f) helps to locate previous number expressions on the tape.

Since there are never more than 5 blanks between two marked squares, we can, beginning with the reading head scanning a square within the minimum region of the tape which contains all marked squares, easily find its beginning and its end: go left (resp. right) until a string of 6 blank squares is found.

We introduce a subroutine Y such that if the reading head is scanning a square which falls within the minimum tape region which contains all marked squares and 6 blanks to the right of the last marked square, carrying out Y will enable the reading head to find the last marked square and end up scanning the open, provided there are initially no more than 5 blanks between any two marked squares. We note that this is useful because at the beginning of each of the programs we are actually going to use below, the square initially under scan always falls within the region just specified. We introduce first a subroutine X which enables us to find the nearest marked square to the left of the square initially under scan (or itself, if it is marked) and end up one square to the right.

Subroutine X: $1.C14$, $2.\leftarrow$, $3.C14$, $4.\leftarrow$, $5.C14$, $6.\leftarrow$, $7.C14$, $8.\leftarrow$, $9.C14$, $10.\leftarrow$, $11.C14$, $12.\leftarrow$, $13.C14$, $14.\rightarrow$, $15.\rightarrow$, $16.\leftarrow$.

Using X, we can define Y:

(i) Subroutine Y: $1.X$, $2.\leftarrow$, $3.\rightarrow$, $4.C3$, $5.\rightarrow$, $6.C3$, $7.\rightarrow$, $8.C3$, $9.\rightarrow$, $10.C3$, $11.\rightarrow$, $12.C3$, $13.\rightarrow$, $14.C3$, $15.\rightarrow$, $16.\leftarrow$.

Next we introduce a subroutine which enables us to add 1 to the last number on the tape and end up with the reading head scanning the last marked square. We use the obvious notation $\leftarrow^n$ (resp. $\rightarrow^n$) when $\leftarrow$ (resp.$\rightarrow$) is repeated n times:

(ii) Subroutine A: $1.Y$, $2.\leftarrow^4$, $3.*$, $4.\rightarrow$, $5.\leftarrow$; or simply Y, $\leftarrow^4$, $*$, $\rightarrow$, $\leftarrow$; or even just Y, $\leftarrow^4$, $*$.

This subroutine can be reiterated to enable us to add any given constant k to the last number. In general, for any subroutine or program Π, we shall use the notation Π^k to represent that Π is repeated k times, and Π^1 is identified with Π. In the case of A, the purpose of A^k could be achieved more directly by using $(A, \rightarrow^2, *)$, $(A, \rightarrow^2, *, \rightarrow^2, *)$, etc. But it is preferable to use reiterations of whole subroutines, when possible.

Similarly with Subroutine Y, we can also introduce a subroutine H which enables us to find the nearest big gap to the left of the square under scan (if there is such a big gap) and end up with the reading head scanning the middle square of the big gap. Note that if the square initially under scan is not marked, we have to find a marked square first.

(iii) Subroutine H: $1.C4$, $2.\leftarrow$, $3.C1$, $4.\leftarrow$, $5.C4$, $6.\leftarrow$, $7.C4$, $8.\leftarrow$, $9.C4$, $10.\leftarrow$, $11.C4$, $12.\leftarrow$, $13.C4$, $14.\rightarrow^2$, $(15.\rightarrow$, $16.\leftarrow.)$

The success of the subroutine again depends on the fact that we do not allow more than 5 blanks between two marked squares. We could also get one in-

dependently of this fact if we replace "$14.\rightarrow^{2}$" by the following: $14.\leftarrow$, $15.C19$, $16,\leftarrow$, $17.C4$, $18.C16$, $19.\rightarrow^{3}$.

To give direction to our procedure, we recall that our purpose is to prove that all recursive functions are B-computable, and that our proof depends on the possibility of getting all recursive functions from six schemata. Of these six, the first three are direct schemata telling us unconditionally that functions defined by them are recursive, the remaining three are conditional ones. We pause to state the first three schemata, which are simpler, and explain where we are going.

Schema (I). $\varphi(x) = x + 1$.

Schema (II). $\varphi(x_1, \cdots, x_n) = q$, q a given constant.

Schema (III). $\varphi(x_1, \cdots, x_n) = x_i$, i a given constant among $1, \cdots, n$.

To prove, for instance, that the function defined by (I) is B-computable, we find a program such that when the argument value x is given on the tape and the machine begins by having its reading head scanning a square within the minimum tape portion which contained all initially marked squares and 6 blanks to the right of the last marked square, the reading head will find the last marked square in the representation of the number x, move right six squares (leaving thereby a big gap), start to copy the number x in the consecutive P-squares, add 1 after the copying is finished, and indicate that the operation is completed. We shall then have the answer since the last number on the tape will be $x + 1$.

To obtain such a program, the most difficult part is to do the copying whose result of course has to vary with the number to be copied. Let us assume for the moment that we have obtained a subroutine for copying:

(iv) Subroutine I_m: this enables us, beginning at the square under scan, to copy in successive alternate squares the m-th number (counting from right to left) which lies to the left of the nearest big gap which precedes (is to the left of) the square under scan, and end up with the reading head scanning the open.

The closing steps of eliminating the last big gap and having the machine scan the open are combined in one subroutine:

(v) Subroutine Z: $1.H, 2.*, 3.Y$; or, since there is no conditional transfer, simply, H, $*$, Y.

Using Z, the subroutines A, Y already introduced, plus the subroutine I_m (to be introduced formally below), we can now prove that all functions defined by the schemata (I), (II), (III) are B-computable.

The programs needed are:

(I) Y, I_1, A, Z.

(II) $Y, *, A^{q-1}, Z$; when $q = 1$, simply $Y, *, Z$.

(III) Y, I_{n-i+1}, Z.

Hence, to complete this part of our proof, we only have to construct the subroutine I_m. We shall deliberately make it more complicated than necessary in order that the same number expression can be copied more than once. In order to copy a number expression, we have to mark the A-squares between the marked P-squares of the original number expression as we go along, in order to keep track of how far we have advanced. If we use up all these auxiliary squares in the first copying, we shall not be able to make a second copy from the original and we do not wish to make a second copy from the first copy since we wish to operate on it. In general, we contrive to allow the possibility of copying a given number twice, so that we can operate on one copy and reserve one clean copy for future "reference" (i.e., further copying).

These complications are of course only necessitated by our decision not to permit erasing. Indeed, this is a crucial step of the whole proof. Since we may wish to copy a number expression an indefinite number of times and since with each copying we have to mark some squares to keep track of how far we are along in our action, we may feel that erasing is indispensable. The difficulty is solved by the simple trick just described: copy the same expression twice, copy one of the copies twice, etc.

We introduce a few subroutines.

(vi) Subroutine D: $1.C4, 2.\leftarrow^2, 3.C6, 4.\leftarrow^2, 5.C1, 6.\rightarrow, 7.\leftarrow$.

This enables the reading head, when scanning a P-square, to find and scan the last P-square of the nearest preceding number expression. The process can be repeated so that D^m gives us the last (marked) P-square of the m-th (counting from right to left) number expression which precedes the square under scan.

(vii) Subroutine G: $1.\rightarrow, 2.\leftarrow^2, 3.C2, 4.\leftarrow^2, 5.C2, 6.\rightarrow$.

When scanning a P-square, this enables the reading head to find the nearest two successive blank A-squares to its left and end up scanning the P-square between them.

(viii) Subroutine K: $1.C3, 2.C5, 3.\leftarrow, 4.C1, 5.\leftarrow$.

When scanning a marked square, find the nearest blank square to its left and end up scanning the square immediately preceding that blank.

(ix) Subroutine $M(a)$: $1.C3, 2.Ca, 3.\leftarrow^2, 4.C6, 5.A, Ca, 6.\rightarrow, 7.C6, 8.*, A^2$.

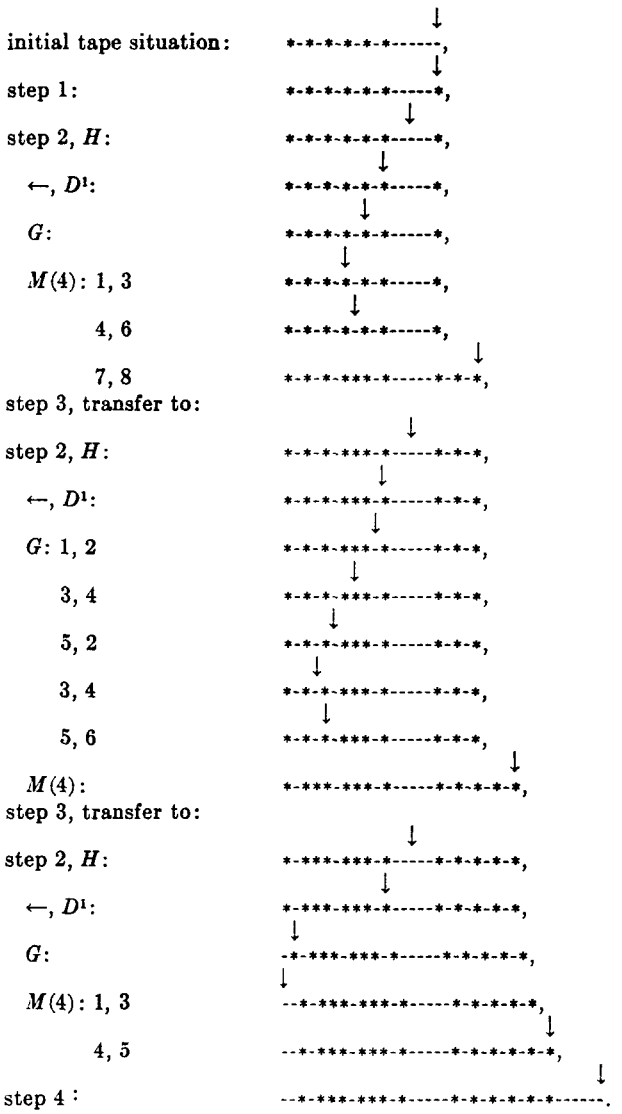
When scanning a P-square, if it is blank, this asks the machine to follow instruction a (which "plugs" this subroutine into a larger routine or program), if marked, then scan the immediately preceding P-square: if it is blank, then add one to the last number expression on the tape and then follow instruction a; if it is marked, find and mark the nearest blank square to its right, add 2 to the last number expression on the tape and stop.

We are ready to define I_m:

(x) Subroutine I_m: $1.*, 2.H, \leftarrow, D^m, G, M(4), 3.C2, 4.Y$.

To illustrate how this works, we give a diagram of a simple example (copying the single number 6), with $\downarrow$ indicating the square under scan, $*$ for marked,

- for blank squares:



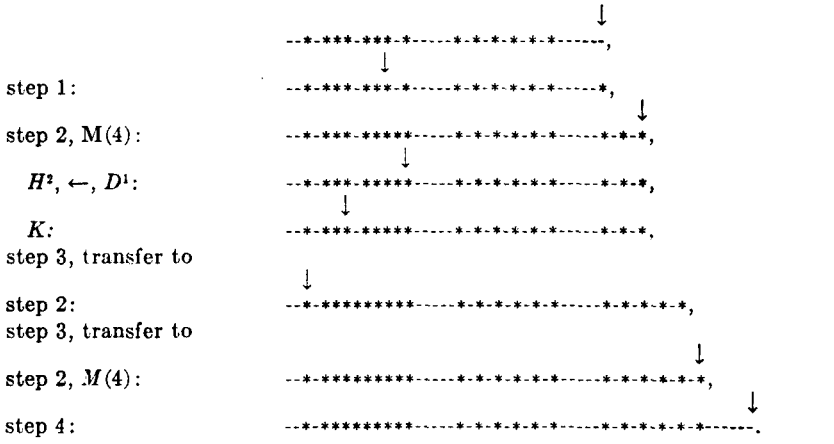
This example should make a little clearer the subroutines G, $M(a)$, and I_m. The primary purpose of $M(a)$ is to enable the machine to copy two units at a time and use up only at most half of the available A-squares in the number expression to be copied, in order to leave room for the making of a second copy, It leads to three different courses of action according as (1) no more left to copy; (2) only one more unit left; and (3) at least 2 more units yet to be copied.

Once we possess I_m, our proof that all functions defined by the schemata (I)–(III) are B-computable is complete. It remains to be shown that from

given B-computable functions only B-computable functions can be generated by the other three schemata. For this purpose, a few more subroutines are necessary, notably the ones for making the second copy of a given number expression.

(xi) Subroutine J_m : $1.*, H^2, \leftarrow, D^m, \leftarrow^2, 2.M(4), H^2, \leftarrow, D^m, K, 3.C2, 4.Y$.

This enables the machine to make a second copy of the m-th (counting from right to left) number expression to the left of the second (counting from right to left) big gap which precedes the square under scan. To illustrate it, we take the last line of the preceding example as the initial tape situation (taking $m = 1$):



We shall often wish to copy not only a single number expression but a succession of number expressions. We easily get two extensions of I_m and J_m.

(xii) $\bar{I}_m$: $I_m , \leftarrow^2, I_{m-1} , \leftarrow^2 , \cdots , \leftarrow^2, I_1$. For example, $\bar{I}_2$ is merely $I_2 , \leftarrow^2, I_1$; $\bar{I}_3$ is $I_3 , \leftarrow^2, I_2 , \leftarrow^2 , I_1$; etc.

(xiii) $\bar{J}_m$: $J_m , \leftarrow^2, J_{m-1} , \leftarrow^2, \cdots , \leftarrow^2, J_1$.

We shall also use, at one place, the notation $\bar{J}_m - J_1$ for the subroutine obtained from $\bar{J}_m$ by omitting $\leftarrow^2, J_1$ at the end.

A slight variant of I_m is to copy $y - 1$ instead of y, the m-th number to the left of the nearest big gap preceding the square under scan.

(xiv) Subroutine L_m : $1.*, 2.H, D^m, \leftarrow^2, G, M(4), 3.C2, 4.Y$.

This differs from I_m only in the insertion of $\leftarrow^2$ in step 2. It is of course only applicable when $y > 1$.

We are now ready to deal with the three remaining schemata.

(IV) If $\chi_1 , \cdots , \chi_m$ and ψ are recursive (resp. B-computable), then the function φ defined by the following schema is also recursive (resp. B-computable):

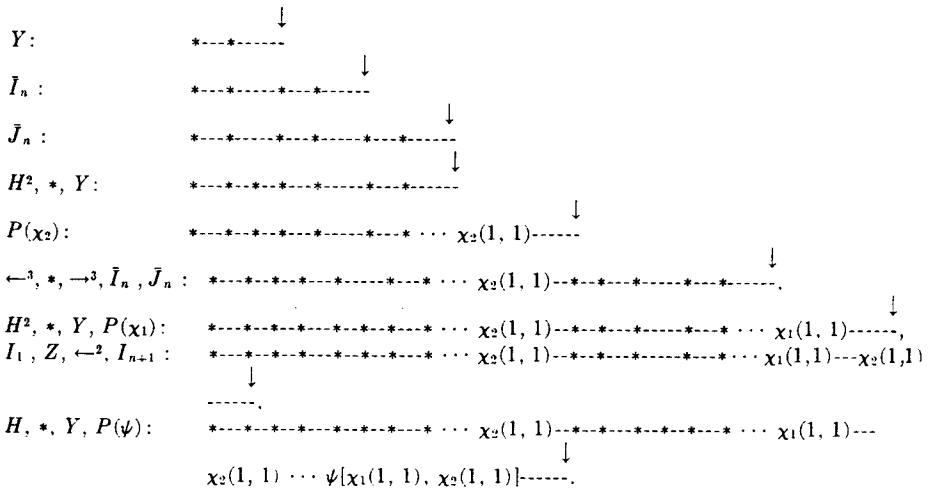
$$\varphi(x_1 , \cdots , x_n) = \psi[\chi_1(x_1 , \cdots , x_n), \cdots , \chi_m(x_1 , \cdots , x_n)].$$

We assume that the programs for $\psi, \chi_1 , \cdots , \chi_m$ are respectively $P(\psi), P(\chi_1), \cdots , P(\chi_m)$, we are to find a program $P(\varphi)$ for the function φ. Intuitively the

program $P(\varphi)$ will do the following: copy the argument values $x_1, \cdots, x_n$ twice, keep the first copy clean and find the value of $\chi_m(x_1, \cdots, x_n)$ with the second copy and the program $P(\chi_m)$, then make two copies from the clean copy of $x_1, \cdots, x_n$, and get the value of $\chi_{m-1}(x_1, \cdots, x_n)$ with one copy and $P(\chi_{m-1})$; repeating the process, we get the values of $\chi_m(x_1, \cdots, x_n), \cdots, \chi_1(x_1, \cdots, x_n)$; copy all of them in the order $\chi_1(x_1, \cdots, x_n), \cdots, \chi_m(x_1, \cdots, x_n)$ and then apply the program $P(\psi)$, we get the value of $\varphi(x_1, \cdots, x_n)$. The program $P(\varphi)$ is as follows (taking for illustration, $m = 2$):

$$Y, \bar{I}_n, \bar{J}_n, H^2, *, Y, P(\chi_2), \leftarrow^3, *, \rightarrow^3, \bar{I}_n, \bar{J}_n, H^2, *, Y, P(\chi_1), I_1, Z, \leftarrow^2, I_{n+1}, H, *, Y, P(\psi).$$

Slight complications are needed in the program to assure that the dispersed values of $\chi_m(x_1, \cdots, x_n), \cdots, \chi_1(x_1, \cdots, x_n)$ can be collected together. We illustrate this program with a simple example with $n = 2$, $x_1 = x_2 = 1$ (incidentally, the use of Y before $P(\chi_1)$ or $P(\psi)$ is redundant but makes the illustrations clearer):



Next we have the schema for primitive recursion:

$$\text{(V)} \quad \begin{cases} \varphi(1, x_2, \cdots, x_n) = \psi(x_2, \cdots, x_n), \\ \varphi(z + 1, x_2, \cdots, x_n) = \chi[z, \varphi(z, x_2, \cdots, x_n), x_2 \cdots, x_n]. \end{cases}$$

Given programs $P(\psi)$ and $P(\chi)$, we are to obtain a program $P(\varphi)$ for the function $\varphi(y, x_2, \cdots, x_n)$. Intuitively, the number of applications of $P(\chi)$ in evaluating $\varphi(y, x_2, \cdots, x_n)$ is unbounded; for a given y, we have to apply $P(\chi)$ $y - 1$ times. This is accomplished by testing successively as we go along whether y is 1, $y - 1$ is 1 or $y - 2$ is 1 or etc. Thus we evaluate $\psi(x_2, \cdots, x_n)$ by $P(\psi)$ and test whether y is 1. If y is 1, we copy $\psi(x_2, \cdots, x_n)$ as answer to $P(\varphi)$; if y is not 1, we evaluate $\chi[1, \psi(x_2, \cdots, x_n), x_2, \cdots, x_n]$ and test whether $y - 1$ is 1. If $y - 1$ is not 1, we evaluate

$$\chi\{2, \chi[1, \psi(x_2, \cdots, x_n), x_2, \cdots, x_n], x_2, \cdots, x_n\}$$

and test whether $y - 2$ is 1. And so on. The program $P(\varphi)$ is:

1. Y, $\bar{I}_n$, $\leftarrow^2$, $*$, $\rightarrow^6$, $\bar{J}_{n-1}$, H^2, $*$, Y, $P(\psi)$, $\leftarrow^2$, I_{n+1}, $\leftarrow^6$,
2. $\leftarrow^2$, $C4$,
3. $\rightarrow^2$, $C5$,
4. $\rightarrow^6$, $\bar{I}_n$, A, Y, L_1, I_{n+2}, $\bar{J}_n - J_1$, H^2, $*$, Y, $P(\chi)$, $\leftarrow^2$, L_{n+1}, $\leftarrow^6$, $C2$,
5. $\rightarrow^4$, I_2, Z.

We give an example with $n = 2$, $y = 3$, $x_2 = 2$.

1. Y: ↓ *-*-*---*-*------

$\bar{I}_n$, $\leftarrow^2$, $*$, $\rightarrow^6$, $\bar{J}_{n-1}$, H^2, $*$, Y: ↓ ***-*---*-*--*--*-*-*---*-*---*------*-*------ (copy y, $x_2, \cdots, x_n$, then write 1, then big gap, then copy $x_2, \cdots, x_n$ again, and then mark first big gap, end up scanning the open).

$P(\psi)$, $\leftarrow^2$, I_{n+1}, $\leftarrow^6$: [***-*---*-*--*--***-*---]*-*---*------*-* $\cdots$ $\psi(x)$---*-*-* (evaluate
(y x y x 1 x y)

$\psi(x_2, \cdots, x_n)$ then copy y and end up scanning the end of y).

2. $\leftarrow^2$, $C4$ (when following step 1, this tests whether y is 1. If y is 1, then follow step 3 and step 5 and copy $\psi(x_2, \cdots, x_n)$, the calculation is complete. If y is not 1, follow step 4: this is the case in the present example).

4. $\rightarrow^6$, $\bar{I}_n$, A, Y, L_1, I_{n+2}, $\bar{J}_n - J_1$, H^2, $*$, Y: [*-*---*--*--*-* $\cdots$ $\psi(x)$---]*-*-*---*-*
(x 1 x y x)

↓
---*-*------*---$\psi(x)$---*-*------ (copy $x_2, \cdots, x_n$, and replace 1 by 2, leave big gap,
(2 1 x)

then copy $2 - 1, \psi(x_2, \cdots, x_n), x_2, \cdots, x_n$, and mark earliest big gap, and end up scanning the open).

$P(\chi)$, $\leftarrow^2$, L_{n+1}, $\leftarrow^6$, $C2$: [*-*-*---]*-*---*-*------*--- $\psi(x)$---*-* $\cdots$ $\chi[1, \psi(x), x]$---*-* ↓
(y x 2 1 x $y-1$)

(evaluate $\chi[1, \psi(x_2, \cdots, x_n), x_2 \cdots, x_n]$ and write down $y - 1$, ready to transfer to step 2 for testing whether $y - 1$ is 1).

2. This now tests whether $y - 1 = 1$. Since $y - 1 \neq 1$ in our example, we repeat step 4 for $y - 2$:

4. $\rightarrow^6, \bar{I}_n, A, Y, L_1, I_{n+2}, \bar{J}_n - J_1, H^2, *, Y$: [*-*---*-*--*--*---$\psi(x)$---*-* $\cdots$ $\chi[1, \psi(x), x]$
(x 2 1 x)

↓
---]*-*---*-*---*-*-*------*-*--- $\chi[1, \psi(x), x]$---*-*------ (copy $x_2, \cdots, x_n$ and replace
($y-1$ x 3 2 x)

2 by 3, leave big gap, then copy $3 - 1$, $\chi[1, \psi(x_2, \cdots, x_n), x_2, \cdots, x_n], x_2, \cdots, x_n$, mark earliest big gap and end up scanning the open).

$P(\chi)$, $\leftarrow^2$, L_{n+1}, $\leftarrow^6$, $C2$: [*-*---]*-*---*-*-*------*-*--- $\chi[1, \psi(x), x]$---*-* $\cdots$ $\chi[2, \chi[1,$
($y-1$ x 3 2 x)

↓
$\psi(x), x], x]$---* (evaluate $\chi[2, \varphi(2, x_2, \cdots, x_n), x_2, \cdots, x_n]$ and write down $y - 2$
(1 $y-2$)

ready to transfer to step 2 for testing whether $y - 2$ is 1).

2. This now tests whether $y - 2 = 1$. Since $y - 2 = 1$ in our example, we transfer to step 3 and then to step 5.

5. This now merely gives $\chi[2, \varphi(2, x_2, \cdots, x_n), x_2, \cdots, x_n]$ as final answer, marks remaining big gap, and ends up scanning the open: [*-*---*-*-*--*--*-*--- $\chi[1, \psi(x),$

x 3 ↓ 2

$x]$---*-* $\cdots$ $\chi[2, \chi[1, \psi(x), x], x]$---*]---$\chi[2, \chi[1, \psi(x), x], x]$------.

x $y-2$

Finally, we have to deal with the schema about the μ-operator ("the smallest number such that"):

(VI) $\varphi(x_1, \cdots, x_n) = \mu_y[\chi(x_1, \cdots, x_n, y) = 1]$.

The construction of $P(\varphi)$ from $P(\chi)$ is somewhat similar to that in the case of schema (V) but simpler. For any given $x_1, \cdots, x_n$, we evaluate first $\chi(x_1, \cdots, x_n, 1)$ and see whether its value is 1. If yes, we stop; if no, we evaluate $\chi(x_1, \cdots, x_n, 2)$. And so on. The program $P(\varphi)$ is as follows:

1. Y, $\tilde{I}_n$, $\leftarrow^2$, $*$, $\rightarrow^6$, $\tilde{J}_n$, $\leftarrow^2$, $*$,
2. H^2, $*$, $P(\chi)$, $\leftarrow^3$, $*$, $\leftarrow^3$, $C4$,
3. $\rightarrow^2$, $C5$,
4. $\rightarrow^6$, $\tilde{I}_{n+1}$, A, $\rightarrow^6$, $\tilde{J}_{n+1}$, A, $C2$,
5. $\rightarrow^4$, I_1, Z.

This completes the proof of the theorem that all recursive functions are B-computable. Thus, every recursive function is defined by a finite succession of conditions such that each condition is of one of the forms (I)–(VI) and, if it is not of one of the forms (I)–(III), the function symbols involved on the right-hand side of the condition are introduced in previous conditions. Hence, we are to assemble the programs for these conditions in a corresponding order to get the program for the final recursive function introduced by the set of conditions. Since this assembling process is mechanical, we could, if we wish, design a program to assemble the subroutines (I)–(VI) for any given definition of some recursive function. That would be a fairly simple form of "automatic programming". We shall, however, not delay over this.

We shall call a function a B-computable complete function if the function is defined for all argument values. It follows from the coextensiveness of B-computable and (partial) recursive functions that B-computable complete functions are exactly the general recursive functions.

We digress to comment on recursive definitions as they are given in mathematical logic. Just as we define speed functions for programs of B and their corresponding functions in terms of the time taken (number of steps needed) to evaluate any given input tapes, we can also define speed functions for recursive functions introduced by the schemata (I)–(VI) according to the number of steps needed to evaluate the function from a given definition, by certain standard rules (Kleene [2], p. 264, p. 326). Corresponding to each set of equa-

tions E which defines a recursive function $f(n)$, there is some speed function $f_s(n)$ which, for each constant n_0, gives an upper bound to the number of lines (derived equations) needed for deriving logically the value of $f(n_0)$ from E. Obviously each recursive function has many definitions and each definition (set of equations E) has many corresponding speed functions. It is a rather simple matter to give a corresponding speed function for each function defined by the schemata (I)–(VI). Call them (I_s)–(VI_s).

$$
\begin{aligned}
&(I_s)\ \phi_s(x) = 1,\\
&(II_s)\ \phi_s(x_1, \cdots, x_n) = 1,\\
&(III_s)\ \phi_s(x_1, \cdots, x_n) = 1,\\
&(IV_s)\ \phi_s(x_1, \cdots, x_n) = (\chi_1)_s(x_1, \cdots, x_n) + \cdots + (\chi_m)_s(x_1, \cdots, x_n) + m + 1 + \\
&\qquad \psi_s[\chi_1(x_1, \cdots, x_n), \cdots, \chi_m(x_1, \cdots, x_n)],\\
&(V_s)\ \begin{cases} \phi_s(1, x_2, \cdots, x_n) = \psi_s(x_2, \cdots, x_n) + 1, \\ \phi_s(z+1, x_2, \cdots, x_n) = \phi_s(z, x_2, \cdots, x_n) + 1 + \chi_s[z, \phi(z, x_2, \cdots, x_n), x_2, \\ \qquad \cdots, x_n] + 1, \end{cases}\\
&(VI_s)\ \phi_s(x_1, \cdots, x_n) = 1 + \sum_{i=1}^{\mu_y[\chi(x_1, \cdots, x_n, y)=1]} [\chi_s(x_1, \cdots, x_n, i) + 3].
\end{aligned}
$$

This digression brings out the point that in many cases although a function is B-computable or recursive, we have no idea in advance how long it will take before the machine grinds out the answer, or the equations yield the desired line for each given argument. This is so even if we are only concerned with B-computable complete functions or general recursive functions. With regard to recursive functions, the indeterminate element is concentrated in schema (VI) and in the corresponding (VI_s). In order that a recursive function defined by (VI) be general recursive, there is the requirement that for all $x_1, \cdots, x_n$, *there exists* y, such that $\chi(x_1, \cdots, x_n, y) = 1$. The condition, however, gives in general no information as to how big a number y could satisfy the equation $\chi(x_1, \cdots, x_n, y) = 1$, for given $x_1, \cdots, x_n$.

The definitions for speed functions are of course not directly applicable to digital computers actually in use. It is, nonetheless, thought that these idealized definitions might give some clue to the study of more practical cases.

4. *Basic instructions*

From our definition of a program, if we write only a single instruction in each line, there are clearly, for every n, n^{n+4} possible $(n+2)$-lined programs, since in each line, the instruction can be any one of: $\rightarrow$, $\leftarrow$, $*$, $C1, \cdots, C(n+1)$, and the last two lines are always $n+1.\rightarrow$, $n+2.\leftarrow$.

There are other possible basic instructions which we have avoided to use. Chief among them are erasing E, the unconditional transfer Ux which instructs the machine to follow instruction x independently of the content of the square under scan, and the dual conditional transfer $C'x$ which instructs the machine to follow the instruction x when the square under scan is blank and to follow the next instruction otherwise. Of these, erasing introduces a new basic act while the transfers do not. If we leave out E, there are theoretically 16 possible types of basic instruction: if the square under scan is marked, we can do four different

kinds of things, viz., $\rightarrow$, $\leftarrow$, $*$, follow instruction x for some given x; similarly if the square under scan is blank. Accordingly we have:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$*$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$*$	$*$	$*$	$*$	$\leftarrow$	$\leftarrow$	$\leftarrow$	$\leftarrow$	w	w	w	w
$-$	$\rightarrow$	$*$	$\leftarrow$	x	$\rightarrow$	$*$	$\leftarrow$	y	$\rightarrow$	$*$	$\leftarrow$	z	$\rightarrow$	$*$	$\leftarrow$	u

The variables x, y, z, w, u can in turn take indefinitely many different values. Given N, there will be $(3 + N)^2$ possible different basic instructions in any program with N lines and hence $N^{(3+N)^2}$ possible N-lined programs. The basic steps $\rightarrow$, $*$, $\leftarrow$ are respectively 1, 6, 11 in the above table, while the unconditional and the two conditional transfers are special cases of 16. In general, if we use M basic symbols (instead of just the single symbol $*$), we shall have $M + 2$ basic acts ($\leftarrow$, $\rightarrow$ and printing each symbol), $M + 1$ possible square contents, $(M + 3)^{M+1}$ possible types of basic instructions, $(M + 1 + N)^{M+1}$ possible basic instructions for programs with N lines. If we allow erasing, the numbers will be $M + 3$, $M + 1$, $(M + 4)^{M+1}$, $(M + 2 + N)^{M+1}$ respectively.

We shall confine our attention to the machine B and the seven simple types of instructions Cx, $C'x$, Ux, $\rightarrow$, $\leftarrow$, $*$, E applicable to it. A corollary of what we have proved could be described very roughly as: computations which can be performed with the help of $C'x$, Ux, E can already be done with $\rightarrow$, $\leftarrow$, $*$, Cx alone. This of course does not mean that given $\rightarrow$, $\leftarrow$, $*$, Cx, everything which can be done with $C'x$, Ux, E can also be done without them.

Indeed, there are many things which we can do when we permit erasing but which we cannot do otherwise. Erasing is dispensable only in the sense that all functions which are computable with erasing are also computable without erasing. For example, if we permit erasing, we can set up programs in such a way that if the machine begins by scanning the argument values of a function on the tape, by carrying out the program, only the function value (the answer) appears on the tape at the end of the operation, everything else having been erased. Such is obviously impossible in general, if we do not permit erasing. For example, if the answer happens to be shorter than the question.

There are simple things which can be done with the help of $C'x$ or Ux, but which Cx, $\rightarrow$, $\leftarrow$, $*$ alone cannot do. Suppose we know there is a marked square somewhere to the left of the square under scan but do not know how far away it is. We cannot construct a program with Cx, $\rightarrow$, $\leftarrow$, $*$ which will always enable us to find the nearest marked square on the left without introducing new marked squares, because given any explicit program with (say) n lines, the reading head cannot arrive at any marked square if there are n or more blank squares between the square under scan and the nearest marked square. This is so because only $\leftarrow$ can carry the reading head leftward, Cx cannot produce any periodic action unless some marked square were encountered. Yet it is easy to do the thing either with $C'x$ alone or with Ux and Cx. Thus: $1.\leftarrow$, $2.C'1$, this will stop when and only when a marked square is encountered. Or: $1.\leftarrow$, $2.C4$, $3.U1$, $4.\rightarrow$, $5.\leftarrow$; the steps 4 and 5 are of course just to say "stop". Similarly, using $C'x$ only we cannot always find the nearest preceding blank if we do not know how far away it is.

Given Ux and Cx, $C'x$ is dispensable; given Ux and $C'x$, Cx is dispensable; given Cx and $C'x$, Ux is dispensable. Suppose we have a program with k lines and there is a line: $m.C'n (m, n \leqq k)$. We can eliminate $C'n$ by modifying the line $m.C'n$ and the lines following it: if $n \geqq m$, replace $m.C'n$ by $m.C(m + 2)$, $m + 1.U(n + 1)$, and increase the index of every later line by 1 (replace $m + i$ by $m + i + 1$); if $n < m$, replace $m.C'n$ by $m.C(m + 2)$, $m + 1.Un$, and increase the index of every later line by 1. In the unusual case when $k = m$, we have to add two more lines: $m + 2.\rightarrow$, $m + 3.\leftarrow$. Similarly, we can eliminate Cx by using Ux and $C'x$. The elimination of Ux by Cx and $C'x$ is also similar: we simply replace $m.Un$ by $m.Cn$, $m + 1.C'n$ (or $m.C(n + 1)$, $m + 1.C'(n + 1)$) and readjust later lines accordingly.

It is of interest to know that there are simple things which we cannot do even with C, C', and U. Assume we know that there is somewhere on the tape a marked square and the reading head is scanning a blank square on the tape; we wish to find a program which will always enable us to find a marked square and stop there, but introduce no new marked squares. The natural thing to suggest would, for example, be: $\rightarrow$, $\leftarrow^2$, $\rightarrow^3$, $\leftarrow^4$, etc.; in other words, look left and right in alternation. We wish to prove that no program using Cx, $C'x$, $\rightarrow$, $\leftarrow$, $*$, Ux can do this. Clearly, $*$ should be left out since we do not allow the introduction of new marked squares. We need not consider Ux, since we can replace its applications by applications of Cx and $C'x$. Assume there is a given arbitrary N-lined program with $\rightarrow$, $\leftarrow$, Cx and $C'x$. We wish to prove that it cannot do what we want. Suppose it does not contain a line of the form $m.C'n$ with $n < m$. If there are more than N squares between the nearest marked square and the square initially under scan, then such a program cannot carry us to a marked square. Hence, the only interesting case is a program which contains a line of the form $m.C'n$, $n < m$.

Let P be an arbitrary such program. We proceed to prove that there is an initial tape situation such that the initial tape contains a marked square but P does not enable us to find it. Assume P contains N lines among which K are of the form $m.C'n$, $n < m$. We consider first how P would behave when facing initially a completely blank tape. There are two possibilities. In carrying out the program P, either there is one of the K lines of the form $m.C'n$ $(n < m)$ which we encounter more than once, or there is none. In the second case, since only such lines enable us to return to previous lines, the whole program will be carried out in no more than $(K + 1)N$ steps. Hence, if there are more than $(K + 1)N$ squares between the square initially under scan and the marked square, the program P does not enable us to find the marked square. In the other case, let $m_1.C'n_1$ $(n_1 < m_1)$ be the line which, in carrying out the program, we encounter a second time at the earliest stage. We have taken no more than $(K + 1)N$ steps before we encounter $m_1.C'n_1$ for the second time. Moreover, since we are assuming a tape initially blank, the steps from the first visit of the line $m_1.C'n_1$ to the second will forever repeat themselves completely. Call the square under scan at the first visit the origin. In the process of carrying out the program between the two visits, there will be (say) scanned N_1 squares to the left of the origin, N_2 to the right, and the reading head will be scanning at the

second visit a square (the "new origin"), N_3 squares to, say, the right of the origin. We know that $N_1, N_2, N_3 < (K + 1)N$. Since the "origin" will keep on shifting to the right, we see that if the initially marked square is more than $2(K + 1)N$ to the left of the square initially under scan, the program P does not enable us to find the square.

If erasing is permitted, then we have much more freedom. For example, we can design a program with $*$, $\rightarrow$, $\leftarrow$, Cx, E only which will enable us to find some marked square if there is anywhere a marked square on the tape. Essentially, we begin with the square under scan and go left and right in alternation ($\leftarrow$, $\rightarrow$, $\leftarrow^2$, $\rightarrow^2$, $\leftarrow^3$, $\rightarrow^3$, etc.), marking each blank square on the left (resp. right) side if the square immediately preceding (resp. succeeding) it is also blank. When we find a marked square beyond the portions we have marked, we eliminate the marks we introduce and go back to the marked square. The routine has 42 lines:

1.$C41$, 2.$\leftarrow$, 3.$C41$, 4.$\rightarrow$, 5.$\rightarrow$, 6.$C41$, 7.$\leftarrow$, 8.$*$, 9.$\leftarrow$, 10.$\leftarrow$, 11.$C24$, 12.$\rightarrow$, 13.$*$, 14.$\rightarrow$, 15.$C14$, 16.$\rightarrow$, 17.$C33$, 18.$\leftarrow$, 19.$*$, 20.$\leftarrow$, 21.$C20$, 22.$\rightarrow$, 23.$C9$, 24.$\rightarrow$, 25.$\rightarrow$, 26.$C25$, 27.$\leftarrow$, 28.E, 29.$\leftarrow$, 30.$C28$, 31.$\leftarrow$, 32.$C41$, 33.$\leftarrow$, 34.$\leftarrow$, 35.$C34$, 36.$\rightarrow$, 37.E, 38.$\rightarrow$, 39.$C37$, 40.$\rightarrow$, 41.$\rightarrow$, 42.$\leftarrow$.

Given erasing, we can also derive Cx and $C'x$ from each other. We prove how it is possible that given any program using $\rightarrow$, $\leftarrow$, $*$, Cx, $C'x$, Ux, E we can find a program using $\rightarrow$, $\leftarrow$, $*$, Cx, E which performs the same function. As we have shown before, we can eliminate Ux by Cx and $C'x$. Now we wish to eliminate also $C'x$. Take the first occurrence of $C'x$ in the given program. Suppose the line is $m.C'n$ and suppose $n < m$. We construct the new program by (i) leaving the lines 1 to $n - 1$ unchanged; (ii) replace lines n and $n + 1$ by: $n.C(n + 2)$, $n + 1.E$; (iii) renumber the original lines n to $m - 1$ as $n + 2$ to $m + 1$ respectively; (iv) replace the original line m by: $m + 2.C(m + 5)$, $m + 3.*$, $m + 4.C(n + 1)$; (v) renumber all the original lines $m + i$ as $m + i + 4$; (vi) adjust references to these lines in other conditional transfers accordingly. We leave it to the reader to verify that the new program does the same thing as the original. If $n \geqq m$, similar constructions can be made. Repeating the process, we can eliminate all occurrences of $C'x$.

It appears unlikely that we could delete any of the four types of basic operations $*$, Cx, $\rightarrow$, $\leftarrow$ without adding other operations instead, and still compute all recursive functions. To give an exact proof of the indispensability of each of the four types of instruction, we could proceed as in §2: represent all possible tape contents by positive integers, consider all the possible programs obtained by using only three of the four basic types of instruction, survey all the possible transformations which these programs can perform on initial input tapes, and prove that they do not include all recursive functions. For instance, suppose we use only $*$, Cx, and $\rightarrow$. Consider, for simplicity, just functions of one argument. First, we use a simple function $g(x)$ which maps the set of all possible input tape contents (or a simple recursive subset of it) into the set of positive integers. Then we need another simple function $h(x)$ which maps all the possible final output tape contents into the set of positive integers. Analogously with the functions θ_τ, τ_τ in §2, we can define a function $\Delta_\tau(x)$ such that $\Delta_\tau(g(\iota))$ gives the number

of the output content for an arbitrary program made out of $*$, Cx, $\rightarrow$, with number π, and an arbitrary input tape content with number ι. It then follows that, for every program Π (with number π) on such a machine, $h[\Delta_\pi(g(\iota))]$ is the function computed by it. It could then be shown, on account of the form of $\Delta_\pi(x)$, that only very simple recursive functions can be represented in the form $h[\Delta_\pi(g(\iota))]$. For example, if h, g are primitive recursive, then all one-placed general recursive functions which can be expressed in the form $h[\Delta_\pi(g(\iota))]$ are primitive recursive. Instead of trying to work out details of the argument, we present intuitive arguments relative to the particular way of representing questions and answers we use.

The necessity of $*$ can be argued on perfectly general ground. Thus, it seems reasonable to represent the same argument values in the same way even when we are concerned with different functions. Moreover, to be unambiguous, we must not interpret the same final tape output as different answers for different questions. This excludes the possibility of reading answers directly from the questions (data) initially given on the tape. For example, given 3 and 5, we may wish to get the value of $3 + 5$ or the value of 3^5. If the programs for addition and exponentiation yielded the same final tape situation for the given argument values 3 and 5, we would not be allowed to say, "well, it depends on how you read the result, it is $3 + 5$ if you read it one way, etc." This condition is sufficient to indicate that $*$ is indispensable. Without $*$, the answers must always be the same as the questions are the same as one another for the same argument values.

If we use only $\rightarrow$, $\leftarrow$, $*$, we would get the same changes from the initial tape content for all different argument values of the same function, as we exercise no judgment over the initial question and make no choice from the possible alternative courses. Roughly then a program for a function would only serve as a name for the function rather than provide a method for computing its values. Take, for example, the function $m \times n$. There is no program for it, because given any program made out of $\rightarrow$, $\leftarrow$, $*$, it has only a fixed number N of occurrences of the instruction $*$, and, for any given m and n, no more than $m + n + N$ $*$'s can occur on the final tape, although we can easily find m_1 and n_1 such that $m_1 \times n_1 > m + n + N$.

To prove that $\leftarrow$ is indispensable, we assume that we have only $\rightarrow$, $*$, Cx on the machine. Such a machine would have a rather restricted memory since the reading-writing head cannot go back (leftwards) and consult what it did to previous squares. Assume a given program π. At each square, the head can only do two things: $*$ and $\rightarrow$. Since we assume that the initial tape contains only finitely many marked squares, if we let the head begin by scanning a square within the marked region, the result of performing Π will either have the head stop within the marked region of the input tape or have it continue to go in some periodic manner or have it operate on an additional portion which has a bound that depends only on Π but is independent of the initial tape situation (i.e., the given argument values). Consider again the function $m \times n$. Suppose a program Π of N lines is given which is intended to compute it, the head begins by scanning a square in the printed portion of the input tape. If the head stops

within the initially printed portion, we of course do not get the value of $m \times n$. Assume, therefore, the head has left that region and is scanning a blank to the right of that region and is going to perform line k of the program II. The interesting point is that, once the head is scanning the open, no single conditional transfer Cx in II can function twice without introducing a circular loop. Thus, if the i-th line is Cx and it is to function twice, then each time it must be scanning the rightmost marked square and goes to the x-th line in II. Since there is no $\leftarrow$, if the machine comes back to the i-th line again and the square under scan is again marked, it must repeat the whole process and come back to the i-th line once more scanning a marked square. If no conditional transfer can function twice, then since there are at most N conditional transfers in II which has only N lines, and since transfer can at most make the machine repeat N lines, there can be at most N^2 steps and at most N^2 new $*$'s can be marked in the new region of the tape, provided we wish to exclude circular loops. Since no circular loop should occur in a program for $m \times n$ and since we can always find m_1, n_1 such that the value of $m_1 \times n_1$ is greater than N^2, there is no program with $\rightarrow$, $*$, Cx only which can compute the function $m \times n$.

For similar reasons, $\rightarrow$ is indispensable; and also we cannot compute all recursive functions with just Ux $\rightarrow$, $\leftarrow$, $*$ (replacing Cx by the unconditional transfer). It is, however, not known to the author whether $C'x$, $\rightarrow$, $\leftarrow$, $*$ are sufficient in some nontrivial sense. For instance, if we use the method of representing computations developed in the preceding section, it is not clear how $C'x$ can enable us to go through an indefinitely long string of marked squares or whether that is not necessary. One might wish to reverse the roles of blanks and marked squares; but then we have to replace $*$ by E. Rather trivially, E, $\rightarrow$, $\leftarrow$, $C'x$ give a sufficient set from duality considerations.

We note that there is an interesting machine (call it "machine W") which is closely related to machine B but much easier to use: add erasing, so that W has five types of basic operations, viz. $\rightarrow$, $\leftarrow$, $*$, Cx, E. As we have shown above, the instructions Ux and $C'x$ can be derived from these so that we can freely use them too. That all recursive functions are computable on such a machine follows directly from our result on B-computability. Indeed, from Kleene [2] and our constructions in the preceding section, it is fairly obvious that for machine W, we do not need the auxiliary squares, and we can so arrange the matter that in computing a recursive function for given arguments, if the machine stops at all, it will stop with a string of marked squares whose number is exactly the answer sought. In this way, we get a more or less unique natural normal representation of questions and answers. For most purposes, it would seem more attractive to use machine W than machine B.

It remains an open question whether we can dispense with auxiliary squares and still be able to compute all recursive functions by programs consisting of only basic steps $\rightarrow$, $\leftarrow$, $*$, Cx. Of course, it is not necessary to use every other square as the auxiliary square. If we do not mind complications, we can take any fixed n and use every n-th square as the auxiliary square.

5. *Universal Turing machines*

The basic machine B (or the machine W) is fictitious not only in that it assumes an indefinitely expandable tape (which may be viewed as a serial storage), but also because no finite internal storage (a parallel storage) is adequate for storing every program that is involved in proving that all (partial) recursive functions are B-computable. In the proof, we allow ourselves the privilege of using, for every n, programs with more than n instruction words. To many, this assumption of an indefinite parallel storage whose units all are accessible at any moment is even more repulsive than permitting the tape to expand as needed.

One way to get around this difficulty about storage units is to use, instead of the general-purpose machine B for all recursive functions, one special-purpose machine M_i for each recursive function f_i. Thus, for each function f_i, we have available a program for the machine B which will compute values of the function by following the program. Instead of using the storage unit to keep the instructions, we can construct a machine M_i which will carry out the particular program automatically every time it is scanning a tape. Such special-purpose machines could be machines for doing just addition, machines for doing just multiplication, and so on. These correspond more closely than B to what are known as Turing machines.

There is also a method of using a single machine with a finite internal storage unit on which all B-computable functions can be computed. This is by using what Turing calls a universal machine. Intuitively it is very plausible that we can design such universal machines, because, since we have one infinite serial storage (the tape) anyway, we can trade the unbounded internal storage for additional tape by having programs for particular functions stored on the tape.

On the basic machine, B several different programs can, of course, compute the same function in the sense that confronted with the same initial argument values, the programs always yield the same rightmost number on the tape. Therefore, since we can build a corresponding special-purpose machine for each program, we may also have many structurally different machines which correspond to the same function.

Roughly, a universal machine is one which can do what every special-purpose machine does. This it does by imitating in a uniform manner each special-purpose machine M_i and is therefore, different from the basic machine B which permits flexibility in the programs. A universal machine is a machine built specially for the purpose of imitation. We may either store a fixed program in the internal storage of machine B once and for all or build the program into the machine and dispense with the internal storage unit altogether. The universal machine is like a special-purpose machine in that it has only a serial storage (the tape); it is, nevertheless, general-purpose by having the programs for individual functions transported to the tape.

It is possible to construct a universal machine U which is again only capable of the four types of basic instructions. Or to put it differently, it is possible to write up a program for the machine B such that when scanning a tape which

contains in addition to the argument values of a function, also a number which represents (a program or its corresponding special-purpose machine for) the function, the machine B, by following the program, will imitate the behavior of the particular machine for the function and compute the value of the function for the arguments given on the tape.

Since many special-purpose machines may correspond to one function, it is desirable to distinguish a functional universal machine and a structural one: the former is able to imitate all special-purpose machines in the sense that there is an effective method of representing every special-purpose machine (or even just at least one machine for each function) on the tape such that afterwards it will yield the same answer for the same question, while the latter is able to do more in that it also imitates the moves (or the carrying out of the program steps) of the special-purpose machine in question. It follows that a functional universal machine need only imitate at least one of the many possible special-purpose machines (or programs) for each function to the extent of yielding the same answers to the same questions. For instance, on a functional universal machine it is permissible to represent all different special-purpose machines corresponding to the same function by the same initial tape situation. Every structural universal machine is a functional one, but not vice versa. Actually it is a little harder to construct a structural universal machine, while a functional universal machine can be derived more or less as an immediate corollary of the results of §3 and known results in recursive function theory. We shall leave the former which is quite complex for a separate paper and discuss merely the latter here. Either of these will yield a positive solution to the question which Professor G. W. Patterson raised in conversation: is it possible to design a one-tape nonerasing universal Turing machine? As a matter of fact, he probably had in mind a direct construction of a structural universal machine.

Since the B-computable functions are all and only the partial recursive functions, it seems sufficient to use the following result of recursive function theory: there is an effective correlation of all definitions for recursive functions with positive integers (called their Gödel numbers) such that for every n, there is a recursive function $V_n(z, x_1, \cdots, x_n)$, often written $U(\mu_y T_n(z, x_1, \cdots, x_n, y))$, which has the property that for every n-placed recursive function $f_e(x_1, \cdots, x_n)$ with the Gödel number e, $V_n(e, x_1, \cdots, x_n)$ coincides with $f_e(x_1, \cdots, x_n)$, for all the argument values $x_1, \cdots, x_n$ for which f_e is defined (Kleene [2], p. 330). Since the functions $V_n(z, x_1, \cdots, x_n)$ are recursive, they are B-computable. We can therefore find a program on the machine B or alternatively construct a special-purpose machine for each $V_n(n = 1, 2, \cdots)$ which will compute every n-placed recursive function once we put its Gödel number on the input tape.

An obvious defect of this is that we do not have a really functional universal machine, but one for all singulary B-computable functions, one for all binary functions, etc. It is, however, known that every recursive function can be reduced to a singulary function because we can effectively enumerate for every n all n-tuples of positive integers and introduce recursive functions $[\ ]_1, \cdots, [\ ]_n$, so that $[x]_i$ is the i-th number of the x-th n-tuple, and we can reduce an

n-placed function $f(x_1, \cdots, x_n)$ to the one-placed function $f([x]_1, \cdots, [x]_n)$. One might, therefore, wish to say that $V_1(z, x)$ is already a universal function except for the slight drawback that functions with different arguments, which we ordinarily regard as different, may become indistinguishable. Thus, for example, two different functions $g(x, y)$ and $h(x, y, z)$ may satisfy the conditions $g([x]_1, [x]_2) = f(x)$ and $h([x]_1, [x]_2, [x]_3) = f(x)$, for the same function $f(x)$; then $f(x), g(x, y), h(x, y, z)$ become indistinguishable.

To remedy this, we seem to need a recursive or B-computable function with indefinitely many arguments. This is a rather natural notion if we think in terms of the machines (say, the machine B). Given an input containing a sequence of number expressions, the machine will first count the number of number expressions and then proceed in a uniform manner to calculate the function value, taking into consideration the number of arguments initially given. On the other hand, it is also not unnatural to reconstrue the schemata (II)–(VI) for defining recursive functions to include cases with indefinitely many arguments, For instance, we may wish to use $\sum_1^n x_i$ or $\prod_1^n x_i$ for indefinite n. Indeed. it can be proved exactly that if we allow for functions with indefinitely many arguments, recursiveness and B-computability are still coextensive. Thus, if we permit each function in the schemata (II)–(VI) to contain indefinitely many parameters, we can still, with some extra care, prove that all recursive functions in the extended sense are B-computable in the sense that for each such function $f(x_1, \cdots, x_n)$, we can get a program of machine B so that given $x_1, \cdots, x_n$ for arbitrary n on the tape, we can get by the program the value of $f(x_1, \cdots, x_n)$. Since this is not important for our principal result, we shall not enter into details. In the paper on the structural universal machine, we shall have occasion to see more explicitly how a function with indefinitely many arguments can be computed.

Meanwhile, we assume given a recursive function $g_n(x_1, \cdots, x_n)$ (and a program for computing it) whose value is k if and only if for every $n, x_1, \cdots, x_n$, $(x_1, \cdots, x_n)$ is the k-th n-tuple (of positive integers). Then we define $V(z, x_1, \cdots, x_n) = V_1(z, g_n(x_1, \cdots, x_n))$. There is then a program on the machine B for computing V, since we have programs for V_1 and g. The special-purpose machine realizing the program for V is then a functional universal machine.

Each special-purpose machine, including the functional universal machine for V, as well as the structural universal machine not described here, can be realized physically, for example, by modifying the specification of an idealized computer in Burks and Copi [3]. In each case, we can either construct the machine B (or the machine W) with its internal storage suitably restricted and store the program for the particular machine in the internal storage once and for all, or simply construct a physical realization of the program and dispense with an internal storage altogether. So far as the physical realization of a universal Turing machine is concerned, the one proposed by Moore [4] requires much less physical equipment than either of the two universal machines we envisage. Ours are of interest in that we have reduced the number of different types of basic in-

structions to a bare minimum. It should be clear that any actual general-purpose digital computer could be viewed as a realization of the machine B (or W), provided we imagine that the internal storage could be expanded as much as we wish.

6. *Proving machines*

In conclusion, we permit ourselves to speculate a bit wildly and make a few idle general comments.

While mathematical logic had often been criticized for its uselessness, most professional logicians do not seem to have been overwhelmed by the extensive application of logic to the construction and use of computing machines in recent years. There is a strong feeling that the useful part of logic does not coincide or even overlap with the interesting part; or even a suspicion that what is interesting in logic is not useful, what is useful is not interesting. Yet it cannot be denied that there is a great deal of similarity between the interests and activities of logicians on the one side and designers and users of computers on the other. Both groups are interested in making thoughts articulate, in formalization and mechanization of more or less vague ideas. Certainly logicians are not more precise and accurate than the machine people who are being punished for their errors more directly and more vividly. Just as logicians speak of theorems and metatheorems, there are programs and metaprograms. Just as logicians distinguish between using and mentioning a word, automatic coding must observe the distinction between using an instruction and talking about it. Just as logicians contrast primitive propositions with derived rules of inference, there is the distinction between basic commands and subroutines. Shouldn't there be some deeper bond between logic and the development of computers?

What strikes the eye but is probably not of much theoretical interest is the possibility of using machines to perform known decision procedures.

It is known that in a number of domains of logic and mathematics, there are decision procedures, i.e., effective procedures by which, given any statement in these domains, we can decide in a finite number of steps whether or not it is a theorem. Examples include elementary geometry and algebra, arithmetic with merely addition or merely multiplication, theory of truth functions, monadic predicate calculus. Intutitively, to have an effective procedure for decision amounts to the possibility of constructing a machine to make the decision. It is theoretically possible, for each of the known decision procedures, to construct one machine to perform it. For example, this has been done for the theory of truth functions. And it is only economic and engineering considerations which have thus far prevented the construction of machines to perform the other decision procedures. There is, therefore, the fairly interesting problem of constructing, for instance, a machine for monadic predicate calculus. This would be more properly a "logic machine" because, as is often asserted, all syllogistic inferences can be carried out in monadic predicate calculus.

A related, and probably more practical, problem is to try to program these

decision procedures on the ordinary general-purpose computers. These can perhaps lead to two kinds of useful results: (a) a library of subroutines for all or most of the known decision procedures that is ready for use, sometimes with slight modifications, for most ordinary computers now in existence; (b) certain operations which are often needed in programming the decision procedures may lead to the addition or modification of the basic operations in computing machines; for example, it may lead to the replacement of one or more operations by a more useful new operation.

The trouble with these questions is not that they are inhumanly difficult but rather that they are neither urgent for practical purposes nor intellectually sufficiently exciting. In contrast, the questions involved in the imitation of mind by machine or in the attempt to study the philosophy of mind by comparing mind with machine are surely fascinating but quite often we cannot even formulate the problems clearly, or, when we have more specific problems such as mechanical translation or chess-playing, there is little basic conceptual difficulty but a good deal of "engineering" tasks which require a large amount of skilled labour.

What has been discussed less frequently is the possibility of using machines to aid theoretical mathematical research on a large scale. One main contribution of mathematical logic is the setting up of a standard of rigour which is, at least by intention, in mechanical terms. Thus, we learn from mathematical logic that most theorems of mathematics can be proved mechanically within certain axiom systems which are formalistically rather simple. It follows that most mathematical problems can be viewed as inquiring whether certain statements follow from certain axioms, or, since proofs and theorems of any given axiom system can be enumerated effectively, whether certain statements occur in such enumerations. In other words, while in a computation problem, we ask what the value of a computable complete function $f(x)$ is, for some given value of x; in a provability problem, we ask whether there exists some x, such that $f(x)$ is the given number. In other to answer the first question for a given number n_0, we need only carry out the procedure for $f(x)$ as applied to n_0; in order to answer the second question for a given n_0, we have to compute $f(1)$, $f(2)$, $f(3)$, etc., until we arrive, by luck, at a number x_0 such that $f(x_0) = n_0$. If it happens that n_0 is not a theorem, we shall never be able to stop in our search for n_0 from the values of $f(1)$, $f(2)$, etc. In terms of recursive function theory, our question is to ask whether a number belongs to a given recursively enumerable set (i.e., the set of all numbers y such that there exists some x, $f(x) = y$, where f is a given general recursive function).

Incidentally, in our previous considerations, we have assumed that at each stage only finitely many squares are marked. If we permit arbitrary distributions of infinitely many initially marked squares on the tape, we can, roughly speaking, compute values of a function recursive relative to the arbitrary function determined by the initial tape situation (we have, in Turing's terminology, an oracle machine).

To come back to the function $f(x)$ which enumerates theorems, since it is

general recursive, it is by our main theorem a B-computable complete function and has a program $\Pi(f)$ on the machine B. It is not hard to devise a new program, using $\Pi(f)$, which will test for every given number n_0 whether there exists some x, such that $f(x) = n_0$. Roughly the program does this: write down the number n_0 on the tape and, after a big gap, the number 1, apply $\Pi(f)$, and then compare its result with n_0, if same, stop and write 1, otherwise, write 2 (in general, copy preceding argument value and add 1), and apply $\Pi(f)$, compare result with n_0, if same, stop and write 1, otherwise, etc. Since we can define a procedure on the machine B for comparing two numbers, we can get the desired program. In this way we get a partial recursive function $f_e(y)$ such that $f_e(y) = 1$ if and only if there exists a number x, $f(x) = y$; otherwise, $f_e(y)$ is undefined. Thus, given any axiom system (e.g., for arithmetic, or for set theory, or for analysis), we can devise a program or construct a special-purpose machine so that feeding any statement in the notation of the system into the machine will eventually produce an answer 1, if and only if the statement happens to be a theorem.

The standard objections against studying mathematics by such machines are twofold: since our interest is to decide whether a particular statement is a theorem, the method is in general futile as we can never run through the infinitely many arguments for the function f and unless we happen to be fortunate to get a positive result, we shall never know the answer no matter how long we run; secondly, even when the answer happens to be yes, it will be usually a long, long time before we hit upon the answer, since we have to examine all the possible proofs. In reply, we may observe that there are often rather modest aims of research which can be aided more directly, and even in the questions of discovering more ambitious theorems, there is no reason why shortcuts in the testing procedure, elimination of superfluous cases, and technological advancement will not bring the tasks within the range of practical feasibility.

So far as modest research goals are concerned, we may give a few examples. A good deal of brain power has been spent on studying the independence of axioms in the propositional calculus. Apparently there is in general no decision method for such questions. Yet in most cases the independence could be proved by matrices of a few rows and columns (usually less than 5 rows and 5 columns), or disproved by rather short derivations from the initial axioms and rules of inference. Since the matrices and the derivations can be listed and tested mechanically, there is little doubt machines can greatly aid such researches. More generally, we often wish to test whether an alleged proof is correct. If the proof were presented in full detail as in some books on mathematical logic, a machine test would be immediate. The problem becomes more interesting when, as is usual, the alleged proof is only presented in sketch. The situation is rather like (say) picking 20 lines from a proof of 1000 lines. The problem is more or less one of reconstructing 1000 lines from the 20 lines which are given. When an alleged proof is wrong, we can no longer reconstruct a proof out of the 20 lines. Yet, if, for example, we can handle all proofs in a system with less than 100 lines, then it is very probable that we can handle most proofs with 1000 lines when we are given a summary of 20 lines for each proof. Or again, sometimes

we suspect that a statement is either a theorem with a short proof or not a theorem at all. Sometimes we feel that a certain statement is a theorem and that certain statements are necessary in its proof. In such cases, the machine usually only has to accomplish a more restricted task in order to confirm and disprove our hunches. In many cases, we should not ask the machine to check all possible cases, but just to perform certain "crucial experiments" in order, for example, to disentangle a few exceedingly confusing steps when we are convinced that our main direction is correct.

A fundamental result of Herbrand has the effect that any derivation of a theorem in a consistent axiom system corresponds to a truth-functional tautology of a form related to the statement of the theorem and the axioms of the system in a predetermined way. This and the possibility already mentioned of viewing axiom systems as proof-grinding machines can both be used to bring about the application of computing machines to the investigation of the question of derivability in general, and inconsistency (i.e., derivability of contradictions) in particular of axiom systems. There is, of course, no reason why we should wish to deny ourselves the privilege of introducing ingenious devices to reduce the great complexity of the combinatorial problems involved in such applications of machines. When machines are extensively used as aids to mathematical discoveries, we shall have a more objective standard of originality of ideas in terms of the magnitude of labour required for a machine to discover them.

If we compare, for example, such possible applications with programming machines to play chess or checkers, the proving machines or proving programs have at least this much advantage on their side: our opponent is Nature or, if one prefers, the platonic world of ideas whose predominant purpose is not, as far as we know, to defeat us at the game.

Surely the dimension of magnitude which we are initially concerned with is staggering: there are so many possible proofs in any interesting axiom system, there are so many truth-functional tautologies. But how do we know whether significant assistance to mathematical research will or will not emerge from such application of machines until preliminary probing has been undertaken on a fairly large scale? If it were thought that such application would have direct bearing on military or business activities, more incentive for looking into it would undoubtedly have been manufactured to direct major effort to the question. Who knows but that such combined application of machines and mathematical logic will not, in the long run, turn out to be more efficient aid to the advance of science, even for the purpose of bigger bombs, longer range missiles, or more attractive automobiles? The logicians on their part will certainly feel happy when other mathematicians find logic an indispensable tool for their own researches. Or perhaps applications of logic in the theory and use of machines will generate interesting new logic just as physics generates important new mathematics.

We often feel that we cannot design machines which are more clever than their designers. We feel that there is an essential difference between such a task and that of designing machines which are physically stronger than their designers.

The concept of proving machines shows that this is not so. If, for example Fermat's or Goldbach's conjecture is indeed provable in one of the usual formal systems of mathematics, there is nothing absurd about the belief that a proof will be first discovered by a machine. The important point is that we are trading qualitative difficulty for quantitative complexity. On account of the great restriction on the mind's ability to handle quantitative complexities, we find it more necessary to rely on insight, ingenuity, and vague intuition. Using machines, we find our ability in this respect increased tremendously and it is but natural to expect that we can then go a longer way even with less ingenuity. The grinding out of proofs and the *selection* of the right ones (i.e., the comparison of their conclusions with proposed conjectures) are both mechanically simple procedures. There is no difficulty in imagining that such machines will be more clever than man in performing the tasks of discovery of theorems and confirmation of conjectures faster and more reliably.

We can instruct a proving machine to select and print out theorems which are short but require long proofs (the "deep" theorems). We can also similarly instruct machines to generate interesting conjectures: e.g., short statements which are neither provable nor refutable by proofs of a preassigned high upper bound of complexity; presumably Fermat's and Goldbach's conjectures will occur in some such class with a fairly high bound.

There is a reasonable sense in which the basic machine B or a universal Turing machine can solve all solvable mathematical problems and prove all provable mathematical statements. Thus, if there is a proof for a mathematical statement, it must be, as a proof, expressible in exact terms, beginning with accepted statements and continuing in a systematic manner. We have therefore a program on the machine B which will lead from the initial statements (axioms) to the conclusion.

If we compare Gauss with a universal Turing machine in regard to their mathematical abilities, we have to admit that Gauss, unlike the imagined machine, did not have an infinite mind or an infinite memory. But then he did not have to solve all problems either. What is peculiar is rather that he could create so much more mathematics with a brain by no means proportionally larger than that of an average man. One would have to be considerably more clever if he were to design a machine of a given size to imitate Gauss (in his mathematical activities) rather than an average college mathematics student. But at the present stage, we may feel that the difference is relatively small compared with the gap between a "student mathematics machine" and any digital computer currently in operation.

REFERENCES

1. A. M. Turing, On computable numbers, *Proc. London Math. Soc.*, Series 2, 24 (1936), 230–265.
2. S. C. Kleene, *Introduction to Metamathematics*, 1952.
3. A. W. Burks and I. M. Copi, The logical design of an idealized general-purpose computer, *J. Franklin Inst.*, 261 (1956), 299–314, 421–436.
4. E. F. Moore, A simplified universal Turing machine, *Proc. ACM, Sept. 8, 1952*, 1953.