Computability of Recursive Functions*

J. C. SHEPHERDSON

University of Bristol, England†

AND

H. E. STURGIS

University of California, Berkeley, USA

1. *Introduction*

As a result of the work of Turing, Post, Kleene and Church [1, 2, 3, 9, 10, 11, 12, 17, 18] it is now widely accepted[1] that the concept of "computable" as applied to a function[2] of natural numbers is correctly identified with the concept of "partial recursive." One half of this equivalence, that all functions computable by any finite, discrete, deterministic device supplied with unlimited storage are partial recursive, is relatively straightforward[3] once the elements of recursive function theory have been established. All that is necessary is to number the configurations of machine-plus-storage medium, show that the changes of configuration number caused by each "move" are given by partial recursive functions, and then use closure properties of the class of partial recursive functions to deduce that the function computed by the complete sequence of moves is partial recursive. Until recently all proofs [4, 6, 12, 13, 19, 20] of the converse half of the equivalence, namely, that all partial recursive functions are computable, have consisted of proofs that all partial recursive functions can be computed by Turing machines,[4] which are certainly machines in the above sense. Although

* Received December, 1961.

† Visitor at the University of California, Berkeley, 1958–1959, when most of this work was done.

[1] There are some finitists or intuitionists who might deny that all general recursive functions are computable, or even assert that the class of general recursive functions is not well-defined. However, by speaking of partial recursive functions we avoid this difference of opinion. For there is surely no doubt that the routines given here and elsewhere will actually compute the value of a given recursive function for a given argument at which the function is defined, and will go on computing forever if the function is not defined at that argument. Of course, there may now be a difference of opinion as to whether a given partial recursive function is general recursive, i.e. defined for all arguments; in fact, the question of whether such a function is defined for one particular argument can be as difficult as the Fermat conjecture. But disagreement on this or on the equivalent question of whether the corresponding computational routine terminates or not does not affect the completely finitist proof that for arguments for which the function *is* defined the routine will compute its value.

[2] Not necessarily defined for all arguments.

[3] Although the belief that all "computations" can be carried out by such a device must be taken as an act of faith or a definition of computation.

[4] Or Markov algorithms, which are similarly restrictive.

not difficult, these proofs are complicated and tedious to follow for two reasons: (1) A Turing machine has only one head so that one is obliged to break down the computation into very small steps of operations on a single digit. (2) It has only one tape so that one has to go to some trouble to find the number one wishes to work on and keep it separate from other numbers. The object of this paper is first to obtain, by relaxing these restrictions, a form of idealized computer which is sufficiently flexible for one to be able to convert an intuitive computational procedure with little change into a program for such a machine. Since this sort of computer plus a given finite program clearly can be regarded as a finite, discrete, deterministic device (plus unlimited storage), a very simple proof can be given to show that all partial recursive functions are computable. We then gradually reintroduce restrictions (1) and (2), passing through a sequence of definitions of intermediate forms of machine and ending with a form from which we can not only obtain directly the computability of all partial recursive functions by a Turing machine[5] with only two tape symbols ("mark" and "blank") but by a very slight change, also the strong result of Wang [20] that erasing is dispensable and that "shift left one square", "shift right one square", "mark a blank square", "conditional transfer" (jump if square under scan is marked) are adequate. In fact, by making another slight change we can decide affirmatively[6] the question raised by Wang [20, p. 84] whether the "conditional transfer" can be replaced by the "dual conditional transfer" (jump if square under scan is blank). The intermediate forms of machine or computational procedure are, we think, of some interest in their own right. For example, in Section 8 we note that a general-purpose computer could be built using one binary tape and two heads, the right-hand one being a writing head which can move only to the right and can print only when moving, the left-hand one a reading head which can also move only to the right and can read only when moving (and may destroy whatever it reads in the process of reading it). In other words, the simple "push-button" or "push-down" store,[7] in which "cards" with 0 or 1 printed on them are added only at the top and taken off to be read only at the bottom, is a universal computing machine. In Section 10 we show that theorems (including Minsky's results [21]) on the computation of partial recursive functions by machines with one and two tapes can be obtained rather easily from one of our intermediate forms. So we might sum up by saying that we have tried to carry a step further the "rapprochement" between the practical and theoretical aspects of computation suggested and started by Wang [20]. However, we do not discuss questions of economy in programming; our aim is to show as simply as possible that certain operations can be carried out. In the interests of reada-

[5] A fact which is important at least to metamathematicians, since it is the basis of many undecidability proofs.

[6] This has also been established recently (by a different method) by C. Y. Lee [24].

[7] We are grateful to A. L. Tritter for pointing out that our use of these expressions is nonstandard; apparently a "push-down" store is an LIFO (last-in, first-out) store, whereas we describe an FIFO (first-in, first-out) store.

bility we have relegated to an appendix certain computational details and supplementary remarks.

Note. There have recently appeared papers by Ershov [5], Kaphengst [8] and Peter [15] which also provide simple proofs of the computability of all partial recursive functions by various kinds of idealized machines or computational procedures. These are all similar to each other and to the methods of this paper but have interesting differences in approach. Ershov starts from a very wide and elegant definition of algorithm, which is particularly suitable for dealing with the theory of programming of digital computers; Peter starts from a general form of block diagram, and Kaphengst from an idealization of a digital computer. We comment later (Appendix A) in more detail on the differences between the operations used in these approaches and those used here; although all the sets of operations are equivalent, the present method appears to be best adapted to our purpose of starting from operations in terms of which all partial recursive functions are easily computable, progressively breaking these down into simpler operations, and ending with the very few basic operations of a non-erasing Turing machine.

Kaphengst's approach is interesting in that it gives a direct proof of the universality of present-day digital computers, at least when idealized to the extent of admitting an infinity of storage registers each capable of storing arbitrarily long words. The only arithmetic operations needed are the successor operation and the testing of two numbers for equality (other operations of the usual kind for transferring numbers from other addresses to and from the mill and the order register are also needed, of course). The proof of this universality which has been tacitly assumed by all concerned with electronic computers since their inception seems to have been first written down by Hermes, who showed in [7] how an idealized computer could be programmed to duplicate the behavior of any Turing machine.

2. *Unlimited Register Machine* (*URM*)

This, our first and most flexible machine, consists of a denumerable sequence of registers numbered 1, 2, 3, $\cdots$, each of which can store any natural number 0, 1, 2, $\cdots$. Each particular program, however, involves only a finite number of these registers, the others remaining empty (i.e. containing 0) throughout the computation. The basic *instructions* (orders, commands) are as follows (here $\langle n\rangle$, $\langle n'\rangle$ denote respectively the content of register n before and after carrying out the instruction):

a.	$P(n)$:	*add* 1 *to the number in register* n, i.e. $\langle n'\rangle = \langle n\rangle + 1$.
b.	$D(n)$:	*subtract* 1 *from the number in register* n, i.e. $\langle n'\rangle = \langle n\rangle - 1$. $(\langle n\rangle \neq 0)$.
c.	$O(n)$:	*"clear" register* n, i.e. *place* 0 *in it*, i.e. $\langle n'\rangle = 0$.
d.	$C(m, n)$:	*copy from register* m *into register* n, i.e. $\langle n'\rangle = \langle m\rangle$.
e.	$J[E1]$:	*jump to exit* 1.
f.	$J(m)[E1]$:	*jump to exit* 1 *if register* m *is empty*.

Notes.

(1) This set of instructions is chosen for ease of programming the computation of partial recursive functions rather than economy; it is shown in Section 4 that this set is equivalent to a smaller set.

(2) There are infinitely many instructions in this list since m, n range over all positive integers.

(3) In instructions a, b, c, d, the contents of all registers except n are supposed to be left unchanged; in instructions e, f, the contents of all registers are unchanged.

(4) The P, D of a, b, stand for PRINT, DELETE, which is what they amount to when we pass to the next stage of representing the natural number x by a sequence of x marks.

(5) Instruction b is used in our programs only when register n is non-empty, so we leave the definition of the machine incomplete to the extent that we do not specify what would happen if it were applied to an empty register (e.g. no effect at all, or STOP without result).

(6) Instruction d is defined only for $m \neq n$; we make this restriction rather than saying this is an instruction $C(n, n)$ which does nothing at all, since the subroutine we give later for $C(m, n)$ in terms of the other instructions would go on computing forever if $m = n$ (it would continually subtract and add 1 to n).

Instructions a, b, c, d, are called *single-exit* instructions and are said to have only the *normal exit* or exit 0. This means that when they occur in a program there is no choice to be made; the machine simply proceeds to the next line of the program. Instruction f, $J(m)[E1]$, however, is thought of as a *two-exit* instruction: if register m is non-empty, take the normal exit (i.e. proceed to the next line of the program); if register m is empty, take exit 1. When this instruction occurs in a program it will always be in the form $J(m)[n]$, indicating that to take exit 1 you proceed to line n. Instruction e, $J[E1]$ is similarly thought of as a two-exit instruction; in this case, however, the "normal" exit is never taken.

Although we do not have here any basic instructions with more than two exits it is convenient to give a definition of program which would apply also in such a case, since for later machines we wish to use subroutines with more than two exits. So we define a *program* (or routine) as a finite sequence of l *lines*, each line being of the form $I[m_1, \cdots, m_k]$, where $I[E1, \cdots, Ek]$ is an instruction, k is the number of non-normal exits of I, and $m_1, \cdots, m_k$ are integers between 1 and $l+1$ (where it is understood that if $k = 0$ the line simply consists of I alone). In following such a program the machine starts on line 1 and proceeds thus: when on line i. $I[m_1, \cdots, m_k]$ it carries out instruction I and proceeds to line $i+1$, $m_1, \cdots$, or m_k depending on whether the state of the registers is such that the 0th (normal), 1st, $\cdots$, or kth exit of instruction I is to be taken; on arriving at the non-existent line $l+1$, it stops. For example, the program

1. $J(n)[4]$
2. $D(n)$
3. $J[1]$

could be written more fully:

1. *Proceed to line* 2 *if register n is non-empty, to line* 4 (*i.e.* stop) *if it is empty.*
2. *Subtract* 1 *from number in register n.*
3. *Jump to line* 1.

It is easily seen to have the same effect as $O(n)$.

Following Wang, we make extensive use of subroutines. A subroutine S is like a program except that (like an instruction) it may have several exits, e.g. we use subroutines such as $\bar{O}(n)[E1]$, "clear register n and proceed to exit 1":

1. $J(n)[E1]$
2. $D(n)$
3. $J[1]$

To obtain a definition for subroutines of this kind we have only to take the above definition of a program and allow the $m_1, \cdots, m_k$ to range over $E1, \cdots, Ek$ as well as $1, \cdots, l+1$. Here k will be the number of non-normal exits of the subroutine. The basic theorem about subroutines of this kind, which (following Wang and other writers on computing machines) we take as being sufficiently obvious not to need a formal proof (which is a little tedious) is that if such subroutines are used as new single instructions in the formation of other subroutines and programs and so on, then all the resulting programs could be obtained with the original set of basic instructions. The formal proof of this is obtained by showing how to expand these subroutines in terms of basic instructions whenever they occur in other routines or subroutines. For illustration, consider the case of a routine or subroutine U with l lines:

$$\begin{array}{ll} 1. & U_1 \\ \vdots & \vdots \\ j. & U_j \\ \vdots & \vdots \\ l. & U_l \end{array}$$

whose jth line is of the form $S[m_1, \cdots, m_k]$ where $S[E1, \cdots, Ek]$ is a $(k+1)$-exit subroutine expressed in terms of basic instructions by m lines:

$$\begin{array}{ll} 1. & S_1 \\ \vdots & \vdots \\ m. & S_m \end{array}$$

To eliminate S, simply replace U_j by these m instructions and convert all jump references so that they go to the correct line in the new program; the resulting program is

$$\begin{array}{rl} 1. & U_1' \\ \vdots & \vdots \\ j-1. & U_{j-1}' \\ j. & S_1'' \\ \vdots & \vdots \\ j+m-1. & S_m'' \\ j+m. & U_{j+1}' \\ \vdots & \vdots \\ l+m-1. & U_l' \end{array}$$

where U_i' $(i = 1, \cdots, j-1, j+1, \cdots l)$ is obtained from U_i as follows: if U_i is $I[n_1, \cdots, n_r]$ then U_i' is $I[n_1', \cdots, n_r']$, where $n' = n$ unless $j < n \leqq l+1$, in which case $n' = n+m-1$. Similarly, if S_i is $I[n_1, \cdots, n_r]$ then S_i'' is $I[n_1'', \cdots, n_r'']$, where $n'' = n+j-1$ if $1 \leqq n \leqq m+1$, $n'' = m_i'$ if $n = E_i$ $(i = 1, \cdots, k)$.

3. *Computability of Partial Recursive Functions by the URM*

A single-valued function (not necessarily defined for all arguments) whose arguments and values range over the natural numbers is *partial recursive* if it can be obtained from the initial functions of schemata I, II, III below by means of a finite number of applications of schemata[8] IV, V, VI:

I. $S(x_1) = x_1+1$
II. $O^n(x_1, \cdots, x_n) = 0$
III. $U_i^n(x_1, \cdots, x_n) = x_i$
IV. [COMPOSITION] *If $h, g_1, \cdots, g_m$ are partial recursive so is the function f defined by* $f(x_1, \cdots, x_n) = h(g_1(x_1, \cdots, x_n), \cdots, g_m(x_1, \cdots, x_n))$.
V. [PRIMITIVE RECURSION] *If g, h are partial recursive so is the function f defined by* $f(0, x_2, \cdots, x_n) = g(x_2, \cdots, x_n)$,
$f(z+1, x_2, \cdots, x_n) = h(z, f(z, x_2, \cdots, x_n), x_2, \cdots, x_n)$.
VI. [LEAST NUMBER OPERATOR] *If g is partial recursive so is the function f defined by* $f(x_1, \cdots, x_n) = \mu y[g(x_1, \cdots, x_n, y) = 0]$.

Note. In schema VI, the "μy", "the least y such that", is to be interpreted thus: $f(x_1, \cdots, x_n)$ is defined to be y_0 when $g(x_1, \cdots, x_n, y_0) = 0$ *and* $g(x_1, \cdots, x_n, y)$ is *defined* but non-zero for $y < y_0$; if no such y_0 exists, then f is undefined.

We now show that all partial recursive functions are computable by the URM in the following sense: for each partial recursive function f of n arguments and each set of natural numbers $x_1, \cdots, x_n, y, N$ ($y \neq x_i$, for $i = 1, \cdots, n$; $x_1, \cdots, x_n, y \leqq N$) there exists a routine $R_N(y = f(x_1, \cdots, x_n))$ such that if $\langle x_1 \rangle, \cdots, \langle x_n \rangle$ are the initial contents of registers $x_1, \cdots, x_n$, then if $f(\langle x_1 \rangle, \cdots, \langle x_n \rangle)$ is undefined the machine will not stop; if $f(\langle x_1 \rangle, \cdots, \langle x_n \rangle)$ is defined the machine will stop with $\langle y \rangle$, the final content of register y, equal to $f(\langle x_1 \rangle, \cdots, \langle x_n \rangle)$, and with the final contents of all registers $1, 2, \cdots, N$ except register y the same as their initial contents.[9] This is the most convenient form to choose for the intuitive proof that all partial recursive functions are computable, since we wish to preserve the arguments for subsequent calculations; if, however, a final routine is wanted which leaves only the value of the function and erases the contents of all registers less than or equal to N except y, this can obviously be obtained from the above routine by adding the instructions $O(1), \cdots, O(y-1), O(y+1), \cdots, O(N)$. What we must now give are subroutines for computing outright the initial functions of schema I, II, III, and for schemata IV, V, VI subroutines for computing f from given subroutines for computing g, h. We give these below:

I. SUBROUTINE $R_N(y = S(x))$
  1. $C(x, y)$
  2. $P(y)$
II. SUBROUTINE $R_N(y = O^n(x_1, \cdots, x_n))$
  1. $O(y)$
III. SUBROUTINE $R_N(y = U_i^n(x_1, \cdots, x_n))$
  1. $C(x_i, y)$

[8] It is convenient in stating these to allow functions of 0 arguments (i.e. constants) so that the n in schemata II–VI ranges over the values $0, 1, 2, \cdots$.

[9] Obviously we cannot hope to preserve the contents of *all* registers except y—we must have some place to do "rough work"—it is important in the induction to have this lower bound N on the addresses of registers possibly disturbed.

IV. SUBROUTINE $R_N(y = f(x_1, \cdots, x_n))$ USING SUBROUTINES FOR g, h, WHERE f IS DEFINED BY SCHEMA IV THUS:
$f(x_1, \cdots, x_n) = h(g_1(x_1, \cdots, x_n), \cdots, g_m(x_1, \cdots, x_n))$

1. $R_{N+1}(N+1 = g_1(x_1, \cdots, x_n))$
$\vdots \quad \vdots \quad \vdots$
m. $R_{N+m}(N+m = g_m(x_1, \cdots, x_n))$
$m+1$. $R_{N+m}(y = h(N+1, \cdots, N+m))$

Note. Registers $N+1, \cdots, N+m$ are used to hold $g_1, \cdots, g_m$ since all registers $1, \cdots, N$ (except y) must be left unchanged by R_N.

VI. SUBROUTINE FOR $R_N(y = f(x_1, \cdots, x_n))$ USING SUBROUTINE FOR g WHERE f IS DEFINED BY VI, THUS: $f(x_1, \cdots, x_n) = \mu y[g(x_1, \cdots, x_n, y) = 0]$

1. $O(y)$
2. $R_{N+1}(N+1 = g(x_1, \cdots, x_n, y))$
3. $J(N+1)[4]$, $P(y)$, $J[2]$

Here and later we no longer number each line of a subroutine; this means simply that we are using for our lines certain subroutines. Clearly, the only instructions which need to be numbered are those to which a jump is made. Two other abbreviatory techniques are worth introducing now, viz. if I is an instruction or subroutine, then I^n stands for the result of performing I n times, i.e. for the subroutine 1. I, 2. I, $\cdots$, n. I. Similarly, if I is a single exit instruction or subroutine which does not affect register n then $I^{\langle n \rangle}$ stands for the result of performing I $\langle n \rangle$ times and reducing $\langle n \rangle$ (the number in register n) to zero; it can be obtained thus: 1. $J(n)[2]$, I, $D(n)$, $J[1]$.

V. SUBROUTINE $R_N(y = f(x_1, \cdots, x_n))$ USING SUBROUTINES FOR g, h WHERE f IS DEFINED BY SCHEMA V, THUS: $f(0, x_2, \cdots, x_n) = g(x_2, \cdots, x_n)$,
$f(z+1, x_2, \cdots, x_n) = h(z, f(z, x_2, \cdots, x_n), x_2 \cdots, x_n)$:

1. $R_N(y = g(x_2, \cdots, x_n))$, $O(N+1)$
2. $\{R_{N+2}(N+2 = h(N+1, y, x_2, \cdots, x_n))$, $C(N+2, y)$, $P(N+1)\}^{\langle x_1 \rangle}$
3. $C(N+1, x_1)$

This completes the proof that all partial recursive functions are computable by the URM. We have simply followed the intuitive argument by which one convinces oneself that one could in fact compute all values of all functions definable by I,$\cdots$,VI. We have chosen a set of basic instructions large enough to make the programming straightforward. Kleene [12, p. 363] proceeds somewhat similarly: "An intuitive calculation by schemata (I)-(VI) is accomplished by repetitions of a few simple operations, such as copying a number previously written (at a determinate earlier position), adding or subtracting one, deciding whether a given number is 0 or not 0. We shall first construct some [Turing] machines to perform such operations as these." However, he does not give explicit programs in terms of these operations but proceeds immediately to the one-dimensional tape and the construction of particular Turing machines. By deferring these steps we are able to get his result and the stronger result of Wang quite simply from the same intermediate form.

4. *Reduction of Basic Instructions*

We now try to reduce the instructions to a smaller and simpler set. The most obvious candidate for such replacement is the copy instruction d. The subroutine which springs to mind for defining this in terms of the other instructions is to keep adding one into register n and subtracting one from register m until the latter is empty. This certainly copies the contents of register m into register n but unfortunately it destroys the original. We can avoid this by making two copies at once and afterwards copying one of them back into register m. However, this will not give exactly $C(m,n)$, since the original contents of the register ($N+1$, say), used to hold the second copy, will have been destroyed. What we can obtain in this way is a bounded copy subroutine $C_N(m,n)$ defined for m, $n \leqq N$, $m \neq n$, thus:

$C_N(m,n)$: *Clear register n, copy contents of register m into it, leaving contents of all registers* $1, \cdots, n-1, n+1, \cdots N$ *(including m) unchanged.*

Before considering this subroutine, note that it can be used instead of $C(m,n)$ in the above routines, since an appropriate bound N can always be determined for the number of registers whose contents we need to keep unchanged. Indeed, consider the bounded analogues of all our basic instructions:

a₁ .	$P_N(n)$	d₁ .	$C_N(m, n)$
b₁ .	$D_N(n)$	e₁ .	$J_N[E1]$
c₁ .	$O_N(n)$	f₁ .	$J_N(m)[E1]$

(for all m,n,N with $m,n \leqq N$), these being defined as having exactly the same effect as the original unsubscripted instructions, except that whereas the latter were required to leave the contents of all registers (except n) unchanged, the new weaker operations are only assumed to leave all registers (except n) less than or equal to N unchanged.

It is easily seen that these operations, with suitable bounding subscript N, could replace the original ones in all the routines given so far, since each of these needs only a bounded number of registers (in subroutines I–VI the number of registers needed is N, N, N, $N+m$, $N+2$, $N+1$, respectively). In fact, this is true of every routine regarded as a function from a given finite set of registers $\{1, \cdots, N_0\}$ to a given finite set $\{1, \cdots, N_1\}$ of registers. For each routine, R is finite and is unchanged by its own operation[10] so there exists a number N_R such that R neither affects nor is affected by any registers greater than N_R; hence if we take $N = \max\{N_0, N_1, N_R\}$ and bound all operations by N the resulting program will be equivalent from this point of view. Since this is the only way we do regard routines we can say that the bounded set of operations is equivalent[11] to the original set.

[10] Since the URM, unlike present-day electronic computers, has no means of working on and altering its program.

[11] If, however, a routine is regarded as establishing such a function for all N_0, N_1 or as a function of the whole infinite totality of registers, then the bounded operations are weaker. No single program formed from the bounded operations can be given which always (regardless of what particular effect the actual N-bounded operations may have on the registers

We now give a series of reductions of this set of bounded instructions.

1. SUBROUTINE FOR d_1, $C_N(m,n)$ IN TERMS OF a_1, b_1, c_1, e_1, f_1
   1. $O_N(n)$, $O_{N+1}(N+1)$
   2. $\{P_{N+1}(N+1), P_{N+1}(n)\}^{(m)}$
   3. $\{P_{N+1}(m)\}^{(N+1)}$

It is to be understood here that the instructions J, $J(m)$, which are involved when lines 2, 3 are expanded (in accordance with the definition of $I^{(m)}$ given in Section 3 above), are also given the appropriate bounding subscript $N+1$.

Note. It is interesting to compare Wang's way [20, p. 73] of dealing (in slightly different circumstances, *viz.* a non-erasing Turing machine) with this difficulty that the original is destroyed in the process of copying. He arranges for the original to be not completely destroyed but only "defaced" so that it is possible (by a different routine) later to copy once more from the defaced original; in this second copying the original is completely destroyed. This leads to somewhat more complicated programs than our method; however, it is only by eliminating the copy operation at this stage, where we can still create space for "rough work" just by bringing in another register, that it is possible to see easily that it can be done this way; the resulting program for making two "simultaneous" copies with a non-erasing Turing machine would involve a large number of operations of permuting the contents of the significant part of the tape.

2. SUBROUTINE FOR c_1, $O_N(n)$ IN TERMS OF b_1, e_1, f_1
   1. $J_N(n)[2]$, $D_N(n)$, $J_N[1]$

We now show how to eliminate the jumps $\mathrm{e}_1,\mathrm{f}_1$ in terms of the dual $\bar{\mathrm{f}}_1$ of f_1:

$\bar{\mathrm{f}}_1$. $\bar{J}_N(m)[E1]$: *jump to exit* 1 *if register m is non-empty*

3. SUBROUTINE FOR f_1, $J_N(m)$ $[E1]$ IN TERMS OF e_1, $\bar{\mathrm{f}}_1$
   1. $\bar{J}_N$ $(m)[2]$, J_N $[E1]$

4. SUBROUTINE FOR e_1, $J_N[E1]$ IN TERMS OF a_1, $\bar{\mathrm{f}}_1$
   1. $P_{N+1}(N+1)$, $\bar{J}_{N+1}(N+1)[E1]$

We have now shown:

4.1. *For each natural number N_0 and each program P of the URM, there exists a program having the same effect as P on registers* $1, \cdots, N_0$ *and composed only of instructions from the following list* $(N = 1, 2, \cdots, n = 1, 2, \cdots)$

a_1.	$P_N(n)$:	$\langle n'\rangle = \langle n\rangle+1$
b_1.	$D_N(n)$:	$\langle n'\rangle = \langle n\rangle-1$
$\bar{\mathrm{f}}_1$.	$\bar{J}_N(n)[E1]$:	*jump to exit* 1 *if* $\langle n\rangle \neq 0$,

greater than N) has the same effect for all N on the first N registers (or on all registers) as the operation $C(m, n)$. Note, however, that subroutines (1) and (4) given below for defining COPY and JUMP in terms of the other operations disturb the contents of only one register; so if there is one additional register 0 available for this use (an "arithmetic unit" or "mill") then these subroutines do show that the original set of instructions a,$\cdots$,f is equivalent to the set a,b,c,f (or a,b,c,f) in the strong sense that for each program P of the original URM (plus the new register 0) there exists a program P_1 in the reduced set of instructions which has the final effect as P on all the registers 1, 2, 3,$\cdots$. The same applies to the set a,b,$\bar{\mathrm{f}}$, although here slight changes are necessary in programs (1) and (2) to avoid register 0 being used simultaneously in conflicting ways. The remaining set mentioned in Appendix A, a,b,f plus an initial 0, is adequate only if a second additional register is available to hold this 0.

where the subscript N denotes that the content of registers $N+1$, $N+2$, $\cdots$ may be altered by the instruction. In particular, all partial recursive functions are computable using these instructions only.

This set of instructions is fairly obviously minimal; for a fuller discussion and comparison with the operations used by Kaphengst [8], Ershov [5] and Peter [15], see Appendix A; reductions in the number of registers used are considered in Sections 7, 8, 10.

5. *Partial Recursive Functions Over a General Alphabet*

When computing a function f of natural numbers and using, say, the decimal representation, it is sometimes convenient to think of the corresponding function f^1 from decimal expressions to decimal expressions defined by $f^1(d)$ equal to the decimal representation of $f(n)$, where n is the number of which d is the decimal representation. For example, if we wish to write a program for the computation of such a function it is in the last analysis the function f^1 which must be considered. In this case, as has been shown in Section 4 for example, it is enough to show how to obtain the decimal functions corresponding to the functions $S(x) = x+1$, $P(x) = x-1$, i.e. how to add and subtract 1 from numbers expressed in decimal notation. However, for some of the more complex ways of representing natural numbers which are considered later, it is easier to work throughout with functions of expressions or "words" over a general alphabet. By an *alphabet* $\mathfrak{A}$ we mean a finite set $\{a_1, \cdots, a_s\}$ of objects called *letters*; a *word* over the alphabet $\mathfrak{A}$ is a finite sequence $a_{i_1} \cdots a_{i_r}$ ($r = 0$ is allowed; this gives the null word $\wedge$) of letters of $\mathfrak{A}$; $W(\mathfrak{A})$ denotes the set of words over $\mathfrak{A}$. By analogy with the usual definition of partial recursive function of natural numbers quoted in Section 3, we may define the partial recursive functions over $\mathfrak{A}$ (i.e. with arguments and values in $W(\mathfrak{A})$) to be the functions obtained by application of the following schemata:

I$_i$*. $(i = 1, \cdots, s)$ $S_{a_i}(x_1) = x_1 a_i$

II*. $\wedge^n(x_1, \cdots, x_n) = \wedge$

III*. $U_i^n(x_1, \cdots, x_n) = x_i$

IV*. *If $h, g_1, \cdots, g_m$ are partial recursive over $\mathfrak{A}$, so is the function f defined by* $f(x_1, \cdots, x_n) = h(g_1(x_1, \cdots, x_n), \cdots, g_m(x_1, \cdots, x_n))$

V*. *If g, h_i $(i = 1, \cdots, s)$ are partial recursive over $\mathfrak{A}$, so is the function f defined by* $f(\wedge, x_2, \cdots, x_n) = g(x_2, \cdots, x_n)$, $f(za_i, x_2, \cdots, x_n) = h_i(z, f(z, x_2, \cdots, x_n), x_2, \cdots, x_n)$, $(i = 1, \cdots s)$.

VI$_i$*. $(i = 1, \cdots, s)$. *If g is partial recursive over $\mathfrak{A}$, so is the function f defined by* $f(x_1, \cdots, x_n) = \mu_i y[g(x_1, \cdots, x_n, y) = \wedge]$, where $\mu_i y[g(x_1, \cdots, x_n, y) = \wedge]$ means "the shortest word y composed entirely of a_i (i.e., of one of the forms $\wedge, a_i, a_i a_i, a_i a_i a_i \cdots$) such that $g(x_1, \cdots, x_n, y) = \wedge$ and $g(x_1, \cdots, x_n, y_1)$ is defined (and $\neq \wedge$) for all y_1 of this form shorter than y.

Notes.

(1) The variables $x_1, \cdots, x_n, y, z$ range over $W(\mathfrak{A})$.

(2) xa_i denotes the concatenation of x and a_i, i.e. the word obtained by placing a_i on the right-hand end of the word x.

(3) The partial recursive functions of natural numbers are included if the natural num-

ber n is identified with the n-letter word a_1^n, i.e. $a_1a_1 \cdots a_1$ on the single-letter alphabet $\mathfrak{A}_1 = \{a_1\}$.

(4) It might appear more natural to use in VI* a μ-operator giving the first word (no restriction on its form) in a certain fixed ordering of $W(\mathfrak{A})$ which satisfied the given condition. However, this would commit us to assigning (arbitrarily) this fixed ordering. As far as generality goes, the two forms are easily seen to be equivalent provided the ordering is primitive recursive, i.e. using the above identification of natural numbers with words in $\mathfrak{A}_1$ provided the function $n(x)$ giving the number of word x in the ordering and the function $W(x)$ whose value for $x = a_1^n$ is the nth word in the ordering (and whose value for words not composed entirely of a_1 may be assigned arbitrarily) are primitive recursive over $\mathfrak{A}$, i.e. definable by schemata I*-V* only. (The usual lexicographic ordering certainly satisfies this condition.)

(5) It can easily be seen that it would have been enough to have only one of the s schemata VI_i^*. We include them all for the sake of symmetry.

(6) It is more usual to define partial recursive functions over $\mathfrak{A}$ in terms of a Gödel-numbering of $W(\mathfrak{A})$ by saying that a function is partial recursive over $\mathfrak{A}$ when the corresponding function of Gödel-numbers is partial recursive. This is easily seen to be equivalent to the definition given provided the Gödel-numbering is primitive recursive (see note 4 above); the familiar Gödel-numberings certainly are). The present approach seems to us to be more natural; it is similar to that of Post [17], Markov [13], and Smullyan [22].

6. *Computability of Partial Recursive Functions over $\mathfrak{A}$ by the URM($\mathfrak{A}$)*

We now give the parallel for a general alphabet to the arguments of Sections 3, 4. The details are so similar to the case already dealt with that we shall relegate them to Appendix B and merely state the final result—that all partial recursive functions over $\mathfrak{A}$ are computable on the URM($\mathfrak{A}$) whose instructions are: ($N = 1, 2, \cdots; n = 1, 2, \cdots; i = 1, \cdots, s$).

a_i.	$P_N^{(i)}(n)$:	*place a_i on the (right-hand) end of $\langle n\rangle$*
b_i.	$D_N(n)$:	*delete the first (left-most) letter of $\langle n\rangle$ ($\langle n\rangle \neq \wedge$)*
f_i'.	$J_N^{(i)}(n)[E1]$:	*jump to exit 1 if $\langle n\rangle$ begins with a_i*

Notes.

(1) The subscript N, as in Section 4, signifies that, apart from making the operations described above, the contents of registers $1, \cdots, N$ are unaltered, although the contents of $N+1, N+2, \cdots$ may be changed.

(2) As before, $\langle n\rangle$ denotes the content of register n.

(3) Instruction b_i will be used only when $\langle n\rangle$ is non-null.

(4) The reason for choosing operations of adding at the end and deleting and jumping from the beginning of a word is that it is the simplest combination to use for building up the subroutines for copying and primitive recursion. It is clear that one must jump and delete from the same end in order to be able to do anything useful, but the addition of letters to a word could take place either at the other end (as here) or at the same end, since it is easily seen that one could then reverse a word if one wished to add to the other end. However, this reversal needs a second register; in the later reduction to a single register only the combination given above (and its opposite) is adequate.

7. *Limited Register Machine (LRM)*

Observe now that the URM can be replaced by a machine which has at any time a finite but variable number N of registers and with instructions possibly depending on N. The Limited Register Machine (LRM) has for the numerical (single letter alphabet) case the following instructions:

$\mathfrak{a}_1$.	$P_N(n)$:	*add* 1 *to* $\langle n \rangle$
$\mathfrak{b}_1$.	$D_N(n)$:	*subtract* 1 *from* $\langle n \rangle$
$\bar{\mathfrak{f}}_1$.	$\bar{J}_N(n)[E1]$:	*jump to exit* 1 *if* $\langle n \rangle \neq 0$
$\mathfrak{h}_1$.	$N \rightarrow N+1$:	*bring in a new register, numbered* $N+1$
$\mathfrak{i}_1$.	$N \rightarrow N-1$:	*remove (empty) register* N

In the general alphabet case we speak of an LRM($\mathfrak{A}$) and replace instructions $\mathfrak{a}_1$, $\mathfrak{b}_1$, $\bar{\mathfrak{f}}_1$ above by

$\mathfrak{a}_1$.	$P_N^{(i)}(n)$:	*place* a_i *on the end of* $\langle n \rangle$
$\mathfrak{b}_1$.	$D_N(n)$:	*delete the first letter of* $\langle n \rangle$
$\mathfrak{f}_1'$.	$J_N^{(i)}(n)[E1]$:	*jump to exit* 1 *if* $\langle n \rangle$ *begins with* a_i

where $\mathfrak{A} = \{a_1, \cdots, a_s\}$ and the range of i is $1, \cdots, s$. [As in Appendix C, $\mathfrak{b}_1, \mathfrak{f}_1'$ can be replaced by a combined SCAN AND DELETE operation $\mathfrak{s}_1$. $Scd_N(n)[E1, \cdots, Es]$.][12]

Here the N denotes that the instruction has the indicated effect when the total number of registers is N; we do not care what would happen if it were applied when the number of registers is different from N. The range of N is $1, 2, 3 \cdots$ for all instructions except $\mathfrak{h}_1$ where it is $0, 1, 2, 3, \cdots$; the range of n is $1, 2, \cdots, N$. It is supposed that the above instructions have exactly the effect specified, i.e. do not alter the contents of other registers.

We first show how to obtain a stronger form $N \rightarrow_1 N-1$ of $\mathfrak{i}_1$, "*remove the not necessarily empty register* N":

1. $P_N(N)$
2. $D_N(N)$, $\bar{J}_N(N)[2]$
3. $N \rightarrow N-1$

This is for the single-letter alphabet. For the general alphabet, replace $P_N(N)$ by $P_N^{(1)}(N)$ and $\bar{J}_N(N)[2]$ by $J_N^{(1)}(N)[2], \cdots, J_N^{(s)}(N)[2]$.

The above instructions, apart from h_1, i_1, are exactly analogous to the bounded forms of the instructions for the URM. We have

7.1. *All partial recursive functions are computable by the LRM.*

To prove this we need only take each routine $R_N(y = f(x_1, \cdots, x_n))$ etc., previously given for the URM with bounded instructions, find the maximum bounding subscript M which occurs, replace all the bounding subscripts by M

[12] However, these two sets are not completely equivalent. The subroutine for $\mathfrak{s}_1$ in terms of $\mathfrak{b}_1, \mathfrak{f}_1'$ given in Appendix C works with the new meaning but the subroutine for $\mathfrak{f}_1'$, $J_N^{(i)}(n)$, in terms of $\mathfrak{s}_1$ must be modified by the insertion of $N \rightarrow N+1$, $N+1 \rightarrow N+2$ at the beginning of the first line, and $N+2 \rightarrow_1 N+1$, $N+1 \rightarrow_1 N$ at the end of the last line (where $N \rightarrow_1 N-1$ is defined by 1. $Scd_N(N)[1, \cdots, 1]$, $N \rightarrow N-1$). Even with this modification the subroutine is not equivalent to $J_N^{(i)}(n)$, since if the jump is taken it is taken from a position where there are $N+2$, not N registers in use. But when it occurs in a complete program with only the single (i.e. normal) exit, this can be compensated for by replacing the line m, say, to which the jump is taken, by two lines m. $N \rightarrow N+1, N+1 \rightarrow N+2$, $m+1$. $N+2 \rightarrow_1 N+1, N+1 \rightarrow_1 N$, old line m, taking the jump now to line $m+1$ and suitably renumbering all other lines and jumps to them.

and, if $M > N$, add instructions $N \to N+1$, $N+1 \to N+2$, $\cdots$, $N+M-1 \to N+M$ at the beginning and $N+M \to_1 N+M-1$, $\cdots$, $N+1 \to_1 N$ at the end.

In connection with later reductions to Turing Machines we note here two special cases of subroutines for the computation of a partial recursive function $f(x_1, \cdots, x_n)$ which are of use. If we take $R_{n+1}\,(n+1 = f(1, \cdots, n))$ and precede it by $n \to n+1$, we get a routine which when started with the first n registers containing $x_1, \cdots, x_n$ finishes with $n+1$ registers containing $x_1, \cdots, x_n$, $f(x_1, \cdots, x_n)$. If we add a routine for copying the contents of register $n+1$ into register 1 and deleting all registers except register 1, we get as final form a single register containing $f(x_1, \cdots, x_n)$, which is the neatest way of displaying the answer, although the former routine which preserved the arguments was useful in the inductive proof that all partial recursive functions were computable.

8. *Reduction to a Single-Register Machine (SRM)*

Instead of speaking of registers of the LRM, we may think of the state of its storage medium at any time as a sequence $\langle 1\rangle, \cdots, \langle N\rangle$ of numbers, or in the general case, words over an alphabet $\mathfrak{A}$. This suggests yet another way of looking at the matter; namely, we can think of $\langle 1\rangle, \cdots, \langle N\rangle$ as a single word A on the alphabet $\mathfrak{A} \cup \{,\}$. From this point of view, however, the basic instructions of the LRM($\mathfrak{A}$) are rather complicated, involving as they do changes in the middle of the word A. It is natural to try to follow Post [17] and replace these operations by simpler ones which affect only the beginning and end of A.[13] The obvious set to try is the analogue of the set we have used for the LRM, i.e., to regard A as the content of a Single-Register Machine (SRM) which has the same instructions applicable to this register as the LRM does for each of its registers:

a.	$P^{(i)}$:	*add a_i to the end of A*
b.	D:	*delete the first letter of A*
f′.	$J^{(i)}[E1]$:	*jump to exit 1 if A begins with a_i*

Here we suppose that alphabet $\mathfrak{A} \cup \{,\}$ is labelled $a_0(,)$, $a_1, \cdots, a_s$ so that i runs from 0 to s in these instructions. Since there is now only one fixed register, we have shorn the instructions of all subscripts and other marks which are now unnecessary.

We now show how to obtain subroutines for the operations of the LRM($\mathfrak{A}$) in terms of these basic instructions applied to the single word $A = A_1, \cdots, A_N$ where $A_1, \cdots, A_N$ stand for the contents of registers $1, \cdots, N$ of the LRM. The key to this is a subroutine T for transferring a word on $\mathfrak{A}$ from the beginning of the string to the end, i.e. for sending $A_1, A_2, \cdots, A_N$ into $A_2, \cdots, A_N, A_1$. We first define the jump $\bar{J}$, jump if $A \neq \wedge$ (if $N > 1$ the word A is always non-null since it contains at least a comma, so that in this case $\bar{J}$ is an unconditional jump): $\bar{J}[E1] = J^{(0)}[E1], J^{(1)}[E1], \cdots, J^{(s)}[E1]$.

[13] The fact that Post was concerned with generating sets of words whereas we are concerned with programs yielding at most one result makes it difficult to use his results directly. In fact it appears to be easier to proceed in the other direction and obtain his results from ours. (See footnote 14.)

We now define T as:

```
1.    $P^{(0)}$
2.    $J^{(1)}[3], \cdots, J^{(s)}[s+2], J^{(0)}[s+3]$
3.    $D, P^{(1)}, \bar{J}[2]$
 ⋮     ⋮
s+2.  $D, P^{(s)}, \bar{J}[2]$
s+3.  $D$
```

We can now obtain the LRM($\mathfrak{A}$) operations in the natural way—by bringing the word we want to operate on to the beginning (for operations b_1, f_1') or end (for operation a_1) by applying T the appropriate number of times, carrying out the corresponding operation of the SRM and then restoring the word to its original position by T. In full:

a_1.	$P_N^{(i)}(n)$	$= 1.$	$T^n, P^{(i)}, T^{N-n}$
b_1.	$D_N(n)$	$= 1.$	T^{n-1}, D, T^{N-n+1}
f_1'.	$J_N^{(i)}(n)[E1]$	$= 1.$	$T^{n-1}, J^{(i)}[2], T^{N-n+1}, \bar{J}[3]$
		$2.$	$T^{N-n+1}, \bar{J}[E1]$
h_1.	$N \rightarrow N+1$	$= 1.$	$P^{(0)}$
i_1.	$N \rightarrow N-1$	$= 1.$	T^{N-1}, D

Here (as before) T^n stands for $T, \cdots, T$ (n times). Taking this together with 7.1 we have

8.1. *All partial recursive functions over $\mathfrak{A}$ are computable by a single-register machine with alphabet $\mathfrak{A} \cup \{,\}$ and operations*

a. $P^{(i)}$: $A \rightarrow Aa_i$
b. D: $a_iA \rightarrow A$
f′. $J^{(i)}[E1]$: *jump to exit* 1 *if A begins with a_i*

or (see Appendix C)

a. $P^{(i)}$: $A \rightarrow Aa_i$
s. $Scd[E1, \cdots, E(s+1)]$: *scan the first letter of A; if $A = \wedge$, take the normal exit; if first letter of A is a_i, delete this and take exit $i+1$ ($i = 0, \cdots, s$).*

These results may be improved slightly; namely, the operations f′,s need not be defined when $A = \wedge$. For we can easily write the above programs so that f′,s are never applied to blank words. One way of doing this is simply to introduce an additional register (i.e. comma) in the second line of the program and remove it in the line before the last.

Here i ranges from 0 to s where $\mathfrak{A} = \{a_1, \cdots, a_s\}$ and a_0 is the comma. As noted in Section 7 the program for computing a function f can be written so that applied to $A_1, \cdots, A_n$ it yields $A_1, \cdots, A_n, f(A_1, \cdots, A_n)$ or so that it yields simply $f(A_1, \cdots, A_n)$. In fact if $f_1, \cdots, f_m$ are m partial recursive functions of $A_1, \cdots, A_n$ it is clear that we can write a program which yields $f_1(A_1, \cdots, A_n), \cdots, f_m(A_1, \cdots, A_n)$. If we take $\mathfrak{A}$ as the single letter alphabet $\{1\}$ and use 0 as a comma, this shows that all partial recursive functions of natural numbers are computable by a machine with a single one-way tape, two tape symbols 0, 1 and

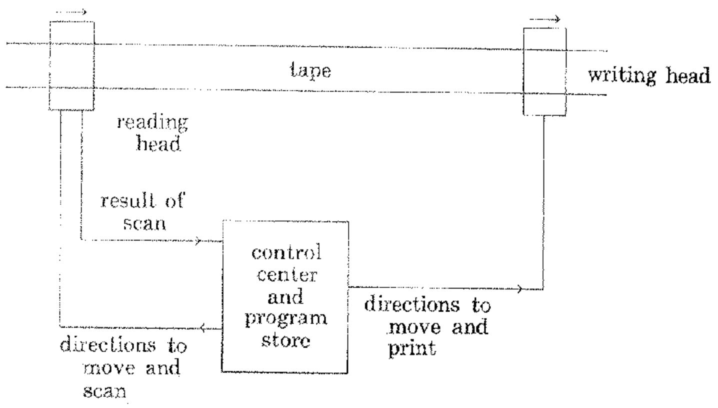
tape
writing head
reading head
result of scan
control center and program store
directions to move and print
directions to move and scan

Fig. 1

two heads—a reading head at the left-hand end and a writing head at the right-hand end, each capable of moving to the right only and connected by a suitable control center and program store (Fig. 1). The adequacy of instructions a,s (see Appendix C) shows that both reading and writing heads need be capable only of reading and writing while moving—so that the tape could be magnetic tape. The reading head "deletes" simply by moving one square to the right; since the tape can never be scanned again when it has passed to the left of the reading head it does not matter if in the process of moving and scanning the reading head destroys the tape completely.[14]

[14] These instructions a,s (in the weakened form where s is not applied to a null word) provide a simple transition to Post normal systems. If we have a program of m lines on an alphabet $\mathfrak{A} = \{a_0, \cdots, a_s\}$ we consider a Post normal system on alphabet $\mathfrak{A}' = \{a_0, \cdots, a_s, q_1, \cdots, q_m\}$ obtained as follows: for each line of the program of the form $i.\ Scd[i_1, \cdots, i_{s+1}]$ introduce "productions" $q_ia_jP \rightarrow Pq_{i_{j+1}}$ $(j = 1, \cdots, s+1)$, for each line $i.\ P^{(j)}$ add productions $q_iP \rightarrow Pa_jq_{i+1}$, and finally add the productions $a_jP \rightarrow Pa_j$ $(j = 0, \cdots, s)$ (for getting the q back to the beginning again). It is easily proved that if W, W_1 are words on $\mathfrak{A}$ then $q_1W \Rightarrow q_mW_1$ by these productions if and only if the program started on W would end with W_1. So if we now take such a program for the computation of a partial recursive function $f(n)$ which is defined on a nonrecursive set and takes the value 0 when defined, we obtain a normal system such that the problem whether $q_11^n \Rightarrow q_m$ is unsolvable. The reverse system is one in which the problem $q_m \Rightarrow q_11^n$ is unsolvable, i.e. this system with initial assertion q_m has unsolvable decision problem. An argument used by Post [23, p. 5] shows that the same is true for the system with all the productions made symmetrical, viz. $q_ia_jP \leftrightarrow Pq_{i_{j+1}}$, etc. Using the fact that for every recursively enumerable set S of words over $\mathfrak{A}$ there exists a function f taking the value 0 on S and undefined outside we see that for each such set S there exists a normal system (symmetrical if desired) on an alphabet including $\mathfrak{A}$ such that, if W is a word on $\mathfrak{A}$, q_1W is derivable if and only if W belongs to S. To get the full result of Post [17], or rather a result which implies it, we must get rid of the q_i here. The easiest way to do this is to use a trick of Post's: start with alphabet

$$\mathfrak{A}'' = \{a_0, \cdots, a_s, \bar{a}_0, \cdots, \bar{a}_s, q_1, \cdots, q_m, \bar{q}_1, \cdots, \bar{q}_m\}$$

instead of $\mathfrak{A}'$ and replace the above productions by $q_ia_jP \rightarrow P\bar{q}_{i_{j+1}}$, $q_iP \rightarrow Pa_j\bar{q}_{i+1}$, $\alpha P \rightarrow P\bar{\alpha}$ (for all $\alpha \in \mathfrak{A}''$, $\bar{\bar{a}}_i$, $\bar{\bar{q}}_i$ being defined as a_i, q_i respectively), $P \rightarrow P\bar{q}_1$.

Another physical realization of the SRM($\mathfrak{A}$) is that of a stack of cards, each printed with a symbol from $\mathfrak{A}$, which can be added to only at the top and read and removed only at the bottom. If instructions a, s are used we need to examine the bottom card only when it is removed so we see that a binary "push-down" (push-button) store with instructions

(1) *add card at top printed* 0
(2) *add card at top printed* 1
(3) *remove bottom card; if printed* 0 *jump to instruction* m_1
if printed 1 *jump to instruction* m_2

is a universal computer; i.e. supplied with a suitable program of instructions of this type it can compute any partial recursive function in the sense that if the stack of cards is initially $1^{x_1}\ 0\ 1^{x_2}\ 0\ \cdots\ 1^{x_n}$ (where 1^{x_1} stands for a stack of x_1 cards marked 1) then it will finally be $1^{f(x_1,\cdots,x_n)}$. As shown in Appendix C, instruction (3) can be weakened to

(3′) *remove bottom card; if printed* 0 *proceed to next instruction*
if printed 1 *jump to instruction* m.

It is of some interest to notice that the above instructions can be still further weakened by placing the comma, used only for punctuation purposes, in a less privileged position than the other letters—namely, by omitting the jump operations on the comma—and having to know the number of commas in a word before operating on it. The weaker set we wish to consider (which will be used in Section 9 to obtain the universality of weak forms of Turing machine) is:

a$_1$. $P_N^{(i)}$: *add* a_i *to the end of* A $(i = 0, \cdots, s)$
b$_1$. D_N : *delete the first letter of* A
f$_1'$. $J_N^{(i)}[E1]$: *jump to exit* 1 *if* A *begins with* a_i $(i = 1, \cdots, s)$

Here N (which takes values $1, 2, \cdots$) is one more than the number of commas in A; i.e. it is the number of words on $\mathfrak{A}$ which A represents; these instructions are to be used only on words containing the correct number of commas. The result we want is

8.2. *Theorem* 8.1 *holds for the weaker set of instructions* {a$_1$, b$_1$, f$_1'$}.

Since we no longer have $J^{(0)}$, the jump-on-comma, the previous subroutine for transferring a word from beginning to end no longer works. However, if S is any subroutine *which jumps when finished* (i.e. never takes the normal exit 0) we can obtain (for $N > 1$) a subroutine $T_N(S)$ which, started on a word A_1, $A_2, \cdots, A_N$, transfers the first word onto the end of the last one and then performs subroutine S, i.e. goes to $A_2, \cdots, A_N A_1$ and then performs S.[14a] $T_N(S)$ is equal to:

1. $J_N^{(1)}[2], \cdots, J_N^{(s)}[s+1], D_N, S$
2. $D_N, P_N^{(1)}, \{1\}$
$\vdots$ $\vdots$
$s+1$. $D_N, P_N^{(s)}, \{1\}$

[14a] The {1} here is simply an abbreviation for the instruction of line 1.

We now define by induction on r a subroutine $R_N(r)[E1]$ which, started on $A_1, \cdots, A_r, a_i A_{r+1}, \cdots, A_N$ sends this into $a_i A_{r+1}, \cdots, A_N, A_1, \cdots, A_r$ and jumps to exit 1:

(1) $r = 0$. $\quad R_N(0)[E1] = 1. \quad J_N^{(1)}[E1]$
(2) $r > 0$. $\quad R_N(r+1)[E1] = 1. \quad P_N^{(0)}, T_{N+1}(R_N(r)[E1])$

Now a subroutine which started on $A_1, \cdots, A_N$ sends this into $a_i A_1, \cdots, A_N$ and jumps to exit 1:

$J_N^*[E1] = 1. \quad P_N^{(0)}, P_{N+1}^1, T_{N+1}^{(1)}(R_N(N-1)[E1])$

Finally, T_N, which started on $A_1, \cdots, A_N$ sends this into $A_2, \cdots, A_N, A_1$:

$T_N = 1. \quad P_N^{(0)}, T_{N+1}(J_N^*[2])$
$\quad 2. \quad D_N$

Now as above we define for $N > 1$:

a$_1$. $P_N^{(i)}(n) \quad = 1. \quad T_N^n, P_N^{(i)}, T_N^{N-n}$
b$_1$. $D_N(n) \quad = 1. \quad T_N^{n-1}, D_N, T_N^{N-n+1}$
h$_1$. $N \to N+1 = 1. \quad P_N^{(0)}$
i$_1$. $N \to N-1 = 1. \quad T_N^{N-1}, D_N$
f$_1'$. $J_N^{(i)}(n)[m] \quad = 1. \quad T_N^{n-1}, J_N^{(i)}[m+1], T_N^{N-n+1}$,

with, in f$_1'$, the compensation[15,12] of replacing line m by

$m. \quad T_N^{n-1}, m+1. \quad T_N^{N-n+1}$, old line m.

For $N = 1$, h$_1$ is defined as above, i$_1$ is undefined, and a$_1$, b$_1$, f$_1'$ are defined by:

a$_1$. $P_1^{(i)}(1) = P_1^{(i)}$
b$_1$. $D_1(1) = D_1$
f$_1'$. $J_1^{(i)}(1)[E1] = J_1^{(i)}[E1]$.

9. *Reductions To Turing Machines* (*TM*)

Note first that the passage from a program to a table of internal state transitions is immediate—simply assign an internal state for each line of the program (when written out in full).

So we have only to concern ourselves with getting the instructions into the TM form, i.e. motion (one square left and right) printing and scanning by a single head. Formulated in terms of instructions, a TM is a program of instructions for a machine which has a single reading-writing head moving on a linear

[15] In this case the use of "compensated" subroutines is inevitable; for the given instructions a$_1$, b$_1$, f$_1'$ provide no means of jumping from a word $A_1, \cdots, A_N$ when A_1 is null, so if $n \neq 1$ and A_n begins with a_i but A_1 is null then $J_N^{(i)}(n)$, which calls for a jump, cannot be obtained by an ordinary subroutine on a$_1$, b$_1$, f$_1'$.

tape which is marked off into squares and is infinite in both[16] directions. At any given time, the head "covers" just one square; it is capable of reading from and printing on this square only. The symbols it can print are 0 (blank, a_0) and the symbols from some non-null alphabet $\mathfrak{A} = \{a_1, \cdots, a_s\}$. The basic instructions are:

	L	*move the head one square left*
	R	*move the head one square right*
$(i = 0, \cdots, s)$	$P^{(i)}$	*print* a_i[17] (i.e. *erase the symbol on the square under the head and replace it by* a_i)
	Sc	*scan the square under the head; if the symbol printed on it is* a_i *take exit* $i+1$ $(i = 0, \cdots, s)$

For the sake of easier comparison with our earlier formulation and with the results of Wang [20] we shall single out $P^{(0)}$, print 0, and denote it by E (erase), and replace Sc by the equivalent set of instructions

$(i = 0, \cdots, s)$ $J^{(i)}$: *jump to exit* 1 *if the scanned symbol is* a_i

So the set we consider is

$$L, \quad R, \quad P^{(i)}\ (i = 1, \cdots, s), \quad E\ (\text{i.e. } P^{(0)}), \quad J^{(i)}\ (i = 0, \cdots, s).$$

We shall see later that E and $J^{(0)}$ are dispensable.

We now propose to use the 0 as a comma and represent a word $A_1, \cdots, A_N$ of the SRM$(\mathfrak{A} \cup \{,\})$ by

$$\text{------}0\!\downarrow\! A_1 0 A_2 0 A_3 \cdots 0 A_N 0^\infty$$

where $\downarrow$ indicates the standard position of the reading head[18] on the first square to the right of the arrow and the horizontal line on the left indicates that we do not care what is printed on the tape there. This leaves a certain ambiguity—the same tape also represents the sequences $A_1, \cdots, A_N, \wedge$; $A_1, \cdots, A_N, \wedge, \wedge$; etc. This is of no concern since in computing partial recursive functions over $\mathfrak{A}$ we deal always with a known number N of words over $\mathfrak{A}$.

To tie up with our previous results, we must give subroutines for carrying out the basic instructions of the SRM$(\mathfrak{A} \cup \{,\})$. As weakened in 8.2, we clearly

[16] Here we follow Post [16], Kleene [12] and Wang [20] rather than Turing [18], who used a one-way infinite tape. However, as Wang remarks, the two-way tape machine as used here (with the head never moving to the left of its initial position) is weaker than the machine with a one-way tape and a specially marked initial square, since it is deprived of the use of this as a fixed point.

[17] In some formulations of TMs the requirement is made that one should not order *print* a_i when the scanned square already has a_i on it. It is easily seen that one can always write programs so as to avoid this since the scan operation allows one to observe the square first before deciding whether to print; in fact the $s(s+1)$ weaker operations, "*replace* a_i *by* a_j" $(i, j = 0, \cdots, s; i \neq j)$ could, for the same reason, replace the $s+1$ operations $P^{(i)}$.

[18] Kleene [12] and Wang [20] take the standard position of the reading head at the right-hand end of the expression; it clearly makes little difference which we choose; the left-hand position saves a few orders in our subroutines.

cannot get the original N-independent orders in view of the ambiguity of our representation.[19] First we need:

SUBROUTINE R_0: *proceed to next blank to the right*
1. $R, J^{(1)}[1], \cdots, J^{(a)}[1]$

Similarly, L_0. Now subroutines for operations a_i, b_1, f_1' of the SRM($\mathfrak{A} \cup \{,\}$) are:

a_i. $P_N^{(i)} = 1.\ L, R_0{}^N, P^{(i)}, L_0{}^N, R$
b_1. $D_N = 1.\ E, R$
f_1'. $J_N^{(i)}[E1] = J^{(i)}[E1]$

Taking this together with 8.2 we obtain the result:

9.1. *Every partial recursive function f of n arguments over the alphabet $\mathfrak{A}$ is computable by a Turing machine over the alphabet $\mathfrak{A} \cup \{0\}$ in the following sense: if the initial tape configuration is*

$$\text{------}0{\downarrow}x_1 0 x_2 \cdots x_{n-1} 0 x_n 0^\infty$$

then, if $f(x_1, \cdots, x_n)$ is undefined the machine will not stop; if $f(x_1, \cdots, x_n)$ is defined it will stop with tape configuration

$$\text{------}0{\downarrow}x_1 0 x_2 \cdots 0 x_n 0 f(x_1, \cdots, x_n) 0^\infty.$$

[As pointed out in Section 5, this could be replaced by $0{\downarrow}f(x_1, \cdots, x_n)0^\infty$ if desired.]

So (Kleene [12]) if the natural number n is represented by 1^n, i.e. $1 \cdots 1$ (n 1's), then all partial recursive functions of natural numbers are computable on a Turing machine with alphabet $\{0,1\}$, i.e. "blank" and "mark." Of course this is an extremely uneconomical way of representing natural numbers; however, we can easily obtain a corresponding result, e.g. a decimal representation in any scale. Consider, for example, the binary decimal representation—to avoid confusion with the use of 0 as blank, suppose that the symbols used in this are 1 and 2. Now the binary representation of $f(x_1, \cdots, x_n)$ is clearly[20] a partial recursive function over $\{1,2\}$ of the words $\bar{x}_1, \cdots, \bar{x}_n$, which are the binary representation of $x_1, \cdots, x_n$, so that 9.1 shows that a Turing machine over $\{0,1,2\}$ could compute f with respect to the binary representation, i.e. when started with $0{\downarrow}\bar{x}_1 0 \cdots \bar{x}_n 0$ it would finish with $0{\downarrow}\overline{f(x_1, \cdots, x_n)}0$.

Moving toward the results of Wang [20] on the computability of all partial recursive functions by TMs which have no erase operation, the first step is to notice that with a very slight change in the meaning of the basic instructions

[19] This could be achieved, and the ambiguity avoided, if we did not deal with null words or if a symbol different from 0 (and distinct from the letters of $\mathfrak{A}$) were used as the comma. Notice that the weakening which gave rise to the complexity in 8.2, namely, the omission of "*jump on comma*" is not needed for the result 9.1 about ordinary TMs but only for obtaining the results of Wang for a particular representation using a nonerasing machine.

[20] The simplest way of proving this fully is to show that the functions converting from the "tally" representation 1^n of n to the binary representation on $\{1, 2\}$ and vice versa, are primitive recursive functions over $\{1, 2\}$.

the programs just given are applicable to a more general case. Suppose that we have a weak TM whose alphabet $\mathfrak{B}$ contains $\mathfrak{A} \cup \{0\}$ and whose operations are:

	L	*move the head one square left*
	R	*move the head one square right*
	E	*replace the symbol on the scanned square by a symbol in* $\bar{\mathfrak{A}}$
$(i = 1, \cdots, s)$	$P^{(i)}$	*print* a_i *on the scanned square provided this is blank*
$(i = 1, \cdots, s)$	$J^{(i)}[E1]$	*jump to exit* 1 *if the scanned symbol is* a_i

Notes.

(1) $\bar{\mathfrak{A}}$ denotes $\mathfrak{B} - \mathfrak{A}$.

(2) In E we do not stipulate whether and in what way the symbol from $\bar{\mathfrak{A}}$ which replaces the scanned symbol depends on this or on other factors. All we need to know is that the new symbol is in $\bar{\mathfrak{A}}$.

(3) As before $a_1, \cdots, a_s$ are supposed to be the elements of $\mathfrak{A}$.

(4) We use instruction $P^{(i)}$ only when scanned square is blank.

Now let us use $\bar{a}$ to denote an unspecified symbol from $\bar{\mathfrak{A}}$ and agree to represent the sequence $A_1, \cdots, A_N$ of words over $\mathfrak{A}$ by the tape configuration ——$\bar{a}{\downarrow}A_1\bar{a}A_2\bar{a} \cdots \bar{a}A_N0^{\infty}$. Then it is easily seen that all the subroutines just given still function as desired (although R_0, L_0 should now be described somewhat differently, viz. proceed to next $\bar{a}$ to the right, left). So we have

9.2. *Every partial recursive function f over $\mathfrak{A}$ is computable by a weak TM over any alphabet $\mathfrak{B}$ containing $\mathfrak{A} \cup \{0\}$ in the following sense: if the initial tape configuration is* ....$\bar{a}{\downarrow}x_1\bar{a}x_2 \cdots \bar{a}x_n0^{\infty}$ *then the final tape configuration when*

$$f(x_1, \cdots, x_n)$$

is defined is

$$........\bar{a}{\downarrow}x_1\bar{a}x_2 \cdots \bar{a}x_n\bar{a}f(x_1, \cdots, x_n)0^{\infty}$$

[or ....$\bar{a}{\downarrow}f(x_1, \cdots, x_n)0^{\infty}$ if desired].

The simplest case of this which involves a "non-erasing" machine is where $\mathfrak{A}$ is the one-letter alphabet $\{1\}$, where $\mathfrak{B}$ is $\{0, 1, 2\}$ and where operation E consists of replacing the scanned symbol by 2. This is non-erasing in the following sense: the sequence of symbols appearing during the course of computation on any given square has no cycle of length greater than one (as it may, e.g. $0 \to 1 \to 0$, for a normal TM); once a square has had a 1 printed on it the 0 can never be restored; all that can be done is to "degenerate" it further by replacing the 1 by 2. Identifying the natural number n with 1^n gives, in a sense, the simplest non-erasing TM for the computation of all partial recursive functions of natural numbers. Wang's result can now be obtained by mapping this alphabet $\{0, 1, 2\}$ as follows onto a binary alphabet $\{b,*\}$ (b = blank): $0 \to bb$, $1 \to *b$, $2 \to **$.

On this alphabet $\{b,*\}$ the operations Wang uses are

$\leftarrow$:	*move head one square left*
$\rightarrow$:	*move head one square right*
$*$:	*mark the scanned square* (i.e. print $*$)
C:	*jump to exit* 1 *if scanned square is marked.*

To obtain his result[21] on the computability of all partial recursive functions with this representation and these basic operations from the $\{0, 1, 2\}$-case of 9.2 just discussed, we have only to show how to obtain the above operations L, R, E, $P^{(1)}$, $J^{(1)}$. This we do as follows:

When the "old" (i.e. $\{0, 1, 2\}$-machine) head is scanning a symbol 0, 1 or 2, the new head will scan the leftmost of the corresponding pair of symbols from the alphabet $\{b, *\}$. With this convention the subroutines are

L:	$\leftarrow, \leftarrow$
R:	$\rightarrow, \rightarrow$
E:	$*, \rightarrow, *, \leftarrow$
$P^{(1)}$:	$*$
$J^{(1)}[E1]$:	1. $\rightarrow, C[2], \leftarrow, C[E1], \rightarrow$, 2. $\leftarrow$

Wang says [20, p. 84] that he does not know whether C can be replaced by C': *jump to exit 1 if scanned square is blank.* The easiest way of seeing that it can is to change the above convention, use the rightmost of the pair of symbols on $\{b, *\}$ as the standard position of the scanning head, and change the last three subroutines above to:

E:	$*, \leftarrow, *, \rightarrow$
$P^{(1)}$:	$\leftarrow, *, \rightarrow$
$J^{(1)}[E1]$:	1. $\leftarrow, C'[2], \rightarrow, C'[E1], \leftarrow$
	2. $\rightarrow$

The reason for his doubt was "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." The answer we have given is that it is not necessary; in our solution the only pairs of adjacent squares which are both marked are under or to the left of the standard position of the head; in other words we have shown that all "rough work" can (at the cost of many extra permutation steps) be done to the left of, and not in the middle of, the main calculation.

Notes.

(1) Lee's result (cf. footnote 6) on the adequacy of $*$, $\leftarrow$, $\rightarrow$, C' is a little weaker than ours in that he uses additional auxiliary squares, 0 being represented by $bbbb$ and 1 by $*bbb$. These new auxiliary squares are kept permanently blank so that with C' a jump can always be made from them—another way of avoiding the need to go through an indefinitely long string of marked squares. Our treatment above has been complicated by our desire to obtain Wang's results using exactly his form of representation. By means of a slight modification of this, using $*b$ for 1 and $b*$ for the comma (instead of bb), we can write subroutines for "jump on comma" as well as "jump on 1" (with either C or C') and so avoid the need for the more complicated definition of T_N given in 8.2. Note that even with Wang's representation this is not needed for the elimination of erasing but only for the restriction to the single conditional transfer C.

(2) Oberschelp [14] remarks that with the type of machine used by Wang it is not possible to compute each partial recursive function in such a way that the final tape is of the form $0^{-\infty}x_1 0 x_2 \cdots 0 x_n 0 f(x_1, \cdots, x_n) 0^{\infty}$. As he points out, only very simple functions f

[21] Wang actually considers only positive integers. We are able to include 0 without the device of using 1^{n+1} to represent n because we can deal with null words, since we always know how many words we are dealing with.

can be computed in this way (because the machine cannot erase its rough work at all). We have shown here, however, that if one is prepared to tolerate rubbish to the left of the final position of the head this form can be achieved. With the routines given above the actual final form of the $b,*$ tape would be

$$b^{-\infty}*^{2k}(*b)^{x_1}bb(*b)^{x_2}\cdots(*b)^{x_n}bb(*b)^{f(x_1,\cdots,x_n)}b^{\infty}$$

(for some k) so that the rough work takes the simple form of a solid block of completely marked tape. As mentioned above, the final form $b^{-\infty}*^{2k}(*b)^{f(x_1,\cdots,x_n)}b^{\infty}$ could also be achieved if desired.

10. *Reductions to Bounded Number of Registers Without Enlargement of Alphabet*

The reduction to a single-register machine was accomplished in Section 8 only at the cost of enlarging the alphabet from $\mathfrak{A}$ to $\mathfrak{A} \cup \{,\}$. It is interesting to see what reductions in the number of registers are possible without doing this. Starting with the case of a one-letter alphabet $\mathfrak{A} = \{1\}$, i.e. the case where each register stores simply a non-negative integer n in the form $1 \cdots (n \text{ times}) \cdots 1$, our results of Sections 3, 4 form a rather more convenient starting point than TMs do for establishing the following version of a result[22] of Minsky [21]:

10.1. *A single register machine working on non-negative integers and with operations*

(α)	$\times\, k$:	*multiply the number in the register by k*
(β)	$\div\, k$:	*divide the number in the register by k*
(γ)	Div?$k[E1]$:	*test whether the number in the register is divisible by k; if so take exit 1, if not proceed normally to next instruction*

can compute all partial recursive functions f in the following sense: if the number in the register is initially $p_1^{x_1}p_2^{x_2}\cdots p_n^{x_n}$, then it will finally be $p_1^{f(x_1,\cdots,x_n)}$.

Notes.

(1) k is supposed to range over all natural numbers; it will be used only for prime k and it will be shown later that (for functions of one variable) it is enough to have the operations for $k = 2, 3, 5$ only, or with a more complicated representation of argument and value for $k = 2, 3$ only.

(2) p_i denotes the ith prime.

(3) Operation (β) is used only when the number in the register is divisible by k.

(4) We could equally well obtain $p_1^{x_1}\cdots p_n^{x_n}p_{n+1}^{f(x_1,\cdots,x_n)}$ as the number finally in the register.

PROOF. We use the number $p_1^{(1)}\cdots p_N^{(N)}$ to represent the state of the URM. In view of the results of Sections 3, 4 we have merely to show how to perform operations on this number corresponding to the operations a, b, $\bar{\text{f}}$ of the URM. These are evidently obtainable thus:

$P(n)$:	$\times p_n$
$D(n)$:	$\div p_n$
$\bar{J}(n)[E1]$:	Div?$p_n[E1]$

[22] Minsky uses a combined multiply and jump operation and a combined β and γ, viz. *test whether divisible, if so divide and take exit* 1, *if not take exit* 2. It is clear that the present operations can be obtained from these. The results of Appendix C show the adequacy of the set consisting of (α) and (γ'): *Test whether divisible by k, if so divide by k and take exit* 1, *if not take normal exit.*

10.1 shows that a single register is sufficient if complicated enough operations are used—if we want to start with x_1 and finish with $f(x_1)$ we must add the operations $n \to 2^n$, $2^n \to n$. Following Minsky, we proceed to see how many additional registers are needed to replace these by the simple operations of addition and subtraction of one we have used up to now. Consider then a machine with a fixed number N of registers and operations (there is no need for the subscript N now since N is fixed): (for $n = 1, \cdots, N$)

a.	$P(n)$:	*add one to* $\langle n \rangle$
b.	$D(n)$:	*subtract one from* $\langle n \rangle$
$\bar{\text{f}}$.	$\bar{J}(n)[E1]$:	*jump to exit* 1 *if* $\langle n \rangle \neq 0$

We first show that operations α, β, γ can be obtained using one extra register.

Lemma. *If* $n < N$, *then there are programs which end with* $\langle n+1 \rangle = 0$, *do not disturb any registers except* n, $n+1$ *and perform the following operations*:

(α)	$\langle n \rangle \times k$:	*multiply* $\langle n \rangle$ *by* k
(β)	$\langle n \rangle \div k$:	*divide* $\langle n \rangle$ *by* k ($\langle n \rangle$ *supposed divisible by* k)
(γ)	Div?$(\langle n \rangle, k)[E1]$:	*if* $k \mid \langle n \rangle$ *take exit* 1, *if not take exit* 0.

Proof. In Section 4 we showed how to obtain from a, b $\bar{\text{f}}$ programs for $O(n)$: *clear register* n, J (*unconditional jump*), $J(n)$, *jump if* $\langle n \rangle \neq 0$. In one of these, the one for J, we disturbed a register other than n. This can be avoided by using a compensated subroutine: $J[m] = 1.\ P(1),\ \bar{J}(1)[m+1]$ with the compensation of replacing old line m by: $m.\ P(1)$, $m+1.\ D(1)$, $m+2.$ old line m.

So we are at liberty to use all of these, hence also the device $I^{\langle n \rangle}$ introduced in Section 3. Notice that if $\langle n+1 \rangle = 0$, then $(P(n+1))^{\langle n \rangle}$ copies $\langle n \rangle$ into register $n+1$ and clears register n. Our subroutines for α, β, γ above all start with $O(n+1)$, which clears register $n+1$. They continue as follows:

$\langle n \rangle \times k$:	$(P(n+1))^{\langle n \rangle},\ (P(n)^k)^{\langle n+1 \rangle}$
$\langle n \rangle \div k$:	1. $(P(n+1))^{\langle n \rangle}$
	2. $J(n+1)[3],\ (D(n+1))^k,\ P(n),\ J[2]$
Div?$(\langle n \rangle, k)[E1]$:	1. $(P(n+1))^{\langle n \rangle}$
	2. $J(n+1)[E1],\ (D(n+1),\ P(n),\ J(n+1)[3])^{k-1},\ D(n+1),\ P(n),\ J[2]$

Together with 10.1, the lemma gives a second result of Minsky:

10.2. *With the same representation of arguments and values as in* 10.1 *but with operations* a, b, $\bar{\text{f}}$, *two registers are adequate for the computation of all partial recursive functions.*

The next question is how many registers are needed if arguments and values are required to be given in uncoded form. The answer is:

10.3. *A machine with operations* a, b, $\bar{\text{f}}$ *and* $n+2$ *registers is adequate for the computation of all partial recursive functions* f *of* n *variables in the following way*: *start with* $x_1, \cdots, x_n$ *in registers* $1, \cdots, n$, *finish with* $f(x_1, \cdots, x_n)$ *in register* 1.

Proof. In view of 10.2 we have only to show how to replace the $x_1, \cdots, x_n$ in registers $1, \cdots, n$ by $p_1^{x_1} \cdots p_n^{x_n}$ in register 1 with the others clear, and conversely, how to replace $2^{f(x_1, \cdots, x_n)}$ in register 1 by $f(x_1, \cdots, x_n)$ in register 1.

We define for $i = 1, \cdots, n$ a subroutine $red(i)$ which, if $\langle i+2\rangle = 0$ places $p_i^{\langle i\rangle} \times \langle i+1\rangle$ in register i and clears registers $i+1$, $i+2$.

$$red(i): \quad (\langle i+1\rangle \times p_i)^{\langle i\rangle}, \ (P(i))^{\langle i+1\rangle}.$$

Now the required initial routine is simply:

$$P(n+1), red(n), red(n-1), \cdots, red(1)$$

For the final conversion from $2^{f(x_1, \cdots, x_n)}$ to $f(x_1, \cdots, x_n)$ in register 1 we first clear registers 2, 3, then apply:

1. $(P(2))^{\langle 1\rangle}$
2. Div?$(\langle 2\rangle,2)[3],J[4]$
3. $\langle 2\rangle \div 2, P(1), J[2]$

A similar treatment shows that $n+3$ registers are adequate for the final form $x_1, \cdots, x_n$ in registers $1, \cdots, n$; $f(x_1, \cdots, x_n)$ in register $n+1$.

Applying the results of 10.3, 10.2 to the proof of 10.1, we obtain

10.4. *In* 10.1 *for the computation of functions of n variables the operations* α, β, γ *are needed only for* $k = p_1, \cdots, p_{n+2}$. *If the arguments and value are represented in the forms* $p_1^{p_1^{x_1} p_2^{x_2} \cdots p_n^{x_n}}$, $p_1^{p_1^{f(x_1, \cdots, x_n)}}$, α, β, γ *are needed only for* $k = 2, 3$.

For the case of a general alphabet $\mathfrak{A} = \{a_0, \cdots, a_{s-1}\}$ there is a result analogous to 10.3; if $s > 1$ the $n+2$ can be replaced by $n+1$. We shall merely sketch the proof. Let us use as the Gödel number of the word $a_{i_1} \cdots a_{i_r}$ the word a_1^k where $k = i_1 s^{r-1} + i_2 s^{r-2} + \cdots + i_r$. Using two extra registers, the words $x_1, \cdots, x_n$ in each of registers $1, \cdots, n$ can be replaced by their Gödel numbers. Now the Gödel number of $f(x_1, \cdots, x_n)$ is a partial recursive function of the Gödel number of $x_1, \cdots, x_n$; so by 10.3 it can be computed and placed in register 1. Finally, we can (using the two extra registers) replace this Gödel number by the corresponding word. Since the two extra registers are used only for holding words of the form a_1^m, a_1^n, by the results of Section 8 they can be replaced by a single extra register which holds $a_1^m a_0 a_1^n$.

Thus for an alphabet $\mathfrak{A}$ with two or more letters, each partial recursive function f of one variable over $\mathfrak{A}$ can be computed directly with the operations a, b, $\bar{\text{f}}$ by a machine having 2 registers: i.e. if A is placed initially in register 1 then $f(A)$ appears there finally. What can be done by a single register machine? Clearly if we are prepared to allow complicated enough operations, such as replacing A by its Gödel number and vice-versa, then as in 10.1 we can compute all partial recursive functions. But how complicated must these operations be? Rather surprisingly it turns out that the operations PRINT, DELETE and SCAN are enough *provided they can be used at both ends* of the word:

10.5. *Let* $\mathfrak{A} = \{a_0, \cdots, a_{s-1}\}$ *be an s-letter alphabet where* $s \geqq 2$. *Consider the following operations on a word A over* $\mathfrak{A}$ $(i = 0, \cdots, s-1)$:

a_L.	$P_L^{(i)}$:	*print* a_i *on the left-hand end of A*
a_R.	$P_R^{(i)}$:	*print* a_i *on the right-hand end of A*
b_L.	D_L:	*delete the leftmost letter of A*
b_R.	D_R:	*delete the rightmost letter of A*
f_L'.	$J_L^{(i)}[E1]$:	*jump to exit* 1 *if* a_i *is the leftmost letter of A*
f_R'.	$J_R^{(i)}[E1]$:	*jump to exit* 1 *if* a_i *is the rightmost letter of A*

For each partial recursive function f of one variable over $\mathfrak{A}$ there exists a program using only these operations which computes f, i.e. *started on A will, if $f(A)$ is defined, finish with $f(A)$; if $f(A)$ is undefined, will not stop.*

PROOF. We rely on the result of Section 8 that all partial recursive functions over $\mathfrak{A}$ are computable by a single-register machine with alphabet $\mathfrak{A}_1 = \mathfrak{A} \cup \{,\}$ with the operations a, b, f$'$ used previously. We define a mapping ϕ: $\mathfrak{A}_1 \to \mathfrak{A}$ by

$$\phi(a_i) = a_i a_0, \quad \phi(,) = a_0 a_1 . \qquad (i = 0, \cdots, s-1)$$

We have to show how to convert a word A over $\mathfrak{A}$ into $\phi(A)$, how to convert $\phi(A)$ back into A and how if A_1 is a word over $\mathfrak{A}_1$ to perform operations on $\phi(A_1)$ corresponding to the operations a, b, f$'$ on A_1. First, consider the conversion of A into $\phi(A)$. We have to send $A = a_{i_1}a_{i_2}\cdots a_{i_n}$ into $\phi(A) = a_{i_1}a_0a_{i_2}a_0\cdots a_{i_n}a_0$. At first sight this appears to be impossible; since there are no auxiliary letters available there seems to be no way of distinguishing the original A from the subsequently added letters. The key is to have the right-hand head[23] print in a particular pattern and to have it constantly go back and re-examine to see whether the pattern has been disturbed by the left-hand head. When it has, then the original word has been completely coded, so that the process must then stop before part of the word is coded twice. The details are as follows: first, $A = a_{i_1}a_{i_2}\cdots a_{i_n}$ is sent into $a_{i_1}a_{i_2}\cdots a_{i_n}a_1a_1a_0a_0$. Then a loop is entered, the general step starting with the partially coded form $a_{i_j}\cdots a_{i_n}a_1a_1a_0a_0a_{i_1}a_0a_{i_2}a_0\cdots a_{i_{j-1}}a_0$, which reads the leftmost letter a_{i_j}, stores it, replaces it by a_0, and before printing $a_{i_j}a_0$ on the right-hand end checks to make sure that this a_{i_j} replaced by a_0 was a letter of the original word A and not the first of the added a_1's. Routine $T(R, L)$ transfers the word $A_j = a_1a_0a_0a_{i_1}a_0a_{i_2}a_0\cdots a_{i_{j-1}}a_0$ (i.e. the word obtained by going in from the right one letter, then two letters at a time until an a_1 is reached) from the right-hand end to the left-hand end, where it reappears in the form $A_j^1 = a_1a_0a_1a_{i_1}a_1a_{i_2}\cdots a_1a_{i_{j-1}}a_0$, and then checks the rightmost letter to see whether it is a_1 as it was originally, or a_0 as it is when all of A has been coded. If it is a_1 then A_j^1 is transferred by a routine $T(L, R)$ from the left-hand end, where it is recognizable as the word obtained by going in from the left one letter, then two letters at a time until an a_0 is reached, to the right-hand end where it reappears in the form A_j; then $a_{i_j}a_0$ is printed on the right, a_0 is deleted on the left and the loop is re-entered. If it is a_0, A_j^1 is transferred to the right as A_j and the coding is completed by the deletion of $a_0a_1a_0a_0$ on the left. In other words, the coding routine proceeds thus:

$$A = a_{i_1}\cdots a_{i_n} .$$

print $a_1a_1a_0a_0$ on right

$$a_{i_1}\cdots a_{i_n}a_1a_1a_0a_0$$

enter for the first time a loop, jth entry of which is from

$$a_{i_j}\cdots a_{i_n}a_1a_1a_0a_0a_{i_1}a_0\cdots a_{i_{j-1}}a_0$$

replace a_{i_j} by a_0

$$a_0a_{i_{j+1}}\cdots a_{i_n}a_1a_1a_0a_0a_{i_1}a_0\cdots a_{i_{j-1}}a_0$$

apply $T(R, L)$

$$a_1a_0a_1a_{i_1}\cdots a_1a_{i_{j-1}}a_0a_0a_{i_{j+1}}\cdots a_{i_n}a_1$$

[23] We visualize the operations being carried out by two heads, one at each end of the word.

check that right-hand letter is not a_0 ; *if not proceed to apply* $T(L, R)$

$$a_0a_{i_{j+1}}\cdots a_{i_n}a_1a_1a_0a_0a_{i_1}a_0\cdots a_{i_{j-1}}a_0$$

print $a_{i_j}a_0$ *on right, delete* a_0 *on left*

$$a_{i_{j+1}}\cdots a_{i_n}a_1a_1a_0a_0a_{i_1}a_0\cdots a_{i_{j-1}}a_0a_{i_j}a_0$$

and re-enter loop

The closing stages are:

enter loop with

$$a_1a_1a_0a_0a_{i_1}a_0\cdots a_{i_n}a_0$$

replace a_1 *by* a_0

$$a_0a_1a_0a_0a_{i_1}a_0\cdots a_{i_n}a_0$$

apply $T(R, L)$

$$a_1a_0a_1a_{i_1}\cdots a_1a_{i_n}a_0a_0$$

check right-hand letter; it is a_0 , *so apply* $T(L, R)$

$$a_0a_1a_0a_0a_{i_1}a_0\cdots a_{i_n}a_0$$

and delete $a_0a_1a_0a_0$ *on the left, leaving*

$$\phi(A) = a_{i_1}a_0\cdots a_{i_n}a_0$$

In terms of the subroutines $T(R, L)$, $T(L, R)$ (defined below) the program for this is:

1. $(P_R^{(1)})^2,\ (P_R^{(0)})^2$
2. $\sum_{i=0}^{s-1}{}_L\ \{D_L,\ P_L^{(0)},\ T(R, L),\ J_R^{(0)}[3],\ T(L, R),\ P_R^{(i)},\ P_R^{(0)},\ D_L,\ J[2]\}$
3. $T(L, R),\ D_L^4$

Here we have used the absolute jump J. This can easily be programmed thus: $J[m] = 1.\ \ P_L^{(1)},\ J_L^{(1)}[m+1]$ with the compensation of replacing old line m by $m.\ P_L^{(1)},\quad m+1.\ D_L^{(1)},\quad m+2.$ old line m.

We have also used the notation $\sum_{i=0}^{s-1}{}_L F^{(i)}$. This (cf. Appendix B) denotes a subroutine which follows subroutine $F^{(i)}$ if the leftmost letter of the word is a_i, $i = 0, \cdots, s-1$, and does nothing if the word is null. It is obtainable thus:

```
1.     $J_L^{(0)}[2], \cdots, J_L^{(s-1)}[s+1], J[s+2]$
2.     $F^{(0)}, J[s+2]$
 .      .
 .      .
 .      .
s+1.   $F^{(s-1)}, J[s+2]$.
```

We use Σ_R similarly.

The subroutine $T(R, L)$ must send a word of the form $Ba_1a_0a_0a_{i_1}a_0\cdots a_{i_n}a_0$ into $a_1a_0a_1a_{i_1}\cdots a_1a_{i_n}a_0B$. It is obtined in the obvious way by repetition of the operation $Xaa_0 \to a_1aX$, $\quad Xa_1 \to X$ *and stop.*

$T(R, L)$ is equal to:

1. $P_L^{(0)}$
2. $J_R^{(1)}[4]$
3. $D_R,\ \sum_{i=0}^{s-1} R\{D_R,\ P_L^{(i)},\ P_L^{(1)},\ J[2]\}$
4. D_R

The inverse operation $T(L, R)$ can obviously be obtained from this by interchanging ${}^{(0)}$, ${}^{(1)}$ and L, R throughout.

The decoding procedure which sends $\phi(A) = a_{i_1}a_0 \cdots a_{i_n}a_0$ into $A = a_{i_1} \cdots a_{i_n}$ is much simpler; it can be accomplished by the routine:

1. $P_L^{(1)}$
2. $J_R^{(1)}[3], D_R, \sum_{i=0}^{s-1}{}_R \{P_L^{(i)}, D_R, J[2]\}$
3. D_R

It works thus:

start with

$$\phi(A) = a_{i_1}a_0 \cdots a_{i_n}a_0$$

print a_1 *on left*

$$a_1a_{i_1}a_0 \cdots a_{i_n}a_0$$

read right-hand letter; if not a_1, *delete it, transfer next letter from right-hand to left-hand end and repeat*

$$a_{i_n}a_1a_{i_1}a_0 \cdots a_{i_{n-1}}a_0$$
$$\vdots$$
$$a_{i_1}a_{i_2} \cdots a_{i_n}a_1$$

when right-hand letter is a_i *delete it and stop*

$$a_{i_1}a_{i_2} \cdots a_{i_n} = A$$

We must now show how to perform operations on $\phi(A_1)$ which correspond to the operations a, b, f′ performed on A_1.

If we number the letters of the alphabet $\mathfrak{A}_1 = \mathfrak{A} \cup \{,\}$ 0, 1, $\cdots$, $s-1$, s in the order $a_0, \cdots, a_{s-1}, \{,\}$ then we can obtain the operations thus:

a.	$P^{(i)}$:	$P_R^{(i)}, P_R^{(0)}$	$(i = 0, \cdots, s-1)$
	$P^{(s)}$:	$P_R^{(0)}, P_R^{(1)}$	
b.	D:	$D_L{}^2$	
f′.	$J^{(i)}[E1]$:	$J_L^{(i)}[E1]$	$(i = 1, \cdots, s-1)$
	$J^{(0)}[E1]$:	1. $J_L^{(0)}[2], J[4]$	
		2. $D_L, J_L^{(0)}[3], P_L^{(0)}, J[4]$	
		3. $P_L^{(0)}, J[E1]$	
	$J^{(s)}[E1]$:	1. $J_L^{(0)}[2], J[4]$	
		2. $D_L, J_L^{(1)}[3], P_L^{(0)}, J[4]$	
		3. $P_L^{(0)}, J[E1]$	

This completes the proof of 10.5. We show in Appendix E that the set of operations used here is minimal, so for the case of an alphabet with at least two letters, two registers are certainly necessary (and, as mentioned above, sufficient) for the computation of all single-argument partial recursive functions if the original operations a, b, f′ only are allowed. For a one-letter alphabet the left- and right-hand operations are the same; so the result of Appendix E shows that a single register with operations at both ends is not adequate. In this case the best results are 10.1—that a single register is adequate with operations of multiplication, division (plus exponentiation and its inverse if argument and value are required in uncoded form); 10.2—that two registers with operations a, b, f̄ (+1, −1, test whether 0) are adequate with exponential coding of argument and value; and 10.3—that with operations a, b, f̄ and three registers there is no need to code arguments and values.

APPENDIX A. Minimality of Instructions Used in 4.1 Comparison with Similar Systems

The set of instructions in 4.1, viz.

a_1. $P_N(n)$: $\langle n' \rangle = \langle n \rangle + 1$
b_1. $D_N(n)$: $\langle n' \rangle = \langle n \rangle - 1$
$\bar{f}_1$. $\bar{J}_N(n)[E1]$: jump to exit 1 if $\langle n \rangle \neq 0$

is fairly obviously minimal: namely, if the initial configuration was $x, 0, 0, \cdots$, (i.e. x in register 1, all other registers empty), then with b_1, $\bar{f}_1$ alone the only everywhere defined function $f(x)$ whose value could be computed in register 2 would be the zero function; with a_1, b_1 alone, only constant functions; with a_1, $\bar{f}_1$ alone, only functions of the form $f(x) = k$ for $x = 0, f(x) = l$ for $x \neq 0$, where $k \geqq l$.

Concerning reductions in the range of values of N, n for these instructions, it is clear that if they are available for an infinity of values of n, and for each such n an infinity of values of N, we have essentially the same machine. If, however, they are available for only a finite number of values of n or N then they are clearly inadequate to compute functions of all numbers of variables with the method of representation used above, i.e. with the arguments placed in separate registers. (But see Section 10.)

It is natural to ask whether f_1 could be used instead of $\bar{f}_1$. If by this we mean is it possible to write in terms of a_1, b_1, f_1 a subroutine R for each n-ary partial recursive function ϕ such that if $x_1, \cdots, x_n$ are initially placed (say) in registers $1, \cdots, n$ then *regardless of the contents of the other registers* the effect of R will be to place the value of $\phi(x_1, \cdots, x_n)$ in register $n+1$, then the answer is negative. For if all registers are nonempty there is no way of jumping at all since with f_1 this would require first clearing a register and this cannot be done without a jump operation (unless one has an upper bound for the content of some register). However, if we agree always to start with 0 in register 1 then all partial recursive functions can be computed, for we can keep the 0 in register 1 and obtain $J_N = J_N(1)$. Similarly if we are given $c_1 : O_N(n)$ we can again clear a suitable register at the beginning of each program and so obtain J_N. It is easily verified that of the set of instructions $a_1 : P_N(n)$, $b_1 : D_N(n)$, $c_1 : O_N(n)$, $d_1 : C_N(m, n)$, $e_1 : J_N[E1]$, $f_1 : J_N(m)[E1]$, $\bar{f}_1 : \bar{J}_N(m)[E1]$, the only minimal subsets adequate for the computation of all partial recursive functions as above are the ones we have considered, viz. $\{a_1, b_1, e_1, f_1\}$, $\{a_1, b_1, \bar{f}_1\}$, $\{a_1, b_1, c_1, f_1\}$, $\{a_1, b_1, f_1$, fixed 0 register$\}$. For without a conditional jump f_1 or $\bar{f}_1$ the arguments cannot influence the form of the computation at all; without b_1 only their vanishing or nonvanishing can influence it, and without a_1 no values could be written down which were greater than any arguments.

It is interesting to compare the operations used by Kaphengst [8], Ershov [5] and Peter [15].

Kaphengst's PM (programmgesteuerte Rechenmaschine) has a special calculating register "mill", number ∞, and an order register number 0 which contains the address of the next order. The orders themselves are stored in the ordinary registers so the machine, like an actual computer, is capable of doing arithmetic operations on its own program.

However, Kaphengst shows that it can calculate all partial recursive functions without using this facility. It is then essentially similar to a URM plus a special register, number ∞, and the following instructions (for $m, n = 1, 2, \cdots$):

D_1.	$C(m, \infty)$:	*copy contents of register m into mill*
D_2.	$C(\infty, m)$:	*copy contents of mill into register m*
C_1.	$O(\infty)$:	*clear mill*
A_1.	$P(\infty)$:	*add 1 to number in mill*
F_1.	$J(\infty)[E1]$:	*jump to exit 1 if mill is empty*
G_1.	$O_n(\infty)$:	*clear mill if its contents coincide with contents of register n, otherwise place 1 in it, i.e.* $\langle\infty'\rangle = 0$ if $\langle\infty\rangle = \langle n\rangle$, $\langle\infty'\rangle = 1$ *otherwise*
G_2.	$0'(\infty)$:	*clear mill if not already clear; if already clear place 1 in it, i.e.* $\langle\infty'\rangle = 0$ if $\langle\infty\rangle \neq 0$, $\langle\infty'\rangle = 1$ if $\langle\infty\rangle = 0$
H.	*stop*:	*stop if mill is clear (i.e. if* $\langle\infty\rangle = 0$)

It is easily seen that operations C_1, G_2 are definable[24] in terms of other orders and that H (the only form of stop which the PM has) can be defined in terms of F_1 and an ordinary absolute stop. If we remove the mill and consider the effect of the orders on registers 1, 2, 3, $\cdots$ we see that the remaining orders are equivalent to the following orders for a URM

a. $P(n)$
d. $C(m, n)$
f. $J(m)[E1]$
g. $E_k(m, n)$: $\langle k'\rangle = 0$ if $\langle m\rangle = \langle n\rangle$
$\langle k'\rangle = 1$ if $\langle m\rangle \neq \langle n\rangle$

which are easily seen to be a minimal set and to be equivalent[25] to our original set a, $\cdots$, f. In terms of the reduction to ordinary computing machines, Kaphengst's reduction, with all arithmetic operations taking place in only one register, is more apt than ours. However, our ultimate aim is to reduce to simple forms of Turing machines which operate on one bit at a time; from this point of view the operations $P(n)$ and $D(n)$ are simpler than the copy operation. Similar remarks apply to the basic sets described below corresponding to Ershov's and Peter's treatment.

Ershov's class $\mathfrak{A}(\mathfrak{V}_1, \mathfrak{S}_1)$ of operator algorithms differs from the URM in its program structure and treatment, e.g. like the PM its program is stored in the registers. But it is substantially equivalent to a URM with the following instructions

d.	$C(m, n)$	
$\bar{\text{d}}$.	$C_1(m, n)$:	*copy* $\langle m\rangle+1$ *into n, i.e.* $\langle n'\rangle = \langle m\rangle+1$
e.	$J[E1]$	
f*.	$J(m, n)[E1, E2]$:	*jump to exit 1 if* $\langle m\rangle \leqq \langle n\rangle$ *jump to exit 2 if* $\langle m\rangle > \langle n\rangle$

together with the ability to place any constants in any registers at the beginning of the program. Once again it is easy to see by direct construction of subroutines that this set of instructions is equivalent[26] to our original set a, $\cdots$, f and to the set a,d,f,g just given. e is a special case of f* but is listed separately because it is an inseparable part of all Ershov's algorithmic programs. Apart from this, the set of instructions is obviously minimal (although f* is necessary only for one fixed value of m (or n)).

Peter's treatment involves basic operations such as $(x_1, \cdots, x_n) \to (x_1, \cdots, x_n, x_1, \cdots, x_n)$, but with a slight re-formulation it could be regarded as roughly equivalent to a URM with instructions

c.	$O(n)$	
d.	$C(m, n)$	
$\bar{\text{d}}$.	$C_1(m, n)$	
f†.	$J(m, n)$ $[E1, E2]$:	*jump to exit 1 if* $\langle m\rangle = \langle n\rangle$ *jump to exit 2 if* $\langle m\rangle \neq \langle n\rangle$

[24] Using, in the case of G_2, other registers for rough work (holding 0 and 1).
[25] In the sense of 4.1, i.e. in their effect on the contents of any finite set of registers.
[26] In the sense of 4.1, i.e. in their effect on the contents of any finite set of registers.

These (apart from the fact that by using d, c need be available only for one fixed register, number 1 say, $\bar{d}$ only for $m = n = 0$, and f† only for $m = 0, n = 1$) are also clearly minimal and equivalent to the other sets.

To sum up these various minimal systems of instructions one might say that a universal computer working on natural numbers must be capable of producing 0, of adding 1 to a number (i.e. of performing the operations which generate the natural numbers), of copying a number, and either of comparing two numbers for equality or order, or comparing one number with zero and reducing (i.e. by $D(n)$) a number step by step to zero, and directly (e.g. f*) or indirectly (f, g) changing the course of the computation depending on the result of this comparison. The various minimal systems are very similar; from the point of view of proving as quickly as possible the computability of all partial recursive functions Peter's is perhaps the best; for proving their computability by Turing machines a further analysis of the copying operation is necessary along the lines we have taken above.

APPENDIX B. Details of Section 6.

Computability of Partial Recursive Functions over General Alphabet $\mathfrak{A}$ by the URM($\mathfrak{A}$)

We show that all partial recursive functions over $\mathfrak{A}$ are computable on the URM($\mathfrak{A}$) whose instructions are:

a$_i$. $P_N^{(i)}(n)$: *place a_i on the (right-hand) end of $\langle n\rangle$*
b. $D_N^{(n)}$: *delete the first (left-most) letter of $\langle n\rangle$ ($\langle n\rangle \neq \wedge$)*
f$_1'$. $J_N^{(i)}(n)[E1]$: *jump to exit 1 if $\langle n\rangle$ begins with a_i*

Note that Ershov's class $\mathfrak{A}(\mathfrak{V}_2, \mathfrak{S}_2)$ of algorithms of [5] corresponds closely to the above instructions together with $C(m, n)$, with all operations taking place at the beginning of the word. His class $\mathfrak{A}(\mathfrak{V}_3, \mathfrak{S}_3)$ amounts to using the operations

$C(m, n)$
$Jxt(m, n, k)$: $\langle k'\rangle = \langle m\rangle\langle n\rangle$ (juxtaposition or concatenation of $\langle m\rangle$ and $\langle n\rangle$)
$J^*(m, n)[E1, E2]$: *jump to exit 1 if $\langle m\rangle$ ends $\langle n\rangle$*
jump to exit 2 if not

and allowing any constants to be placed in any registers initially. As he says, it is easy to see that these two sets of operations are equivalent (and universal) either by his proof that $\mathfrak{A}(\mathfrak{V}_2, \mathfrak{S}_2)$ is capable of dealing with all Markov algorithms, or our proof below that the URM($\mathfrak{A}$) is capable of computing all partial recursive functions over $\mathfrak{A}$. For a further set of universal operations, see Appendix (C).

We first introduce auxiliary subroutines as in Section 4 for the more complex operations originally used in Sections 2, 3. Slight changes are necessary in some cases due to the fact that there may now be more than one letter in the alphabet.

(1) Subroutine for $\bar{J}_N(n)[E1]$: *jump to exit 1 if $\langle n\rangle \neq \wedge$*
1. $J_N^{(1)}(n)[E1], \cdots, J_N^{(s)}(n)[E1]$
(2) Subroutine for $J_N[E1]$: *jump to exit 1*
1. $P_{N+1}^{(1)}(N+1), \bar{J}_{N+1}(N+1)[E1]$
(3) Subroutine for $J_N(n)[E1]$: *jump to exit 1 if $\langle n\rangle = \wedge$*
1. $\bar{J}_N(n)[2], J_N[E1]$
(4) Subroutine for $\wedge_N(n)$: *clear register n (i.e. place $\wedge$ in it)*
1. $J_N(n)[2], D_N(n), J_N[1]$

We now introduce a convenient abbreviation. Suppose we have subroutines $F_N^{(i)}$ $(i = 1, \cdots, s)$ for performing certain operations. It is convenient to have a subroutine which will follow subroutine $F_N^{(1)}$ if $\langle n \rangle$ begins with $a_1, \cdots$; will follow $F_N^{(s)}$ if $\langle n \rangle$ begins with a_s; and will, say, do nothing if $\langle n \rangle = \wedge$. We denote such a subroutine by $\sum_{i=1}^{s} (n) F_N^{(i)}$ and obtain it thus:

1. $J_N^{(1)}(n)[2], \cdots, J_N^{(s)}(n)[s+1], J_N(n)[s+2]$
2. $F_N^{(1)}, J_N[s+2]$

$\vdots$

$s+1$. $F_N^{(s)}, J_N[s+2]$.

Strictly speaking the Σ should have a subscript N to denote that the additional J, $J(n)$ instructions it involves have subscript N. Since we use it only when all the $F^{(i)}$ have the same subscript N we omit this subscript on the Σ. We have followed the same procedure with the $I^{\langle n \rangle}$ subroutine given in Section 3 above. The analogue of this latter operation, which we denote by $\{I_N^{(i)}\}_i^{\langle n \rangle}$ has the following effect: if $\langle n \rangle = a_{i_1} \cdots a_{i_k}$ then it performs the sequence of operations $I_N^{(i_1)}, \cdots, I_N^{(i_k)}$ (if $\langle n \rangle = \wedge$, does nothing) and reduces $\langle n \rangle$ to $\wedge$, possibly disturbing the contents of registers $N+1$, $N+2, \cdots$.

Here $I_N^{(1)}, \cdots, I_N^{(s)}$ is a given sequence of instructions or subroutines which are supposed not to affect register n. Using the above Σ-notation we can obtain $\{I_N^{(i)}\}_i^{\langle n \rangle}$ thus:

1. $\sum_{i=1}^{s} (n) \{I_N^{(i)}, D_N(n), J_N[1]\}$

Now we can define

(5) SUBROUTINE FOR $C_N(m, n)$: *copy $\langle m \rangle$ into register n*

1. $\wedge_N(n), \wedge_{N+1}(N+1)$
2. $\{P_{N+1}^{(i)}(N+1), P_{N+1}^{(i)}(n)\}_i^{\langle m \rangle}$
3. $\{P_{N+1}^{(i)}(m)\}_i^{\langle N+1 \rangle}$

We now proceed to give subroutines for schemata I*–VI* of Section 5:

I*. SUBROUTINE $R_N(y = Sa_i(x))$

1. $C_N(x, y)$
2. $P_N^{(i)}(y)$

II*. SUBROUTINE $R_N(y = \wedge^n(x_1, \cdots, x_n))$

1. $\wedge_N(y)$

III*. SUBROUTINE $R_N(y = U_i^n(x_1, \cdots, x_n))$

1. $C_N(x_i, y)$

IV*. SUBROUTINE $R_N(y = f(x_1, \cdots, x_n))$ USING SUBROUTINES FOR g, h WHERE f IS DEFINED BY SCHEMA IV*, THUS:

$$f(x_1, \cdots, x_n) = h(g_1(x_1, \cdots, x_n), \cdots, g_m(x_1, \cdots, x_n))$$

1. $R_{N+1}(N+1 = g_1(x_1, \cdots, x_n))$

$\vdots$

m. $R_{N+m}(N+m = g_m(x_1, \cdots, x_n))$

$m+1$. $R_{N+m}(y = h(N+1, \cdots, N+m))$

V*. SUBROUTINE $R_N(y = f(x_1, \cdots, x_n))$ USING SUBROUTINES FOR g, h, WHERE f IS DEFINED BY SCHEMA V* THUS: $f(\wedge, x_2, \cdots, x_n) = g(x_2, \cdots, x_n)$, and $f(za_i, x_2, \cdots, x_n) = h_i(z, f(z, x_1, \cdots, x_n), x_2, \cdots, x_n)$ $(i = 1, \cdots, s)$

1. $R_N(y = g(x_2, \cdots, x_n)), \wedge_{N+1}(N+1)$
2. $\{R_{N+2}(N+2 = h_i(N+1, y, x_2, \cdots, x_n), C_{N+2}(N+2, y), P_{N+2}^{(i)}(N+1)\}_i^{\langle x_1 \rangle}$
3. $C_{N+2}(N+1, x_1)$.

VI$_i$*. SUBROUTINE FOR $R_N(y = f(x_1, \cdots, x_n))$ USING SUBROUTINE g WHERE f IS DEFINED BY VI$_i$*, THUS: $f(x_1, \cdots, x_n) = \mu_i y[g(x_1, \cdots, x_n, y) = \wedge]$

1. $\wedge_N(y)$
2. $R_{N+1}(N+1 = g(x_1, \cdots, x_n, y))$
3. $J_{N+1}(N+1)[4]$, $P_{N+1}^{(i)}(y)$, $J_{N+1}[2]$

This completes the proof that all partial recursive functions are computable by the URM($\mathfrak{a}$).

APPENDIX C. ALTERNATIVE SET OF BASIC INSTRUCTIONS "SCAN AND DELETE" INSTEAD OF "SEPARATE SCAN, DELETE"

There is an alternative set of basic instructions which could have been used in Section 6 instead of a_1, b_1, f_1', *viz.*

a_1.	$P_N^{(i)}(n)$:	*place a_i on end of $\langle n\rangle$*
s_1.	$Scd_N(n)[E1, \cdots, Es]$:	*scan the first letter of $\langle n\rangle$; if $\langle n\rangle = \wedge$ take normal exit, if first letter of $\langle n\rangle$ is a_i delete this and proceed to exit i $(i = 1, \cdots, s)$*

Here s_1, "scan and delete" is an $(s+1)$-exit instruction. This set seems to be of some interest in that it shows that there is never any need to scan a symbol twice, that a general-purpose computer can be built using only scanning devices which destroy the symbol scanned. To see that the new set of instructions is adequate one could write the above programs using s_1 instead of b_1, f_1'— the resulting programs are perhaps slightly simpler since it will be observed that in nearly every case we did delete after scanning. However, one can show that the new set of instructions is actually equivalent to the old. On the one hand, s_1 can easily be obtained using b_1, f_1', viz. $Scd_N(n)[E1, \cdots, Es] = \sum_{i=1}^{s} (n)\{D_N(n), J_N[E_i]\}$. To obtain a_1, f_1' from a_1, s_1 is a little more complicated; we first define

$$D_N(n) = 1.\ \ Scd_N(n)[2, \cdots, 2] \quad \text{and} \quad \wedge_N(n) = 1.\ \ Scd_N(n)[1, \cdots, 1].$$

Then we define $Scd_N'(n)[E1, \cdots, E(s+1)]$, an $(s+2)$-exit instruction which differs from $Scd(n)[E1, \cdots, Es]$ in that when $\langle n\rangle = \wedge$ exit $s+1$ is taken instead of exit 0 (i.e. which provides a jump on $\wedge$ as well as the other jumps):

$$Scd_N'(n)[E1, \cdots, E(s+1)] \\ = 1.\ \ Scd_N(n)[E1, \cdots, Es],\ P_N^{(1)}(n),\ Scd_N(n)[E(s+1), \cdots, E(s+1)]$$

Now construct a subroutine $CP_N(m\colon n_1, n_2, \cdots, n_r)[E1]$ which copies $\langle m\rangle$ onto the end of each of $\langle n_1\rangle, \cdots, \langle n_r\rangle$ (we actually need this only for $r = 1, 2$) and proceeds to exit 1, $\langle m\rangle$ being replaced by $\wedge$.

```
1.    $Scd_N'(m)[2, 3, \cdots, s+1, E1]$
2.    $P_N^{(1)}(n_1), \cdots, P_N^{(1)}(n_r), \{1\}$
⋮         ⋮                     ⋮
s+1.  $P_N^{(s)}(n_1), \cdots, P_N^{(s)}(n_r), \{1\}$
```

Here the {1} is simply used as an abbreviation for the instructions of line 1, viz. $Scd_N'(m)[2, 3, \cdots, s+1, E1]$. We can now obtain $J_N^{(i)}(n)[E1]$ by copying $\langle n \rangle$ out into (cleared) registers $N+1$, $N+2$, copying one of them back again into register n and operating with Scd on the other:

1. $\wedge_{N+1}(N+1)$, $\wedge_{N+2}(N+2)$, $CP_{N+2}(n; N+1, N+2)[2]$
2. $CP_{N+2}(N+1; n)[3]$
3. $Scd_{N+2}[4, \cdots, E1, \cdots, 4]$

where, in line 3, the $E1$ is in the ith place.

For the single-register machine of Section 8 also the instructions a. $P^{(i)}$, b. D, f′. $J^{(i)}[E1]$ can be replaced by a and s:

s. $Scd[E1, \cdots, Es, E(s+1)]$: *scan the first letter of A; if $A = \wedge$ take normal exit, if first letter of A is a_i delete this and take exit $i+1 (i = 0, \cdots, s)$*

However, just as in Section 7 (cf. footnote 12), these two sets of instructions are not completely equivalent—for example, an unconditional jump can be obtained from a, s but not from a, b, f′. The simplest way to show the adequacy of a, s is to repeat the treatment of Section 8 and show how to obtain the LRM operations a_1, b_1, s_1, h_1, i_1 in terms of a, s. The program for T is now:

1. $P^{(0)}$
2. $Scd[s+3, 3, \cdots, s+2]$
3. $P^{(1)}$, {2}

$\vdots$ $\quad\vdots$

$s+2$. $P^{(s)}$, {2}

The programs for a_1, h_1 are the same as before. For the others we write

i_1. $N \to N-1$ = 1. T^{N-1}, $Scd[2, \cdots, 2]$

s_1. $Scd_N(n)[m_1, \cdots, m_s]$ = 1. T^{n-1}, $Scd[2, (m_1+1)', \cdots, (m_s+1)']$
2. $P^{(0)}$, T^{N-n}

with the "compensation" (see footnote 12)[27] of replacing each line m_i ($i = 1, \cdots, s$) by two lines m_i. T^{N-1}, m_i+1. T^{N-n+1}, old line m_i, renumbering all lines and jumps as necessary ($(m_i+1)'$ refers to the final number of the new line m_i+1).

Similarly, the instructions a_1. $P_N^{(i)}$, b_1. D_N, f_1'. $J_N^{(i)}[E1]$ of 8.2 can be replaced by a_1 and

s. $Scd_N[E1, \cdots, Es]$: *scan the first letter of A; if it is a_i delete it and take exit i ($i = 0, \cdots, s$)*[28]*; if A is null do nothing and take the normal exit (i.e. exit number 0—this can occur only if $N = 1$)*

[27] This use of compensating subroutines can actually be avoided here by the use of more complex programs which first duplicate the initial letter of A_n; see the treatment below in terms of the weaker form of *Scd*.

[28] Note that this means that if A begins with a comma, i.e. with a_0, then there is no jump but simply the normal exit to the next line of the program.

The treatment is very similar to that of 8.2. $T_N(S)$ is obtained thus:

$$\begin{array}{rl} 1. & Scd_N[2, \cdots, s+1], S \\ 2. & P_N^{(1)}, \{1\} \\ \vdots & \quad \vdots \\ s+1. & P_N^{(s)}, \{1\} \end{array}$$

Next, by induction on r, a subroutine $R_N{}'(r)[E1]$ is defined which sends $A_1, \cdots, A_r, a_1A_{r+1}, \cdots, A_N$ into $A_{r+1}, \cdots . A_N, A_1, \cdots, A_r$ and jumps to exit 1:

$$\begin{array}{ll} (1)\ r = 0. & R_N{}'(0)[E1] = Scd_N[E1, \cdots, E1] \\ (2)\ r > 0. & R_N{}'(r+1)[E1] = 1.\ \ P_N^{(0)}, T_{N+1}(R_N{}'(r)[E1]) \end{array}$$

Then an unconditional jump

$$J_N[E1] = 1.\ \ P_N^{(0)}, P_{N+1}^{(1)}, T_{N+1}(R_N{}'(N-1)[E1])$$

and

$$T_N = 1.\ \ P_N^{(0)}, T_{N+1}(J_N[2]).$$

Finally, we show how to program the operations a_1, s_1, h_1, i_1 of the LRM($\mathfrak{A}$) (which in Section 7 were shown to be adequate for the computation of all partial recursive functions). a_1, h_1 are dealt with exactly as above and we define:

$$\begin{array}{lrl} \mathrm{i}_1 . & N \to N-1 = 1. & T_N^{n-1}, Scd_N[2, \cdots, 2] \\ \mathrm{s}_1 . & Scd_N(n)[E1, \cdots, Es] = 1. & T_N^{n-1}, Scd_N[2, 3, \cdots, s+1], P_{N-1}^{(0)}, T_N^{N-n}, J_N[s+2] \\ & 2. & T_N^{N-n+1}, J_N[E1] \\ & \vdots & \qquad \vdots \\ & s+1. & T_N^{N-n+1}, J_N[Es]. \end{array}$$

APPENDIX D. Need for "Auxiliary" Squares in Non-erasing TMs

Wang [20] says "it is 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 →, ←, *, 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 nth square as the auxiliary square." Oberschelp [14] shows that with the representation of n by $*^n$ only a very restricted class of "semiperiodic" functions are computable—because once the head gets into a long block of marked squares it cannot alter these in any way so that it has only its finite internal memory to tell it how far it has gone; as a result the actual number of marked squares passed over leaves no trace, only its residue class modulo something. In a sense this shows that, for this particular "tally" representation auxiliary squares *are* necessary. A similar argument shows that very few partial recursive functions over $\mathfrak{A}$ are computable by a TM with alphabet $\mathfrak{A} \cup \{0\}$ which is not allowed to change any of the symbols from $\mathfrak{A}$ into anything else but only to print them on blank squares. However, if one goes back from words on an alphabet to the actual natural numbers it is rather difficult to define what is meant by saying that auxiliary squares are used in a particular representation. Both the representation of n by $(*b)^n$ and the tally representation $(*)^n$ itself "use auxiliary squares" in that they are much longer than the irredundant binary representation of n of length $\log_2 n$. It looks as though all that one can define precisely is the degree of redundancy of the coding—the function $f(n)$ giving the length of the representation of n. From this point of view it is difficult to distinguish between the erasing and nonerasing machines. Both require auxiliary squares for punctuation; if $x_1, \cdots, x_n$ is a sequence of natural numbers you cannot simply take the binary representations of $x_1, \cdots, x_n$, and place them end to end, since there would then be no way of telling where one number stopped and another began, nor of recognizing the end of x_n. It seems to be impossible to avoid having regularly occurring "auxiliary" squares to deal with this punctuation problem so that the ideal coding of length $\log_2 n$ is not attainable even asymptotically. By taking blocks of length k sufficiently large, representing a binary word $u_1, \cdots u_k$ by $u_1 \cdots u_k*$ and leaving b^{k+1} to represent the comma, we can achieve a length of $(1+\epsilon) \log_2 n$ for any $\epsilon > 0$. However we shall now show that the same condensation can be achieved with the nonerasing machine.

First we observe that if we take the binary representation of a number x and, starting from the left, mark it off into blocks of k so that it appears as $u_1, \cdots u_r, v$, where each of $u_1, \cdots, u_r$ is of length k and v is of length greater than 0 and less than k, then we may regard this as a word on a new alphabet $\mathfrak{A}'$ with $2^{k+1}-2$ letters, viz. the 2^k "full" blocks of length k, and the $2^{k-1}+2^{k-2}+\cdots+2$ incomplete blocks of lengths $k-1, \cdots, 1$. Writing x' to denote the word of $\mathfrak{A}'$ corresponding to x in this way, it is clear that x' is computable from x and vice versa, so that if f is a partial recursive function of $x_1, \cdots, x_n$ then $f(x_1, \cdots, x_n)' = g(x_1', \cdots, x_n')$, where g is a partial recursive function of $x_1', \cdots, x_n'$. Hence by 9.2 there is a program on a weak TM over $\mathfrak{A}' \cup \{0\}$ which computes g. Let us use an alphabet $\mathfrak{A}' \cup \{0, e\}$ for this machine, the operation E being the replacement of the scanned symbol by e. We now map the 2^{k+1} symbols of $\mathfrak{A}' \cup \{0, e\}$ back onto the binary alphabet $\{b, *\}$, using the blocks of length $k+1$, with 0 mapped onto the block b^{k+1} and e mapped onto $*^{k+1}$, the mapping being otherwise arbitrary. In this way we have taken the original binary expression for x and replaced each block of k symbols (and the incomplete block at the end) by one of $k+1$ symbols, so that we have achieved the same degree of condensation as before. To complete the proof that the operations of a nonerasing TM are adequate to compute all partial recursive functions with this representation we now show that this last mapping of $\mathfrak{A}' \cup \{0, e\}$ onto $\{b, *\}$ is such that a nonerasing TM on $\{b, *\}$ with operations $\leftarrow$, $\rightarrow$, $*$, C (or C'—this can be dealt with similarly) can carry out the operations of a weak TM on alphabet $\mathfrak{A}' \cup \{0, e\}$. We need a subroutine $R(k+1)[E1, \cdots, E(2^{k+1}-1)]$ with 2^{k+1} exits, which will examine the $k+1$ squares to the right of the head and take exit t if they contain the binary expansion of t ($b = 0$, $* = 1$, most significant place on the left), with the position of the head on exit being on the furthest 1 to the right in this block of $k+1$ if $t \neq 0$, on the original square if $t = 0$ (i.e. is $k+1 - \rho(k+1, t)$ places to the right of the initial position, where $\rho(k+1, t)$

equals the greatest r less than or equal to $k+1$ such that $2^r \mid t$). This can be defined inductively as follows:

$$\begin{array}{rl}
R(1)[E1] = 1. & \rightarrow, C[E1], \leftarrow \\
R(k+1)[E1, \cdots, E(2^{k+1}-1)] = 1. & R(k)[t \rightarrow t+1], \rightarrow^{k+1}, C[E1], \leftarrow^{k+1} \\
2. & \rightarrow, C[E3], \leftarrow, C[E2] \\
\vdots & \quad \vdots \\
t+1. & \rightarrow^{\rho(k,t)+1}, C[E(2t+1)], \leftarrow^{\rho(k,t)+1}, C[E(2t)] \\
\vdots & \quad \vdots \\
2^k. & \rightarrow^{\rho(k,2^k-1)}, C[E(2^{k+1}-1)], \leftarrow^{\rho(k,2^k-1)}, C[E(2^{k+1}-2)]
\end{array}$$

The notation $R(k)[t \rightarrow t+1]$ here means that, for $t = 1, \cdots, 2^k-1$, exit t of $R(k)[E1, \cdots, E(2^k-1)]$ is connected to line $t+1$, i.e. stands for $R(k)[2, \cdots, 2^k]$. If we now number the blocks of $k+1$ squares according to the binary number they represent, as above, then the weak TM operations on these blocks are: $L = \leftarrow^{k+1}$; $R = \rightarrow^{k+1}$; $E = (\rightarrow, *)^{k+1}, \leftarrow^{k+1}$; $P^{(i)} = \rightarrow, *^{i_1}, \rightarrow, *^{i_2}, \cdots, \rightarrow, *^{i_{k+1}}, \leftarrow^{k+1}$, where $i_1 \cdots i_{k+1}$ is the binary expansion of i ($i = 1, \cdots, 2^{k+1}-1$).

$$\begin{array}{rl}
J^{(i)}[m] = 1. & R(k+1)[2, 3, \cdots, (m+1)', \cdots, 2^{k+1}], \rightarrow^{k+1} \\
2. & \rightarrow^{\rho(k+1,1)}, \leftarrow^{\rho(k+1,2)} \\
3. & \rightarrow^{\rho(k+1,2)}, \leftarrow^{\rho(k+1,3)} \\
\vdots & \quad \vdots \\
2^{k+1}. & \rightarrow^{\rho(k+1,2^{k+1}-1)}, \leftarrow^{k+1}
\end{array}$$

with the compensation: replace old line m by

$$m. \ \rightarrow^{k+1-\rho(k+1,i)}, \quad m+1. \leftarrow^{k+1-\rho(k+1,i)}, \quad \text{old line } m.$$

Here the $(m+1)'$ in line 1 is connected to exit i and, as before, stands for the final number of the added line $m+1$.

APPENDIX E. Proof That All Operations at Both Ends Are Necessary for Computing All Recursive Functions with a Single Register Machine Working on the Same Alphabet

THEOREM. *An SRM on alphabet $\mathfrak{A} = \{a_1, \cdots, a_s\}$ with heads operating at the two ends of the word is not capable of computing all one-place recursive functions over $\mathfrak{A}$ unless both heads are capable of printing, deleting and "reading"* (i.e. *making the letter which is scanned influence the future computation in some way such as a conditional transfer*).

PROOF. Let $\mathcal{P}$ be a given program of $l-1$ lines for an SRM whose r.h.h. (right-hand head) is capable of all three types of operation but whose l.h.h. (left-hand head) is not. We shall show that if $\mathcal{P}$ computes a function which takes infinitely many values, then there exist words U, V, U', V' such that, for every word X, $\mathcal{P}$ sends UXV into $U'XV'$. This proves the theorem since it shows that the function f defined by $f(W) = WW$ is not computable by such a machine.

In order to define U, V, we take a word W_0 with the property that when

$\mathcal{P}$ is applied with W_0 as initial word, then at no stage of the computation will the register contain a word with less than $\max\{l, 2\}$ letters. Such a W_0 certainly exists for there are only finitely many ($s^k l$) different possible final outcomes from a position where the word in the register has only k letters, and we are supposing that $\mathcal{P}$ computes a function with infinitely many values. In defining U it is convenient to imagine that the word which is the content of the register at any time is printed on a doubly infinite tape divided into squares (all of which are blank except those occupied by this word) with the square originally occupied by the first letter of W_0 numbered 0 and the squares to the right of this numbered 1, 2, 3, $\cdots$. The l.h.h. starts on square number 0 and, in the course of the application of $\mathcal{P}$ to W_0, passes over only a finite number of squares. Let n_0 be the number of the furthest square to the right reached by the l.h.h., so that the l.h.h. passes at some time over squares $0, 1, \cdots, n_0$ (and possibly over squares with negative numbers, i.e. to the left of square number 0). Now U is defined as the (n_0+1)-letter word whose first letter is the first letter of W_0 and, generally, whose rth letter is the letter which was on square number r when the l.h.h. reached this square *for the first time*. V is defined in a symmetrical way with respect to the right hand end of the word W_0. If we can show that the program $\mathcal{P}$ applied to the word UXV produces the same succession of steps as when it is applied to W_0, then the result of $\mathcal{P}$ must be to leave the X unscanned, i.e. to produce a word $U'XV'$, where U', V' are independent of X. It is clear therefore that the desired result will follow from the following lemma:

LEMMA A. *If the right-hand head scans a certain square, moves off this square to the right*[29] *and later returns to read this square, then the letter printed on that square will be unchanged (i.e. cannot have been changed by the left-hand head), and similarly for the left-hand head.*

We derive this from another lemma:

LEMMA B. *If the left-hand head cannot read and the right-hand head moves at least $l-1$ squares to the right of the present square before returning to scan it, then it never returns.*

PROOF OF LEMMA B. Let the squares be numbered $0, 1, 2, \cdots$ to the right, starting from the present square. Consider, for each of the l squares $0, 1, \cdots, l-1$ the number of the line of the program in which that square is left by the r.hh. *for the last time* on its way out to square $l-1$. Since there are only l lines in the program there must exist i_1, i_2, $0 \leqq i_1 < i_2 \leqq l-1$, which are left in the same program line, l_0, say. This means that after leaving square i_1 in line l_0 the r.h.h., before returning to this square, arrives at square i_2 in line l_0. In other words, when started with the r.h.h. on square i_1 in line l_0, facing blank squares to the right, the computation proceeds, *without the r.h.h. scanning square i_1 or any square to the left of it*, through steps which bring the r.h.h. to square i_2 with the program again on line l_0. Since the l.h.h. cannot read, it cannot cause any change in procedure, so the conditions are now effectively the same as before, i.e. the r.h.h. must now go on to the square $i_2+(i_2-i_1)$,

[29] Remaining on the square is considered to fall under this description also.

leave this in the same line l_0 of the program and so on, i.e. the motion of the machine will consist in the r.h.h. moving cyclically and endlessly to the right.

PROOF OF LEMMA A. We are supposing the l.h.h. is not capable of all three operations. There are thus three cases to be considered:

(1) *The l.h.h. cannot read.* In this case Lemma B immediately gives the result of Lemma A for the r.h.h., since we are supposing that the word in the register is never reduced to fewer than l letters, so that the l.h.h. could only arrive at and alter a certain square if the r.h.h. had proceeded at least $l-1$ squares to the right, in which case Lemma B shows that the r.h.h. never returns to scan the altered square. For the l.h.h. Lemma A is vacuously true, since the l.h.h. is supposed to be unable to read.

(2) *The l.h.h. cannot print.* In this case the l.h.h. certainly cannot alter a square between scannings by the r.h.h.—nor can it delete it since this would imply the word was at some stage reduced to null. Nor can the r.h.h. alter a square scanned by the l.h.h., for since the l.h.h. cannot move left it could only satisfy the conditions of lemma A by remaining on the square and then the r.h.h. could only alter this square by reducing the word to length 1, contrary to hypothesis.

(3) *The l.h.h. cannot delete.* Then the l.h.h. cannot move right so it could only alter a square which had been occupied by the r.h.h. if it was present on this square when the r.h.h. was, i.e. if the word was at some time reduced to one letter, contrary to hypothesis. Also, since the l.h.h. cannot move right, it can only satisfy the conditions of Lemma A itself by staying still, in which case the r.h.h. cannot alter the square it (the l.h.h.) is occupying without reducing the word to length one, contrary to hypothesis.

REFERENCES

1. CHURCH, A. A set of postulates for the foundation of logic. *Ann. Math.* (2) *33* (1932), 346–366; *34* (1933).
2. ——. An unsolvable problem of elementary number theory. *Amer. J. Math. 58* (1936), 345–363.
3. ——. A note on the Entscheidungsproblem. *J. Symb. Logic 1* (1936), 40–41, 101–102.
4. DAVIS, M. *Computability and Unsolvability.* New York, 1958.
5. ERSHOV, A. P. On operator algorithms. (Russian) Dok. Akad. Nauk *122* (1958), 967–970. English translation, *Automat. Express 1* (1959), 20–23.
6. HERMES, H. *Vorlesung über Entscheidungsproblemen in Mathematik und Logik.* Ausarb. Math. Phys. Vorlesungen, Vol. 15. Munster, 1955.
7. HERMES, H. Die Universalität programmgesteuerter Rechenmaschinen. *Math.-Phys. Semsterberichte* (Göttingen) *4* (1954), 42–53.
8. KAPHENGST, H. Eine Abstrakte programmgesteuerte Rechenmaschine. Zeit. Math. Logik Grund. d. Math: *5* (1959), 366–379.
9. KLEENE, S. C. General recursive functions of natural numbers, *Math. Ann. 112* (1936), 727–742.
10. ——. A theory of positive integers in formal logic. *Amer. J. Math. 57* (1935), 153–173, 219–244.
11. ——. λ-definability and recursiveness. *Duke Math. J. 2* (1936), 340–353.
12. ——. *Introduction to Metamathematics*, Ch. 13. Princeton, 1952.

13. Markov, A. A. *Teoriya algorifmov.* Tr. Mat. Inst. Steklov, No. 42. Moscow, 1954.
14. Oberschelp, W. *Varianten von Turingmaschinen, Arch. math. Logik Grund.*, No. 4/1-2 (1958), 53-62.
15. Péter, R. *Graphschemata und rekursive Funktionen, Dialectica 12* (1958), 373.
16. Post, E. L. Finite combinatory processes—formulation, I. *J. Symb. Logic 1* (1936), 103-105.
17. ——. Formal reductions of the general combinatorial decision problem. *Amer. J. Math. 65* (1943), 197-215.
18. Turing, A. M. On computable numbers with an application to the Entscheidungsproblem. *Proc. Lond. Math. Soc.* {2}, *42* (1936-7), 230-265; addendum and corrigendum, *43* (1937), 544-546.
19. ——. Computability and λ-definability. *J. Symb. Logic 2* (1937), 153-163.
20. Wang, H. A variant to Turing's theory of computing machines. *J. ACM 4* (1957), 63-92.
21. Minsky, M. Recursive unsolvability of Post's problem. M.I.T. Lincoln Lab. Report 54G-0023.
22. Smullyan, R. M. *Theory of Formal Systems.* Princeton, 1961.
23. Post, E. L. Recursive unsolvability of a problem of Thue. *J. Symb. Logic 12* (1947), 1-11.
24. Lee, C. Y. Categorizing automata by W-machine programs. *J. ACM 8* (1961), 384-399.