COMPUTATION:

FINITE AND INFINITE MACHINES

MARVIN L. MINSKY

Professor of Electrical Engineering
Massachusetts Institute of Technology

PRENTICE-HALL, INC.
ENGLEWOOD CLIFFS, N. J.

© 1967 by
Prentice-Hall, Inc.
Englewood Cliffs, N. J.

All rights reserved.
No part of this book may
be reproduced in
any form or by any means
without permission in writing
from the publisher.

Current printing (last digit):

15 14

Library of Congress Catalog Card No. 67-12342
Printed in the United States of America

14 VERY SIMPLE BASES FOR COMPUTABILITY

14.1 UNIVERSAL PROGRAM MACHINES WITH TWO REGISTERS

In section 11.1 we introduced "program machines" which could compute any recursive function by executing programs, made up of the two operations below, on the contents of registers—number-containing cells.

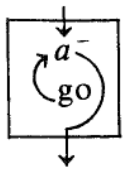
$\boxed{a'}$	Add unity to the number in register **a**, and go to next instruction.
$\boxed{a^-(n)}$	If the number in **a** is not zero, then subtract 1 from **a** and go to the next instruction, otherwise go to the *n*th instruction.

We showed, in 11.2, that these operations working on just five registers were enough to construct an equivalent of any Turing machine, and we remarked in 11.4 that this could be done with just two registers; we will now prove this. But our purpose is not merely to reduce the concept of program machine to a minimum, but to also use this result to obtain a number of otherwise obscure theorems.

We recall (from 11.4) that the operation $\boxed{a^0}$, i.e., put zero in register **a**, can be simulated if we have a register **w** already containing zero; then we can also use $w^-(n)$ as a $\boxed{\text{go}(n)}$ instruction.

For our purposes here it is more convenient to assume that we have

$\boxed{a'}$, $\boxed{a^-(n)}$, and $\boxed{\text{go}(n)}$ at the start. Then



will serve as a^0, and we can assume that we have

$$a^0 \qquad a^-(n) \qquad a' \qquad \text{go}(n)$$

These are the only operations that appear in the diagram of 11.2, so they are shown to be all we need to simulate any Turing machine.

To reduce the machine of Fig. 11.2-1 to one with just two registers **r** and **s**, we will "simulate" a larger number of virtual registers by using an elementary fact about arithmetic—namely that the prime-factorization of an integer is unique. Suppose, for example, that we want to simulate *three* registers **x**, **y**, and **z**. We will begin by placing the number $2^x3^y5^z$ in register **r** and zero in register **s**. The important thing is that from the single number $2^x3^y5^z$ one can recover the numbers x, y, and z simply by determining how many times the number can be divided by 2, 3, and 5 respectively. For example, if $r = 1440$, then $x = 5$, $y = 2$, and $z = 1$. For our purposes, we have only to show how we can obtain the effect of the operations x' and x^-, y' and y^-, and z' and z^-.

INCREMENTING

Suppose we want to increment **x**, that is, add unity to x. This means that we want to replace $2^x3^y5^z$ by $2^{x+1}3^y5^z = 2 \cdot 2^x3^y5^z$. But this is the same as doubling the number in **r**! Similarly, incrementing **y** and **z** is trebling and quintupling (respectively) the contents of **r**. And this is done by the programs of Fig. 14.1-1. The first loop in each program counts **r** down while counting **s** up twice (or three or five times) as fast; the second loop transfers the contents of **s** back into **r**.

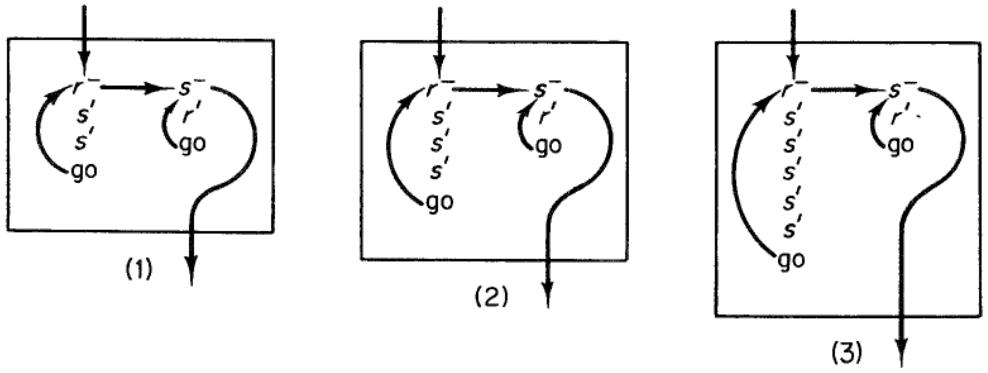


Fig. 14.1-1

Decrementing

Subtracting unity from x is a little more tricky, since we have to determine whether x is zero; if x is zero, we want to leave it unchanged and do a $\boxed{\text{go}}$. If x is not zero, then we want to change $2^x3^y5^z$ into $2^{x-1}3^y5^z$—that is, divide the contents of **r** by two. Similarly, decrementing y and z is (conditional on not being zero) equivalent to dividing by three and five. We illustrate in Fig. 14.1-2 a program for the 5-case.

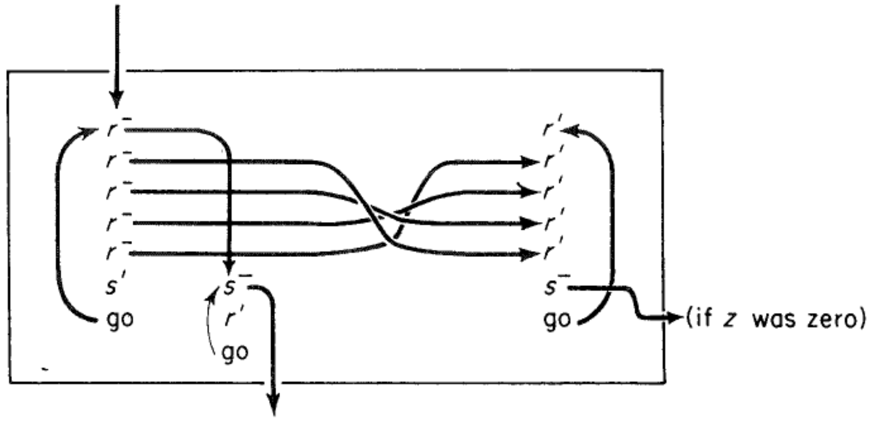


Fig. 14.1-2

The loop to the left does the division, by repeated subtraction. If the division comes out exact—that is, has no remainder—then the lower loop copies the quotient back into **r**. If the division was inexact (i.e., if z was zero), the loop to the right first restores the remainder (which at that moment is stored in the state of the machine—i.e., the location in the program) and then multiplies the quotient by the divisor, putting the result (which is the original contents) back into **r**.

This is all we have to show, for clearly we can do the same for the *five* registers mentioned in 11.2. We simply put in **r** the number $2^m3^n5^a7^z11^w$ and use the same techniques given just above. To build a program machine equivalent to that of Fig. 11.2-1, we take that diagram and replace *each* of its individual instructions by a copy of the equivalent program structure we have just developed. This proves:

Theorem 14.1-1

For any Turing machine T there exists a program machine M_T with just two registers that behaves the same as T (in the sense described in sections 10.1 *and* 11.2) *when started with zero in one register and* $2^a3^m5^n$ *in the other. This machine uses only the operations* $\boxed{\,'\,}$ *and* $\boxed{-}$ *, assuming that the successor instruction contains the "go" information for the next instruction.*

REMARK

Now that we know there are universal program machines that use only *two* registers, we can improve the result of section 12.8. For such a machine there are just four kinds of instructions: "add 1 to R_1," "add 1 to R_2," "subtract (conditional) 1 from R_1," "subtract (conditional) 1 from R_2." In such a machine, we can omit the R letters and just use the I's to separate the two registers, so that we can have a canonical system for such a machine with simple productions like

$$\$_1 I_j \$_2 \to \$_1 1 I_{j+1} \$_2$$

for addition, and

$$\$_1 I_j 1 \$_2 \to \$_1 I_{j+1} \$_2$$
$$\$ I_j \to \$ I_{j'}$$

for the conditional subtraction.

14.2 UNIVERSAL PROGRAM-MACHINE WITH ONE REGISTER

We can get an even stronger result if we admit multiplication and division operations (by constants) as possible machine instructions, for then we can do everything with a single register! The proof of theorem 14.1-1 shows that we need only a single register if we can perform on it the operations "multiply by 2 (or 3, 5, 7, 11)" and "divide by 2 (or 3, 5, 7, 11) conditionally upon whether the division is exact." Now one last observation. The proof of theorem 14.1-1 shows that the effects of multiplication by 2 (or by 3) and division by 2 (or by 3) on a number of the form $2^x 3^y$ are precisely the same as the effects of incrementing **x** (or **y**) and decrementing **x** (or **y**). Furthermore, by theorem 14.1-1, we don't really need anything but these four operations, if we are willing to raise our numbers up to another exponent level. Thus, by applying the result of theorem 14.1-1 to its own proof (!) we obtain:

THEOREM 14.2-1

For any Turing machine T there exists a program machine M_T^* with just one register that behaves the same as T (in the usual sense) when started with $2^{2^a 3^m 5^n} \cdot 3$ in its register. This machine uses only the operations of multiplication and (conditional) division by integers 2 and 3.

PROBLEM 14.2-1. Verify the correctness of this fast-talking proof for Theorem 14.2-1.

PROBLEM 14.2-2. In both theorems 14.1-1 and 14.2-1, the final result was a program machine with *four* instruction types. In one case there were two operations that could each be applied to each of two registers; in the other case, there were *four* operations on one register. Show that in each case we can reduce the number to three, by replacing two of them with one that affects both numbers or registers. For example, in the machine M_T we can eliminate s^- and s' if we keep r^- and r' and adjoin *exchange* (r, s). In M_T^* we can use *multiply by* 6, *divide by* 3, and *divide by* 2. We can even reduce the basis to two instructions: Show that for machine M_T it is sufficient to have:

> Add 1 to r and exchange r and s.
> If r is not zero, subtract 1 from r and exchange;
> otherwise exchange and go(n).

A slight change in input form is required.

DISCUSSION

Theorem 14.2-1 looks, superficially, like a better result than theorem 14.1-1, since one register is less than two. However, from the point of view of Turing's argument (chapter 5), we prefer theorem 14.1-1 because we can think of m', m^-, n', and n^- as unitary, basic, finite actions. On the other hand, the operation of multiplying the contents of a register by two cannot be regarded as a fixed, finite action, because the amount of work involved must grow with the size of the number in the register, beyond any fixed bound. (The same objection could be held against theorem 14.1-1 if the number in the register is represented as a binary string. It must be unary. Why?)

14.3 GÖDEL NUMBERS

The methods of section 14.1 draw their surprising power from the basic fact that one can represent any amount of information by selecting a single integer—or, more precisely, that one can effectively compute such an integer and effectively recover the information from it. Thus, in the proof of theorem 14.1-1 we take an arbitrary quadruple (m, n, a, z) of integers and encode them into the single integer $2^m3^n5^a7^z$. There is nothing new here, for in our original arithmetization of Turing machines (section 10.1) we use the numbers m and n themselves to represent arbitrary sequences of 1's and 0's, and in our discussion (10.3) of enumeration of the partial-recursive functions we envisioned an even more complex encoding of information into single integers. In 12.3 we showed that a single integer could represent an arbitrary list of lists of letters from an alphabet. More generally, as was apparently first pointed out by Gödel [1931], a single integer representation can represent an arbitrary list structure (see also 10.7). Let us define such a correspondence, inductively:

If K is a number, then $K^* = 2^K$
If K is a list $(a, b, c, d, \ldots)$, then $K^* = 3^{a^*}5^{b^*}7^{c^*}11^{d^*}\ldots$

For example,

$$3^* = 2^3$$

$$(2, 4, 6)^* = 3^{2^2} \cdot 5^{2^4} \cdot 7^{2^6}$$

$$(1, (2, 3), 4)^* = 3^{2^1} \cdot 5^{3^{2^2} \cdot 5^{2^3}} \cdot 7^{2^4}$$

$$(0, (0, 0))^* = 3 \cdot 5^{3 \cdot 5}$$

PROBLEM 14.3-1. Show that any two different list structures of integers yield different *-numbers, and describe an effective procedure to recover the list structure from the *-number.

PROBLEM 14.3-2. Which of the following definitions have the property that the list structure can be unambiguously recovered from the number?

$$(1) \begin{cases} K^* = K & \text{(if } K \text{ is a number)} \\ K^* = 2^{a^*}3^{b^*}5^{c^*}7^{d^*}\ldots & \text{(if } K = (\text{a, b, c, d}, \ldots)) \end{cases}$$

$$(2) \begin{cases} K^* = 2K & (K \text{ a number}) \\ K^* = 3^{a^*}5^{b^*}7^{c^*}11^{d^*}\ldots & (K \text{ a list}) \end{cases}$$

$$(3) \begin{cases} K^* = 3^K & (K \text{ a number}) \\ K^* = 3^{2^{a^*}}5^{2^{b^*}}7^{2^{c^*}}11^{2^{d^*}}\ldots & (K \text{ a list}) \end{cases}$$

The idea of a Gödel numbering was important in mathematical logic because while that subject is, on the surface, concerned with notions of number and the foundations of mathematics, it has come more and more to be a theory of *mathematical theories*. It is possible to use the number-theoretic formalisms, as Gödel showed, to talk about the sentences of the theory (and sentences are not numbers) by assigning numbers to sentences and properly interpreting corresponding numbers. In particular this makes it possible to interpret a theory as talking about its own sentences, without paradoxes; and Gödel proved his celebrated theorem, about the impossibility of a non-contradictory theory containing a proof of its self-consistency, using this technique. More recently it has been noted that one can use numbering schemes much simpler than Gödel's for the same effect, avoiding number-theory facts like the unique-factorization theorem. The methods of Smullyan [1961] are particularly elegant; Smullyan was strongly influenced by Post's inclination to deal directly with symbolic expressions rather than numerical representations of them. (So was I, for example, in the proof of theorem 13.3-1). But I believe methods like those of section 10.7, or those of 13.3, that step around arithmetic entirely, are ultimately the clearest and most illuminating.

14.4 TWO-TAPE NON-WRITING TURING MACHINES

It is easy enough to generalize the idea of a Turing machine to a machine with two or more tapes. One way to do this is to specify, with each state of the Turing machine, what it is to do with each tape (what to write, and which way to move) for each *combination* of symbols seen at the set of reading heads. In this formulation one would use, instead of the quintuple $(q_i, s_j, q_{ij}, s_{ij}, d_{ij})$ of the ordinary machine, a specification like the following—for K tapes, a $2K + 3$-tuple:

$$(q_i; s_{i_1 \cdots i_K}; q_{i, i_1 \cdots i_K}; s_{i_{11} \cdots i_{1K}}, \ldots, s_{i_{K1} \ldots i_{KK}}; D_{i_{11} \ldots i_{1K}}, \ldots, D_{i_{K1} \cdots i_{KK}})$$

Another, more convenient, way to describe a multi-tape machine would be in terms of a program machine whose instructions include:

Move tape j in direction d.
Write symbol s_i on tape j.
If head j reads s_i, go to instruction . . . .

As we know, it is very hard to construct Turing machines to do tasks of any significant complexity, even for theoretical purposes, and no one would consider using one for a practical purpose. It is much easier to sketch out the organization of a multi-tape Turing machine for a complex computation if one simply assigns different tapes to different memory functions; then it is not necessary to resort to tricky punctuation devices. For example, it is very easy to describe a universal Turing machine using two tapes—one for simulating the tape of, and one for holding the description of, the (one-tape) machine being imitated.

PROBLEM. Show how to construct a simple universal machine using two tapes.

It should come as no surprise that, in spite of the greater apparent power of multi-tape machines, their ultimate computation range is the same as usual, namely, the computation of any partial-recursive function. This can be shown by describing how to make a conventional, one-tape, Turing machine that imitates K-tape machines.

In our original, heavy-handed construction of a universal Turing machine (section 7.3), we reserved two regions of a machine's tape for different purposes. One of these—the description region—was finite, but it could just as well have been infinite. The same method won't work for more than two tapes, of course, since a tape has only two directions; but one can get the effect of K tapes simply by assigning every Kth square of the tape to be used for imitating a given tape. We leave the construction as an exercise.

PROBLEM. Show that any K-tape machine can be simulated by a one-tape machine.

Although K-tape machines, in general, are equivalent to one-tape machines, this conceivably might not be so if we impose restrictions on what can be done with the tapes. It comes, therefore, as something of a surprise that:

THEOREM 14.4-1

Any computation that can be done by a Turing machine can be simulated by a machine with two semi-infinite (single-ended) tapes which can neither read nor write on its tapes, but can only sense when a tape has come to its end.

In view of theorem 14.1-1, the proof is hardly worth writing down. We use the *length* from the reading heads to the ends of the two tapes as our representation of m and n; the operations m' and n' move the heads away from the ends, and m^- and n^- move the heads towards the ends, conditional on reaching an end of tape.

Another consequence of the same situation is this corollary:

COROLLARY 14.4-1

Any Turing-machine computation can be simulated by a Turing machine whose tape is always entirely blank, save for at most three 1's.

For one can construct a Turing machine, equivalent in the (m, n) sense to the given machine, which works with a tape of the form

$$0\ 0\ 0 \ldots 0\ \mathit{1}\ 0 \ldots (m \text{ zeros}) \ldots 0\ \mathit{1}\ 0 \ldots (n \text{ zeros}) \ldots 0\ \mathit{1}\ 0$$

PROBLEM. Show how to construct the machine for the proof of the corollary.

14.5 UNIVERSAL NON-ERASING TURING MACHINES

We can now demonstrate the remarkable fact, first shown by Wang [1957], that for any Turing machine T there is an equivalent Turing machine T_N that *never changes a once-written symbol*! In fact, we will construct a two-symbol machine T_N that can only change blank squares on its tape to 1's but can not change a 1 back to a blank.

Our proof will have two parts. First we will show, given T, how to make an equivalent machine T'_N that uses four symbols 0, A, B, C and is subject to the *symbol-changing restriction* that the only changes permitted are

$$\boxed{0 \to A, 0 \to B, 0 \to C, A \to B, A \to C, B \to C}$$

Then we will show how to make the non-erasing machine T_N out of this T'_N by making the two-symbol T_N work with its tape-squares grouped into binary triplets, so that we can make the identification:

T'_N symbols		T_N symbols
0	$\leftrightarrow$	$\boxed{0\vert 0\vert 0}$
A	$\leftrightarrow$	$\boxed{1\vert 0\vert 0}$
B	$\leftrightarrow$	$\boxed{1\vert 1\vert 0}$
C	$\leftrightarrow$	$\boxed{1\vert 1\vert 1}$

The point is that the permitted symbol-changes in T'_N correspond to changes for T_N that only transform 0's to 1's; no 1's need ever be changed to 0's.

THEOREM 14.5-1

For any Turing machine T there is an equivalent two-symbol machine which can only print 1's on blank squares but can never erase a 1, once printed.

Proof: First we construct the machine T'_N mentioned above. T'_N will work with our standard two-integer (m, n) representation for the content of a Turing-machine tape, and we will represent the state of T's tape of the form

$$\ldots CCCC \ldots CCCBB \ldots BBAA \ldots AA000 \ldots 000 \ldots$$

where there are m B's, n A's, and any number of C's to the left. We have only to show that we can make state diagrams that will perform the basic

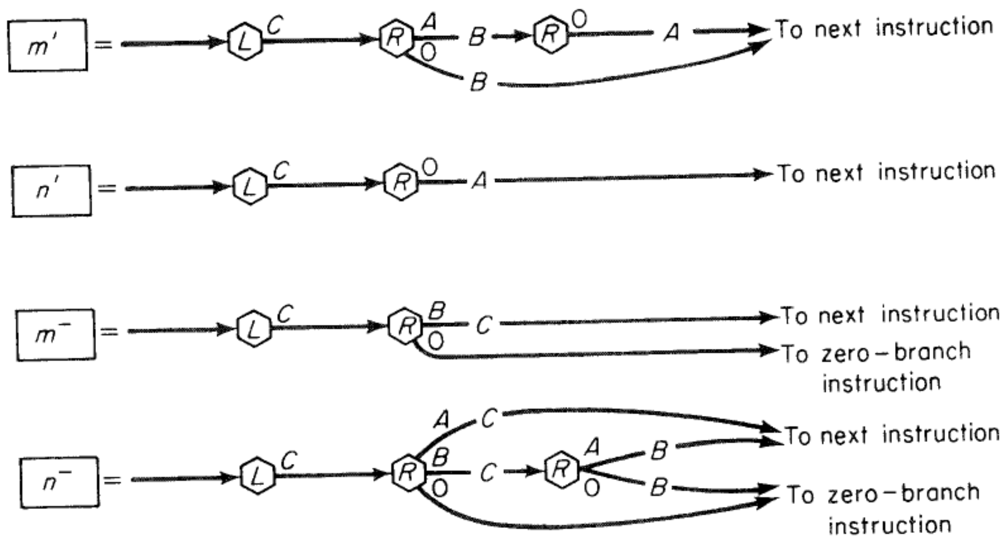
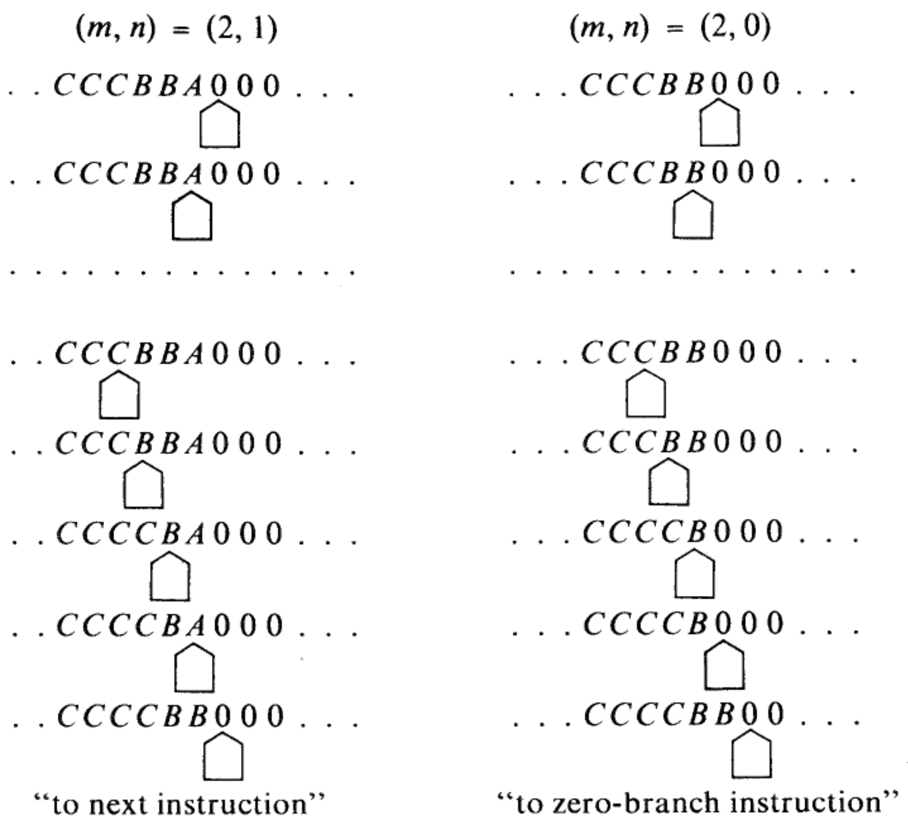


Fig. 14.5-1

operations of theorem 14.1-1, namely, m', n', m^-, and n^-; the latter being conditional on whether m and n are already zero. That is, we have to show that we can increment and (conditionally) decrement m and n—the numbers of B's and A's on T'_N-s tape, without violating the symbol-changing restrictions. Now consider the four state diagrams in Fig. 14.5-1. These do exactly what is wanted. For example, apply the state-diagram for n^- to the strings for $(m, n) = (2, 1)$ and $(2, 0)$, starting somewhere to the right in each case:



In each case the machine first runs to the left until it encounters the block of C's. (It will never move left of the rightmost C.) It then starts back to the right to perform the desired operation. Only operation n^- has any subtlety at all; the machine is supposed to "erase" an A (if there is one); it can only do this by changing that A to a higher letter—B or C. It prepares for this by removing the leftmost B (if there is one); when it meets an A, it will change this to a B, thus reducing the number of A's and restoring the number of B's. The reader can verify that we have also accounted for the exceptional case in which there is no B; finally, in the case that there is no A (that is, if $n = 0$), B is unchanged but the machine takes a different exit from the state-diagram. Note also that in each case the machine ends somewhere to the right of the block of C's, so that we can safely link the output of one diagram to the input of another.

To obtain the machine T'_N we now simply find a program machine equivalent to T by using theorem 14.1-1. Then we take each instruction of the program machine and replace it by the appropriate one of the four state-diagram machines just exhibited. Last, we realize the "jump" or "control" structure of the program machine by making corresponding connections of the inputs and outputs of our set of little state diagrams. Since none of the state diagrams violate the symbol-changing restriction, we have constructed T'_M as required.

To complete the proof of theorem 14.5-1, we have only to show how to convert T'_M into a two-symbol non-erasing machine T_M. To do this, we have to provide the mechanism, mentioned earlier, through which T_M will work with its tape as though it were formed of triplets of squares. We will do this by replacing *each state* of T'_M by a state network that operates on triplets of squares in accord with

$$0 \leftrightarrow \boxed{0}\boxed{0}\boxed{0}, \quad A \leftrightarrow \boxed{1}\boxed{0}\boxed{0}, \quad B \leftrightarrow \boxed{1}\boxed{1}\boxed{0}, \quad C \leftrightarrow \boxed{1}\boxed{1}\boxed{1}$$

For example, the state of $(L)\xrightarrow{C}$, which means "move left (by triplets) until encountering the triplet $\boxed{1}\boxed{1}\boxed{1}$, is realized in T'_N by the network of Fig. 14.5-2, where the loop-structure is used to make sure that the system always moves three squares. The right-moving states of T'_M are replaced by similar structures; for example, the right side of Fig. 14.5-3 would replace the left side, and the right side of Fig. 14.5-4 would replace the left side. The latter example shows again how a searching state keeps moving along by repeating triplets until it finds one of the symbols it is searching for.

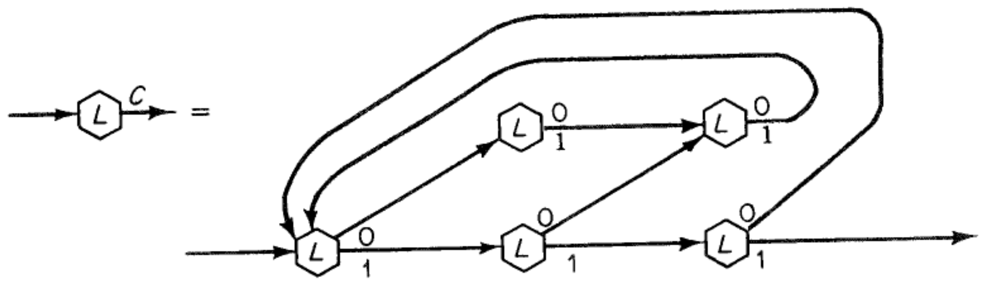


Fig. 14.5-2

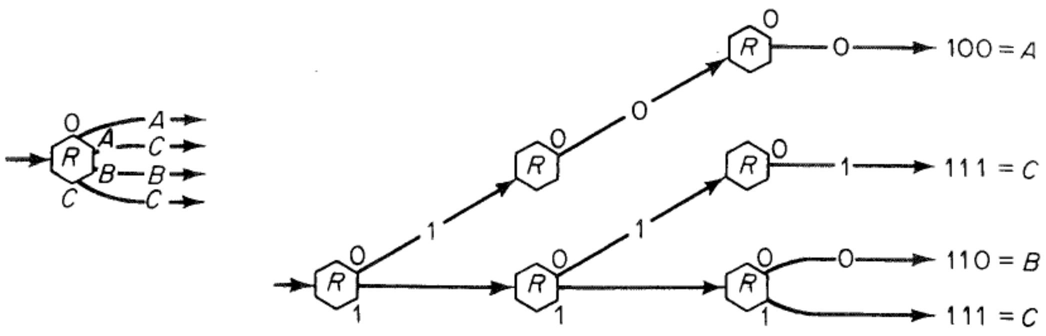


Fig. 14.5-3

PROBLEM. The machine T_M just described wipes out all prior records of its computation, by changing all old symbols to C's. It is possible to make a machine similar in operation except that a permanent record of all prior steps in the computation (that is, all steps of the original machine T, being simulated) is retained. Construct such a machine T_M^* with the property that, if (m_t, n_t) are the (m, n)-representation of T's tape at time t, then T_M^*'s tape at time t has the form $m_1n_1m_2n_2 \ldots m_tn_t \ldots$ in some straightforward sense. For example, one can use a five-level "non-erase" encoding

$$\ldots EEE \ldots EEED^{m_1}C^{n_1}D^{m_2} \ldots D^{m_{t-1}}C^{n_{t-1}}B^{m_t}A^{n_t}000 \ldots 000 \ldots$$

First make a state diagram that will "copy" m_tn_t by converting the above string into

$$\ldots EEE \ldots EEED^{m_1}C^{n_1}D^{m_2} \ldots D^{m_{t-1}}C^{n_{t-1}}D^{m_t}C^{n_t}B^{m_t}A^{n_t}000 \ldots$$

and then show how to modify the "copy" structure to realize each of the operations m', n', m^-, n^-, without violating the symbol-changing restrictions. Compare this system with the universal Post system of theorem 13.3-1, which also keeps a record of all previous work.

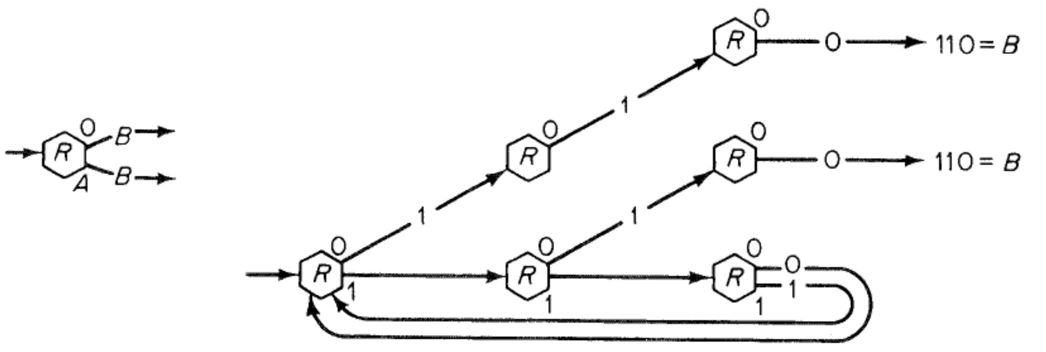


Fig. 14.5-4

14.6 THE PROBLEM OF "TAG" AND MONOGENIC CANONICAL SYSTEMS

While a graduate student at Princeton in 1921, Post [1965] studied a class of apparently simple but curiously frustrating problems of which the following is an example:

> Given a finite string S of 0's and 1's, examine the first letter of S. If it is 0, delete the first three letters of S and append 00 to the result. If the first letter is 1, delete the first three letters and append 1101. Perform the same operation on the resulting string, and repeat the process so long as the resulting string has three or more letters.
>
> For example, if the initial string is 10010, we get

```
10010
   101101
      1011101
         11011101
            111011101
               0111011101
                  101110100
                     1101001101
                        10011011101
                           110111011101
                              1110111011101
                                 01110111011101
                                    1011101110100
                                       11011101001101
                                          111010011011101
                                             0100110111011101
                                               >011011101110100
                                                   01110111010000
                                                      1011101000000
                                   repeats               11010000001101
                                                            100000011011101
                                                               0000110111011101
                                                                 <011011101110100
```

The string has grown, but it has just repeated itself (and hence will continue to repeat the last six iterations forever). Suppose that we start with a different string S'. The reader might try, for example, $(100)^7$, that is, 100100100100100100100, but he will almost certainly give up without answering the question: "Does this string, too, become repetitive?" In fact, the answer to the more general question "Is there an effective way to decide, for any string S, whether this process will ever repeat when started with S?" is still unknown. Post found this (00, 1101) problem "intractable," and so did I, even with the help of a computer. Of course, unless one has a theory, one cannot expect much help from a computer (unless *it* has a theory) except for clerical aid in studying examples; but if the reader tries to study the behavior of 100100100100100100 without such aid, he will be sorry.

Post mentions the (00, 1101) problem, in passing, in his [1943] paper—the one that announces the normal-form theorem—and says that "the little progress made in the solution . . . of such problems make them candidates for unsolvability." As it turns out he was right. While the solvability of the (00, 1101) problem is still unsettled (some partial results are discussed by Watanabe [1963]), it is now known that *some* problems of the same general character are unsolvable. Even more interesting is the fact that there are systems of this class that are universal in the sense of theorem 14.1-1; namely, there is a way to simulate an arbitrary Turing-machine computation within a "tag" system.

DEFINITION

A tag system is a Post normal canonical system that satisfies the conditions: If $A = (a_1, \ldots, a_n)$ is the alphabet of the system, and

$$g_i\$ \rightarrow \$h_i \qquad (i = i, \ldots, n)$$

are its productions, then

> (1) All the antecedent constant strings g_i have the same length P.
> (2) The consequent string h_i depends *only on the first letter* of the associated g_i.

For example, in the (00, 1101) problem just mentioned, there are really eight productions, forming the system with $P = 3$:

000\$ → \$00	100\$ → \$1101
001\$ → \$00	101\$ → \$1101
010\$ → \$00	110\$ → \$1101
011\$ → \$00	111\$ → \$1101

Because of the fact that the consequent h_i is determined by the first letter only of g_i, and that the number of letters in the g's is a constant P, the tag systems all have the character of the (00, 1101) problem; namely, to operate a tag system, one has to:

> Read the first letter a_i.
> Erase P letters from the front of the string.
> Append the associated consequent string h_i to the end of the string.

It is very important to observe that the very definition of a tag system gives it a property not found, generally, in normal or other canonical systems; namely, a tag system is *monogenic*.

DEFINITION

A Post canonical system (or any other logical string-manipulation system) is *monogenic* if, for any string S, there is at most one new string S' that can be produced from it (in one step).

Clearly a tag system is monogenic since, for any string, what happens to it depends only on its first letter; two different strings can be produced only if a string has two different first letters, which would be absurd. What is the importance of the monogenic property? It is that, if a string-manipulation system is monogenic, then it is like a machine in all important respects, for it defines a definite *process* or sequence of things that happen—and these can be regarded as happening in real time, rather than as mere theorems about an unchanging mathematical world or space.

In fact, one can imagine a special machine associated with a tag system. (See Fig. 14.6-1.) This machine is a little like a Turing machine except that

(1) There are two heads, one for reading and one for writing.
(2) The tape begins at a source, runs through the writing head, has an arbitrarily long piece of "slack,"

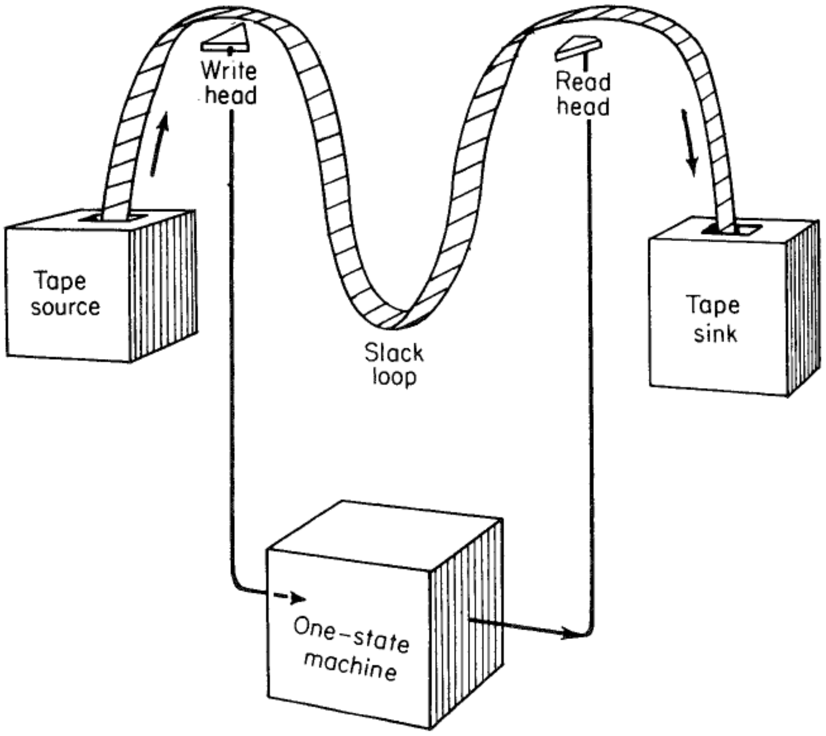


Fig. 14.6-1

then runs through the reading head, and finally disappears forever into a "sink." *It can move only in one direction.*

(3) The finite-state part of the machine must be able to read a symbol, advance the tape P squares, and write the appropriate h_i with the write head. *But otherwise, the machine has no internal states*!

Note that the tag machine cannot erase a symbol. It would do it no good to erase a symbol, since *it can only read a symbol at most once*!

The name "tag" comes from the children's game—Post was interested in the decidability of the question: Does the reading head, which is advancing at the constant rate of P squares per unit time, ever catch up with the write head, which is advancing irregularly. (We do not require the h_i's to have the same lengths.) Note, in the (00, 1101) problem, that the read head advances three units at each step, while the write head advances by two or four units. Statistically, one can see, the latter has the same average speed as the former. Therefore, one would expect the string to vanish, or become periodic. One would suppose this for most initial strings, because if the chances are equal of getting longer or shorter, then it is almost certain to get short, from time to time. Each time the string gets short, there is a significant chance of repeating a previously written string, and repeating once means repeating forever, in a monogenic process. Is there an initial string that grows forever, in spite of this statistical obstacle? No one knows. All the strings I have studied (by computer) either became periodic or vanished, but some only after many millions of iterations!

THEOREM 14.6-1 (Cocke [1964])

For any Turing machine T there exists a tag system T_T that behaves like T, in the (m, n) sense of 11.2, *when given an axiom that encodes T's tape as $Aa\quad aa\quad aa \ldots aa\quad Bb\quad bb\quad bb \ldots bb$ with m aa's and n bb's. The tag system T_T* has deletion number $P = 2$.

COROLLARY 14.6-1

Computability with monogenic normal systems is equivalent to computability with general-recursive functions, Turing machines, general canonical systems, etc.

Proof: We will construct separate tag systems for each of the states of the machine T. Then we will link these together (by identifying certain letters of the different alphabets) to form a single tag system that behaves like the whole machine T. We will begin by constructing a tag system that behaves like the right-hand side of Fig. 11.2-1.

We have been accustomed to thinking of a Turing machine as operating according to the scheme of Fig. 14.6-2. It is equivalent, more con-

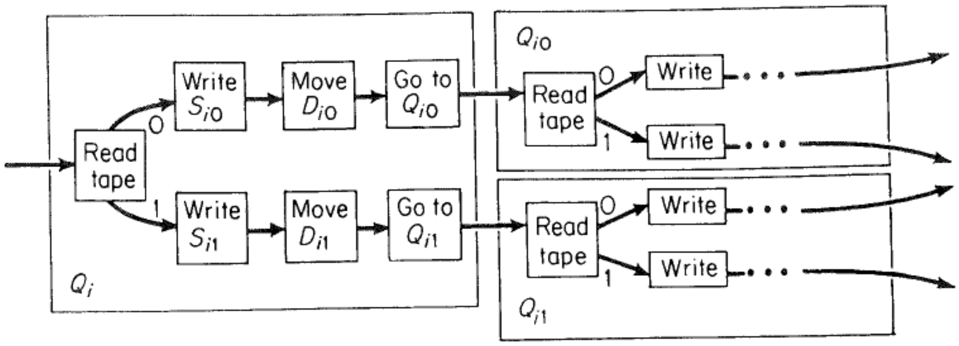


Fig. 14.6-2

venient here, and actually fundamentally simpler to think of the Turing machine as made up of states according to the scheme of Fig. 14.6-3. Looked at this way, a Turing machine will have twice as many states, but in some sense these were concealed in it already, for it must have had a secret pair of states to remember what it had read while it was writing something else.

PROBLEM. Obviously states in the new system do not correspond exactly to states in the old system. The new states are also quintuples, but of the form:

(State	Write	Move	Read: if 0 go to	if 1 go to)
Q_i	S_i	D_i	Q_{i0}	Q_{i1}

and there is only one quintuple for each state, rather than two. Use this formulation of Turing machines to simplify the development of section 11.3.

Now to realize such a state by a tag system, we will exhibit a set of productions that will have the effect of a *move-right* state. In such a case, we will want to change m and n so that

$$m \to 2m + S_i$$

$$n \to H(n) = \begin{cases} n/2 & \text{if } n \text{ is even,} \\ (n-1)/2 & \text{if } n \text{ is odd.} \end{cases}$$

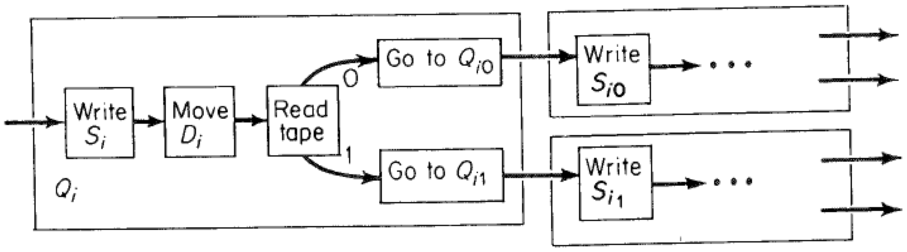


Fig. 14.6-3

and we must prepare the system so that it will next go to state Q_{i0} if n is even, and to Q_{i1} if n is odd.

Let us begin with the string

$$Aa(aa)^m Bb(bb)^n$$

where $@^n$ means to repeat the string $@$ n times. Our first productions will be

$$\boxed{\begin{array}{l} A \rightarrow Cc \\ a \rightarrow cccc \end{array}} \quad \text{or} \quad \boxed{\begin{array}{l} A \rightarrow Cccc \\ a \rightarrow cccc \end{array}}$$

depending on whether S_i is supposed to be 0 or 1. (This depends only on what state we are in.) *For brevity, we will abbreviate tag productions by showing only the antecedent first letter and the consequent string*, since writing the $a_i x \ \$ \rightarrow \$h_i$ form is so redundant. In every case P, the number of letters deleted, is 2. Applying the productions above leads to $Bb(bb)^n Cc(cc)^{m'}$ where m' is either $2m$ or $2m+1$ depending on whether or not the state required the machine to write 1 or 0 on the square it is leaving. The key problem is now to determine whether n is even or odd. The procedure for doing this is initiated by the productions

$$\boxed{\begin{array}{l} B \rightarrow S \\ b \rightarrow s \end{array}} \quad \text{i.e.,} \quad \begin{array}{l} Bx\$ \rightarrow \$S \\ bx\$ \rightarrow \$s \end{array}$$

which result in the string

$$Cc(cc)^{m'} Ss^n$$

and

$$\boxed{\begin{array}{l} C \rightarrow D_1 D_0 \\ c \rightarrow d_1 d_0 \end{array}}$$

which lead to

$$Ss^n D_1 D_0 (d_1 d_0)^{m'}$$

Now the oddness or evenness of n is, at last, to have an effect; for

$$\boxed{\begin{array}{l} S \rightarrow T_1 T_0 \\ s \rightarrow t_1 t_0 \end{array}}$$

yields either

$$D_1 D_0 (d_1 d_0)^{m'} T_1 T_0 (t_1 t_0)^{(n-1)/2} \quad \text{or} \quad D_0 (d_1 d_0)^{m'} T_1 T_0 (t_1 t_0)^{n/2}$$

depending on whether n was odd or even, respectively. This has a profound effect, because in the first case the system will see only subscript '1' symbols from now on, while in the other case it will see only subscript '0' symbols! Hence, we can control what happens now with two distinct sets of productions:

$$\overset{n\text{ odd}}{\boxed{\begin{array}{l} D_1 \to A_1a_1 \\ d_1 \to a_1a_1 \end{array}}} \qquad \overset{n\text{ even}}{\boxed{\begin{array}{l} D_0 \to a_0A_0a_0 \\ d_0 \to a_0a_0 \end{array}}}$$

which yields

$$T_1T_0(t_1t_0)^{n'}A_1a_1(a_1a_1)^{m'} \quad \text{or} \quad T_0(t_1t_0)^{n'}a_0A_0a_0(a_0a_0)^{m'}$$

where we have written n' for $(n - 1)/2$ or $n/2$ as the case may be. Finally,

$$\overset{n\text{ odd}}{\boxed{\begin{array}{l} T_1 \to B_1b_1 \\ t_1 \to b_1b_1 \end{array}}} \qquad \overset{n\text{ even}}{\boxed{\begin{array}{l} T_0 \to B_0b_0 \\ t_0 \to b_0b_0 \end{array}}}$$

produce the desired final strings

$$A_1a_1(a_1a_1)^{m'}B_1b_1(b_1b_1)^{n'=(n-1)/2} \quad \text{or} \quad A_0a_0(a_0a_0)^{m'}B_0b_0(b_0b_0)^{n'=n/2}$$

The important thing about these two possible final results is that *they are in entirely separate alphabets*. This means that we can now write different productions to determine what will next become of the string in the case that n was even and in the case that n was odd. Thus we are, in effect, able to lay out the structure of a program. What should we do, in fact? We write a set of productions like the ones above for each state Q_i of the Turing machine, using entirely different alphabets for each. Then we link them; whenever an exit state Q_{i1} is the state Q_j, we make the output letters A_1, a_1, B_1, and b_1 of Q_i the same as the input letters A, a, B, and b of Q_j. Similarly we identify the output letters A_0, a_0, B_0 and b_0 of Q_{i0} with the input letters of whatever state is Q_{i0}. Thus, we can simulate the interconnections of states of an arbitrary Turing machine by combining in one large tag system, all tag productions described above. This completes the proof of theorem 14.6-1.

14.7 UNSOLVABILITY OF POST'S "CORRESPONDENCE PROBLEM"

In 1947 Post showed that there is no effective procedure to answer questions of the following kind:

THE CORRESPONDENCE PROBLEM

Given an alphabet A and a finite set of pairs of words (g_i, h_i) in the alphabet A, is there a sequence $i_1i_2\ldots i_N$ of selections such that the strings

$$g_{i_1}g_{i_2}\dots g_{i_N} \quad \text{and} \quad h_{i_1}h_{i_2}\dots h_{i_N}$$

formed by concatenating—writing down in order—corresponding g's and h's are identical?

While Post's original proof of the unsolvability of such problems is complicated, the result of 14.6—that monogenic normal systems can be universal—makes it very simple to prove; for we can show that any procedure that could effectively answer all correspondence questions could equally well be used to tell whether any tag, or other monogenic normal system will ever reach a halting symbol, and this is equivalent to telling whether any Turing machine computation will halt.

Proof: Let M be a monogenic normal system with axiom A and productions $g_i\$ \to \$h_i$. Now suppose that this system happens to terminate by eventually producing a string Z which does not begin with any of the g_i's. Define G_i and H_i as follows:

$$\text{if } g_i = a_pa_q\dots a_t, \text{ let } G_i \text{ be } \boxed{Xa_pXa_qX\dots Xa_t}$$

and

$$\text{if } h_j = a_ua_v\dots a_z, \text{ let } H_i \text{ be } \boxed{a_u\,Xa_vX\dots a_zX}$$

Now consider the correspondence system:

$$\begin{array}{ccc} G_0 & G_i & \bar{Z}XY \\ \updownarrow & \updownarrow & \updownarrow \\ X\bar{A}H_0 & H_i & Y \end{array}$$

where X's are placed *after* each letter of A and *before* each letter of Z to form $\bar{A}$ and $\bar{Z}$. Y is a new letter.

ASSERTION

This system will have a matching pair of identical strings if and only if the monogenic normal system ($g_i\$ \to \$h_i$), *when started with* A, *terminates with the string* Z. In fact, if there is any solution to the correspondence question, there is just one, and that solution is (for the G's) the sequence of antecedents and (for the H's) the sequence of consequents encountered in producing Z from A.

If we can establish the truth of the assertion, then the unsolvability of the general correspondence follows, for one can adapt any Turing-machine halting problem to a question of whether the machine in question reaches a certain special state with a blank tape; then, in the normal system, we can reduce this to the question of terminating in a particular string Z. (Or, we can simply equate this to the unsolvable halting problem for tag systems.)

Why is the assertion true? Let us first note what the X's and Y's are for. The X's are to make sure that *if a matching pair exists at all, it must begin with the transcription $X\bar{A}$ of the axiom A.* This is assured by the fact that the *only way an H string can start with an X is by starting with $X\bar{A}H_0$.* And since all G *strings must start with an* X, we can be sure that *any matching set starts with $X\bar{A}$.* Similarly, any matching pair of strings must end with a transcription of Z—more precisely, must end with $\bar{Z}XY$—because a G string can't end in X and an H string can end only with X or Y. It follows that if there is any solution at all to this correspondence problem, then the solution must be a string which can be resolved into the two forms:

$$G_0G_{i_1}, G_{i_2} \ldots G_{i_N}\bar{Z}XY$$
$$X\bar{A}H_0H_{i_1}H_{i_2} \ldots H_{i_N}Y$$

But if this is the case, then it follows that the sequence of G_{i_j}'s is exactly the sequence that would be followed by the original monogenic normal system (g_i, h_i)! We can see this inductively: we have already established that the H string must begin with $X\bar{A}$. Then the G string *must* begin with G_0. Why? Because the system is *monogenic*! That means that the beginning of axiom A can be matched only by g_0. Then g_0 determines h_0—the string to be added to A—and let us remove G_0 from the front. Then there is *only one* G_{i_1} that can match the beginning of the remaining string; this corresponds to the g_{i_1} that the normal system would apply at the next step. Then G_{i_1} determines H_{i_1}—the string that is to be added to what is left after deleting G_{i_1} from the front. Again, G_{i_2} is determined, because the system is monogenic and it, too, must be precisely the production antecedent the normal system would use at its second stage.

Thus the sequence of G_i's must be the same as that of the g_i's underlying the monogenic normal system (and hence must represent the steps on the computation of the still-further underlying Turing-machine computation). If the process terminates, then the last G that was used will be followed in the string by Z—by definition that which will remain after no more productions can be applied.

Therefore, if the strings match, then they must both be the sequence mentioned in the assertion.

So far, the only monogenic normal systems we have are the tag systems. We could, however, have used theorem 14.1-1 more directly to show that *monogenic normal systems are universal*: Consider an arbitrary two-register machine, and represent its state by a word of the form

$$I_j \quad 111\ldots111 \quad K_j \quad 111\ldots111$$

Then, using the same methods as we used in 12.6, we can realize the instruction types of theorem 14.1-1 as follows.

Conditional subtract from first register:

$$
\begin{array}{rcll}
I_jK_j\$ & \rightarrow & \$I_{j'}K_{j'} & \text{(i.e., go to } I_{j'} \text{ if register is empty)} \\
1\$ & \rightarrow & \$1 & \qquad\qquad \textit{or} \\
I_j1\$ & \rightarrow & \$I_{j+1} & \text{(subtract 1 and go to } I_{j+1}) \\
K_j\$ & \rightarrow & \$K_{j+1} &
\end{array}
$$

Conditional subtract from second register:

$$
\begin{array}{rcll}
I_j\$ & \rightarrow & \$I_j^* & \text{(rotate to examine second register)} \\
K_jI_j^*\$ & \rightarrow & \$K_j^*I_{j'} & \text{(go to } I_{j'} \text{ if register is empty)} \\
K_j^*\$ & \rightarrow & \$K_{j'} & \\
K_j1\$ & \rightarrow & \$K_{j+1} & \text{(subtract 1 and go to } I_{j+1}) \\
I_j^*\$ & \rightarrow & \$I_{j+1} &
\end{array}
$$

Add to first:

$$I_j\$ \rightarrow \$I_{j+1}1, \qquad K_j\$ \rightarrow \$K_{j+1}$$

Add to second:

$$I_j\$ \rightarrow \$I_{j+1}, \qquad K_j\$ \rightarrow \$K_{j+1}1$$

PROBLEM. Can you make a similar construction for three or more registers?

14.8 "SMALL" UNIVERSAL TURING MACHINES

The existence of universal machines is surprising enough, and it is startling to find that such machines can be quite simple in structure. One may ask just how small they can be; but to answer this, one needs to have some measure of size, or complexity, of a machine. Several measures can be defended; Shannon [1956] suggests that one might consider the product of the number of symbols and the number of states, since, as he shows, this product has a certain invariance. One can exchange states and symbols without greatly changing this product. To count the number of quintuples would be almost the same.

In this section we describe the universal Turing machine with the smallest known state-symbol product. This machine is the most recent entry in a sort of competition beginning with Ikeno [1958] who exhibited a six-symbol, ten-state "(6, 10)" machine, Watanabe [1960] (6, 8), Minsky [1960] (6, 7), Watanable [1961] (5, 8), Minsky [1961] (6, 6), and finally the

(4, 7) machine of this section (described in Minsky [1962]). The reader is welcome to enter the competition (I believe that a certain one of the (3, 6) machines might be universal, but can't prove it)—although the reader should understand clearly that the question is an intensely tricky puzzle and has essentially *no* serious mathematical interest. To see how tricky things can become, the reader can refer to my 1962 paper describing the (6, 6) machine; it is much more complicated than the machine of this section.

14.8.1 The four-symbol seven-state universal machine

The very notion of a universal Turing machine entails the notion of *description*; the machine to be simulated has to be described, on the tape of the universal machine, in the form of some code. So, also, must the initial tape or data for the simulated machine be described. One way to do this encoding is to write, almost literally, the quintuples for the simulated machine on the tape of the universal machine; this is what we did for the machine of chapter 7. On the other hand, there is no particular virtue in the quintuple formulation, and one might be able to get a simpler universal machine with some other representation. Nevertheless, we must not go too far, for if one is permitted an arbitrary partial-recursive computation to do the encoding and is permitted to let the code depend on the initial data, then one could use as the code the result of the Turing-machine computation itself, and this would surely be considered a cheat! (It would give us a (2, 0) machine, since the answer could be written in unary on a tape, and no computer would be necessary.) We have to make some rule, e.g., that nothing like full computation power may be spent on the encoding. Informally, *this will be guaranteed if the encodings for the machine structure and for the data are done separately*. Then we can be sure that the machine was not applied to the data during the encoding process. This condition, we claim, justifies what we do below. More technically, one might require, for example, that the encoding process be a primitive-recursive, symbol-manipulation operation on the input; this, too, would guarantee that if the resulting machine is universal, this is not due to some power concealed in the encoding process. Davis [1956] discusses this question. We will present first the encoding for our machine and then its state-symbol transition table.

The four-symbol, seven-state machine will work by simulating an arbitrary $P = 2$ tag system. We know, by theorem 14.6-1, that if a machine can do this, it must be universal. We know, by the proof of theorem 14.6-1, that representing the quintuples of an arbitrary Turing machine in the form of a $P = 2$ tag system is a tedious but trivial procedure. Since one can see in advance for any such representation how much work will be involved, the conversion is a primitive-recursive operation. In fact, it involves only writing down sixteen productions for each quintuple.

Suppose, then, that the tag system is:

$$\begin{aligned}
&\text{Alphabet: } a_1, a_2, \ldots, a_m\\
&\text{Productions: } a_1 \to a_{11}a_{12}\ldots a_{1n_1}\\
&\qquad\qquad\qquad a_2 \to a_{21}a_{22}\ldots a_{2n_2}\\
&\qquad\qquad\qquad\quad \ldots\ldots\\
&\qquad\qquad\qquad a_m \to a_{m1}a_{m2}\ldots a_{mn_m}
\end{aligned}$$

where n_i is the number of letters in the consequent of a_i. For each letter a_i we will need a number N_i computed as follows:

$$N_1 = 1$$
$$N_{i+1} = N_i + n_i$$

so that $N_i = 1 + n_1 + n_2 + \cdots + n_{i-1}$. Note that this is just one more than the number of letters in the production consequents preceding that of a_i. We shall see the reason for this definition shortly.

We will represent any string (e.g., an axiom) $a_r a_s \ldots a_z$ on U's tape by a string of the form

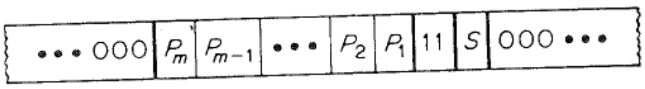
$$S = y^{N_r}Ay^{N_s}A\ldots Ay^{N_z}$$

so that N_i, used as the length of a string of y's, is used to represent a_i to the machine. The A's are spacers.

The productions will be represented as follows. Define

$$P_i = \underline{11}\,0^{N_{in_i}}\underline{01}\ldots\underline{01}\,0^{N_{i2}}\underline{01}\,0^{N_{i1}}$$

so that the representation of the consequent of a_i begins with 11 and then has representations of the consequent's letters—in reverse order—separated by 01's. (We use strings of 0's here instead of strings of y's.) Now, finally, we can describe the whole of U's tape; it is:



The secret of the encoding is this: The pairs 11 and 01 are used as punctuation marks; 11 marks the beginning of a production and 01 marks spaces between letters in a production consequent. *There are exactly n_i punctuation marks in the* ith *production P_i*, and there is one extra 11 just to the left of S. Hence, *there are exactly N_i punctuation marks between S and the beginning of P_i*. So the code y^{N_i} chosen to represent a_i contains exactly the information needed to locate the production corresponding to that letter!

The machine will use only the letters already introduced: 0, 1, y, and A. It is understood that '0' is also the blank symbol on the remainder of the infinite tape. Table 14.8-1 is the state-symbol table for the machine; it is understood that if no new state is given, the machine remains in its present state.

Table 14.8-1

	q_1	q_2	q_3	q_4	q_5	q_6	q_7
y	$0\ L$	$0\ L/1$	$y\ L$	$y\ L$	$y\ R$	$y\ R$	$0\ R$
0	$0\ L$	$y\ R$	HALT	$y\ R/5$	$y\ L/3$	$A\ L/3$	$y\ R/6$
1	$1\ L/2$	$A\ R$	$A\ L$	$1\ L/7$	$A\ R$	$A\ R$	$1\ R$
A	$1\ L$	$y\ R/6$	$1\ L/4$	$1\ L$	$1\ R$	$1\ R$	$0\ R/2$

The machine starts in q_2 at the first symbol in S. It seems useless to try to explain the machine, except by following it through an example, because its various functions are all mixed up. Generally, states q_1 and q_2 read the first symbol in S, locate and mark the corresponding production P_i, and erase the first symbol in S. States q_3, q_4, q_5, and q_6 then copy the production consequent at the end of S. (The copying works from inside to out; this is why the productions were written backwards.) When the end of the production is detected (by q_4 and q_7 finding the 11), then q_7 restores the tape, removing the marking of the production region and, incidentally, erasing another symbol from S. Since two symbols were erased from S, and the appropriate production is copied, we have a $P=2$ tag process.

The problem in making a "small" machine is to avoid use of new letters for marking. All marking of working places for this machine is done by interchanging 0's and y's and 1's and A's.

If q_3 meets a 0, the machine halts. It turns out that this can happen only under special conditions; but these conditions will come about, eventually, if the special string $P_H = 110101$ is used as a production and this production is referenced. So if any of the letters of the tag system are supposed to cause a halt, we assign to them the production P_H. The number n_H assigned to the halt symbol P_H is 3.

Constructing and following an example is tedious. Here is a simple one.

AN EXAMPLE

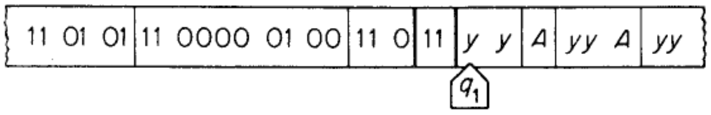
We will code the machine for the simple tag system

$$a_1 \rightarrow a_2$$
$$a_2 \rightarrow a_2 a_3$$
$$a_3 \rightarrow \text{halt}$$

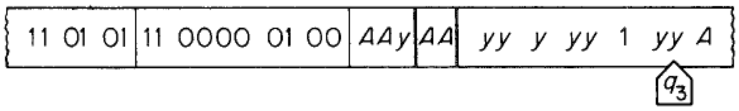
For this system,

$n_1 = 1$	$N_1 = 1$	$P_1 = 11\ 0$
$n_2 = 2$	$N_2 = 2$	$P_2 = 11\ 0000\ 01\ 00$
	$N_3 = 4$	$P_3 = 11\ 01\ 01$ (the halt production)

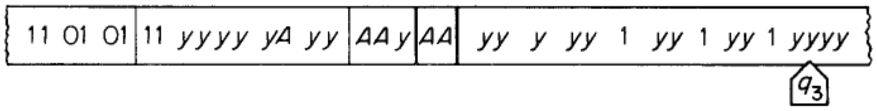
If we start with axiom $a_2a_2a_2$, this is encoded as $yyAyyAyy$ and the tape is:



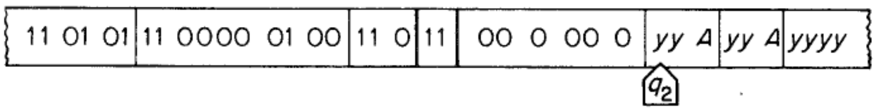
The machine marks symbols to the left, finding two punctuation groups, and then goes to the right in state q_6, writing an A at the end. The tape is then



Note that two deletions have been made from the front of the tag string! Now the production is copied (backwards) at the end; two 0's, and A and four more 0's, forming yy 1 $yyyy$:



On the next trip to the left, the machine encounters the 11, meaning that copying is to stop; the machine enters q_7 and restores the tape to the form



This is, in effect, like the starting state (q_2 has the same effect as q_1) except that the string $a_2a_2a_2$ has been replaced, as it should be, by $a_2a_2a_3$.

If you trace the operation through to the end, you will see how the string next becomes $a_3a_2a_3$. Following that, after some curious struggles, the symbol a_3 will cause the machine to halt; it first writes AA and this sequence eventually causes state q_3 to encounter a zero.

14.8.2 Structure of universal machines

One might suppose, or hope, that the property that a Turing machine is universal should imply some interesting conclusion about its state diagram. But it seems there is nothing much to say about this, in general, because there are universal machines with structures so trivial that one can draw no interesting conclusions. Suppose, for example, we make a straightforward machine for a $P=2$ tag system. Let the axiom S be written out on the tape

$$0\ 0\ 0 \ldots S \ldots 0\ 0\ 0$$

and start the machine at the beginning of S with the state diagram shown in Fig. 14.8-1. Now we can make such a tag machine for any Turing machine, by section 14.6; so we can also do it for some universal Turing machine; hence there is a machine with this structure that is universal.

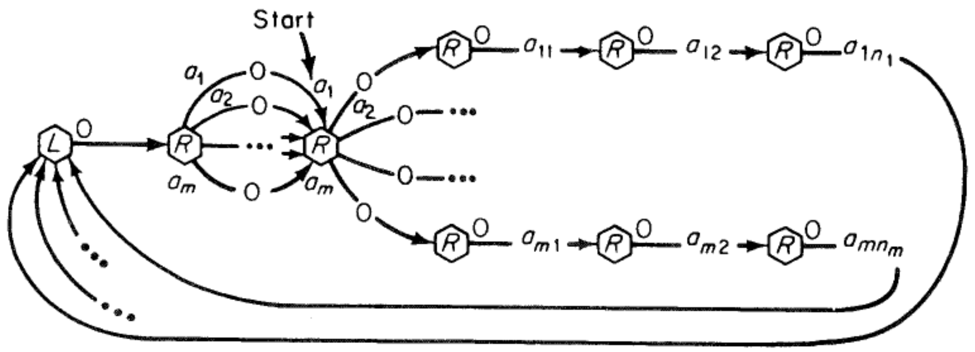


Fig. 14.8-1

There simply doesn't seem to be any structure required that is any more complicated than one needs to make a multiplication machine. Perhaps this is not surprising in view of theorems like theorem 14.2-1 and the difficulty of excluding full recursion, e.g., minimization, in any machine that has any iterative (loop) ability. In any case, the demonstration by Shannon (1956) that, allowed enough symbols, one can replace any Turing machine by a two-state machine shows that the structure of the state diagram can be hidden in the details of operation and not clearly represented in the topology of the interstate connections.

PROBLEM. Choose any two-symbol, two-state machine and show that it is *not* universal. Hint: Show that its halting problem is decidable by describing a procedure that decides whether or not it will stop on any given tape. D. G. Bobrow and the author did this for all (2, 2) machines [1961, unpublished] by a tedious reduction to thirty-odd cases (unpublishable).