A Turing Machine Simulator

M. W. Curtis

Wesleyan University, Middletown, Connecticut

Abstract. A description of a Turing machine simulator, programmed on the IBM 1620, is given. As in the papers by Wang and Lee, Turing machines are represented as programs for a computer. Allowance has been made for the usage of subroutines in the programming language. Also included are some remarks about writing a Universal Turing Machine Program and some experimental evidence that the state-symbol product is not the only measure of complexity of a Turing machine.

1. *Introduction*

Many authors [1–4] have noted that Turing machines [5–6], and finite automata [8] and [2] may be represented by programs written for a computer that is equipped with a potentially infinite tape and a read-write head. The instruction set from which these programs are constructed consists of operations of two types. The first type are operations on the tape and the second are program control operations. There are no instructions that allow for program modifications during execution time.

In writing complicated programs for any computer it is desirable and often necessary to have an actual computer to check (debug) them on. This is true for programmed Turing machines. For this purpose we have designed such a Turing machine and programmed a simulation for it on the IBM 1620. This work was carried out with two particular projects in mind: (1) as an educational device for a digital analysis course at Wesleyan University in which the students had to design a Universal Turing machine; and (2) as a tool for a research project at Wesleyan University directed toward the study of Turing machines and finite automata.

2. *The Programmed Turing Machine*

As originally conceived by Turing [6], this abstract computer is equipped with a potentially one-way infinite tape which is divided up into squares. It performs its calculations on this tape by writing characters in the squares, and being able at any time to make decisions based on the finite number of characters (tape content) already written on the tape. The read-write head scans one character at a time.

Translating this general idea of a computer into a specific form we now specify (1) the set of characters which may be written on the tape, (2) the programming language that it can interpret, and (3) conventions for using the tape.

1. *The Character Set.* The character set chosen was determined by programming and hardware considerations of the 1620 computer which was used in the simulation. However it should also be convenient for other computers since it

conforms to the FORTRAN character set. It consists of: all capital Latin letters A–Z; the decimal digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9; *, \$, @, + and −. The left and right parentheses and the comma are not in the character set. The character, B is reserved to denote a "blank" square on the tape.

2. *The Programming Language.* This language consists of a certain basic set of instructions that are written down in a specified format. A program is a sequence of these basic instructions. The instruction format was chosen for its readability and similarity to other symbolic programming systems on actual computers. The basic instructions are slight modifications of those proposed by Wang [1].

RN :	Shift the read-write head N squares to the right on the tape where N is an integer $0 \leqq N < 100$.
LN :	Shift N squares to the left, $0 \leqq N < 100$.
$W(x)$:	Write x on the scanned square, where x is an element of the character set.
$T(\alpha, x)$:	Conditional transfer; if the scanned character is x, transfer control to the instruction whose name (symbolic address, see below) is α; otherwise go to the next instruction in sequence.
$T(\alpha)$:	Unconditional transfer to instruction at address α.

Our simulator will accept any symbolic address, α, consisting of 2–4 of the alphameric characters available on the 1620. A symbolic address may be the name of only one instruction of a program, but only those instructions referred to in a transfer instruction need have symbolic addresses. In writing a program for a Turing machine, the instructions are listed in sequence and executed in sequential order except as changed by the transfer instructions. The symbolic address is written to the left of the instruction designated.

END :	Signifies the end of the program and causes the tape content to be printed out on the 1620 typewriter.

An essential departure from [1] and [3] is the inclusion in the programming language of an instruction similar to the execute instruction in IPL-5 and the CALL statement in FORTRAN. Its usage is explained in detail in Section 3. Its format is:

NAME$(A_1, A_2, \cdots, A_k)$:	Where the A_i are symbolic instruction addresses or characters of the Turing machine and NAME is the symbolic address of the first instruction of the subroutine.

3. *Tape Conventions.* In the simulator, the Turing machine tape consists of a finite string of digits in the 1620 core storage, 2 digits per tape square. The simulator initializes the tape by marking B's (blanks) in each square. Then the programmer's intitial tape content (i.e. the data for his program) is read in starting at the left end of the tape. When control is passed to the Turing machine program, the leftmost square is scanned first. Thus, the simulator provides a tape extending to the right. However the programmer may use multiple tapes, or a two-way open tape by marking off the tape in sections and programming accordingly. Note that B's are read in as part of the data.

A sample program for the Turing machine given in Kleene [5, p. 364] is shown below. This machine prints a 1 on the first blank square at or to the right of the scanned square.

Symbolic Address	*Instruction*
LOC1	T(LOC2, B)
	R1
	T(LOC1)
LOC2	W(1)
	END

Suppose the initial tape appears as follows: $\bar{x}_1 x_2 \cdots x_k B \cdots$ where the bar indicates the scanned character and the x's are nonblank characters. Then the final tape would be: $x_1 x_2 \cdots x_k \bar{1} \cdots$.

Note. This program will work for a machine with an arbitrary character set containing 1 and B. The above instruction set is not a minimal one for writing programs representing Turing machines (see [1]) but was made as flexible as possible while still retaining the essential features of a Turing machine. In consideration of execution speed and program efficiency the flexibility of the transfer instructions is useful.

It is shown in [2] and [3], that one may also represent finite automata with programs on this computer with the restriction that one does not use the left-shift command.

3. *Subroutines*

Both Turing [6] and Wang [1] use subroutines with variables as arguments (called *skeletal tables* by Turing) in their developments, but mainly as a convenient short-hand notation for their exposition. It was understood that in the actual machine these skeleton tables were copied down where they appeared with fixed arguments replacing the variables. We have tried to follow the more usual concept of subroutine as it appears in actual computer technology where the subroutine is a subprogram transferred to or executed from the main program. The arguments are supplied to the subroutine by the execute instruction of the main program and they are substituted for the variables appearing in the subroutine. The order of substitution is governed by an expression of the form, NAME(α_1, α_2, $\cdots$, α_k), where the α_i denote the symbolic addresses and characters which play the role of variables in the subroutine. An expression of this type must precede each subroutine.

Two examples of subroutines equivalent to the skeletal programs in Turing [6, pp. 236–7] are given below. These subroutines will work on any tape format provided that the left end of the tape is marked by an "E" on the first two squares. The variables in the following subroutine definitions are LOC1, LOC2 and A. Note that we use the same notation for variables as for arguments. The distinction is made by the first expression of the subroutine which designates them as variables.

```
                  Program                                 Comment
Example 1
              SUB1(LOC1,LOC2,A)             This subroutine finds the first A on the tape
    SUB1      T(S1,E)                       and goes to LOC1. If no A exists, it goes to
              L1                            LOC2.
              T(SUB1)
    S1        T(LOC1,A)
              T(S3,B)
              R1
              T(S1)
    S3        R1
              T(LOC2,B)
              T(S1)
              END

Example 2
              SUB2(LOC1,LOC2,C)             This subroutine erases first C and goes to
    SUB2      SUB1(S1,LOC2,C)               LOC1. If there is no C, it goes back to
    S1        W(B)                          LOC2.
              T(LOC1)
              END

Example 3
              SUB3(LOC1,D)                  Erase all D's on the tape and go to LOC1.
    SUB3      SUB2(SUB3,LOC1,D)
              END
```

Although Turing's subroutine in Example 3 is quite elegant in its definition, a more efficient routine that would not need to proceed to the beginning of the tape after each erasure could be written as follows:

```
Example 4
              SUB3(LOC1,A)
    SUB3      SUB1(S1,LOC1,A)
    S1        W(B)
    S3        R1
    S4        T(S1,A)
              T(S2,B)
              T(S3)
    S2        R1
              T(S1,A)
              T(LOC1,B)
              T(S3)
              END
```

In both Examples 1 and 4 we could have incorporated another subroutine that would search right for the first A, or if no A, it would go to LOC1.

4. *The Simulator*

In this section we discuss some of the essential features of the 1620 program that interprets the Turing machine programming language described above.

The simulator program can be broken down into four blocks, explained below.

1. *Pass 1 of the loader.* This block sets up symbol tables for the symbolic addresses in the subroutines and the main program.

2. *Pass 2 of the loader.* The Turing program is coded and loaded into consecutive digit positions of core memory with record marks separating instructions and symbolic addresses replaced by the actual core addresses of instructions.

3. *Loading the computational tape.* The consecutive digit positions in memory that serve as squares on the tape (2 digits per square) are marked with B's; and then the user's initial tape content is read in.

4. *Interpretive mode.* With the exception of the execute subroutine command the instructions are very easy to interpret. The core address of the scanned character and the current instruction address are the most important things to keep track of. A shift instruction of N squares only requires that we add $\pm 2N$ to the scanned character address. To interpret a write instruction, we need to transfer the coded form of the character in the write instruction to the contents of the scanned character address. To execute a transfer, we replace the current instruction address with the instruction address in the transfer instruction.

The subroutines are interpreted by the same part of the simulator as the main program, where the arguments are referred to using the indirect addressing feature of the 1620. This feature allows any level of indirect addressing so the nesting of subroutines along with their arguments poses no special problems. We keep track of the instruction addresses of the execute instructions by storing them in push-down storage lists.

If we now consider the speed of the Turing machine instructions as interpreted by the simulator we can say that each type of instruction takes a certain fixed amount of time (roughly speaking) to execute with the exception of the execute subroutine instruction where the execution time is a function of the number of arguments. The important thing to notice is that the shift instructions take a fixed amount of time and do not depend on the number of squares to be shifted.

5. *The Universal Machine*

For the definition of a Universal Turing Machine, U, the reader is referred to [6]. Essentially, the idea is to write the program, P, of any machine on U's tape and have U interpret the instructions of P. The design of a Universal Turing machine, U, is determined by the encoding scheme that is chosen to represent the "special-purpose machine," P, and its computation on U's tape. Thus, the problem of designing a Universal Machine is that of simulating a class of Turing computers on a single computer or interpreting the programming language of one computer on another.

Given the Turing machine language as described above, it is natural to try to design a universal machine by simulating a modified version of the language, discarding the nonessential parts of the notation. To simplify matters we restrict our attention to special-purpose machines that have the character set

{0, 1, B} (see [7]). We also restrict the language to shifts of only one square at a time.

In writing the program for U we will try to minimize the time that U takes to interpret an arbitrary special-purpose program P, written in the modified programming language.

It is believed that time as well as the state-symbol product is a factor in the measure of the complexity and efficiency of total computation of a Turing machine. Research into this question is currently being conducted at Wesleyan using the simulator described here.

At this point interested readers might care to go to the Appendix for a detailed explanation of such a Universal machine. We will continue here with a general discussion of techniques in designing an encoding scheme to represent P that will save time in U's execution.

Because U's view of its own tape is one-dimensional, much of its time is spent in shifting loops which look for special marks. The most time spent in these shifting loops is in two parts of U's program.

(1) The part that shifts from an instruction in P to the scanned character in P's computational part of the tape.

(2) The part that searches for the next instruction of P.

One can minimize the time in (1) by marking the current instruction and P's scanned character by a single mark and then shift directly to that mark. This involves no copying and subsequent trial and error searching which is very costly.

Let us now consider the timing involved in (2). Let machine U consider the next instruction of P to be immediately to the right of the current instruction unless that instruction is a transfer. This eliminates much of the effort spent in searching for the next instruction which occurs in Turing's formulation, for example. In order to speed up the interpretation of the transfer instruction, we modify the language of P as follows. We mark each addressable instruction of P's program with the same mark. The transfer instruction, $T(k)$, now means "transfer to the kth addressable instruction." We represent k on U's tape in stroke notation (i.e. a string of k consecutive 1's).

The interpretation of $T(k)$ involves shifting back and forth to locate the kth addressable instruction. The time required by U to interpret $T(k)$ is determined as follows: Let

C = the time it takes to shift from the location of $T(k)$ on U's tape to the beginning of U's tape;

C_i = the time it takes to shift from the location of the ith addressable instruction of P to the beginning of U's tape;

t_i = the nominal time to execute $T(i)$.

Then,

$$t_1 = 3(C + C_1), \qquad t_2 = 2(C + C_1) + 3(C + C_2)$$

and in general,

$$t_k = C + C_k + 2\sum_{i=1}^{k} (C + C_i). \tag{1}$$

The nominal time gives a good approximation to the actual interpretation time. (We have neglected some inconsequential intermediate operations.)

In effect, Lee [3] requires each instruction of P to be addressable. Thus for each instruction $T(k)$ in a program P with n instructions using the above scheme, a corresponding program P′ using Lee's scheme would have the instruction $T(k')$, where $n \geqq k' \geqq k \geqq 1$ and therefore require $t_{k'} \geqq t_k$. Hence a Universal program, in our system, identical to Lee's Universal program except for the method of interpreting transfers, would run faster on P than the Lee machine would on P′.

An alternate procedure that would result in even faster running time for many programs is to allow for two transfer instructions: transfer forward, $TR(k)$, and back $TL(k)$, to the kth addressable instruction, when k is counted from the current transfer instruction. The nominal time required for U to interpret $TR(k)$, denoted by t_k^R, can be obtained from (1) by setting $C = 0$ and substituting $C_{i+j} - C$ for C_j, where $C_i \leqq C \leqq C_{i+1}$. We get,

$$t_k^R = C_{i+k} - C + 2\sum_{j=1}^{k} (C_{i+j} - C). \tag{2}$$

Similarly for $TL(k)$ we get,

$$t_k^L = C - C_{i-k+1} + 2\sum_{j=1}^{k} (C - C_{i-j+1}). \tag{3}$$

For some programs t_k may be less than the equivalent $t_{k'}^L$ where $C_i \leqq C \leqq C_{i+1}$ and $k \leqq i$, $k' = i - k + 1$. By looking at formulas (1), (2) and (3) we have the following:

Remark. For a given k and $C_i \leqq C \leqq C_{i+1}$: (1) if $i \leqq 2k$, then $t_{i-k+1}^L < t_k$ and (2) if $i < k$, then $t_{k-i}^R < t_k$.

We may now construct a U that interprets three types of transfers T, TR and TL. Using the results of the above remark we may conclude that any program P will run at least as fast when coded using all three transfers than when coded using only T or only TR and TL.

6. *Some Experimental Results*

Three Universal Machines in the literature [3, 6, 9] were programmed using the Turing Machine Language described above.

Turing's Universal Machine was not translated directly into our language. Instead we used his encoding scheme for instructions of P with the minor modification that quadruples replace his quintuples. We also adhere to his idea of copying successive total configurations of P. Thus, our version of Turing's Universal Machine is not an exact copy of his but does follow his mode of interpreting P. On the other hand, Lee's Universal Machine Program was transcribed directly, instruction by instruction, into the Turing Machine Language, making only the necessary changes dictated by instruction format.

The transition table for the Watanabe Universal Machine, 8-state × 5-symbol, was programmed in the following way: Let s_i stand for the ith state and c_j the jth character. Then this would be represented by Watanabe as follows:

$\vdots$			c_j	
s_i		$\cdots$	c_k, R or L, s_n	$\cdots$
$\vdots$				

The corresponding program for state s_i used in this paper is the following:

```
SI     T(C1, c1)
       T(C2, c2)
           ⋮
       T(CJ, cj)
           ⋮
CJ     W(ck)
       R1 or L1
       T(SN)
           ⋮
```

TABLE A

(1) Turing				(2) Watanabe		
					0	1
S_1	0	w(1)	S_2	S_1	W(1), R, S_2	W(0), R, S_1
S_1	1	w(0)	S_2	S_2	W(1), L, S_3	W(0), R, S_1
S_2	0	R	S_1	S_3	W(0), R, S_3	W(0), L, S_3
S_2	1	R	S_3			
S_3	0	stop				
S_3	1	w(0)	S_2			

(3) Lee		(4) Wesleyan I		(5) Wesleyan II		(6) Wesleyan III	
1	T(7, 1)	A	T(3, 0)	A	TR(2, 0)	A	TR(2, 0)
2	W(1)	A	W(0)	A	W(0)	A	W(0)
3	R1		R1		R1		R1
4	T(7, 1)		T(1)		TL(2)		T(1)
5	W(1)	A	W(1)	A	W(1)	A	W(1)
6	T(12, 1)		R1		R1		R1
7	W(0)		T(2, 1)		TL(2, 1)		TL(2, 1)
8	R1		END		END		END
9	T(7, 1)						
10	W(1)						
11	T(3, 1)						
12	END						

TABLE 1

Encoding Scheme	*A*	*B*	*C*	*D*	*E*	*F*	*G*
1. Turing	Turing machine delanguage scribed above	266	564	$S_iX_jO\ S_k$ quadruple	18	Any Integer	3
2. Lee	Same as 1 except $T(\alpha)$, $T(\alpha, 0)$ were not used	105	127	R, L, $W(X)$, $T(K, 1)$	2	2	12
3. Watanabe	Same as 1	54	141	$S_iX_jY_{kL}^{R}\ S_l$ quintuple	5	2	3*
4. Wesleyan I	Same as 1	129	269	Same as (2) plus $T(K)$, $T(K, 0)$	16	3	8
5. Wesleyan II	Same as 1	145	277	R, L, $W(X)$ $TR(K, X)$, $TR(K)$ $TL(K, X)$, $TL(K)$	16	3	8
6. Wesleyan III	Same as 1	167	343	The union of (4) and (5)	16	3	8

A = Instruction set for U.
B = Number of instructions for U with subroutines.
C = Number of instructions for U without subroutines.
D = Instruction set for P that U can interpret.
E = Cardinality of U's character set.
F = Cardinality of character set available for P.
G = Number of instructions or states for P.

* Since no provision is made for halting in the Watanabe formulation a do-nothing state was added to "stop" the computation.

TABLE 2

Encoding Scheme	*A*	*B*	*C*	*Execution Time*
Turing	445	92,444	936	31 minutes
Lee	204	15,118	207	2 minutes, 12 seconds
Watanabe	1129	36,719	28,303	39 minutes
Wesleyan I	51	1,680	84	40 seconds
Wesleyan II	54	1,453	84	39 seconds
Wesleyan III	52	1,403	76	36 seconds

A = the number of tape squares used by U to execute P.
B = the number of tape squares shifted by U in executing P.
C = the number of symbols written and erased by U in executing P.

Our own versions of a Universal Turing Machine (Wesleyan I, II, III) were written to interpret the three types of transfers described in Section 5.

All six universal machines were tested using a special-purpose machine P that complements a binary tape and stops when it encounters two consecutive zero's on the input tape.

The machine P appears as shown in Table A in the various systems tested. Within small variations, it is intuitively clear that these programs for P are

TABLE 3

Encoding Scheme	
Turing	$8k \leqq L \leqq 32k^2 + 80k$
Lee	$5k \leq L \leq 80k^2 + 120k + 30$
Watanabe	$3.2^{2k} \leqq L$
Wesleyan I	$2k \leq L \leq 4k^2 + 14k$
Wesleyan II	$2k \leqq L \leqq 2k^2 + 18k$
Wesleyan III	$2k \leq L \leq 2k^2 + 18k$

"minimal," i.e. the complementing task is a very simple one and these programs do only what is essential. Comparative results are given in Tables 1–3.

In Table 2 the input tape was 01100 and this tape is the minimal length tape that requires all instructions and all table entries of P to be executed by each U.

It is also important to note how much tape is required to encode an arbitrary 2-symbol, k-state machine P, for each Universal Machine mentioned above. We have the following conservative bounds in Table 3: Let L equal the number of squares of U's tape taken up by the program P, and k equal the number of states for P.

In interpreting the results in Tables 1–3, one should take the following facts into consideration. The timing of the Universal Turing and Watanabe machines is not a fair representation of their capabilities as no attempt was made to rewrite these as efficient programs. We simply translated Watanabe's machine as stated above, without taking advantage of features in our language, but in such a way that the number of squares shifted and the number of symbols written is the same as if one had simulated the transition table directly. The same applies to Lee's machine.

We recognize that the execution time of a given U depends on the programming skill used in constructing U in a particular language. As we have said, the Watanabe machine was penalized by the direct translation into our language (i.e. using his encoding scheme for P, we could have programmed a more efficient U. For this reason, we have included the data in columns A, B, C.)

For a given encoding scheme one tries to design U so that the number of squares shifted and number of symbols written will be a minimum for arbitrary P (this may not be possible). It is believed that the U's tested approach this aim, even though the programming for U may not be the most efficient timewise. (For example, many unessential instructions of the transfer type may be executed. This most likely is the case for the Watanabe program.)

If this is the case, then the data in columns A, B and C of Table 2 are evidence in support of our remarks in Section 5 that the state-symbol product is not the only criterion for measuring the complexity of a Turing Machine.

Acknowledgments. The development of the Turing machine simulator was carried out at the Wesleyan Computer Laboratory under the guidance of Professor E. K. Blum and was partially supported by NSF Grant GP 1356. Special thanks are due to Robert Travis for his able assistance in programming the simulator and to Professor Blum for his advice and suggestions in writing this

paper. The idea for the different transfer schemes in P's program came from conversations with Robert Travis and Peter Hagen. Turing's Universal Program was programmed by Richard Currie.

REFERENCES

1. WANG, H. A varient to Turing's theory of computing machines. *J. ACM 4*, 1 (Jan. 1957), 63–92.
2. LEE, C. Y. Categorizing automata by *W*-machine programs. *J. ACM 8*, 3 (July 1961), 384–399.
3. ——. Automata and finite automata, *BSTJ 39*, 5 (1960), 1267–1695.
4. MINSKY, M. L. Recursive unsolvability of Post's problem of "Tag" and other topics in theory of Turing machine. *J. Math. 74*, 3 (Nov. 1961).
5. KLEENE, S. *Introduction to Metamathematics*. Van Nostrand, New York, 1952.
6. TURING, A. M. On computable numbers with an application to the Entscheidungsproblem. *Proc. Lond. Math Soc. 2*, 43 (1936), 230–265.
7. SHANNON, C. A universal Turing machine with two internal states. In *Automata Studies*, Princeton U. Press, 1956.
8. RABIN, M. O., AND SCOTT, D. Finite automata and their decision problems. *IBM J Res. Develop. 3* (1959), 114–125.
9. WATANABE, S. 5-symbol 8-state and 5-symol 6-state Universal Turing Machines. *J. ACM 8*, 4 (Oct. 1962), 476–483.

APPENDIX. A Universal Turing Machine

We shall state the conventions for representing a special-purpose machine on U's tape. U's tape is divided up into two sections. The first section, U_1, contains P's program and the second, U_2, is reserved for P's computation. The language for P is a modified version of that given in Section 5. It is as follows:

R	: shift right one square
L	: shift left one square
$W\alpha$	: write $\alpha \in \{0, 1, B\}$
$Tk = \alpha$	: If the scanned character is α, transfer to the kth addressable instruction starting at the beginning of the tape. k is written in stroke notation.
$TRk = \alpha$	: If scanned character is α, transfer to the kth addressable instruction to the right of the current instruction.
$TLk = \alpha$	: If scanned character is α, transfer to the kth addressable instruction to the left of the current instruction.

Subroutines are not provided for. The beginning of the tape is marked by ///// . U_1 is separated from U_2 by @@@@@ . This convention makes shifting back and forth from U_1 to U_2 faster. The instructions of P are interpreted in sequence until a transfer instruction is encountered.

Instructions are written on consecutive squares, each instruction separated by a ".".

Alternate squares in U_2 are available for P's computation and the scanned character on P's computational tape is marked by an "S" on the square to the left. To mark the current instruction being interpreted, a "C" is written over the "." to the left. Addressable instructions are marked by an "A" on the square to the right of the ".". While interpreting transfer instructions "$" and "*" are used in the searching procedures.

```
*  UNIVERSAL TURING MACHINE PROGRAM
*      SHIFT LEFT ROUTINE
SUBROUTINE
       SLT(.)
SLT    T(S1,.)
       L1
       T(SLT)
S1     END
*       SHIFT RIGHT ROUTINE
SUBROUTINE
       SRT(.)
SRT    T(S1,.)
       R1
       T(SRT)
S1     END
*       SHIFT TO BEGINNING OF TAPE ROUTINE
SUBROUTINE
       SBT(.)
SBT    T(S2,/)
       L5
       T(SBT)
S2     END
*     FIND AND MARK NEXT SEQUENTIAL INSTRUCTION
SUBROUTINES
       SFNI(C)
SFN    SLT(C)
       W(.)
       R1
       SRT(,)
       W(C)
       END
*     SHIFT TO SCANNED CHARACTER
SUBROUTINE
       SHSS(S)
SHSS   T(S1,@)
       R5
       T(SHSS)
S1     SRT(S)
       END
*    FIND FIRST A ON TAPE
SUBROUTINE
       SFST(A)
SFST   SBT(.)
       SRT(A)
       END
PROGRAM
PROG   SBT(.)
       W(C)        MARK FIRST INSTRUCTION
       SRT(@)
       SLT(B)
       W(S)       MARK INITIAL SCANNED SYMBOL
       SBT(.)
S5     SRT(C)

*      SKIP OVER ADDRESSES
S4     R1
S3     T(S1,A)
       T(S2)
S1     R1
*     INTERPRET A RIGHT SHIFT INSTRUCTION
S2     T(RS,R)
*     INTERPRET A LEFT  SHIFT INSTRUCTION
       T(LS,L)
*     INTERPRET A WRITE        INSTRUCTION
       T(WR,W)
*     INTERPRET A TRANSFER     INSTRUCTION
       T(TR,T)
*     INTERPRET A END
       T(END,E)
*    ILLEGAL INSTRUCTION QUIT
       T(END)
*     GO TO NEXT INSTRUCTION
S5     SLT(C)
       SFNI(C)
       T(S4)
*       THIS SECTION INTERPRETS A TRANSFER INSTRUCTION
TR     SRT(.)
       L2
*     TEST IF STRAIGHT TRANSFER
       T(T1,*)
       T(T4)   STRAIGHT TRANSFER
T1     R1
*     TEST FOR POSSIBLE TRANSFER
       T(T3,1)
       T(T6,0)
       SHSS(S)
       R1
       T(T4,B)
       T(S5)
T6     SHSS(S)
       R1
       T(T4,0)
       T(S5)                             EXIT TO NEXT INSTRUCTION
T3     SHSS(S)
       R1
       T(T4,1)
       T(S5)
T4     SFST(C)
       W(.)
       SRT(T)
       R1
```

```
*    TEST TYPE OF TRANSFER TO INTERPRET
      T(TL,L)
      T(TR1,R)
      W(*)
      SBT(.)
T8    SRT(A)
      W($)
      SBT(.)
      SRT(*)
      W(1)

      R1
      T(T7,1)
      SBT(.)
      SRT($)
      W(A)
      L1
      W(C)
      T(S4)           GO TO MARKED INSTRUCTION
T7    W(*)
      SBT(.)
      SRT($)
      W(A)
      R1
      T(T8)
*   MATCH LOCATIONS FOR TRANSFER ROUTINE
TR1   R1
      W($)
TS3   SRT(A)
      W(*)
      SLT($)
      W(1)
      R1
      T(TS1,1)
      SRT(*)
      W(A)
      L1
      W(C)                          GO TO MARKED INSTRUCTION
      T(S4)
TS1   W($)
      SRT(*)
      W(A)
      R3
      T(TS3)
TL    R1
      W($)
TLS2  SLT(A)
      W(*)
      SRT($)
      W(1)
      R1
      T(TLS1,1)
      SLT(*)                        GO TO MARKED INSTRUCTION
      W(A)
      L1
      W(C)
      T(S4)
TLS1  W($)
      SLT(*)
      W(A)
      L1
      T(TLS2)
*    THIS SECTION INTERPRETS THE RIGHT SHIFT COMMAND
RS    SHSS(S)
      W(B)
      R2
      W(S)
      T(S5)

*    THIS SECTION INTERPRETS THE LEFT  SHIFT COMMAND
LS    SHSS(S)
      W(B)
      L2
      W(S)
      T(S5)
*    THIS SECTION INTERPRETS THE WRITE COMMAND
WR    R1
      T(W1,1)
      T(W2,0)
      SHSS(S)
      R1
      W(B)
      T(S5)
W2    SHSS(S)
      R1
      W(0)
      T(S5)
W1    SHSS(S)
      R1
      W(1)
      T(S5)
*  POSITION THE TAPE HEAD AT COMPUTATIONAL PART OF TAPE FOR EXIT
END   T(E1,@)
      R5
      T(END)
E1    END
```

Fig. 1

For the program (6) in Table A the initial tape reads:

$$/////.\mathrm{ART11} = 0.\mathrm{AW0.R.T1.AW1.R.TL11} = 1.\mathrm{E@@@@@B0B1B1B0B0BBB} \cdots$$

An intermediate tape content while interpreting the first and then the fifth instruction is:

```
/////.ATR1$ = 0.AW0.R.T1.*W1.R.TL11 = 1.E@@@@@S0B1B1B0B0BBB ···
/////.ATR11 = 0.AW0.R.T1CAW1.R.TL11 = 1.E@@@@@S1B1B1B0B0BBB ···
```

The final tape content is:

```
/////.ATR11 = 0.AW0.R.T1.AW1.R.TL11 = 1CE@@@@@B1B0B0B1S0BBB ···
```