APPENDIX B

TURING MACHINES

Throughout this book, we have relied heavily on the description of an algorithm (given in Chapter 2) as a detailed, step-by-step, finite sequence of instructions that are mechanical and unambiguous for performing a task. This is a rather informal and intuitive notion: How detailed must an algorithm be? What kinds of instructions are allowed? What does "mechanical" mean? How can ambiguity be avoided? The purpose of this appendix is to try to make the notion of "algorithm" a bit more precise by clarifying the notion of a mechanical procedure.*

From: Schagrin, Rapaport, & Dipert (1985), Logic: A Computer Approach (New York: McGraw-Hill)

*The presentation of Turing machines in this appendix is based on Clark and Cowell, 1976, pp. 44–49.

From: Schagrin, Rapaport, & Dipert (1985): Logic: A Computer Approach (McGraw-Hill)

Turing's Analysis of Computation

We have said that an algorithm is a "mechanical procedure" for carrying out some task in a finite number of steps. If we limit ourselves to tasks that can be described in some language (either a natural language or a programming language), then we can think of such a procedure as a "computation." In 1937, the English logician Alan Turing (1912–1954) presented an analysis of this notion in an important paper, "On Computable Numbers, with an Application to the *Entscheidungsproblem*," that led, among other things, to the notion of the stored-program computer. (The English translation of the German word *Entscheidungsproblem* is "decision problem"—the question of whether a procedure for a given task will produce a definite affirmative or negative answer in a finite number of steps.)

Turing's analysis begins by considering what it means for a computation to be carried out "mechanically." At the very least, we would need:

1. A "computer"—either a person or a machine—to do the computation,
2. Some scratch paper (a memory device) to do the computation on,
3. A deterministic program for the computer to follow—that is, a finite sequence of instructions for performing the computation by manipulating symbols on the scratch paper.

We can refine this somewhat informal picture. Let us suppose that the mind of the person (or computer) can only be in a finite number of different "states." Also, let's suppose for the sake of convenience that the person (or computer) has memorized the program (so that we only have to concern ourselves with two things: the computer and the scratch-pad "memory").

As for the scratch-pad memory, let's suppose that we have lots of paper—not necessarily an actually infinite amount of paper, but enough paper so that if we need more, we can always get it. And, so that we can describe precisely what's written on the paper, let's imagine that it is crosshatched into small squares (like graph paper), each containing one symbol, so that we can systematically locate any symbol in any square on any page of the scratch pad (for instance, by starting at the square in the first row and first column of the first page and examining each square in turn until we either find the symbol or else reach the last square on the last page without having found it).

Finally, imagine that the person (or computer) doing the computing can only see a finite, bounded number of squares on the scratch pad at any one time.

As the next refinement, let us now assume the following:

1. The person (or computer) sees only one square (containing at most one symbol) of the scratch pad at a time.
2. (a) The scratch pad is a linear tape (like an adding-machine tape), divided into squares, that is potentially infinite in both directions (that is, we can always add an extra square on either end).

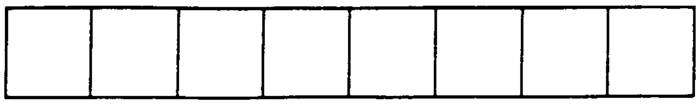


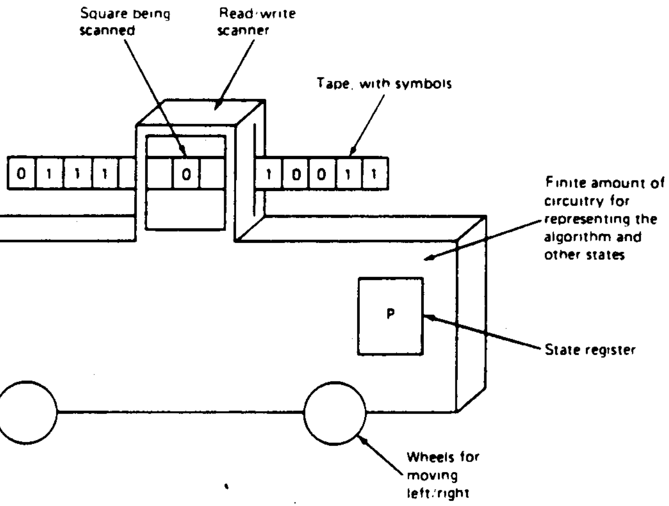
Figure B-1 A Turing-machine tape.

(b) Each square on the tape has a '1' or a '0' printed on it at the start (this is the input).

3. The program that the person (or computer) memorized doesn't consist of complex instructions (such as "Find the word MULTIPLY on the tape"); instead, it consists of a finite sequence of instructions of the following five kinds:
   (a) START.
   (b) IF your state of mind is P
       and you are scanning symbol **S**
         THEN
         (i) Change **S** to **S**′.
         (ii) Change your state of mind to Q.
   (c) IF your state of mind is P
       and you are scanning symbol **S**
         THEN
         (i) Move 1 square to the right.
         (ii) Change your state of mind to Q.
   (d) IF your state of mind is P
       and you are scanning symbol **S**
         THEN
         (i) Move 1 square to the left.
         (ii) Change your state of mind to Q.
   (e) STOP.

Below, we shall make these instructions even more precise. The result of this is a "Turing machine," which may be imagined as a physical machine on wheels, with a unit for reading and writing on the tape and a "register" to indicate what state it is in.

Figure B-2 An imaginary, physical Turing machine.


Square being scanned
Read/write scanner
Tape, with symbols
0 1 1 1
0
1 0 0 1 1
Finite amount of circuitry for representing the algorithm and other states
P
State register
Wheels for moving left/right

APPENDIX B

Turing Machines

A Turing machine can be defined as follows: It consists of a certain set, called the *memory set,* and a program constructed from elements of specified sets of *operations* and *tests.*

The memory set is the formal analog of the tape. Each element of the set consists of two items: (1) a string of '0's and '1's (representing the information stored on the tape) and (2) a positive integer (representing the symbol currently being scanned by the Turing machine). For instance,

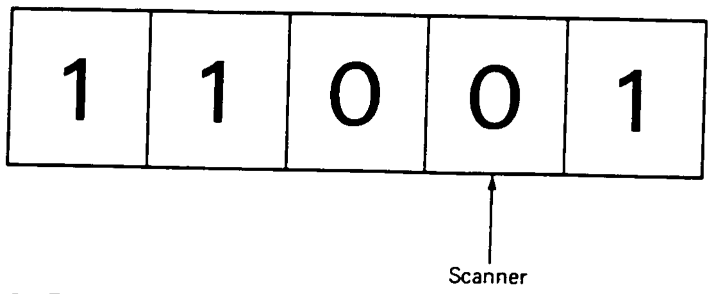


Figure B-3 The Turing-machine tape represented by (11001, 4) or by $110\underline{0}1$.

would be represented by the memory-set element

(11001, 4)

where the first element of this pair is the string and the second is the square being scanned, or by the symbol

$110\underline{0}1$

where the underline represents the position of the scanner. The "empty" tape—the tape with *no* little squares of paper—is represented by any element whose scanning number is 0; we'll represent the empty tape with the symbol *e.*

The Turing-machine programming language (TM) that we shall use consists of seven operations and four tests. The operations are:

1. START.
2. PRINT-0. This changes the symbol currently being scanned to 0. More precisely, PRINT-0 changes memory-state *e* to (0, 1) and memory-state $s_1s_2 \ldots s_{i-1}\underline{s_i}s_{i+1} \ldots s_k$ to $s_1s_2 \ldots s_{i-1}\underline{0}s_{i+1} \ldots s_k$. (Of course, if s_i were 0 to begin with, PRINT-0 would not change the tape.)
3. PRINT-1. (Exercise: Describe the behavior of PRINT-1.)
4. LEFT. This moves the scanner one symbol to the left, adding on a new square with a 0 in it if necessary. (*Remember:* Our Turing machine is on wheels; *it* moves, not the tape.) More precisely, LEFT leaves memory-state *e* alone; it changes $s_1 \ldots \underline{s_i} \ldots s_k$ to $s_1 \ldots \underline{s_{i-1}}s_i \ldots s_k$; and it changes $\underline{s_1} \ldots s_k$ to $\underline{0}s_1 \ldots s_k$.

5. RIGHT. This operation moves the scanner one symbol to the right, adding a new square with a 0 in it if necessary. That is, it leaves *e* alone; it changes $s_1 \ldots \underline{s_j} \ldots s_k$ to $s_1 \ldots s_j\underline{s_{j+1}} \ldots s_k$; and it changes $s_1 \ldots \underline{s_k}$ to $s_1 \ldots s_k\underline{0}$.
6. ERASE. This operation does nothing unless the scanner is at the left-hand end or the right-hand end of the tape, in which case it erases the symbol being scanned and cuts off the square. That is, ERASE leaves *e* and $s_1 \ldots \underline{s_j} \ldots s_k$ alone; but it changes $\underline{s_1}s_2 \ldots s_k$ to $\underline{s_2} \ldots s_k$, and it changes $s_1 \ldots s_{k-1}\underline{s_k}$ to $s_1 \ldots \underline{s_{k-1}}$; also, it causes a 1-square tape with only one symbol on it to disappear (that is, $\underline{s}$ becomes *e*).
7. STOP.

The tests are:

1. 0? and 1? These two test whether the symbol being scanned is 0 or 1. Formally, if the tape is *e*, then the result of the tests is FALSE; and if the tape is $s_1 \ldots \underline{s_j} \ldots s_k$, then the result is TRUE if $s_i = s$ and FALSE otherwise (where s is 0 or 1, depending on the test).
2. LEFTEND? Tests whether the scanner is at the left-hand end of the tape. If it is, or if the tape is empty, the test result is TRUE; otherwise, it is FALSE.
3. RIGHTEND? Tests whether the scanner is at the right-hand end of the tape. If it is, or if the tape is empty, the test result is TRUE; otherwise, it is FALSE.

Turing-Machine Programs

Let us see what some of our simple algorithms look like when translated into programs in TM.

Negation

We'll begin with the algorithm for FNEG, that is, for computing the truth value of the negation of a sentence (see Chapter 3). The first decision we need to make is how to "code" truth values so that they can be expressed on the tape. One obvious way is to represent a TRUE sentence with a 1-square tape containing a '1' and a FALSE sentence with a 1-square tape containing a '0'.

Next, we must decide what the output should look like. One possibility is to have our algorithm end with a 2-square tape whose first symbol is the original truth value and whose second symbol represents the negation of that truth value, with the scanner on the second symbol. (*Remember:* We can add new squares to our tape whenever we want.) Thus for input '$\underline{1}$', the output would be '$1\underline{0}$', and for input '$\underline{0}$', the output would be '$0\underline{1}$'.

So, we'll need to scan the input tape and, if there's a '$\underline{1}$', move right and print a '0', but if there's a '$\underline{0}$', we'll move right and print a '1'.

In TM we shall let the "states of mind" be represented by M_0, M_1, M_2, M_3, etc. We shall always begin (START) in the state of mind M_0. The program for computing FNEG in the manner we have just described is:

```
1. START.
2. IF you are in $M_0$ and are scanning 0,
     THEN
   (a) RIGHT.
   (b) Change state of mind to $M_1$.
3. IF you are in $M_0$ and are scanning 1,
     THEN
   (a) RIGHT.
   (b) Change state of mind to $M_2$.
4. IF you are in $M_1$ and are scanning 0,
     THEN
   (a) PRINT-1.
   (b) Change state of mind to $M_2$.
5. IF you are in $M_2$,
     THEN STOP.
```

Notice that you will never be in M_1 and scanning 1, so that possibility is not covered in the program.

It is easier to represent the program using a flowchart, where your position on the chart corresponds to a "state of mind":

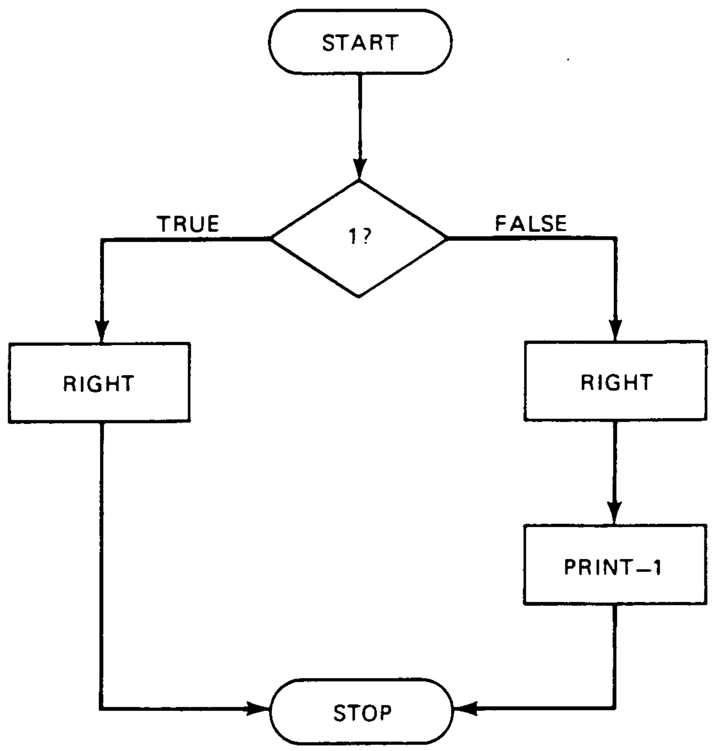


Figure B-4 TM program (in flowchart form) for computing the truth value of a negation.

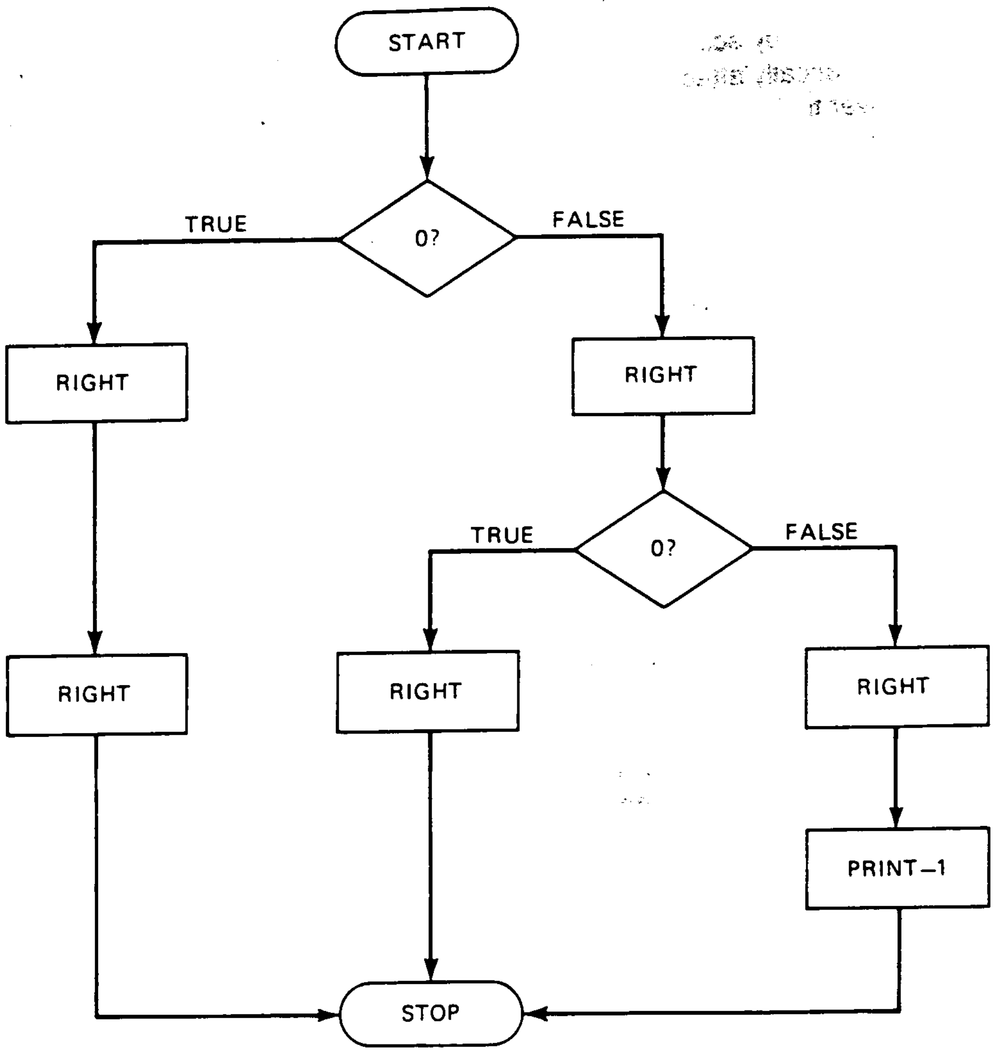


Figure B-5 TM program for computing the truth value of a conjunction.

9. An algorithm for adding two positive integers. Let the input-output coding be as follows: The addition problem $m + n$ will be representd by m '1's, followed by a '0', followed by n '1's. A nice notation for this is:

   1^m01^n

   The output tape will consist of $m + n$ '1's, that is:

   1^{m+n}

Palindromes

As a final example of TM programming, we shall sketch out a program that decides if the input tape contains a palindrome. A *palindrome* will be any string of '0's and '1's that reads the same backward and forward. For instance,

000
010
001100

are palindromes, but

01
110
0011110

are not. Clearly, a 1-square tape always contains a palindrome. It will be useful to consider the empty tape to be a palindrome too.

The input will be a tape containing the string to be tested, with the scanner on the first (leftmost) symbol. The output will be a 1-square tape with a '1' if the input is a palindrome and a '0' otherwise. Two examples are as follows:

Input	**Output**
$\underline{1}01$	$\underline{1}$
$\underline{1}00$	$\underline{0}$

The algorithm that we shall use is the following:

```
1. IF the tape is empty
      THEN OUTPUT '1' [since it is a palindrome].
2. IF the tape has only 1 square
      THEN OUTPUT '1' [since it is a palindrome].
3. IF the tape has 2 or more squares
      THEN
      (a) Compare the symbol in the leftmost square with the symbol in the
          rightmost square.
      (b) IF they match
              THEN
              (i)  Erase both symbols.
              (ii) GO TO step 1.
      (c) IF they do not match
              THEN
              (i)  Erase the tape.
              (ii) OUTPUT '0' [since it is not a palindrome].
4. STOP.
```

(You should try this algorithm on a few examples to convince yourself that it works.)

This time, we shall use top-down design and stepwise refinement to write the program (and we shall leave some of the details as exercises). To do this, we shall introduce a new flowchart symbol—a circle—to represent steps that will have to be refined into the basic TM operations and tests.

The top-level program is:

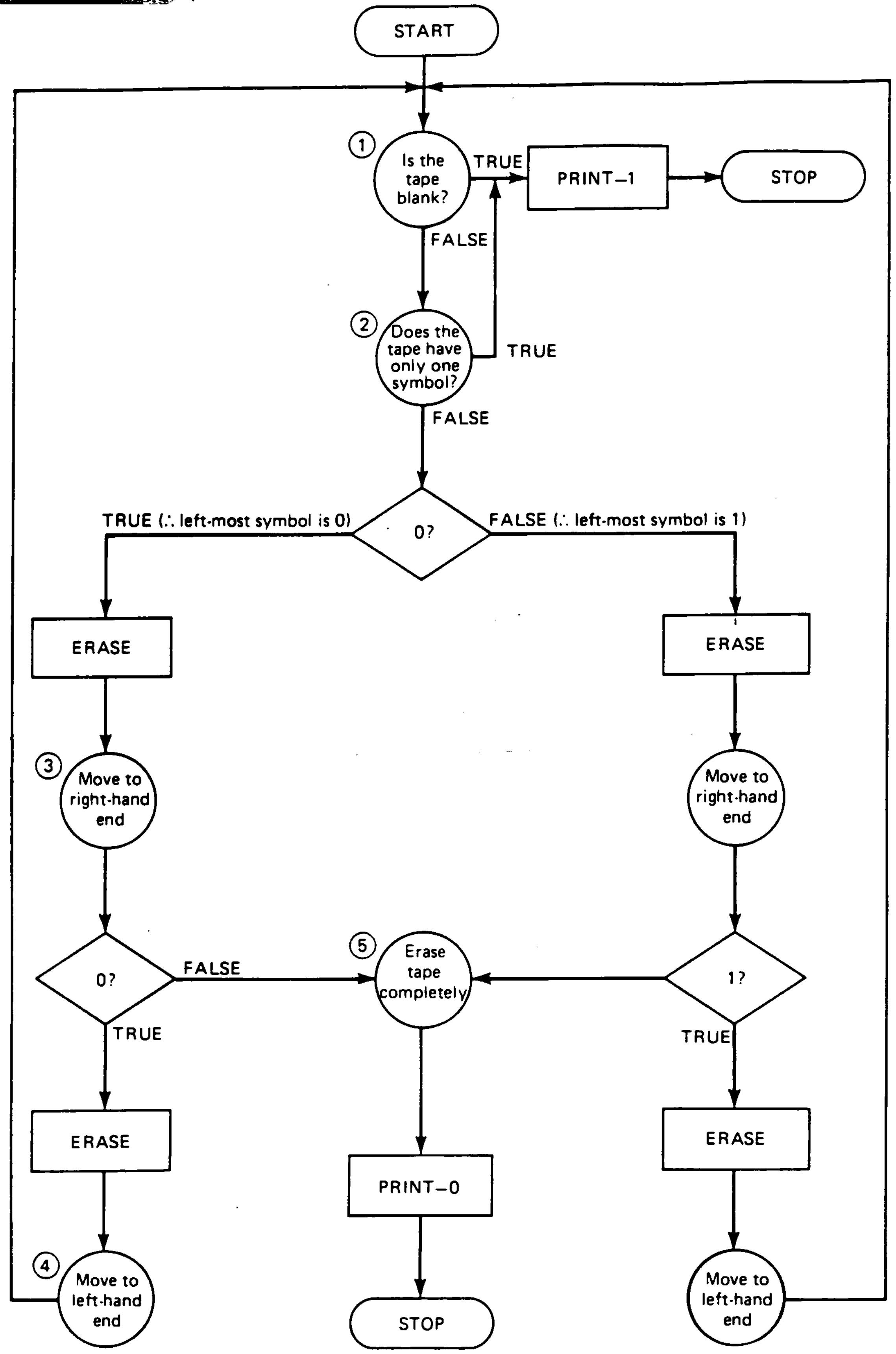


Figure B-6 TM program for the palindrome problem.

The circles labeled 1 to 5 represent procedures that must be refined. Here is the refinement of procedure 1:

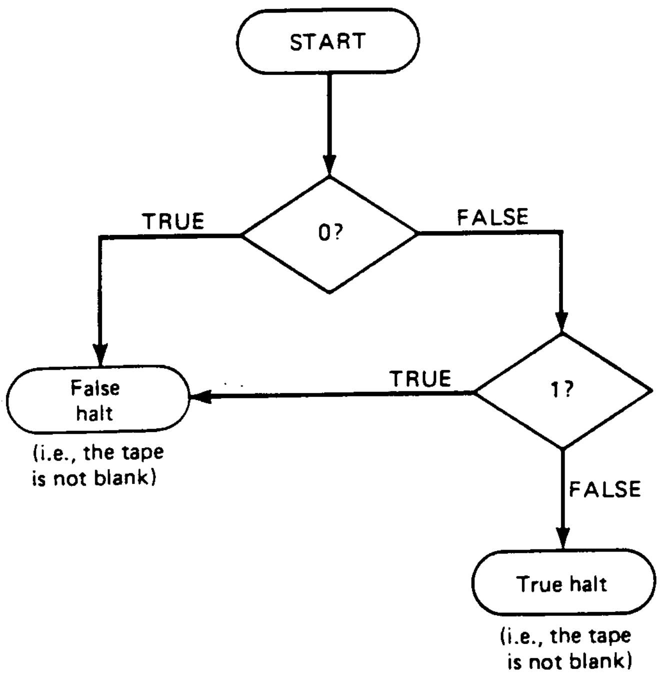


Figure B-7 TM program for procedure 1 of the palindrome program.

And here is the refinement of procedure 5:

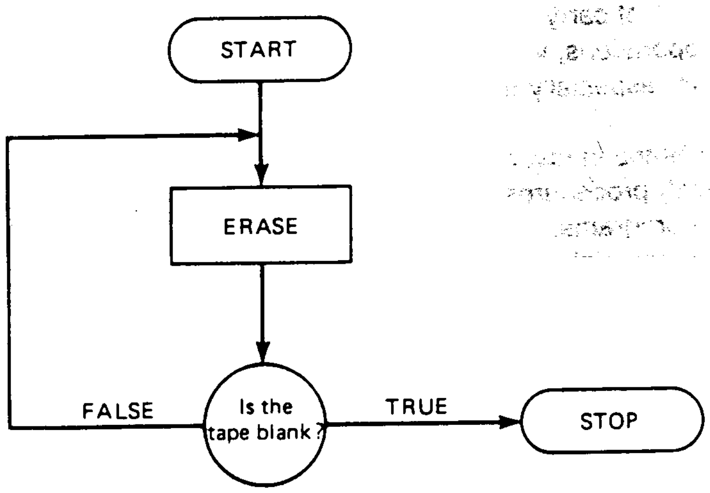


Figure B-8 TM program for procedure 5 of the palindrome program.

Exercises

10. Refine procedure 2.
11. Refine procedure 3.
12. Refine procedure 4.
13. Refine the given refinement of 5 (in Figure B-8).
14. Refine procedure 5 without using procedure 1. (*Hint:* Use LEFTEND?)

Church's Thesis

Let us consider the relationship between a Turing machine and our notion of an algorithm.

One version of *Church's thesis* (see Chapter 14), known as the *Church-Turing thesis,* says that our intuitive notion of a mechanical computation can be *defined* as any computation that can be carried out by a Turing machine. This cannot be proved (because our notion of a mechanical computation is intuitive and vague, and proofs require precise notions), but there is a substantial amount of evidence to suggest the following two points:

1. Any computation that seems intuitively "mechanical" can be carried out on a Turing machine.
2. Any computation that can be carried out on a Turing machine is intuitively "mechanical."

The main evidence for statement 2 is Turing's analysis of computation that we discussed in an earlier section. It seems, intuitively, that any computation that can be done by a Turing machine is intuitively mechanical.

We said that the thesis cannot be proved; but if it is false, then it *can* be falsified: If anyone were to deny the Church-Turing thesis, it would have to be because he or she denied statement 1, that is, claimed that there might be an intuitively mechanical procedure that a Turing machine could not carry out—that, perhaps, Turing machines, with all their limitations of size and of operations, were not powerful enough. (After all, those instructions seem rather simplistic, especially if the only symbols allowed are '0' and '1'.)

But there are two sorts of evidence in favor of statement 1. First, there is empirical evidence: All (intuitively mechanical) procedures that have been constructed so far can be translated into Turing-machine programs.

Second, all "rival" theories of what might count as being "mechanically computable" have turned out to be equivalent to the theory of Turing-machine computability.

Some of these other theories of what is "mechanically computable" are:

1. Church's lambda calculus—the notion he used in the original formulation of the thesis; it was later used as the basis for the programming language LISP.
2. Markov algorithms—which later became the basis for the programming language SNOBOL.

3. The theory of partial-recursive functions, which lies at the foundation of Gödel's incompleteness theorem, and which also can be considered to underlie much of the recent theory of "structured" programming.
4. The Herbrand-Gödel theory of recursion equations—which later became the basis for the programming language ALGOL (and thus, indirectly, its descendant Pascal).
5. The theory of register machines—which is a mathematical analogue of a "von Neumann" digital computer.