Chapter 8

Introduction to Turing Machines

(2015/12/9)

Rothenberg, Germany

Outline

8.0 Introduction

8.1 Problems that Computers Cannot Solve

8.2 The Turing Machine (TM)

8.3 Programming Techniques for TM’s

8.4 Extensions to the Basic TM

8.5 Restricted TM’s

8.6 TM’s and Computers

8.0 Introduction

- **Concepts to be taught ---**
  - ♦ Studying questions about what languages can be defined by any computational device.
  - ♦ There are specific problems that cannot be solved by computers! --- undecidable!
  - ♦ Studying the Turing machine which seems simple, but can be recognized as an accurate model for any physical computing device.

8.1 Problems That Computers Cannot Solve

- **Purpose of this section ---**

  To provide an informal proof (C-programming-based brief proof) of a specific problem that computers *cannot* solve.

- **The problem is:**

  Whether the first thing that a C program prints is

  hello, world.

  - ♦ We will give the intuition behind the formal proof.

8.1.1 Programs that print "Hello, World"

- A C program that prints "Hello, World" is:

```
main()
{
        print("hello, world\n");
}
```

  - ♦ Define a "*hello, world problem*" to be:

    Determine whether a given C program, with a given input, prints *hello, world* as the first 12 characters in what it prints.

  - ♦ Describe the problem *alternatively* using symbols:

    Is there a program *H* that could examine any program *P* and any input *I* for *P*, and tell whether *P*, run with *I* as its input, would print *hello*, *world*?

    (A program *H* means an algorithm in concept here.)

    - The answer is: *undecidable*!
    - That is, there exists no such program *H*.
    - We can prove this by contradiction next.

8.1.2 Hypothetical "Hello, World" Tester

- We want to prove that no program *H*, called *hypothetical* "Hello, World" *tester*, as mentioned above exists by contradiction using the following steps.

♦ Step 1 --- assume H exists in a form as shown in Fig. 8.1 (Fig 8.3 in the textbook).

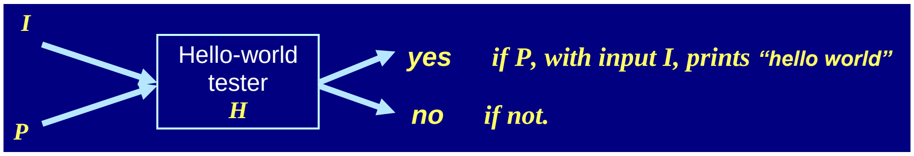

Fig. 8.1 A hypothetical "Hello, World" tester.

♦ Step 2 --- transform H into another form H_2 in a simple way which can be done by C programs.

♦ Step 3 --- prove that H_2 does not exist and so that H does not exist, either.

■ Implementation of Step 2 above ---

(1) Transform H to H_1 in a way as illustrated by Fig. 8.2 (Fig. 8.4 in the textbook).

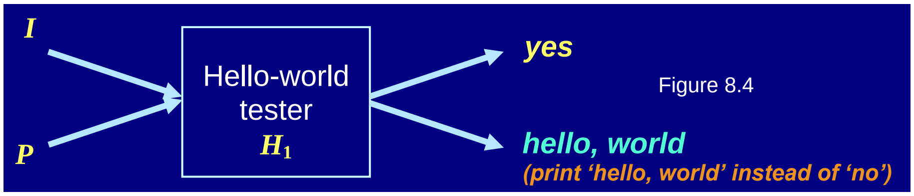

Fig. 8.2 A transformed "hello-world tester" H_1.

(2) Transform H_1 to H_2 in a way as illustrated by Fig. 8.3 (Fig. 8.5 in the textbook).

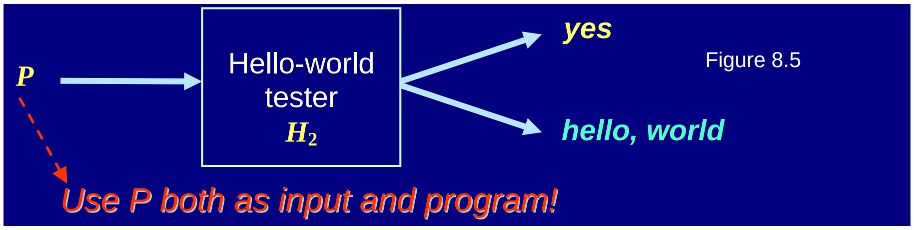

Fig. 8.2 A second transformed "hello-world tester" H_2.

■ The function of H_2 constructed in Step 2 is ---

given any program P as input,

if P prints hello, world *as first output, then H_2 makes output yes;*
if P does not prints hello, world *as first output, then H_2 prints* hello, world.

■ Implementation of Step 3 above (proving H_2 does not exist) ---

♦ Let P for H_2 in Fig. 8.2 (last figure) be H_2 itself, as illustrated in Fig. 8.3 (Fig. 8.6 in the textbook).

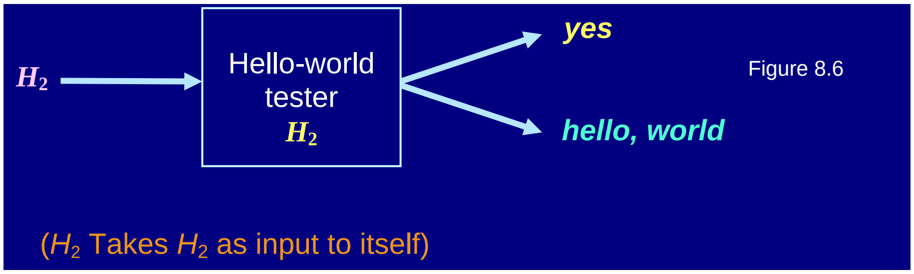

Fig. 8.3 A second transformed "hello-world tester" H_2 taking itself as input.

- ♦ Now, we have the following reasoning (assuming the term "box" means "Hello-world tester" ---

(1) If

the box H_2, given itself as input, makes output yes,

then according to the above-described function of H_2, this means that

the box H_2, given itself as input, prints hello, world *as the first output.*

But this is contradictory because we just suppose that

the box H_2, given itself as input, makes output yes.

(2) The above contradiction means the other alternative must be true since there are only two choices, that is ---

the box H_2, given itself as input, prints hello, world *as the first output.*

But according to the above-described function of H_2, this means that

such H_2, when taken as input to the box H_2 (itself), will make the box H_2 to make output yes.

This is a contradiction again because we just say that

the box H_2, given itself as input, prints hello, world *as the first output.*

- ♦ Since both cases lead to contradiction, we conclude that the assumption that H_2 exists is wrong by the principle of contradiction for proof.

- ♦ H_2 does not exist $\Rightarrow$ H_1 does not exist (otherwise, H_2 must exist)
  $\Rightarrow$ H does not exist (otherwise, H_1 must exist), done!
  ("$\Rightarrow$" means "imply" here)

- ■ The above *self-contradiction* technique, similar to the *diagonalization* technique (to be introduced later), was used by Alan Turing for proving undecidable problems.

8.1.3 <u>Reducing One Problem to Another</u>

- Now we have an undecidable problem, which can be used to prove other undecidable problems by a technique of *problem reduction*.

  - That is, if we know P_1 is undecidable, then we may *reduce P_1 to a new problem P_2*, so that we can prove P_2 undecidable by contradiction in the following way
    - *If P_2 is decidable, then P_1 is decidable.*
    - *But P_1 is known undecidable. So, contradiction!*
    - *Consequently, P_2 is undecidable.*

- An illustration of the above idea is illustrated in Fig. 8.4.

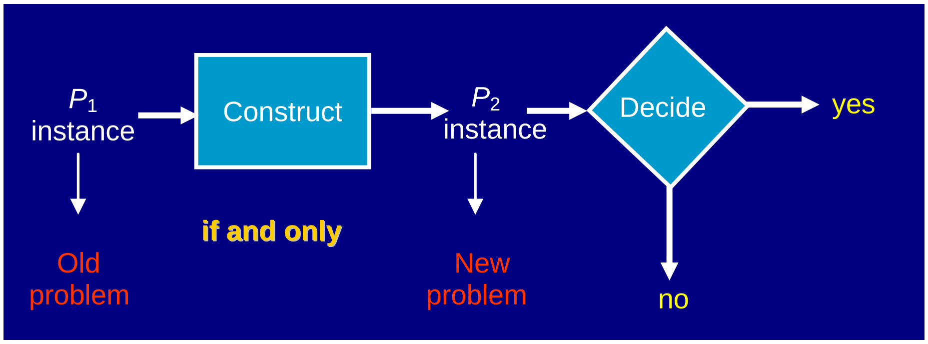

Fig. 8.4 An illustration of reducing one problem to another.

- **Example 8.1 ---**

  We want to prove a new problem P_2 (called ***calls-foo*** *problem*):

  *"does program Q, given input y, ever call function **foo**?"*

  to be undecidable.

  Solution:

  - Reduce P_1: *the hello-world problem* to P_2: *the **calls-foo** problem* in the following way:
    - If *Q* has a function called ***foo***, *rename it and all calls* to that function ⇒ a new program Q_1 doing the same as *Q*. ("⇒" means "leading to" here)
    - Add to Q_1 a function ***foo*** doing nothing & *not* being called ⇒ a new program Q_2.
    - Modify Q_2 to remember the first 12 characters that it prints, storing them in a global array *A* ⇒ a new program Q_3.
    - Modify Q_3 in such a way that whenever it executes any output statement, it checks *A* to see if it has written 12 characters or more, and if so, whether *hello, world* are the first characters. In that case (i.e., if so), call the new function ***foo*** ⇒ a new program *R* with input *y*.

  - Now,
    - if *Q* with input *y* prints *hello, world* as its first output, then *R* will call ***foo***;
    - if *Q* with input *y* does not print *hello, world*, then *R* will never call ***foo***.
  - That is, program *R*, with input *y*, calls ***foo*** if and only if program *Q*, with input *y*, prints *hello, world*.

  - So, if we can decide whether *R*, with input *y*, calls ***foo***, then we can decide whether *Q*, with input *y*, prints *hello, world*.

- But *the latter is impossible* as has been proved before, so the former is impossible.

■ The above example illustrates how to reduce a problem to another as illustrated in Fig. 8.4.

8.2 The Turing Machine

■ **Concepts to be taught ---**

- The study of decidability provides guidance to programmers about what they might or might not be able to accomplish through programming.
- Previous problems are dealt with programs. But *not* all problems can be solved by programs.
- We need a simple model to deal with other decision problems (like grammar ambiguity problems)
- The *Turing machine* is one of such models, whose configuration is easy to describe, but whose function is the most versatile:

  all computations done by a modern computer can be done by a Turing machine.

  (a hypothesis which is not proved but believed so far!)

8.2.1 The Quest to Decide All Mathematical Questions ---

■ **History ---**

- At the turn of 20th century, D. Hilbert asked:

  "*whether it was possible to find an algorithm for determining the truth or falsehood of any mathematical proposition*."

  (in particular, he asked if there was a way to decide *whether any formula in the 1st-order predicate calculus, applied to integer, was true*)

- In 1931, K. Gödel published his ***incompleteness theorem***:

  "*A certain formula in the predicate calculus applied to integers could not be neither proved nor disproved within the predicate calculus*."

- The proof technique is ***diagonalization***, resembling the *self-contradiction* technique used previously (invented by Turing).

■ **Natures of computational model ---**

- *Predicate calculus* --- declarative
- *Partial-recursive functions* --- computational (a programming-language-like notion)
- *Turing machine* --- computational (computer-like)
  (invented by Alan Turing several years before true computers were invented)

■ **Equivalence of *maximal* computational models ---**

All maximal computational models compute the same functions or recognize the same languages, having the same power of computation.

- *Unprovable* **Church-Turing hypothesis (or thesis) ---**

  Any general way to compute will allow us to compute only the partial-recursive functions (or equivalently, only what the Turing machine or modern-day computers can compute).

8.2.2 Notion for the Turing Machine

- **A model for Turing machine ---** as shown in Fig. 8.5.

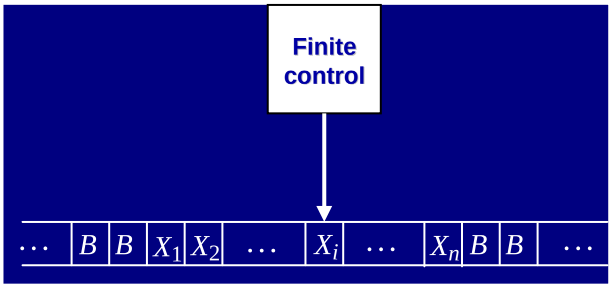

Fig. 8.5 A model for the Turing machine.

- **A move of Turing machine includes ---**
  - change state;
  - write a tape symbol in the cell scanned;
  - move the tape head left or right.

- **Formal definition ---**

  A Turing machine (TM) is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ where

  - Q: a finite set of states of the finite control;
  - Σ: a finite set of input symbols;
  - Γ: a set of tape symbols, with Σ being a *subset* of it;
  - δ: a transition function $\delta(q, X) = (p, Y, D)$ where
    - q: the current state, in Q;
    - X: a tape symbol being scanned;
    - p: the next state, in Q;
    - Y: the tape symbol written on the cell being scanned, used to replace X;
    - D: either L (left) or R (right) telling the move direction of the tape head;
  - q_0: the start state, in Q;
  - B: the blank symbol in Γ, not in Σ (should not be an input symbol);
  - F: the set of final (or accepting) states.
  - A TM is a deterministic automaton with a two- way infinite tape which can be read and written in either direction.

- **A nature of the Turing machine ---** A TM is a *deterministic* automaton with a two-way *infinite* tape which can be *read* and *written* in *either direction*.

8.2.3 Instantaneous Descriptions for Turing Machine

The *instantaneous description* (ID) of a TM ---

The ID of a TM is represented by $X_1X_2\ldots X_{i-1}qX_iX_{i+1}\ldots X_n$ in which

- q is the current state;
- the tape head is scanning the ith symbol X_i from the left;
- $X_1X_2\ldots X_n$ is the portion of the tape between the leftmost and the rightmost nonblank symbols.

Moves of a TM ---

- The moves of a TM M are denoted by $\vdash_M$ or $\vdash$.
- If $\delta(q, X_i) = (p, Y, L)$ (a leftward move), then we write the following to describe the left move:

$$X_1X_2\ldots X_{i-1}qX_iX_{i+1}\ldots X_n \vdash_M X_1X_2\ldots X_{i-2}pX_{i-1}YX_{i+1}\ldots X_n.$$

- Right moves are defined similarly.

Example 8.2 ---

Design a TM to accept the language $L = \{0^n1^n \mid n \geq 1\}$.

- The machine may be designed by the following steps.
  - Starting at the left end of the input.
  - Change 0 to an X.
  - Move to the right over 0's and Y's until a 1.
  - Change 1 to Y.
  - Move left over Y's and 0's until an X.
  - Look for a 0 immediately to the right.
  - If a 0 is found, change it to X and repeat the above process.

- An example illustrating the above steps is as follows (the blue character indicates the position of the reading head).

0011 → X011 → X0Y1 → XXY1 → … → XXYY → XXYYB

- The TM is defined formally as follows:

$$M = (\{q_0\sim q_4\}, \{0, 1\}, \{0, 1, X, Y, B\}, \delta, q_0, B, \{q_4\})$$

  - Transition table for δ is as shown in Table 8.1.
  - The moves to accept the input string w = 0011 are as follows (use $\Rightarrow$ instead of $\vdash$):

$$q_00011 \Rightarrow_1 Xq_1011 \Rightarrow_2 X0q_111 \Rightarrow_4 Xq_20Y1 \Rightarrow_5 q_2X0Y1 \Rightarrow_7 Xq_00Y1 \Rightarrow_1 XXq_1Y1 \Rightarrow_3 XXYq_11$$
$$\Rightarrow_4 XXq_2YY \Rightarrow_6 Xq_2XYY \Rightarrow_7 XXq_0YY \Rightarrow_8 XXYq_3Y \Rightarrow_9 XXYYq_3B \Rightarrow_{10} XXYYBq_4B.$$

where the red numbers on the right sides of the arrows "⇒" in the moves are used to specify the used transitions according to Table 8.1.

Table 8.1. The transition table for the TM of Example 8.2.

	symbol				
state	0	1	X	Y	B
q_0	$(q_1, X, R)_1$	-	-	$(q_3, Y, R)_8$	-
q_1	$(q_1, 0, R)_2$	$(q_2, Y, L)_4$	-	$(q_1, Y, R)_3$	-
q_2	$(q_2, 0, L)_5$	-	$(q_0, X, R)_7$	$(q_2, Y, L)_6$	-
q_3	-	-	-	$(q_3, Y, R)_9$	$(q_4, B, R)_{10}$
q_4	-	-	-	-	-

(Note: red numbers are used to distinguish the transitions.)

8.2.4 Transition Diagrams for TM's

- **Notations ---**
  - If $\delta(q, X) = (p, Y, L)$, we use label $X/Y \leftarrow$ on the arc.
  - If $\delta(q, X) = (p, Y, R)$, we use label $X/Y \rightarrow$ on the arc.

- **Example 8.3 ---**

The transition diagram for Example 8.2 is as shown in Fig. 8.6 (Fig. 8.10 in the textbook).

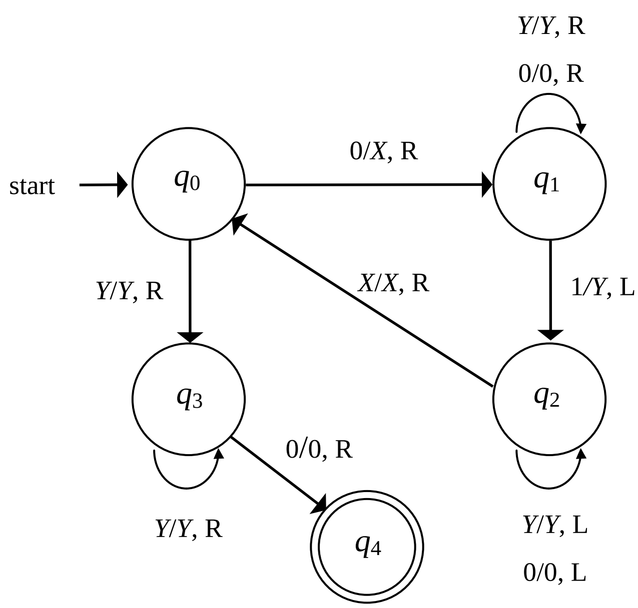

Fig. 8.6 Transition diagram of Example 8.3.

- **Example 8.4 ---**

The TM may use as a *function-computing machine*. *No final state is needed*. For details, see the textbook (pp. 331-334) and later sections.

8.2.5 The Language of a TM

- **Definition ---**

  Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be a TM. The *language* accepted by M is

  $$L(M) = \{w \mid w \in \Sigma^* \text{ and } q_0 w \overset{*}{\underset{M}{\vdash}} \alpha p \beta \text{ with } p \in F\}.$$

  - A string w need not be processed to its end; as long as the machine enters a final state, w can be accepted.
  - The set of languages accepted by a TM is often called the *recursively enumerable language or RE language.*
    - The term "RE" came from computational formalism that predates the TM.

8.2.6 TM's and Halting

- Another notion for accepting strings by TM's --- *acceptance by halting*.

- **Definition ---**

  We say a TM *halts* if it enters a state q scanning a tape symbol X, and there is no move in this situation, i.e., $\delta(q, X)$ is *undefined*.

  - Acceptance by halting may be used for a TM's functions other than accepting languages like Example 8.4 and Example 8.5.
  - We assume that a TM always halts when it is in an accepting state.
  - It is not always possible to require that a TM halts even when it does not accept.

- **Properties of Halting ---**
  - Languages with TM's that do halt eventually, regardless whether or not they accept, are called *recursive languages* (considered in Sec. 9.2.1)
  - TM's that always halt, regardless of whether or not they accept, are a good model of an "algorithm."
  - So TM's that always halt can be used for studies of decidability (see Chapter 9).

8.3 Programming Techniques for TM's

- **Concepts to be taught ---**
  - Showing how a TM computes.
  - Indicating that TM's are as powerful as conventional computers.
  - Even some extended TM's can be simulated by the original TM.

- **Section 8.2 revisited ---**
  - TM's may be used as a computer as well, not just a language recognizer.

  - **Example 8.4 (not taught in the last section) ---**

    Design a TM to compute a function denoted by "$\dot{-}$" called *monus*, or *proper subtraction* defined by

$$m \dot{-} n = m - n \qquad \text{if } m \geq n;$$
$$= 0 \qquad \text{if } m < n.$$

- Assume input integers m and n are put on the input tape separated by a 1 as 0^m10^n *(two unary numbers using 0's separated by a special symbol 1).*
- The TM is $M = (\{q_0, q_1, \ldots, q_6\}, \{0, 1\}, \{0, 1, B\}, \delta, q_0, B)$.
- *No final state is needed.*
- M conducts the following computation steps:
  1. find its leftmost 0 and replaces it by a blank;
  2. move right, and look for a 1;
  3. after finding a 1, move right continuously
  4. after finding a 0, replace it by a 1;
  5. move left until finding a blank, & then move one cell to the right to get a 0;
  6. repeat the above process.
- The transition table of M is as shown in Table 8.2.

Table 8.1. The transition table for the TM of Example 8.4.

	symbol		
state	0	1	B
q_0	(q_1, B, R)	(q_5, B, R)	-
q_1	$(q_1, 0, R)$	$(q_2, 1, R)$	-
q_2	$(q_3, 1, L)$	$(q_2, 1, R)$	(q_4, B, L)
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	(q_0, B, R)
q_4	$(q_4, 0, L)$	(q_4, B, L)	$(q_6, 0, R)$
q_5	(q_5, B, R)	(q_5, B, R)	(q_6, B, R)
q_6	-	-	-

- Moves to compute $2 \dot{-} 1 = 1$:
  $q_0\underline{0}010 \Rightarrow_1 Bq_1\underline{0}10 \Rightarrow_3 B0q_1\underline{1}0 \Rightarrow_4 B01q_2\underline{0} \Rightarrow_5 B0q_3\underline{1}1 \Rightarrow_9 Bq_3\underline{0}11 \Rightarrow_8 q_3\underline{B}011 \Rightarrow_{10}$
  $Bq_0\underline{0}11 \Rightarrow_1 BBq_1\underline{1}1 \Rightarrow_4 BB1q_2\underline{1} \Rightarrow_6 BB11q_2\underline{B} \Rightarrow_7 BB1q_4\underline{1} \Rightarrow_{12} BBq_4\underline{1}B \Rightarrow_{12}$
  $Bq_4\underline{B}BB \Rightarrow_{13} B0q_6\underline{B}B$ halt! (with one 0 left, correct)

- Moves to compute $1 \dot{-} 2 = 0$:
  $q_0\underline{0}100 \Rightarrow Bq_1\underline{1}00 \Rightarrow B1q_2\underline{0}0 \Rightarrow Bq_3\underline{1}10 \Rightarrow q_3\underline{B}110 \Rightarrow Bq_0\underline{1}10 \Rightarrow BBq_5\underline{1}0 \Rightarrow$
  $BBBq_5\underline{0} \Rightarrow BBBBq_5\underline{B} \Rightarrow BBBBBq_6$ halt! (with no 0 left, correct)

■ For details of the following three sections, see the textbook.

8.3.1 Storage in the State
8.3.2 Multiple Tracks
8.3.3 Subroutines

8.4 Extensions to the Basic TM

- **Extended TM's to be studied ---**
  - ♦ Multitape Turing machine
  - ♦ Nondeterministic Turing machine

- The above extensions make no increase of the original TM's power, but make TM's easier to use:
  - ♦ Multitape TM --- useful for simulating real computers
  - ♦ Nondeterministic TM --- making *TM programming* easier.

8.4.1 Multitape TM's

- **A graphic model of a multitape TM** --- shown in Fig. 8.

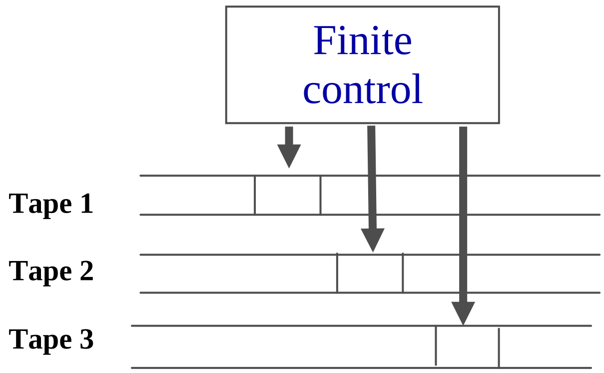

Fig. 8.7 A graphic model of a multitape TM.

- **Function of a multitape TM ---**
  - ♦ Initially,
    - • the input string is placed on the 1st tape;
    - • the other tapes hold all blanks;
    - • the finite control is in its initial state;
    - • the head of the 1st tape is at the left end of the input;
    - • the tape heads of all other tapes are at arbitrary positions.

  - ♦ A *move* consists of the following steps ---
    - • the finite control enters a new state;
    - • on each tape, a symbol is written;
    - • each tape head moves left or right, or *stationary*.

8.4.2 Equivalence of One–tape & Multitape TM's

- **Theorem 8.9 ---**

  Every language accepted by a multitape TM is recursive enumerable.

(That is, the one-tape TM and the multitape one are equivalent)

Proof: see the textbook.

8.4.3 Running Time and the Many-Tapes-to-One Construction

- **Theorem 8.10 ---**

  The time taken by the one-tape TM of Theorem 8.9 to simulate n moves of the k-tape TM is $O(n^2)$.

  Proof: see the textbook.

- **Meaning ---** the equivalence of the two types of TM's is good in the sense that their running times are *roughly the same within polynomial complexity*.

8.4.4 Nondeterministic TM's

- **Definition ---**

  A nondeterministic TM (NTM) has multiple choices of next moves, i.e.,

$$(q, X) = \{(q_1, Y_1, D_1), (q_2, Y_2, D_2), \ldots, (q_k, Y_k, D_k)\}.$$

- The NTM is not more 'powerful' than a deterministic TM (DTM), as said by the following theorem.

- **Theorem 8.11 ---**

  If M_N is NTM, then there is a DTM M_D such that $L(M_N) = L(M_D)$.

  ***Proof*:** see the textbook.

- **Some properties ---**
  - ♦ The equivalent DTM constructed for an NTM in the last theorem may take exponentially more time than the DTM.
  - ♦ It is unknown whether or not this *exponential slowdown* is necessary!
  - ♦ More investigation will be done in Chapter 10.

8.5 Restricted TM's

- **Restricted TM's to be studied ---**
  - ♦ The tape is infinite only to the right, and the blank cannot be used as a replacement symbol.
  - ♦ The tapes are only used as stacks ("stack machines").
  - ♦ The stacks are used as counters only ("counter machines").

- The above restrictions make no decrease of the original TM's power, but are useful for theorem proving.
- Undecidability of the TM also applies to these restricted TM's.

8.5.1 TM's with Semi-infinite Tapes

■ **Theorem 8.12 ---**

Every language accepted by a TM M_2 is also accepted by a TM M_1 with the following restrictions:

♦ M_1's head never moves left of its initial position (so the tape is semi-infinite essential);
♦ M_1 never writes a *blank*.
(i.e., M_1 and M_2 are equivalent)

Proof. See the textbook.

8.5.2 Multistack Machines

■ **Concepts ---**

♦ Multistack machines, which are restricted versions of TM's, may be regarded as extensions of pushdown automata (PDA's).
♦ Actually, a PDA with *two* stacks has the same computation power as the TM.

■ **Definition ---**

A *k*-stack machine is a deterministic PDA with *k* stacks.

♦ See Fig.8.20 for a figure of a multistack TM.

■ **Theorem 8.13 ---**

If a language is accepted by a TM, then it is accepted by a two-stack machine.

Proof. See the textbook.

8.5.3 Counter Machines

■ There are two ways to think of a counter machine.

♦ Way 1: as a multistack machine with each stack replaced by a counter *regarded to be on a tape of a TM*.
- A counter holds any nonnegative integer.
- The machine can only distinguish zero and nonzero counters.
- A move conducts the following operations:
  - changing the state;
  - add or subtract 1 from a counter which cannot becomes negative.

♦ Way 2: as a *restricted* multistack machine with each stack replaced by a counter *implemented on a stack of a PDA*.
- There are only two stack symbols Z_0 and X.
- Z_0 is the initial stack symbol, like that of a PDA.
- Can replace Z_0 only by X^iZ_0 for some $i \geq 0$.
- Can replace X only by X^i for some $i \geq 0$.

♦ For an example of a counter machine of the 2nd type, do the exercise (part a) of this chapter.

8.5.4 The Power of Counter Machines

- Every language accepted by a one-counter machine is a CFL (see the textbook).
- Every language accepted by a counter machine (of any number of counters) is recursive enumerable (see theorems below).

- **Theorem 8.14 ---**

  Every recursive enumerable language is accepted by a three-counter machine.

Proof. See the textbook.

- **Theorem 8.15 ---**

  Every recursive enumerable language is accepted by a two-counter machine.

Proof. See the textbook.

8.6 Turing Machines and Computers

- **In this section, it is shown informally:**
  - ♦ a computer can simulate a TM;
  - ♦ a TM can simulate a computer.

- **That means:**
  - ♦ the real computer we use every day is *nearly* an implementation of the maximal computational model under the following assumptions
    - • the memory space (including registers, RAM, hard disks, …) is infinite in size;
    - • the address space is infinite (*not* only that defined by 32 bits used in most computers today).

- For more details, see the textbook.