9. TURING MACHINES

9.1 Definitions and Examples

- In 1936, English mathematician Alan Turing introduced an abstract machine that is now called a *Turing machine*. This machine, unlike the ones in previous chapters, is capable of performing any computation that can be done by a modern electronic computer.

- Turing based his machine on what a *human* computer could do with pencil and paper, assuming that the human follows a mechanical set of rules that depend only the symbol currently being examined and the human's "state of mind" (both chosen from finite sets).

- A Turing machine has a finite alphabet of symbols (actually two alphabets, an input alphabet and a possibly larger alphabet for use during the computation) and a finite set of states.

- Symbols are written on a "tape," which has a left end and is potentially infinite to the right. The tape is marked off into squares, each of which can hold one symbol. If a square has no symbol on it, it is said to contain the *blank* symbol.

- The reading and writing is done by a *tape head,* which at any time is centered on one square of the tape.

- In the Turing machine model described here—which is similar although not identical to the one proposed by Turing—a move is determined by the current state and the current tape symbol, and consists of three parts:

  1. Replacing the symbol in the current square by another, possibly different symbol.
  2. Moving the tape head one square to the right or left (unless it is already centered on the leftmost square), or leaving it where it is.
  3. Moving from the current state to another, possibly different state.

Definitions and Examples (Continued)

- The tape serves as the input device (the input is the string of nonblank symbols on the tape originally), the memory available during the computation, and the output device (the output is the string of symbols left on the tape at the end).

- The most significant difference between Turing machines and simpler kinds of automata is that a Turing machine is not restricted to a single left-to-right pass through the input.

- A Turing machine can get by with two *final*, or *halting*, states: a state h_a that indicates acceptance and another h_r that indicates rejection.

- The state h_r can be used to indicate a "crash," arising from some abnormal situation in which the machine cannot carry out its mission as expected.

- It is possible that a Turing machine computation does not stop, with the machine continuing to make moves forever.

The Definition of a Turing Machine

- **Definition 9.1:** A *Turing machine* (TM) is a 5-tuple $T = (Q, \Sigma, \Gamma, q_0, \delta)$, where

  Q is a finite set of states, assumed not to contain h_a or h_r, the two *halting* states (the same symbols will be used for the halt states of every TM);
  Σ and Γ are finite sets, the *input* and *tape* alphabets, respectively, with $\Sigma \subseteq \Gamma$; Γ is assumed not to contain Δ, the *blank* symbol;
  q_0, the initial state, is an element of Q;
  $\delta : Q \times (\Gamma \cup \{\Delta\}) \rightarrow (Q \cup \{h_a, h_r\}) \times (\Gamma \cup \{\Delta\}) \times \{\text{R, L, S}\}$ is a partial function (possibly undefined at certain points).

- For elements $q \in Q$, $r \in Q \cup \{h_a, h_r\}$, $X, Y \in \Gamma \cup \{\Delta\}$, and $D \in \{\text{R, L, S}\}$, the formula

  $\delta(q, X) = (r, Y, D)$

  means that when T is in state q and the current tape symbol is X, the machine replaces X by Y, changes to state r, and either moves the tape head one square right, moves it one square left, or leaves it stationary, depending on D.

- When r is either h_a or h_r in the formula, we say that T *halts*. Once it has halted, it cannot move further, since δ is not defined at any pair (h_a, X) or (h_r, X).

- If the tape head is currently on the leftmost square, the current state and tape symbol are q and a, respectively, and $\delta(q, a) = (r, b, \text{L})$, the machine leaves the tape head where it is, replaces the a by b, and enters the state h_r instead of r.

- This terminology and these definitions are not completely standard. Sometimes acceptance is defined to mean *halting* (in any halt state); the only other way computation can terminate is by crashing because there is no move possible.

The Definition of a Turing Machine (Continued)

- Normally a TM begins with an input string $x \in \Sigma^*$ near the beginning of its tape and all other tape squares blank, although sometimes a different rule is used.

- Describing the status of a TM involves specifying the state, the contents of the tape (up to the rightmost nonblank symbol), and the position of the tape head.

- A *configuration* of a TM is represented by a pair

  $(q, x\underline{a}y)$

  where $q \in Q$, x and y are strings over $\Gamma \cup \{\Delta\}$ (either or both possibly null), $a \in \Gamma \cup \{\Delta\}$, and the underlined symbol represents the tape head position.

- For a nonnull string w, writing $(q, x\underline{w})$ or $(q, x\underline{w}y)$ will mean that the tape head is positioned at the first symbol of w.

- If $(q, x\underline{a}y)$ represents a configuration, then y may end in one or more blanks; $(q, x\underline{a}y\Delta)$ represents the same configuration. Usually, however, the string y will either be null or have a nonblank last symbol.

- Writing

  $(q, x\underline{a}y) \mid\!-_T (r, z\underline{b}w)$

  means that T passes from the configuration on the left to that on the right in one move, and

  $(q, x\underline{a}y) \mid\!-_T^* (r, z\underline{b}w)$

  means that T passes from the first configuration to the second in zero or more moves.

- The notations $\mid\!-_T$ and $\mid\!-_T^*$ are usually shortened to $\mid\!-$ and $\mid\!-^*$, respectively, as long as there is no ambiguity.

The Definition of a Turing Machine (Continued)

- Input is provided to a TM by having the input string on the tape initially, beginning in square 1, and positioning the tape head on square 0, which is blank. The *initial configuration corresponding to input x* is therefore the configuration

  $(q_0, \underline{\Delta}x)$

- **Definition 9.2:** If $T = (Q, \Sigma, \Gamma, q_0, \delta)$ is a Turing machine, and $x \in \Sigma^*$, x is accepted by T if there exist $y, z \in (\Gamma \cup \{\Delta\})^*$ and $a \in \Gamma \cup \{\Delta\}$ so that

  $(q_0, \underline{\Delta}x) \vdash_T^* (h_a, y\underline{a}z)$

  The *language accepted by T* is the set $L(T)$ of input strings accepted by T.

- A Turing machine can take one of three actions when given an input string x:

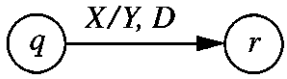
  Accept the string, by entering the state h_a.
  Explicitly reject x, by entering the state h_r.
  Enter an *infinite loop*.

  Although infinite loops are undesirable, they are sometimes inevitable.

- It will often be helpful to draw transition diagrams for Turing machines. The move

  $\delta(q, X) = (r, Y, D)$

  (where D is R, L, or S) will be represented by the following diagram:



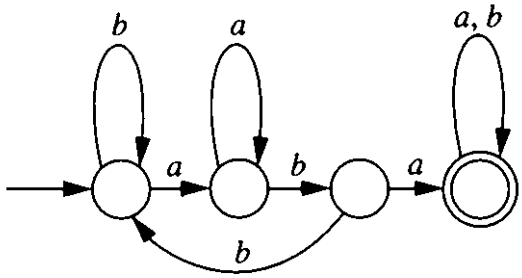
A Turing Machine Accepting {*a*, *b*}*{*aba*}{*a*, *b*}*

- **Example 9.1:** A TM Accepting $\{a, b\}^*\{aba\}\{a, b\}^*$

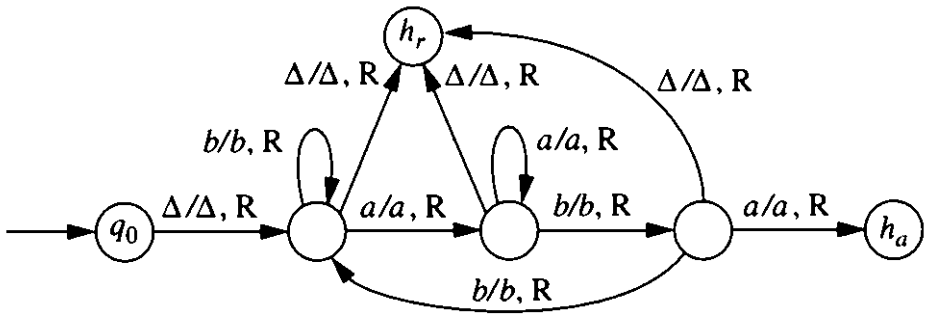
  A Turing machine can easily simulate the behavior of a finite automaton. Consider the language

  $L = \{a, b\}^*\{aba\}\{a, b\}^* = \{x \in \{a, b\}^* \mid x \text{ contains the substring } aba\}$

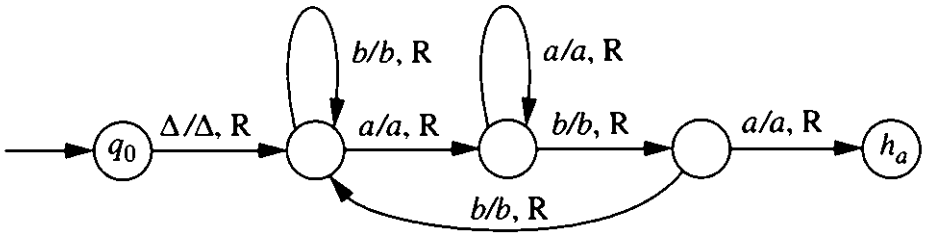
  The following FA recognizes L:



  The transition diagram for a Turing machine that accepts L is similar:



  The transition diagram for the TM becomes even more similar to the one for the FA if transitions to the reject state h_r are omitted:



A Turing Machine Accepting {*a*, *b*}*{*aba*}{*a*, *b*}* (Continued)

It is often convenient to simplify the transition diagram for a TM by omitting transitions to h_r. It is understood that the machine moves to the state h_r for each combination of state and tape symbol for which no move is shown explicitly.

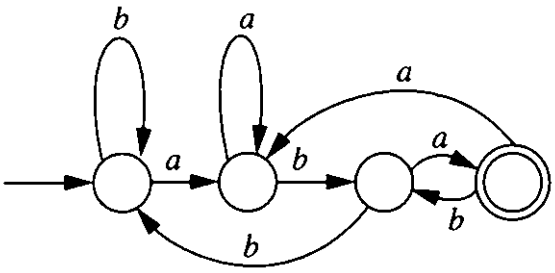
What a TM does with the tape head on a final move to h_r is arbitrary; the general assumption will be that the tape head moves to the right.

As soon as the TM discovers *aba* on the tape, it enters the state h_a and thereby accepts the entire input string, even though it may not have read all of it. Some TMs must read all the input, even if the languages they accept are regular.

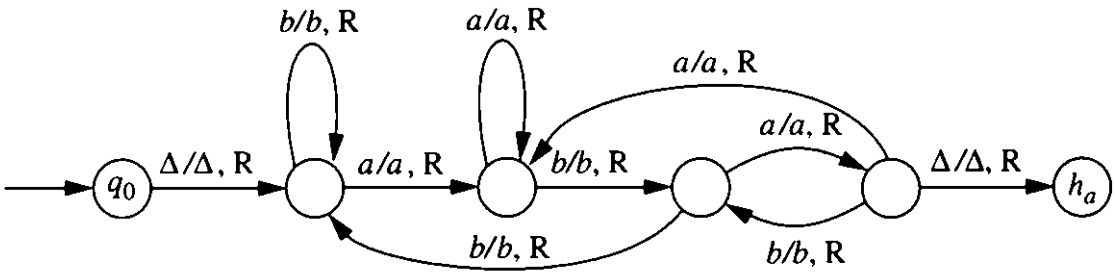
Consider the language

$L_1 = \{x \in \{a, b\}^* \mid x \text{ ends with } aba\}$

which is accepted by the following FA and TM:



(*a*)



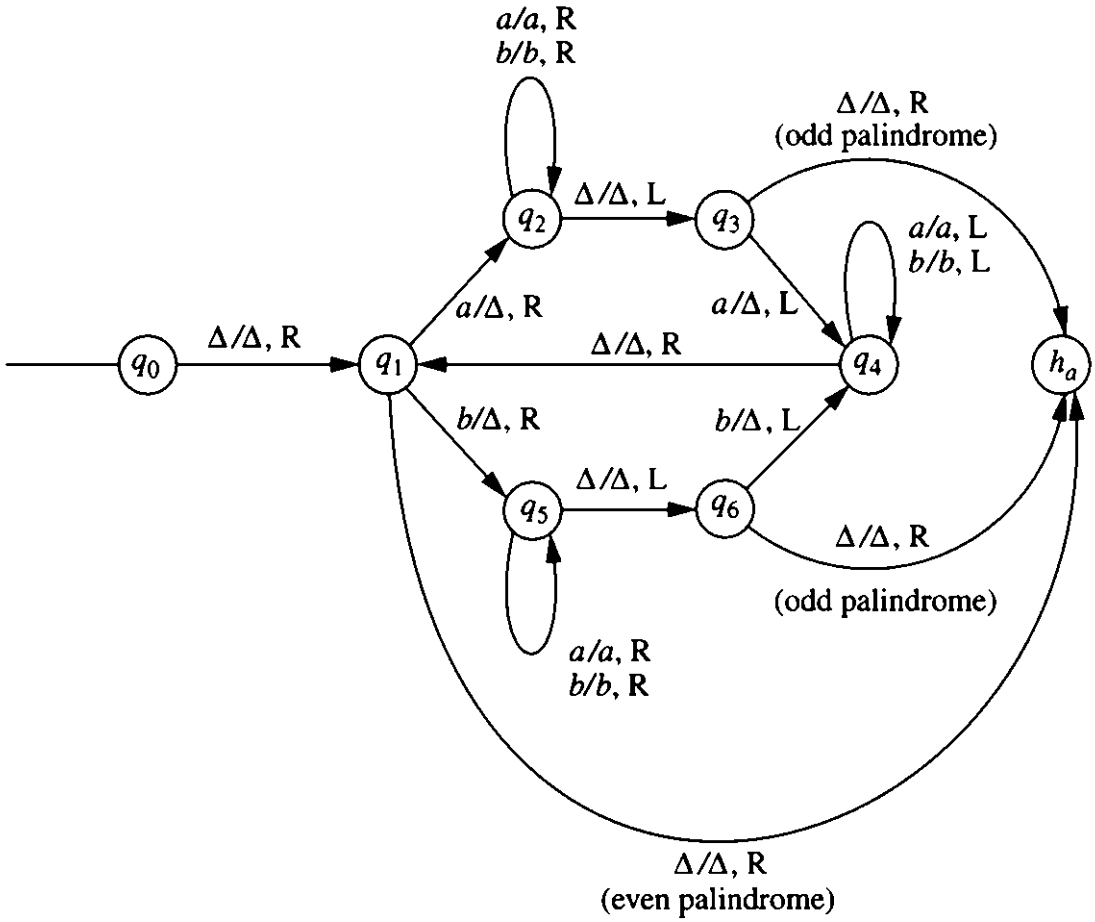
(*b*)

A Turing Machine Accepting *pal*

- **Example 9.2:** A TM Accepting *pal*

  A TM can accept the language *pal* of palindromes over {*a*, *b*} by comparing the symbols on both ends of the input string, erasing them, and then repeating the process.

  A transition diagram for this TM:



  The machine takes the top path each time it finds an *a* at the beginning and attempts to find a matching *a* at the end.

  If it encounters a *b* in state q_3, so that it is unable to match the *a* at the beginning, it enters the reject state h_r. (This transition is not shown explicitly.) Similarly, it rejects from state q_6 if it is unable to match a *b* at the beginning.

A Turing Machine Accepting *pal* (Continued)

A sample computation for a nonpalindrome:

$(q_0, \underline{\Delta}abaa) \vdash (q_1, \Delta\underline{a}baa) \vdash (q_2, \Delta\Delta\underline{b}aa) \vdash^* (q_2, \Delta\Delta baa\underline{\Delta})$
$\vdash (q_3, \Delta\Delta ba\underline{a}) \vdash (q_4, \Delta\Delta b\underline{a}) \vdash^* (q_4, \Delta\underline{\Delta}ba)$
$\vdash (q_1, \Delta\Delta\underline{b}a) \vdash (q_5, \Delta\Delta\Delta\underline{a}) \vdash (q_5, \Delta\Delta\Delta a\underline{\Delta})$
$\vdash (q_6, \Delta\Delta\Delta\underline{a}) \vdash (h_r, \Delta\Delta\Delta a\underline{\Delta})$ (reject)

A sample computation for an even-length palindrome:

$(q_0, \underline{\Delta}aa) \vdash (q_1, \Delta\underline{a}a) \vdash (q_2, \Delta\Delta\underline{a}) \vdash (q_2, \Delta\Delta a\underline{\Delta})$
$\vdash (q_3, \Delta\Delta\underline{a}) \vdash (q_4, \Delta\Delta\underline{\Delta}) \vdash (q_1, \Delta\Delta\underline{\Delta})$
$\vdash (h_a, \Delta\Delta\Delta\underline{\Delta})$ (accept)

A sample computation for an odd-length palindrome:

$(q_0, \underline{\Delta}aba) \vdash (q_1, \Delta\underline{a}ba) \vdash (q_2, \Delta\Delta\underline{b}a) \vdash^* (q_2, \Delta\Delta ba\underline{\Delta})$
$\vdash (q_3, \Delta\Delta b\underline{a}) \vdash (q_4, \Delta\Delta\underline{b}) \vdash (q_4, \Delta\underline{\Delta}b)$
$\vdash (q_1, \Delta\Delta\underline{b}) \vdash (q_5, \Delta\Delta\Delta\underline{\Delta}) \vdash (q_6, \Delta\Delta\underline{\Delta})$
$\vdash (h_a, \Delta\Delta\Delta\underline{\Delta})$ (accept)

A Turing Machine Accepting $\{ss \mid s \in \{a, b\}^*\}$

- **Example 9.3:** A TM Accepting $\{ss \mid s \in \{a, b\}^*\}$

  Consider the following language, which was proved in Chapter 8 not to be context-free:

  $L = \{ss \mid s \in \{a, b\}^*\}$

  The idea behind the TM will be to separate the processing into two parts:

  Finding the middle of the string.
  Comparing the two halves.

  The TM can accomplish the first task by working in from both ends simultaneously, changing symbols to their uppercase versions.

  Once the TM arrives at the middle—which will happen only if the string is of even length—it changes the symbols in the first half back to their original form.

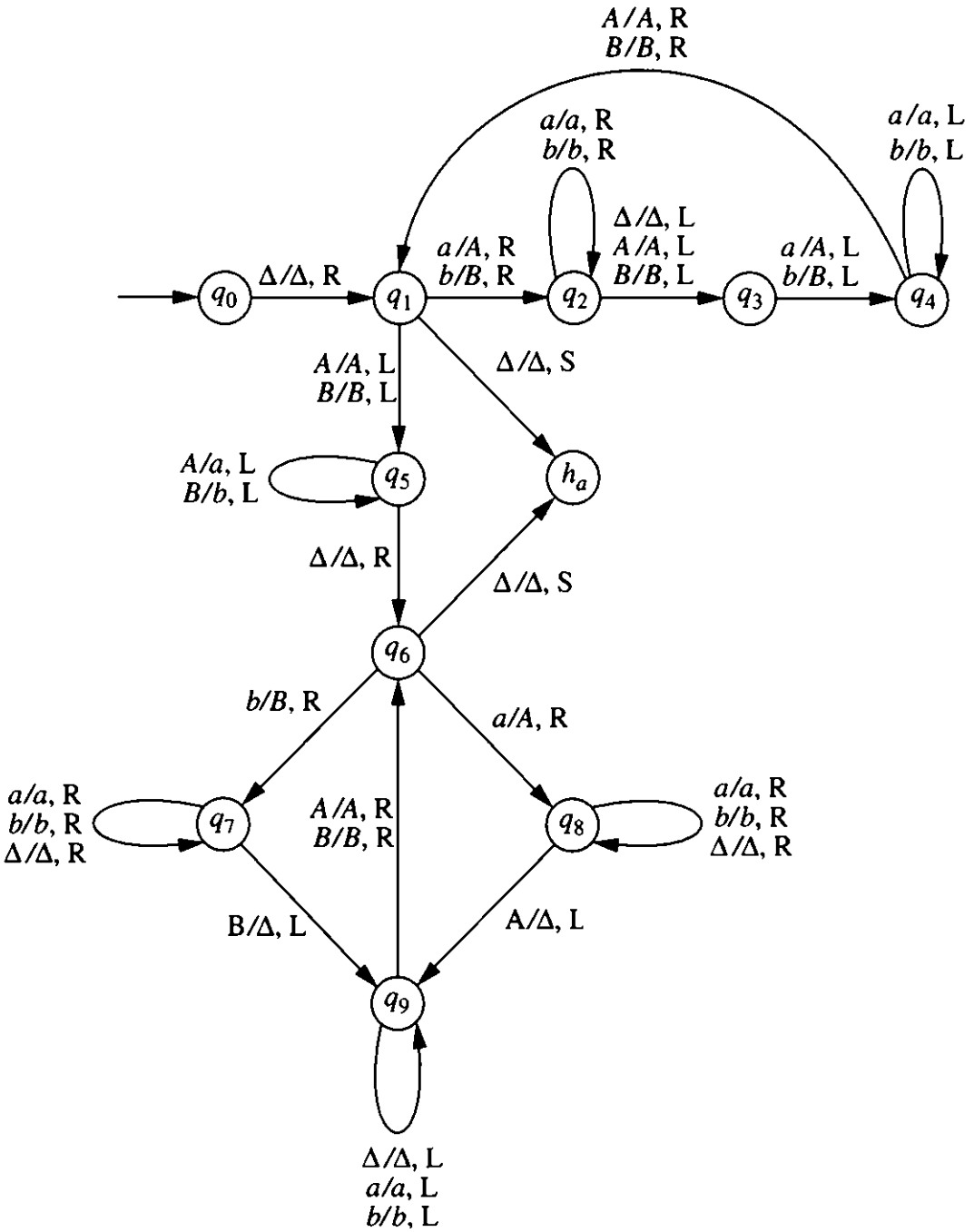
  Next, the TM compares each lowercase symbol in the first half to the corresponding uppercase symbol in the second. The TM tracks its progress by changing lowercase symbols to uppercase and erasing matching uppercase symbols.

  There are two ways that an input string can be rejected:

  If its length is odd, the TM will discover this in the first phase.
  If a symbol in the first half fails to match the corresponding symbol in the second half, the TM will reject the string during the second phase.

A Turing Machine Accepting $\{ss \mid s \in \{a, b\}^*\}$ (Continued)

A transition diagram for the TM just described:



A Turing Machine Accepting $\{ss \mid s \in \{a, b\}^*\}$ (Continued)

A sample rejecting computation:

$(q_0, \underline{\Delta}aba) \vdash (q_1, \Delta\underline{a}ba) \vdash (q_2, \Delta A\underline{b}a) \vdash^* (q_2, \Delta Aba\underline{\Delta})$
$\vdash (q_3, \Delta Ab\underline{a}) \vdash (q_4, \Delta A\underline{b}A) \vdash (q_4, \Delta\underline{A}bA)$
$\vdash (q_1, \Delta A\underline{b}a) \vdash (q_2, \Delta AB\underline{A}) \vdash (q_3, \Delta A\underline{B}A)$
$\vdash (h_r, \Delta AB\underline{A})$ (reject)

A sample computation that rejects in a different way:

$(q_0, \underline{\Delta}abaa) \vdash (q_1, \Delta\underline{a}baa) \vdash (q_2, \Delta A\underline{b}aa) \vdash^* (q_2, \Delta Abaa\underline{\Delta})$
$\vdash (q_3, \Delta Aba\underline{a}) \vdash (q_4, \Delta A\underline{b}aA) \vdash^* (q_4, \Delta\underline{A}baA)$
$\vdash (q_1, \Delta A\underline{b}aA) \vdash (q_2, \Delta AB\underline{a}A) \vdash (q_2, \Delta ABa\underline{A})$
$\vdash (q_3, \Delta AB\underline{a}A) \vdash (q_4, \Delta A\underline{B}AA) \vdash (q_1, \Delta AB\underline{A}A)$
$\vdash (q_5, \Delta A\underline{B}AA) \vdash (q_5, \Delta\underline{A}bAA) \vdash (q_5, \underline{\Delta}abAA)$
(first phase completed)
$\vdash (q_6, \Delta\underline{a}bAA) \vdash (q_8, \Delta A\underline{b}AA) \vdash (q_8, \Delta Ab\underline{A}A)$
$\vdash (q_9, \Delta A\underline{b}\Delta A) \vdash (q_9, \Delta\underline{A}b\Delta A) \vdash (q_6, \Delta A\underline{b}\Delta A)$
$\vdash (q_7, \Delta AB\underline{\Delta}A) \vdash (q_7, \Delta AB\Delta\underline{A}) \vdash (h_r, \Delta AB\Delta A\underline{\Delta})$ (reject)

A sample accepting computation:

$(q_0, \underline{\Delta}abab) \vdash^* \ldots$
(same as previous case, up to 3rd-from-last move)
$\vdash (q_6, \Delta A\underline{b}\Delta B) \vdash (q_7, \Delta AB\underline{\Delta}B) \vdash (q_7, \Delta AB\Delta\underline{B})$
$\vdash (q_9, \Delta AB\underline{\Delta}) \vdash (q_9, \Delta A\underline{B}) \vdash (q_6, \Delta AB\underline{\Delta})$
$\vdash (h_a, \Delta AB\underline{\Delta})$ (accept)

9.2 Computing a Partial Function with a Turing Machine

- A Turing machine T with input alphabet Σ can compute a function f whose domain is a subset of Σ^*.

- For any string x in the domain of f, whenever T starts in the initial configuration corresponding to x, T will eventually halt with the output string $f(x)$ on the tape. Moreover, T should not accept x if x is not in the domain of f.

- A *partial* function f on Σ^* may be undefined at certain points. If it happens that f is defined everywhere on Σ^*, f is said to be a *total* function.

- A TM can handle a function of several variables as well. If the input is to represent the k-tuple $(x_1, x_2, \ldots, x_k) \in (\Sigma^*)^k$, the initial tape will contain all k strings, separated by blanks.

- **Definition 9.3:** Let $T = (Q, \Sigma, \Gamma, q_0, \delta)$ be a Turing machine, and let f be a partial function on Σ^* with values in Γ^*. T is said to compute f if for every $x \in \Sigma^*$ at which f is defined,

  $(q_0, \underline{\Delta}x) \vdash_T^* (h_a, \underline{\Delta}f(x))$

  and no other $x \in \Sigma^*$ is accepted by T.

  If f is a partial function on $(\Sigma^*)^k$ with values in Γ^*, T computes f if for every k-tuple $(x_1, x_2, \ldots, x_k)$ at which f is defined,

  $(q_0, \underline{\Delta}x_1\Delta x_2\Delta\ldots\Delta x_k) \vdash_T^* (h_a, \underline{\Delta}f(x_1, x_2, \ldots, x_k))$

  and no other input that is a k-tuple of strings is accepted by T.

  For two alphabets Σ_1 and Σ_2, and a positive integer k, a partial function $f : (\Sigma_1^*)^k \rightarrow \Sigma_2^*$ is *Turing-computable*, or simply *computable*, if there is a Turing machine computing f.

Computing a Partial Function with a Turing Machine (Continued)

- It is not quite correct to say that a TM computes only one function. However, for any specified k, and any $C \subseteq \Gamma^*$, a given TM computes at most one function of k variables having codomain C.

- Numerical functions of numerical arguments can also be computed by Turing machines. Natural numbers can be represented in "unary," with the integer n represented by the string $1^n = 11\ldots1$.

- **Definition 9.4:** Let $T = (Q, \{1\}, \Gamma, q_0, \delta)$ be a Turing machine. If f is a partial function from $\mathcal{N}$, the set of natural numbers, to itself, T computes f if for every n at which f is defined,

  $(q_0, \underline{\Delta}1^n) \vdash_T^* (h_a, \underline{\Delta}1^{f(n)})$

  and for every other natural number n, T fails to accept the input 1^n.

  Similarly, if f is a partial function from $\mathcal{N}^k$ to $\mathcal{N}$, T computes f if for every k-tuple $(n_1, n_2, \ldots, n_k)$ at which f is defined,

  $(q_0, \underline{\Delta}1^{n_1}\Delta 1^{n_2}\Delta\ldots\Delta 1^{n_k}) \vdash_T^* (h_a, \underline{\Delta}1^{f(n_1,n_2,\ldots,n_k)})$

  and T fails to accept if the input is any k-tuple at which f is not defined.

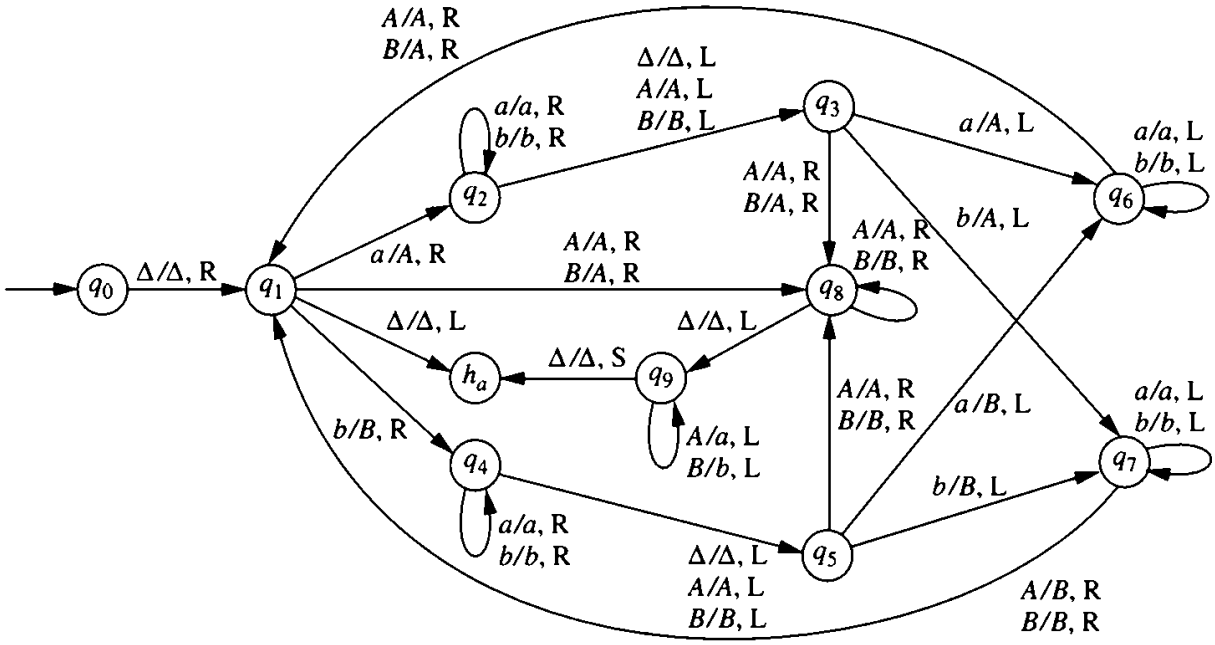
Examples of Computing a Partial Function

- **Example 9.4:** Reversing a String

  Consider the reverse function

  $rev : \{a, b\}^* \rightarrow \{a, b\}^*$

  The following TM reverses the input string "in place," moving from the ends toward the middle, at each step swapping a symbol in the first half with the matching one in the second half:



  In order to keep track of the progress made so far, symbols will also be changed to uppercase.

  When the swaps have been completed, the tape head moves to the end of the string and makes one final pass back to the left, changing all the uppercase symbols back to lowercase.

Examples of Computing a Partial Function (Continued)

A sample computation for an odd-length string:

$(q_0, \underline{\Delta}abb)$ $\vdash (q_1, \Delta\underline{a}bb)$ $\vdash (q_2, \Delta A\underline{b}b)$ $\vdash (q_2, \Delta Ab\underline{b})$
$\vdash (q_2, \Delta Abb\underline{\Delta})$ $\vdash (q_3, \Delta Ab\underline{b})$ $\vdash (q_7, \Delta A\underline{b}A)$
$\vdash (q_7, \Delta\underline{A}bA)$ $\vdash (q_1, \Delta B\underline{b}A)$ $\vdash (q_4, \Delta BB\underline{A})$
$\vdash (q_5, \Delta B\underline{B}A)$ $\vdash (q_8, \Delta BB\underline{A})$ $\vdash (q_8, \Delta BBA\underline{\Delta})$
$\vdash (q_9, \Delta BB\underline{A})$ $\vdash (q_9, \Delta B\underline{B}a)$ $\vdash (q_9, \Delta\underline{B}ba)$
$\vdash (q_9, \underline{\Delta}bba)$ $\vdash (h_a, \underline{\Delta}bba)$

A sample computation for an even-length string:

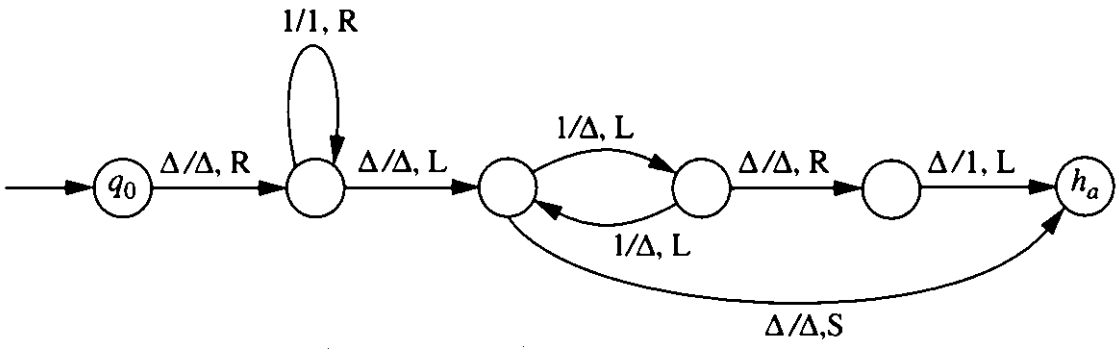
$(q_0, \underline{\Delta}baba)$ $\vdash (q_1, \Delta\underline{b}aba)$ $\vdash (q_4, \Delta B\underline{a}ba)$ $\vdash (q_4, \Delta Ba\underline{b}a)$
$\vdash (q_4, \Delta Bab\underline{a})$ $\vdash (q_4, \Delta Baba\underline{\Delta})$ $\vdash (q_5, \Delta Bab\underline{a})$
$\vdash (q_6, \Delta Ba\underline{b}B)$ $\vdash (q_6, \Delta B\underline{a}bB)$ $\vdash (q_6, \Delta\underline{B}abB)$
$\vdash (q_1, \Delta A\underline{a}bB)$ $\vdash (q_2, \Delta AA\underline{b}B)$ $\vdash (q_2, \Delta AAb\underline{B})$
$\vdash (q_3, \Delta AA\underline{b}B)$ $\vdash (q_7, \Delta A\underline{A}AB)$ $\vdash (q_1, \Delta AB\underline{A}B)$
$\vdash (q_8, \Delta ABA\underline{B})$ $\vdash (q_8, \Delta ABAB\underline{\Delta})$ $\vdash (q_9, \Delta ABA\underline{B})$
$\vdash^* (q_9, \underline{\Delta}abab)$ $\vdash (h_a, \underline{\Delta}abab)$

Examples of Computing a Partial Function (Continued)

- **Example 9.5:** *n* mod 2

  The value of *n* mod 2 can be computed by moving to the end of the input string, making a pass from right to left in which the 1's are counted and simultaneously erased, and either leaving a single 1 (if *n* was odd) or leaving nothing.

  A TM that performs this computation:



Examples of Computing a Partial Function (Continued)

- **Example 9.6:** The Characteristic Function of a Set

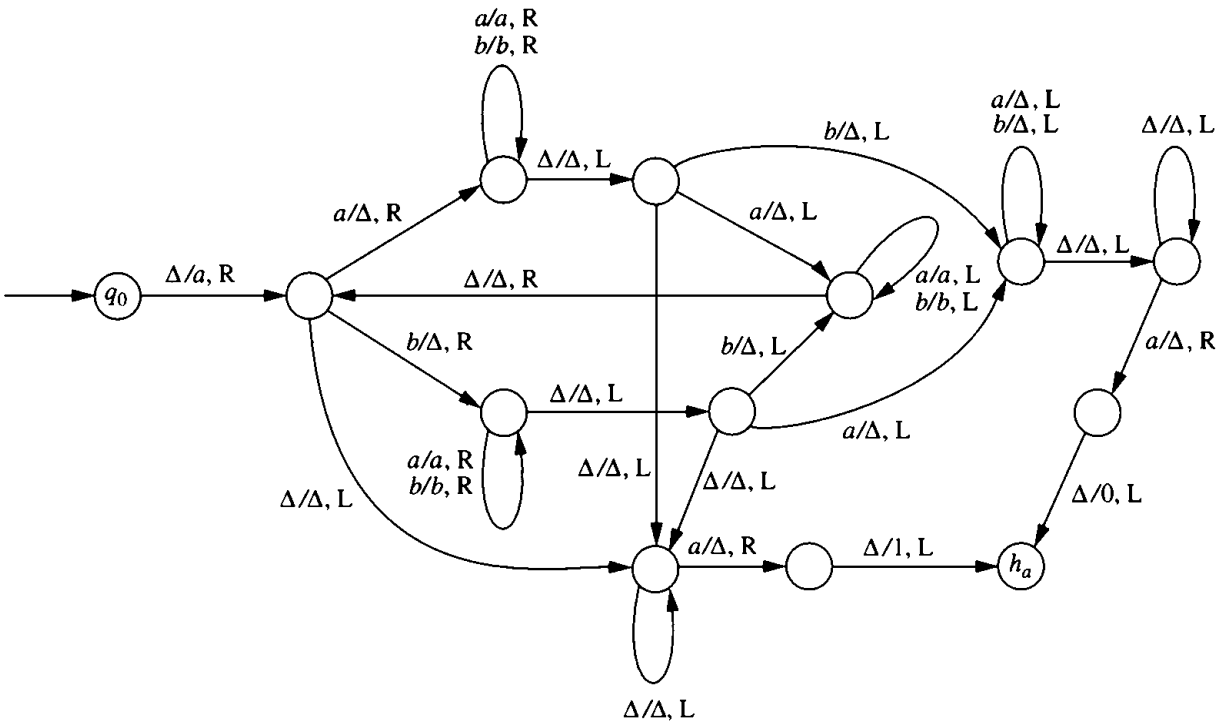
  For any language $L \subseteq \Sigma^*$, the *characteristic function* of L is the function $\chi_L : \Sigma^* \rightarrow \{0, 1\}$ defined by the formula

$$\chi_L(x) = \begin{cases} 1 & \text{if } x \in L \\ 0 & \text{otherwise} \end{cases}$$

  A TM that computes the function χ_L accepts every input. It distinguishes between strings in L and strings not in L by ending up in the configuration $(h_a, \underline{\Delta}1)$ in one case and $(h_a, \underline{\Delta}0)$ in the other.

  If T is a TM that computes χ_L, it is easy to obtain one that accepts L: just modify T so that when it leaves output 0, it enters the state h_r instead of h_a.

  Sometimes it is possible to go the other way. The following TM, which computes χ_L for the language L of palindromes over $\{a, b\}$, was constructed from an earlier TM that accepts L:



Examples of Computing a Partial Function (Continued)

The new TM was obtained from the previous one by identifying the places in the transition diagram where the TM might reject, and modifying the TM so that instead of entering the state h_r it ends up in state h_a with output 0.

For any language L accepted by a TM T that halts on every input string, another TM can be constructed from T that computes χ_L, although the construction may be more complicated than in this example.

However, a TM can accept a language L and still leave the question of whether $x \in L$ unanswered for some strings x, by looping forever on those inputs.

9.3 Combining Turing Machines

- Much of the work that goes on during a TM computation consists of routine, repetitive tasks. Creating a complicated TM can be made easier if it is built from simpler, reusable components that perform these tasks.

- A composite Turing machine will first execute one TM and then another. If T_1 and T_2 are TMs with disjoint sets of nonhalting states, T_1T_2 denotes this composite TM. The set of states is the union of the two sets.

- T_1T_2 begins in the initial state of T_1, and executes the moves of T_1 up to the point where T_1 would halt. Any move that would cause T_1 to halt in the accepting state causes T_1T_2 to move instead to the initial state of T_2.

- From this point on, the moves of T_1T_2 are the moves of T_2. If either T_1 or T_2 would reject during this process, T_1T_2 does also, and T_1T_2 accepts precisely if and when T_2 accepts.

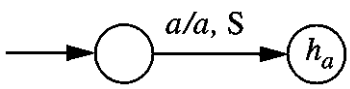
- The following notation can be used to represent the composite machine formed from T_1 and T_2:

  $T_1 \rightarrow T_2$

- The composition can be made conditional, depending on the current tape symbol when T_1 halts. The notation

  $T_1 \xrightarrow{a} T_2$

  stands for the composite machine $T_1T'T_2$, where T' is described by the following diagram:



  The composite machine executes T_1 (rejecting if T_1 rejects, and looping if T_1 loops); if and when T_1 accepts, it executes T_2 if the current tape symbol is a and rejects otherwise.

Combining Turing Machines (Continued)

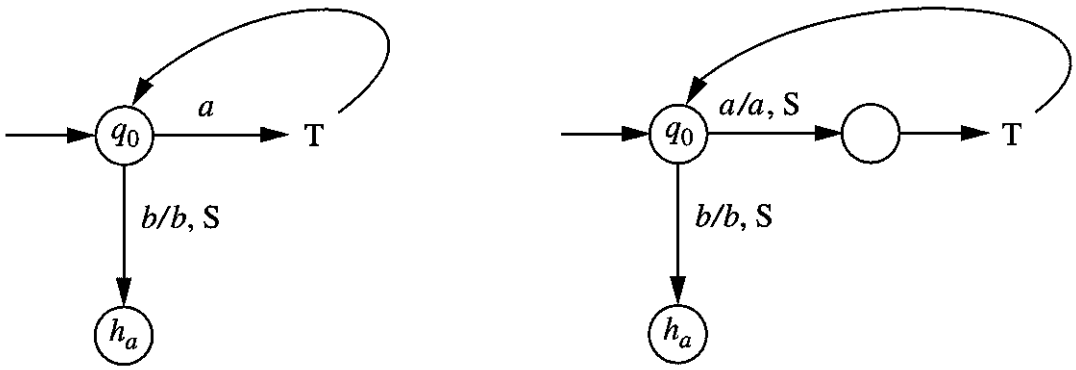
- Some TMs that would only halt in the rejecting state when viewed as self-contained machines (e.g., the TM that executes the algorithm "Move the tape head one square to the left") can be used successfully in combination with others.

- On the other hand, if a TM halts normally when run independently, then it will halt normally when it is used as a component of a larger machine, provided that the tape has been prepared properly before its use.

- For example, a TM T expecting to find an input string z needs to begin in a configuration of the form $(q, y\underline{\Delta}z)$. As long as T halts normally when processing input z in the ordinary way, the processing of z in this way does not depend on y.

- In order to be able to describe composite TMs without having to describe every primitive operation as a separate TM, it is sometimes useful to use a mixed notation in which some but not all of the states of a TM are shown.

- Consider the diagram on the left, which is an abbreviated version of the one on the right:



The diagram means: If the current tape symbol is a, execute the TM T (and then repeat); if it is b, halt in the accepting state; and if it is anything else, reject.

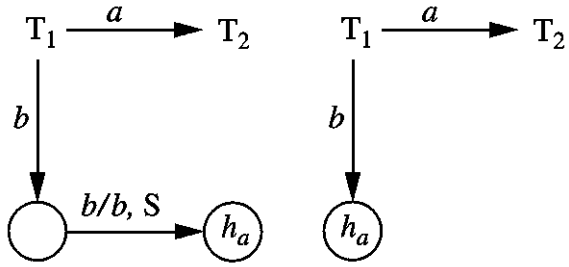
Combining Turing Machines (Continued)

- Although giving a completely precise definition of an arbitrary combination of TMs would be complicated, it is usually clear in specific examples what is involved.

- There is one possible source of confusion, however. Consider a TM T of the form $T_1 \xrightarrow{a} T_2$. If T_1 halts normally scanning some symbol not specified explicitly (i.e., other than a), T rejects. But if T_2 halts normally, T does also—even though *no* tape symbols are specified explicitly.

- One way to avoid this seeming inconsistency would be to say that if T_1 halts normally scanning a symbol other than a, then T halts normally, but then T would then not be equivalent to the composition $T_1T'T_2$.

- A better way is to require that, if at the end of one sub-TM's operation at least one way is specified for the composite TM to continue, then any option that allows accepting at that point must be shown explicitly:



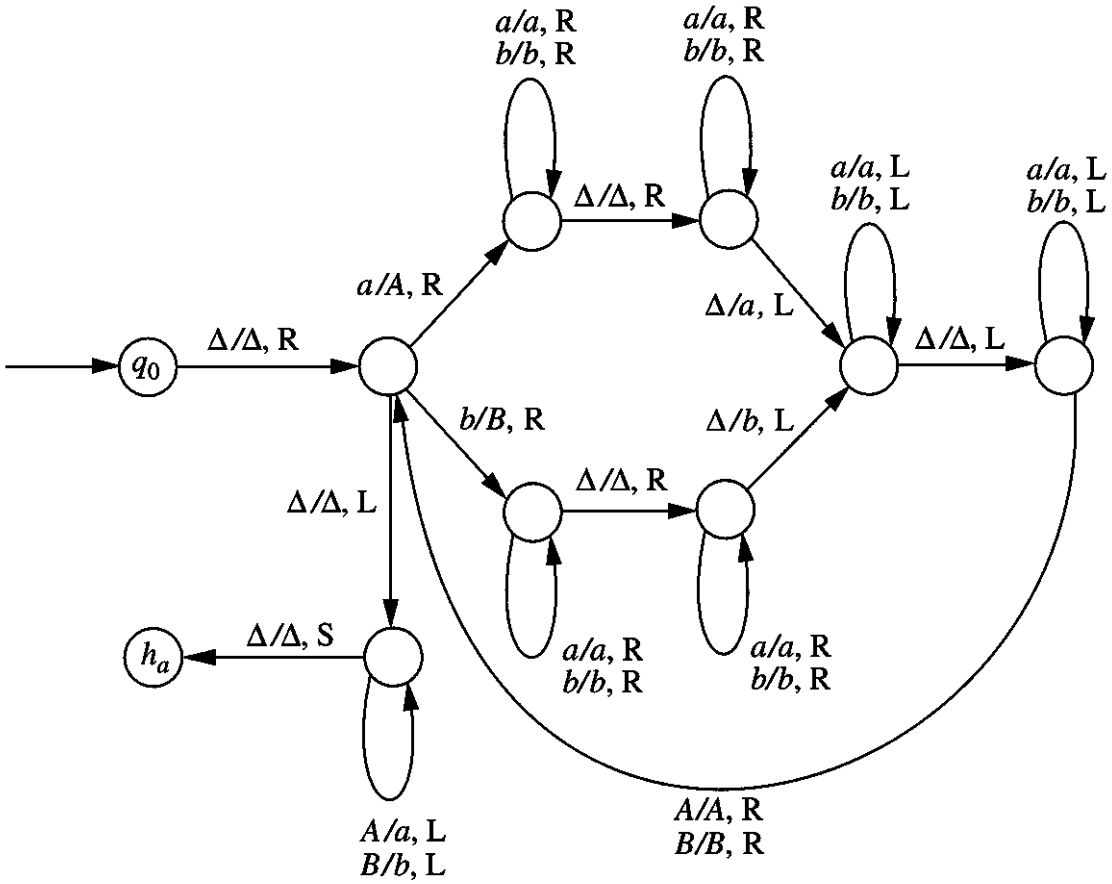
The second figure is a shortened form of the first.

- Some basic TM building blocks are very simple and do not need to be described in detail. Others, such as copying a string, are more involved.

Combining Turing Machines (Continued)

- **Example 9.7:** Copying a String

  The following TM creates a copy of a string over $\{a, b\}$:



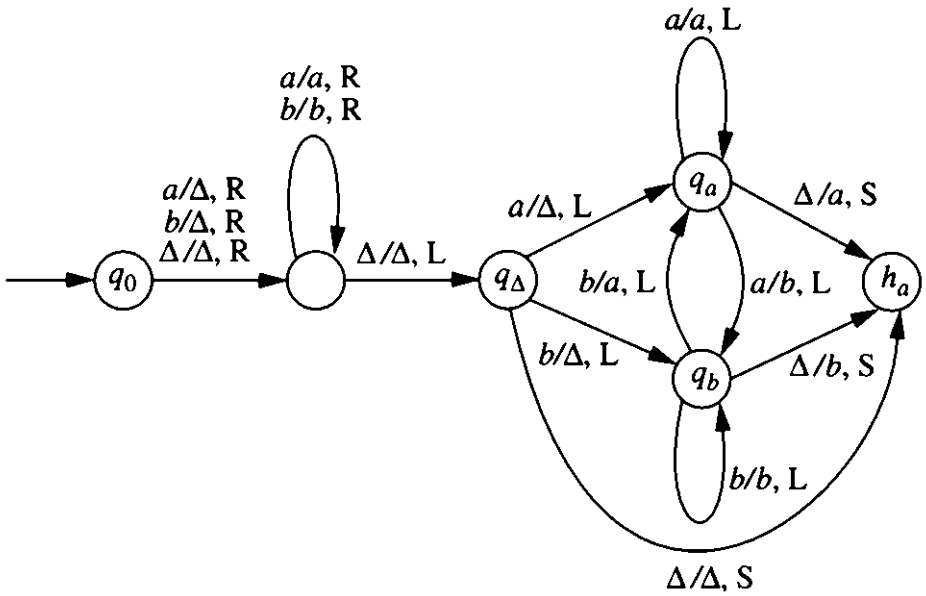
If the initial configuration is $(q_0, \underline{\Delta}x)$, where x is a string of nonblank symbols, then the final configuration will be $(h_a, \underline{\Delta}x\Delta x)$.

The TM copies input symbols one by one, keeping track of its progress by changing the symbols it has copied to uppercase. When the copying is complete, the uppercase symbols are changed back to their original form.

Combining Turing Machines (Continued)

- **Example 9.8:** Deleting a Symbol

  The following TM deletes a symbol from a string over $\{a, b\}$:



  The TM changes the tape contents from $y\underline{a}z$ to $y\underline{z}$, where $y \in (\Sigma \cup \{\Delta\})^*$, $a \in \Sigma \cup \{\Delta\}$, and $z \in \Sigma^*$.

  The TM starts by replacing the symbol to be deleted by a blank. It then moves to the right end of the string z and makes a single pass from right to left, moving symbols one square to the left as it goes, until it hits the blank.

  The states q_a and q_b allow the machine to remember a symbol between the time it erases it and the time it writes it in the next square to the left.

  Inserting a symbol a, or changing the tape contents from $y\underline{z}$ to $y\underline{a}z$, would be done virtually the same way, except that the single pass would go from left to right, and the move that starts it off would write a instead of Δ.

Combining Turing Machines (Continued)

- The *Delete* machine transforms $y\underline{a}z$ to $y\underline{z}$. What if it is called when the tape contents are not $y\underline{a}z$, but $y\underline{a}z\Delta w$, where w is some arbitrary string of symbols?

- At first it might seem that the TM ought to be designed so as to finish with $y\underline{z}\Delta w$ on the tape. However, this is unreasonable, because a Turing machine has no way of finding the rightmost nonblank symbol on a tape.

- In general, a Turing machine is designed to to start in a certain configuration. If it is started in some different configuration, the machine may behave unpredictably, perhaps halting in h_r or looping forever.

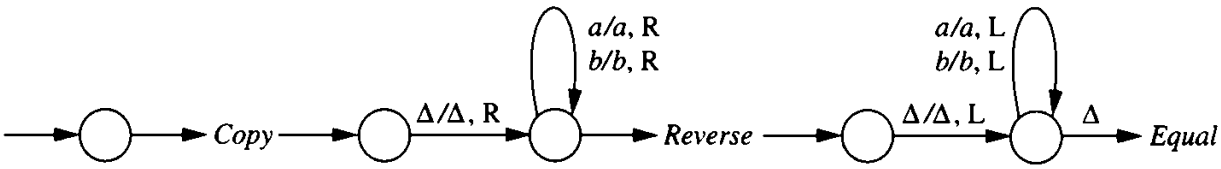
Combining Turing Machines (Continued)

- **Example 9.9:** Another Way of Accepting *pal*

  Suppose that *Copy* is the TM in Example 9.7 and *Reverse* is the one in Example 9.4.

  Let *Equal* be a TM that works as follows: When started with tape $\underline{\Delta}x\Delta y$, where x, $y \in \{a, b\}^*$, *Equal* accepts if and only if $x = y$.

  The following composite TM accepts the language of palindromes over $\{a, b\}$ by comparing the input string to its reverse and accepting if and only if the two are equal:



9.4 Variations of Turing Machines: Multitape TMs

- The basic Turing machine model can be enhanced in several natural ways, such as by adding additional tapes. Using an enhanced TM can make it easier to describe the implementation of an algorithm.

- As it turns out, enhancing the TM model does not give the resulting machine any additional power. For each enhanced TM, there is an ordinary TM that accepts the same input strings and produces the same output.

- The TM model can also be restricted in certain ways without a reduction in power:

  Requiring that in each move the tape head move either to the right or to the left.
  Requiring that a move either write a tape symbol or move the tape head but not both.

Variations of Turing Machines: Multitape TMs (Continued)

- The TM model could be enhanced by allowing the tape to be two-dimensional, or by removing the left end and making the tape potentially infinite in both directions.

- Another enhancement to the TM model would be to add extra tapes. The tapes could be accessed by a single tape head or by multiple heads, one per tape. The second option will now be considered.

- An n-tape Turing machine can be specified by a 5-tuple $T = (Q, \Sigma, \Gamma, q_0, \delta)$. It will make a move on the basis of its current state and the n-tuple of tape symbols currently being examined.

- Since the tape heads move independently, the transition function is the following partial function:

  $\delta : Q \times (\Gamma \cup \{\Delta\})^n \rightarrow (Q \cup \{h_a, h_r\}) \times (\Gamma \cup \{\Delta\})^n \times \{\mathrm{R}, \mathrm{L}, \mathrm{S}\}^n$

- A configuration of an n-tape TM is specified by an $(n + 1)$-tuple of the form

  $(q, x_1\underline{a}_1y_1, x_2\underline{a}_2y_2, ..., x_n\underline{a}_ny_n)$

- The initial configuration corresponding to input string x will be

  $(q_0, \underline{\Delta}x, \underline{\Delta}, \ldots, \underline{\Delta})$

  The first tape is the one used for the input.

- The output of an n-tape TM is the final contents of tape 1. Tapes 2 through n are used only for auxiliary working space.

- An n-tape TM computes a function f if, whenever it begins with an input string x in (or representing an element in) the domain of f, it halts in some configuration $(h_a, \underline{\Delta}f(x), \ldots)$, where the contents of tapes 2 through n are arbitrary, and otherwise it fails to accept.

Variations of Turing Machines: Multitape TMs (Continued)

- It is obvious that for any $n \geq 2$, n-tape TMs are at least as powerful as ordinary 1-tape TMs. To simulate an ordinary TM, a TM with n tapes simply acts as if tape 1 were its only one. Theorem 9.1 shows that the converse is also true.

- **Theorem 9.1.** Let $n \geq 2$ and let $T_1 = (Q_1, \Sigma, \Gamma_1, q_1, \delta_1)$ be an n-tape Turing machine Then there is a one-tape TM $T_2 = (Q_2, \Sigma, \Gamma_2, q_2, \delta_2)$, with $\Gamma_1 \subseteq \Gamma_2$, satisfying the following two conditions.

  1. $L(T_2) = L(T_1)$; that is, for any $x \in \Sigma^*$, T_2 accepts input x if and only if T_1 accepts input x.
  2. For any $x \in \Sigma^*$, if

     $(q_1, \underline{\Delta}x, \underline{\Delta}, \ldots, \underline{\Delta}) \vdash_{T_1}^* (h_a, y\underline{a}z, y_2\underline{a}_2z_2, \ldots, y_n\underline{a}_nz_n)$

     (for some $a, a_i \in \Gamma_1 \cup \{\Delta\}$ and $y, z, y_i, z_i \in (\Gamma_1 \cup \{\Delta\})^*$), then

     $(q_2, \underline{\Delta}x) \vdash_{T_2}^* (h_a, y\underline{a}z)$

  In other words, if T_1 accepts input x, then T_2 accepts input x and produces the same output as T_1.

  Proof: The proof is for $n = 2$. It will be easy to extend to the general case.

  To simulate a machine with two tapes, the one-tape TM T_2 will use a more complicated tape alphabet, so that the symbol on one square can hold the contents of two squares of the original TM.

  T_2's tape can be thought of as having two "tracks," corresponding to the two tapes of T_1. The two tracks do not exist initially but will be created gradually as T_2 moves its tape head farther and farther to the right.

Variations of Turing Machines: Multitape TMs (Continued)

The tape alphabet Γ_2 includes the following kinds of symbols:

1. Ordinary symbols in $\Gamma = \Gamma_1 \cup \{\Delta\}$.
2. Elements of $\Gamma \times \Gamma$. A symbol (X, Y) of this type in square i is thought of as representing X in square i of the first tape and Y in square i of the second.
3. Elements of $(\Gamma \times \Gamma') \cup (\Gamma' \times \Gamma) \cup (\Gamma' \times \Gamma')$, where Γ' contains the same symbols as Γ, marked with $'$. During the simulation, there is one primed symbol on each track to designate the location of the tape head on the corresponding tape of T_1.
4. An extra symbol, #, which is inserted initially into the leftmost square, making it easy to find the beginning of the tape whenever we want.

The next step of T_2, after inserting the symbol #, is to change the blank that is now in square 1 to the symbol (Δ', Δ'), signifying that the "head" on each track is now on square 1, and then move the actual head back to square 0.

From now on, the two tracks will extend as far as T_2 has ever moved its tape head to the right; whenever it advances one square farther, it converts from the old single symbol to the new double symbol.

At this point the actual simulation starts. T_1 makes moves of the form

$\delta(p, a_1, a_2) = (q, b_1, b_2, D_1, D_2)$

where a_1 and a_2 are the symbols in the current squares of the respective tapes.

T_2 will determine T_1's next move by locating the primed symbols on the two tracks of its tape. It carries out the move by making the appropriate changes to both its tracks, including the creation of new primed symbols to reflect any changes in the positions of T_1's tape heads.

T_2 will use its states to "remember" the current state p of T_1.

Variations of Turing Machines: Multitape TMs (Continued)

T_2 will execute the following steps as it simulates a single move of T_1, starting in the leftmost square of its tape.

1. Move the head to the right until a pair of the form (a_1', c) is found (c may or may not be a primed symbol), and remember a_1. Move back to the # at the beginning.
2. Locate the "head" on the second track the same way, by finding the pair of the form (d, a_2') (d may or may not be a primed symbol).
3. If the move of T_1 that has now been determined is (q, b_1, b_2, D_1, D_2) as above, remember the state q, change the pair (d, a_2') to (d, b_2), and move the tape head in direction D_2.
4. If the current square contains #, reject, since T_1 would have crashed by trying to move the tape head off tape 2. If not, and if the new square does not contain a pair of symbols, convert the symbol a there to the pair (a, Δ'); if the new square does contain a pair, say (a, b), convert it to (a, b'). Move the tape head back to the beginning.
5. Locate the pair (a_1', c) again, as in step 1. Change it to (b_1, c) and move the tape head in direction D_1.
6. As in step 4, either reject, or change the single symbol a to the pair (a', Δ), or change the pair (a, b) to the pair (a', b). Return to the beginning of the tape.

Variations of Turing Machines: Multitape TMs (Continued)

Consider the following move of T_1:

$\delta(p, 0, 1) = (q, \Delta, 0, L, R)$

The diagram below illustrates how T_2 would simulate this move:

Original configuration:

#	Δ 0	0′ 1	Δ 0	0 1′	1	Δ	

After step 4:

#	Δ 0	0′ 1	Δ 0	0 0	1 Δ′	Δ		…

After step 6:

#	Δ′ 0	Δ 1	Δ 0	0 0	1 Δ′	Δ	

If T_1 ever enters the halt state h_a during the simulation, T_2 must carry out the following additional steps in order to finish up in the correct configuration:

7. Make a pass through the tape, converting each pair (*a, b*) to the single symbol *a*. (One of these symbols *a* will be a primed symbol.)
8. Delete the #, so that the remaining symbols begin in square 0.
9. Move the tape head to the primed symbol, change it to the corresponding unprimed symbol, and halt in state h_a with the head in that position.

- **Corollary 9.1.** Any language that is accepted by an *n*-tape TM can be accepted by an ordinary TM, and any function that is computed by an *n*-tape TM can be computed by an ordinary TM.

  Proof: The proof is immediate from Theorem 9.1.

9.5 Nondeterministic Turing Machines

- Nondeterminism is convenient but not essential in the case of FAs, whereas the language *pal* is an example of a context-free language that cannot be accepted by any deterministic PDA.

- Turing machines have enough computing power that nondeterminism fails to add any more.

- A *nondeterministic Turing machine* (NTM) $T = (Q, \Sigma, \Gamma, q_0, \delta)$ is defined exactly the same way as an ordinary TM, except that values of δ are subsets, rather than single elements, of the set $(Q \cup \{h_a, h_r\}) \times (\Gamma \cup \{\Delta\}) \times \{\text{R, L, S}\}$. δ is no longer a *partial* function, because now $\delta(q, a)$ is allowed to take the value $\varnothing$.

- The notation for a TM configuration is also unchanged. To say that

  $(p, x\underline{a}y) \vdash_T (q, w\underline{b}z)$

  now means that beginning in the first configuration, there is at least one move that will produce the second. Similarly,

  $(p, x\underline{a}y) \vdash_T^* (q, w\underline{b}z)$

  means that there is at least one sequence of zero or more moves that takes T from the first configuration to the second.

- As before, a string $x \in \Sigma^*$ is accepted by T if for some $a \in \Gamma \cup \{\Delta\}$ and some $y, z \in (\Gamma \cup \{\Delta\})^*$,

  $(q_0, \underline{\Delta}x) \vdash_T^* (h_a, y\underline{a}z)$

- It is hard to define the concept of "output" for NTMs, so they will be considered only as language acceptors.

Nondeterministic Turing Machines (Continued)

- Because every TM can be interpreted as a nondeterministic TM, it is obvious that a language accepted by a TM can be accepted by an NTM. The following theorem shows that the converse is also true.

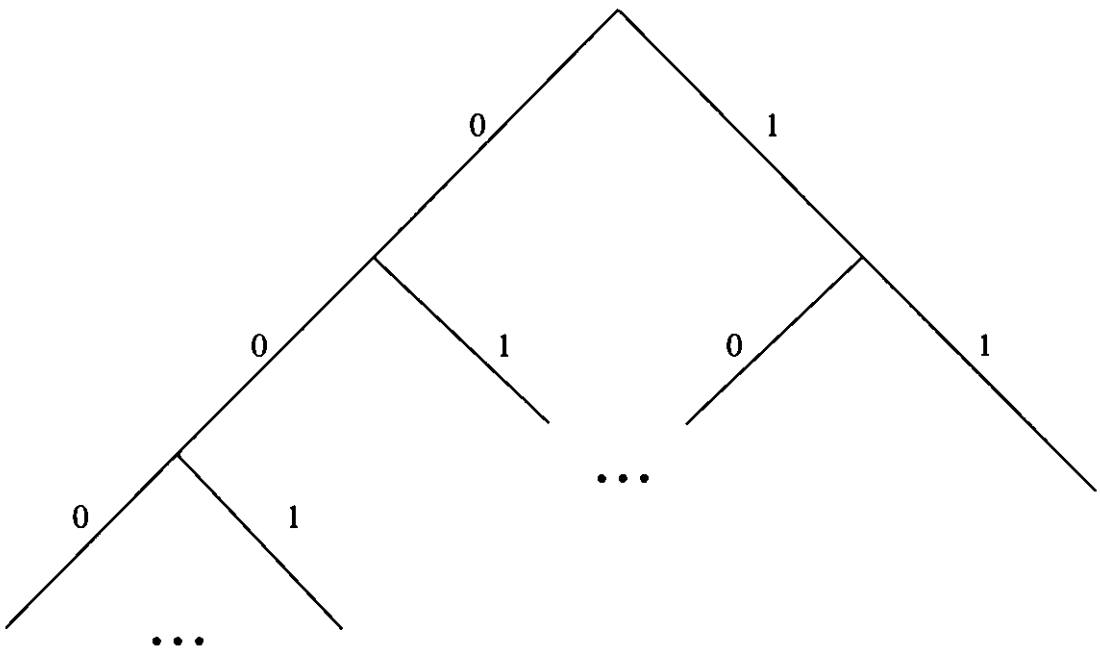
- **Theorem 9.2.** Let $T_1 = (Q_1, \Sigma, \Gamma_1, q_1, \delta_1)$ be an NTM. Then there is an ordinary (deterministic) TM $T_2 = (Q_2, \Sigma, \Gamma_2, q_2, \delta_2)$ with $L(T_2) = L(T_1)$.

  Proof: The strategy for constructing T_2 is to let it try *every* sequence of moves of T_1, one sequence at a time, accepting if and only if it finds a sequence that would cause T_1 to halt in the accepting state.

  Assume for the sake of simplicity that T_1 never has a choice of more than two moves in any configuration. (The proof will easily generalize to larger numbers.)

  Further assume that whenever there are any moves at all, there are exactly two, labeled 0 and 1. (The order is arbitrary, and it is possible that both are actually the same move.)

  For any input string x, a *computation tree* such as the following can be used to represent the sequences of moves T_1 might make on input x:



Nodes in the tree represent configurations of T_1. The root is the initial configuration corresponding to input x, and the children of any node N correspond to the configurations T_1 might reach in one step from the configuration N.

Every interior node has exactly two children; leaves represent halting configurations.

T_2's job is to search the tree for accepting configurations. Because the tree might be infinite, a *breadth-first* approach is appropriate: T_2 will try all possible single moves, then all possible sequences of two moves, and so on.

If $x \in L(T_1)$, then for some n there is a sequence of n moves that causes T_1 to accept input x, and T_2 will eventually get around to trying that sequence.

T_2 will have three tapes:

1. Used only to save the original input string; its contents are never changed.
2. Used to keep track of the sequence of moves of T_1 that T_2 is currently attempting to execute.
3. The "working tape," corresponding to T_1's tape, where T_2 actually carries out the steps specified by the current string on tape 2.

Every time T_2 begins trying a new sequence, the third tape is erased and the input from tape 1 re-copied onto it.

A particular sequence of moves will be represented by a string of binary digits. The string 001 represents the first (i.e., 0th) choice from the initial configuration, followed by the first choice from that configuration, followed by the second choice from that configuration.

There may be strings of digits that do not correspond to sequences of moves, because the first few moves cause T_1 to halt. When T_2 encounters a digit that does not correspond to an executable move, it will abandon the string.

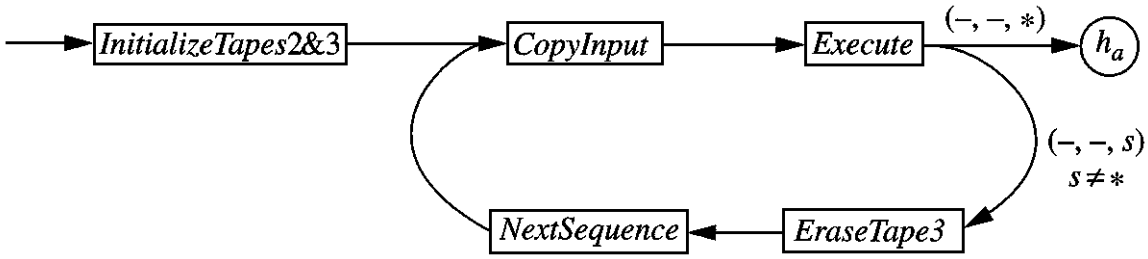
T_2 will generate sequences of moves using the *canonical* ordering of $\{0, 1\}^*$:

Λ, 0, 1, 00, 01, 10, 11, 000, 001, …, 111, 0000, …

If α is a sequence of moves, T_2 generates the next string by interpreting α as a binary number and adding 1, unless $\alpha = 1^k$, in which case the next string is 0^{k+1}.

Nondeterministic Turing Machines (Continued)

T_2 is composed of five smaller TMs called *InitializeTapes2&3*, *CopyInput*, *Execute*, *EraseTape3*, and *NextSequence*, combined in the following way:



InitializeTapes2&3 writes the symbol 0 in square 1 of tape 2, to represent the sequence of moves to be tried first, and places the symbol # in square 0 of tape 3. This marker allows T_2 to detect, and recover from, an attempt by T_1 to move its head off the tape.

CopyInput copies the original input string *x* from tape 1 onto tape 3, so that tape 3 has contents #$\underline{\Delta}x$.

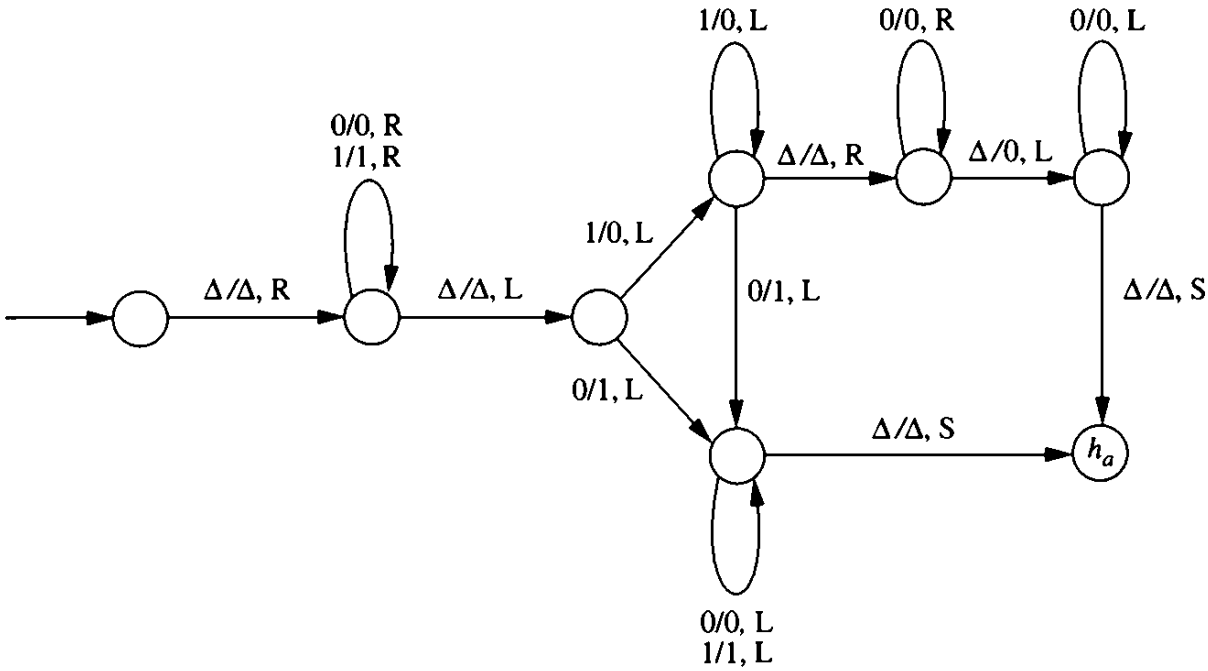
Execute simulates the action of T_1 on this input, by executing the sequence of moves currently specified on tape 2. It finishes with a symbol *s* in the current square of tape 3, and *s* = * if and only if the sequence of moves causes T_1 to accept. In this case T_2 accepts, and otherwise it continues with *EraseTape3*.

EraseTape3 restores tape 3 to the configuration #$\underline{\Delta}$. It is able to complete this operation because the length of the string on tape 2 limits how far to the right the rightmost nonblank symbol on tape 3 can be.

NextSequence updates the string of digits on tape 2 using the operation similar to adding 1 in binary.

Nondeterministic Turing Machines (Continued)

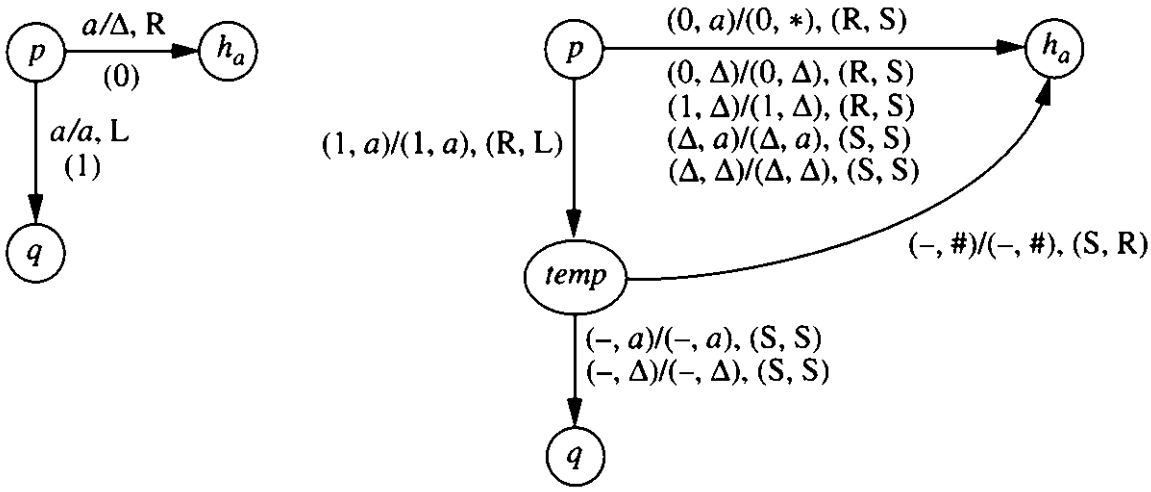
The following figure shows a one-tape version of *NextSequence*; the actual *Next-Sequence* uses the second tape only, ignoring tapes 1 and 3.



Nondeterministic Turing Machines (Continued)

Execute must simulate the sequence of moves of T_1 specified by the string of digits on tape 2.

Describing *Execute* in detail would be very tedious. Instead, the following figure shows a small portion of the transition diagram for T_1 along with the corresponding portion of the diagram for *Execute*:



For simplicity, these diagrams assume that T_1's tape alphabet is $\{a\}$. The second diagram shows only the moves on tapes 2 and 3.

Execute has six moves from state *p*, one for each combination of the three possible symbols on tape 2 (0, 1, and Δ) and the two on tape 3.

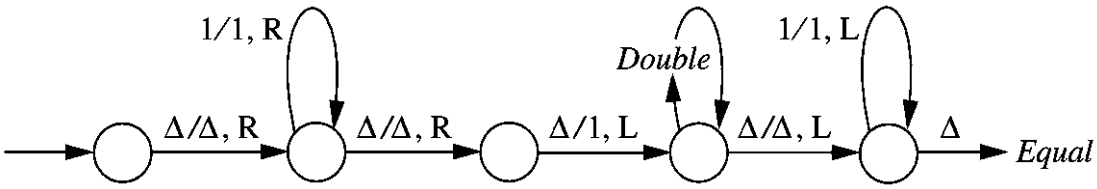
The "temporary" state allows *Execute* to test whether it is possible for T_1 to move left without the head falling off the tape. The symbol # indicates a crash by T_1; *Execute* halts normally after moving the tape head to the right of #.

Nondeterministic Turing Machines (Continued)

- **Example 9.10:** A Simple Example of Nondeterminism

  Let *Double* be a TM that doubles the value of a string of 1's. (*Double* makes a copy of the input and then deletes the blank between the original string and the copy.)

  Let *T* be the following nondeterministic TM:



  T moves past the input string, places a single 1 on the tape, separated from the input by a blank. After positioning the tape head on the blank, it executes *Double* zero or more times (nondeterministically) before returning the head to square 0.

  Finally, *T* executes the TM *Equal*, which accepts if and only if the original input matches the generated string of 1's, which represents some arbitrary power of 2. Therefore, *T* accepts the language $\{1^{2^i} \mid i \geq 0\}$.

  Nondeterminism is not needed in order to accept this language, although it simplifies the TM.

  A deterministic TM could repeatedly divide the original input by 2 until it reaches 1, accepting if the remainder is always zero.

  An easier approach would be to start with 1 and perform a sequence of multiplications by 2, accepting if the result ever matches the original input.

9.6 Universal Turing Machines

- The Turing machines discussed so far have been created to execute a specific algorithm.

- In his 1936 paper, however, Turing anticipated *stored-program* computers, which are capable of executing an arbitrary algorithm stored in memory.

- Turing's "universal computing machine" is a TM T_u whose input consists of a string specifying a (special-purpose) TM T_1 and a string z interpreted as input to T_1. T_u then simulates the processing of z by T_1.

- The first step in creating T_u is to formulate a notational system that can be used to encode both an arbitrary TM T_1 and an input string z over an arbitrary alphabet as strings $e(T_1)$ and $e(z)$ over some fixed alphabet.

- T_u's alphabet will be $\{0, 1\}$, even though T_1 may have a much larger alphabet. Each state of T_1, each tape symbol, and each of the three "directions" (S, L, and R) will be assigned a positive integer value.

- In order to ensure that the encoding is one-to-one, it will be necessary to fix the set of symbols that can be used by TMs.

- **Convention.** Assume that there are two fixed infinite sets $\mathcal{Q} = \{q_1, q_2, \ldots\}$ and $\mathcal{S} = \{a_1, a_2, \ldots\}$ so that for any Turing machine $T = (Q, \Sigma, \Gamma, q_0, \delta)$, we have $Q \subseteq \mathcal{Q}$ and $\Gamma \subseteq \mathcal{S}$.

Universal Turing Machines (Continued)

- A state or a tape symbol can now be represented by a string of 0's that corresponds to its subscript in $\mathcal{Q}$ or S.

- **Definition 9.5:** First, associate to each tape symbol (including Δ), to each state (including h_a and h_r), and to each of the three directions, a string of 0's. Let

  $s(\Delta) = 0$
  $s(a_i) = 0^{i+1}$ (for each $a_i \in S$)
  $s(h_a) = 0$
  $s(h_r) = 00$
  $s(q_i) = 0^{i+2}$ (for each $q_i \in \mathcal{Q}$)
  $s(\mathrm{S}) = 0$
  $s(\mathrm{L}) = 00$
  $s(\mathrm{R}) = 000$

  Each move m of a TM, described by the formula

  $\delta(p, a) = (q, b, D)$

  is encoded by the string

  $e(m) = s(p)1s(a)1s(q)1s(b)1s(D)1$

  and for any TM T, with initial state q, T is encoded by the string

  $e(T) = s(q)1e(m_1)1e(m_2)1\ldots e(m_k)1$

  where $m_1, m_2, \ldots, m_k$ are the distinct moves of T, arranged in some arbitrary order. Finally, any string $z = z_1z_2\ldots z_k$, where each $z_i \in S$, is encoded by

  $e(z) = 1s(z_1)1s(z_2)1\ldots s(z_k)1$

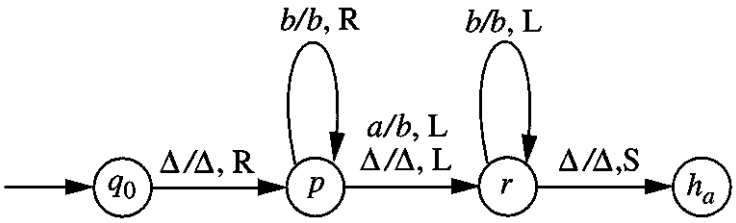
- The 1 at the beginning of the string $e(z)$ is included so that in a composite string of the form $e(T)e(z)$, there will be no doubt as to where $e(T)$ stops.

- Because the moves of a TM T can appear in the string $e(T)$ in any order, there will in general be many correct encodings of T. However, any string of 0's and 1's can be the encoding of at most one TM.

Universal Turing Machines (Continued)

- **Example 9.11:** The Encoding of a Simple TM

  The following TM transforms an input string of *a*'s and *b*'s by changing the left-most *a*, if there is one, to *b*:



  Assume that *a* and *b* are assigned the numbers 1 and 2, so that $s(a) = 00$ and $s(b) = 000$, and that q_0, p, and r are given the numbers 1, 2, and 3, respectively.

  If the six moves are encoded in the order they appear, left-to-right, the first move $\delta(q_0, \Delta) = (p, \Delta, R)$ is represented by the string

  $0^3 10^1 10^4 10^1 10^3 1 = 00010100001010001$

  The entire TM is represented by the string

  0001 000101000010100011 000010001000010001000011 000010010000010001001 1
  000010100000101001 1 000001000100000100010011 000001010101011

  The first part of the string, 0001, identifies the initial state of the TM. Individual moves are separated by spaces for readability.

Universal Turing Machines (Continued)

- The input to the universal TM T_u will consist of a string of the form $e(T)e(z)$, where T is a TM and z is a string over T's input alphabet.

- T_u should accept $e(T)e(z)$ if and only if T accepts input z, and in this case the output from T_u should be the encoded form of the output produced by T on input z.

- It will be convenient to give T_u three tapes:

  1. Used for input and output. Initially contains the input string $e(T)e(z)$.
  2. Used as a working tape during the simulation of T.
  3. Contains the encoded form of the state T is currently in.

- T_u's first step is to move the string $e(z)$ (except for the initial 1) from the end of tape 1 to tape 2, beginning in square 3. Since T begins with its leftmost square blank, T_u will write 01 (because $0 = s(\Delta)$) in squares 1 and 2 of tape 2; square 0 is left blank, and the tape head is positioned on square 1.

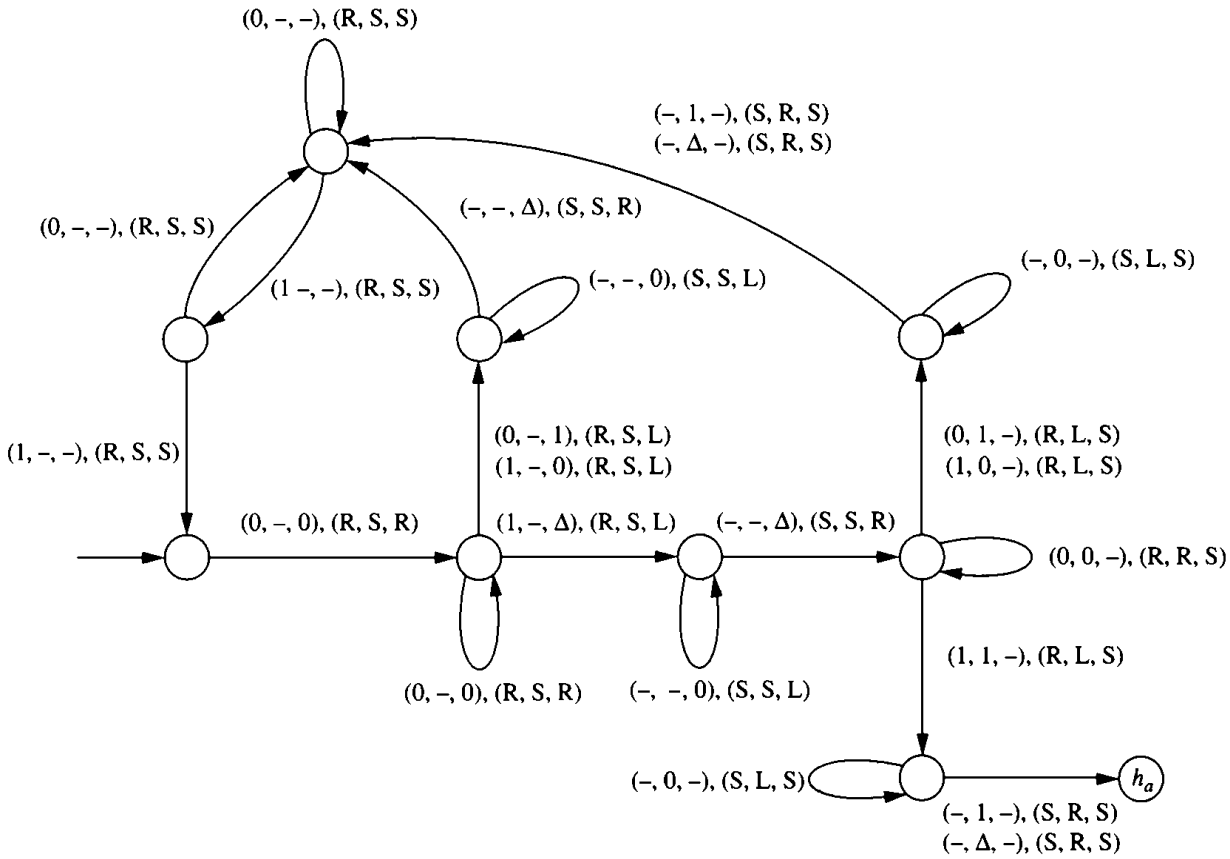
- Next, T_u copies the encoded form of T's initial state from the beginning of tape 1 onto tape 3, beginning in square 1, and deletes it from tape 1.

- After these initial steps, T_u is ready to begin simulating the action of T (encoded on tape 1) on the input string (encoded on tape 2). As the simulation starts, the three tape heads are all in square 1.

Universal Turing Machines (Continued)

- The next move of T at any point is determined by T's state (encoded on tape 3) and the current symbol on T's tape, whose encoding starts in the current position on tape 2.

- In order to simulate this move, T_u must search tape 1 for the 5-tuple whose first two parts match this state-input combination. The following TM performs this search:



Since the search operation never changes any tape symbols, the labels have been simplified. A label of the form

$(a, b, c), (D_1, D_2, D_3)$

means

$(a, b, c) / (a, b, c), (D_1, D_2, D_3)$

Universal Turing Machines (Continued)

- Once the appropriate 5-tuple is found, the last three parts tell T_u how to simulate the move. Suppose that before the search, T_u's three tapes look like this:

  Δ0001010000101000110000100100000100010011000 1…
  Δ01001 0 010001Δ…
  Δ 0 000Δ…

  The corresponding tape of T would be

  Δa $\underline{a}$$b$Δ…

  assuming that the symbols numbered 1 and 2 are a and b, respectively, and T would be in state 2 (the one with encoding 0000). After the search of tape 1, the tapes look like this:

  Δ0001010000101000110000100 1 0 00001000100110001…
  Δ01001 0 010001Δ…
  Δ 0 000Δ…

  The 5-tuple on tape 1 specifies that T's current symbol should be changed to b, the head should be moved left, and the state should be changed to state 3. The final result is

  Δ0001010000101000110000100100000100010011000 1…
  Δ01 0 0100010001Δ…
  Δ 0 0000Δ…

  and T_u is now ready to simulate the next move.

- There are two ways this process might stop.

  T halts abnormally, because there is no move specified or because the move calls for it to move its tape head off the tape. In the first case, the search operation halts abnormally. It is easy to make T_u reject in the second case.

  T accepts. T_u detects this when it processes a 5-tuple on tape 1 whose third part is a single 0. In this case, after T_u has changed tape 2 appropriately, it erases tape 1, copies tape 2 onto tape 1, and accepts.

9.7 Models of Computation and the Church-Turing Thesis

- A TM is not the only way to extend a PDA. One possibility is to add a second stack to a PDA. Another is to replace the stack by a *queue*. Both approaches lead to machines with the same computing power as Turing machines.

- To say that the Turing machine is a general model of computation is simply to say that any algorithmic procedure that can be carried out at all (by a human, a team of humans, or a computer) can be carried out by a TM.

- This statement was first formulated by Alonzo Church in the 1930s, and it is usually referred to as *Church's thesis,* or the *Church-Turing thesis.*

- The Church-Turing thesis is not mathematically precise because there is no precise definition of "algorithmic procedure," and therefore it cannot be proved.

- Since the invention of the TM, however, enough evidence has accumulated to cause the Church-Turing thesis to be generally accepted:

  1. The nature of the model makes it seem likely that all the steps that are crucial to human computation can be carried out by a TM, although a TM may require more moves to do what a human could do in one.
  2. Various enhancements of the TM model have been suggested; in each case, the computing power of the machine is unchanged.
  3. Other theoretical models of computation have been proposed. In every case, the model has been shown to be equivalent to the Turing machine.
  4. Since the introduction of the TM, no one has suggested any type of computation that ought to be included in the category of "algorithmic procedure" and *cannot* be implemented on a TM.

- Adopting the Church-Turing thesis effectively defines an algorithm as a procedure that can be executed on a TM. Such a definition provides a starting point for discussing which problems can be solved algorithmically and which cannot.

- The Church-Turing thesis has another use: a solution to a problem can often be described verbally; translating it into a detailed TM implementation may be tedious but is generally straightforward.