# TURING MACHINES

At this point it will help us to recapitulate the major themes of the previous two parts and outline all the material we have yet to present in the rest of the book all in one large table.

| **Language Defined by** | **Corresponding Acceptor** | **Nondeterminism = determinism?** | **Language Closed Under** | **What Can be Decided** | **Example of Application** |
|---|---|---|---|---|---|
| Regular expression | Finite automaton<br>Transition graph | Yes | Union, product, Kleene star, intersection, complement | Equivalence, emptiness, finiteness, membership | Text editors, sequential circuits |
| Context-free grammar | Pushdown automaton | No | Union, product, Kleene star | Emptiness finiteness membership | Programming language statements, compilers |
| Type 0 grammar | Turing machine, Post machine, 2PDA, *n*PDA | Yes | Union, product, Kleene star | Not much | Computers |

We see from the lower right entry in the table that we are about to fulfill the promise made in the introduction. We shall soon provide a mathematical model for the entire family of modern-day computers. This model will enable us not only to study some theoretical limitations on the tasks that computers can perform, it will also be a model that we can use to show that certain operations *can* be done by computer. This new model will turn out to be surprisingly like the models we have been studying so far.

Another interesting observation we can make about the bottom row of the table is that we take a very pessimistic view of our ability to decide the important questions about this mathematical model (which as we see is called a Turing machine).

We shall prove that we cannot even decide if a given word is accepted by a given Turing machine. This situation is unthinkable for FA's or PDA's, but now it is one of the unanticipated facts of life—a fact with grave repercussions.

There is a definite progression in the rows of this table. All regular languages are context-free languages, and we shall see that all context-free languages are Turing machine languages. Historically, the order of invention of these ideas is:

1. Regular languages and FA's were developed by Kleene, Mealy, Moore, Rabin, and Scott in the 1950s.
2. CFG's and PDA's were developed later, by Chomsky, Oettinger, Schützenberger, and Evey, mostly in the 1960s.
3. Turing machines and their theory were developed by Alan Mathison Turing and Emil Post in the 1930s and 1940s.

It is less surprising that these dates are out of order than that Turing's work predated the invention of the computer itself. Turing was not analyzing a specimen that sat on the table in front of him; he was engaged in inventing the beast. It was directly from the ideas in his work on mathematical models that the first computers were built. This is another demonstration that there is nothing more practical than a good abstract theory.

Since Turing machines will be our ultimate model for computers, they will necessarily have output capabilities. Output is very important, so important that a program with no output statements might seem totally useless because it would never convey to humans the result of its calculations. We may have heard it said that the one statement every program must have is an output statement. This is not exactly true. Consider the following program (written in no particular language):

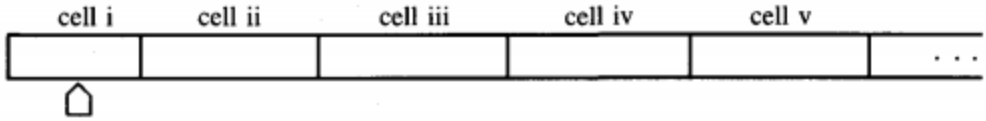
1. READ $X$
2. IF $X = 1$ THEN END
3. IF $X = 2$ THEN DIVIDE $X$ BY 0
4. IF $X > 2$ THEN GOTO STATEMENT 4

Let us assume that the input is a positive integer. If the program terminates naturally, then we know $X$ was 1. If it terminates by creating overflow or was interrupted by some error message warning of illegal calculation (crashes), then we know that $X$ was 2. If we find that our program was terminated because it exceeded our alloted time on the computer, then we know $X$ was greater than 2. We shall see in a moment that the same trichotomy applies to Turing machines.

**DEFINITION**

A **Turing machine,** denoted TM, is a collection of six things:

1. An alphabet $\Sigma$ of input letters, which for clarity's sake does not contain the blank symbol $\Delta$.
2. A TAPE divided into a sequence of numbered cells each containing one character or a blank. The input word is presented to the machine one letter per cell beginning in the left-most cell, called cell i. The rest of the TAPE is initially filled with blanks, $\Delta$'s.




3. A TAPE HEAD that can in one step read the contents of a cell on the TAPE, replace it with some other character, and reposition itself to the next cell to the right or to the left of the one it has just read. At the start of the processing, the TAPE HEAD always begins by reading the input in cell i. The TAPE HEAD can never move left from cell i. If it is given orders to do so, the machine crashes.
4. An alphabet, $\Gamma$, of characters that can be printed on the TAPE by the TAPE HEAD. This can include $\Sigma$. Even though we allow the TAPE HEAD to print a $\Delta$ we call this erasing and do not include the blank as a letter in the alphabet $\Gamma$.
5. A finite set of states including exactly one START state from which we begin execution (and which we may reenter during execution) and some (maybe none) HALT states that cause execution to terminate when we enter them. The other states have no functions, only names:

$$q_1,\ q_2,\ q_3,\ \ldots \qquad \text{or} \qquad 1,\ 2,\ 3,\ \ldots$$

6. A **program,** which is a set of rules that tell us, on the basis of the letter the TAPE HEAD has just read, how to change states, what to print and where to move the TAPE HEAD. We depict the program as a collection of directed edges connecting the states. Each edge is labeled with a triplet of information:

(letter, letter, direction)

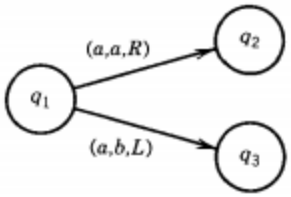
The first letter (either $\Delta$ or from $\Sigma$ or $\Gamma$) is the character the TAPE HEAD reads from the cell to which it is pointing. The second letter (also $\Delta$ or from $\Gamma$) is what the TAPE HEAD prints in the cell before it leaves. The third component, the direction, tells the TAPE HEAD whether to move one cell to the right, $R$, or one cell to the left, $L$.

No stipulation is made as to whether every state has an edge leading from it for every possible letter on the TAPE. If we are in a state and read a letter that offers no choice of path to another state, we *crash;* that means we terminate execution unsuccessfully. To terminate execution of a certain input successfully we must be led to a HALT state. The word on the input TAPE is then said to be *accepted* by the TM.

A crash also occurs when we are in the first cell on the TAPE and try to move the TAPE HEAD left.

By definition, all Turing machines are **deterministic.** This means that there is no state $q$ that has two or more edges leaving it labeled with the same first letter.

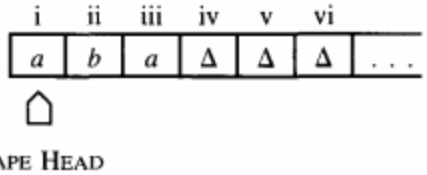
For example,




is not allowed. ■

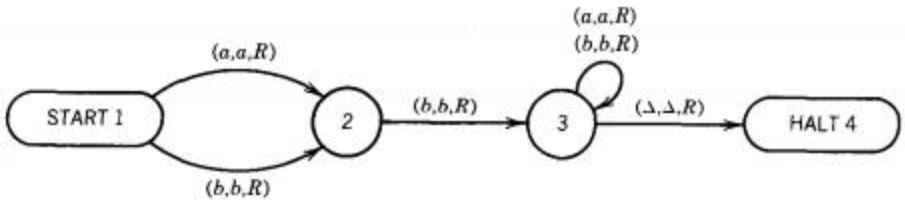
## EXAMPLE

The following is the TAPE from a Turing machine about to run on the input *aba*




TAPE HEAD

The program for this TM is given as a directed graph with labeled edges as shown below




Notice that the loop at state 3 has two labels. The edges from state 1 to state 2 could have been drawn as one edge with two labels.

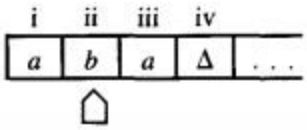
We start, as always, with the TAPE HEAD reading cell i and the program in the start state, which is here labeled state 1. We depict this as

$$\begin{matrix} 1 \\ \underline{a}ba \end{matrix}$$

The number on top is the number of the state we are in. Below that is the current meaningful contents of the string on the TAPE up to the beginning of the infinite run of blanks. It is possible that there may be a Δ inside this string. We underline the character in the cell that is *about to be read.*

At this point in our example, the TAPE HEAD reads the letter $a$ and we follow the edge $(a,a,R)$ to state 2. The instructions of this edge to the TAPE HEAD are "read an $a$, print an $a$, move right."

The TAPE now looks like this:




Notice that we have stopped writing the words "TAPE HEAD" under the indicator under the TAPE. It is still the TAPE HEAD nonetheless.

We can record the execution process by writing:

$$\begin{matrix} 1 \\ \underline{a}ba \end{matrix} \rightarrow \begin{matrix} 2 \\ a\underline{b}a \end{matrix}$$

At this point we are in state 2. Since we are reading the $b$ in cell ii, we must take the ride to state 3 on the edge labeled $(b,b,R)$. The TAPE HEAD replaces the $b$ with a $b$ and moves right one cell. The idea of replacing a letter with itself may seem silly, but it unifies the structure of Turing machines.

We could instead have constructed a machine that uses two different types of instructions: either print or move, not both at once. Our system allows us to formulate two possible meanings in a single type of instruction.

$(a, a, R)$ means move, but do not change the TAPE cell

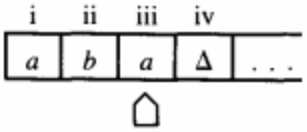
$(a, b, R)$ means move and change the TAPE cell

This system does not give us a one-step way of changing the contents of the TAPE cell without moving the TAPE HEAD, but we shall see that this too can be done by our TM's.

Back to our machine. We are now up to

$$\underset{\underline{a}ba}{1} \rightarrow \underset{a\underline{b}a}{2} \rightarrow \underset{ab\underline{a}}{3}$$

The TAPE now looks like this.




We are in state 3 reading an $a$, so we loop. That means we stay in state 3 but we move the TAPE HEAD to cell iv.

$$\underset{ab\underline{a}}{3} \rightarrow \underset{aba\underline{\Delta}}{3}$$

This is one of those times when we must indicate a $\Delta$ as part of the meaningful contents of the TAPE.

We are now in state 3 reading a $\Delta$, so we move to state 4.

$$\underset{aba\underline{\Delta}}{3} \rightarrow \underset{aba\Delta\underline{\Delta}}{4}$$

The input string *aba* has been accepted by this TM. This particular machine did not change any of the letters on the TAPE, so at the end of the run the TAPE still reads $aba\Delta \ldots$. This is not a requirement for the acceptance of a string, just a phenomenon that happened this time.

In summary, the whole execution can be depicted by the following **execution chain,** also called a **process chain,** or a **trace of execution,** or simply a **trace:**

$$\underset{\underline{a}ba}{1} \rightarrow \underset{a\underline{b}a}{2} \rightarrow \underset{ab\underline{a}}{3} \rightarrow \underset{aba\underline{\Delta}}{3} \rightarrow \text{HALT}$$

This is a new use for the arrow. It is neither a production nor a derivation.

Let us consider which input strings are accepted by this TM. Any first letter, $a$ or $b$, will lead us to state 2. From state 2 to state 3 we require that we read the letter $b$. Once in state 3 we stay there as the TAPE HEAD moves right and right again, moving perhaps many cells until it encounters a $\Delta$. Then we get to the HALT state and accept the word. Any word that reaches state 3 will eventually be accepted. If the second letter is an $a$, then we crash at state 2. This is because there is no edge coming from state 2 with directions for what happens when the TAPE HEAD reads an $a$.

The language of words accepted by this machine is: All words over the alphabet $\{a,b\}$ in which the second letter is a $b$.

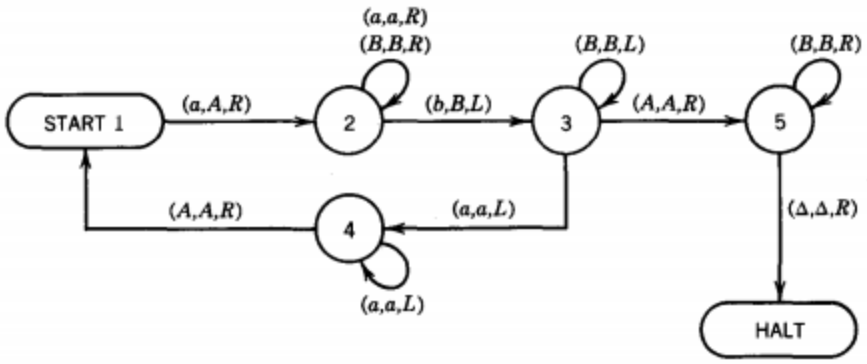
This is a regular language because it can also be defined by the regular expression:

$$\mathbf{(a+b)b(a+b)^*}$$

This TM is also reminiscent of FA's, making only one pass over the input string, moving its TAPE HEAD always to the right, and never changing a letter it has read. TM's can do more tricks, as we shall soon see. ■

## EXAMPLE

Consider the following TM.




We have only drawn the program part of the TM, since initial appearance of the TAPE depends on the input word. This is a more complicated example of a TM. We analyze it by first explaining what it does and then recognizing how it does it.

The language this TM accepts is $\{a^n b^n\}$.

By examining the program we can see that the TAPE HEAD may print any of the letters $a$, $A$ or $B$, or a $\Delta$, and it may read any of the letters $a$, $b$, $A$ or $B$ or a blank. Technically, the input alphabet is $\Sigma = \{a, b\}$ and the output alphabet is $\Gamma = \{a, A, B\}$, since $\Delta$ is the symbol for a blank or empty cell and is not a legal character in an alphabet. Let us describe the algorithm, informally in English, before looking at the directed graph that is the program.

Let us assume that we start with a word of the language $\{a^n b^n\}$ on the TAPE. We begin by taking the $a$ in the first cell and changing it to the character $A$. (If the first cell does not contain an $a$, the program should crash. We can arrange this by having only one edge leading from START and labeling it to read an $a$.) The conversion from $a$ to $A$ means that this $a$ has been counted. We now want to find the $b$ in the word that pairs off with this $a$. So we keep moving the TAPE HEAD to the right, without changing anything it passes over, until it reaches the first $b$. When we reach this $b$, we change it into the character $B$, which again means that it too has been counted. Now we move the TAPE HEAD back down to the left until it reaches the first uncounted $a$. The first time we make our descent down the TAPE this will be the $a$ in cell ii.

How do we know when we get to the first uncounted $a$? We cannot tell the TAPE HEAD to "find cell ii." This instruction is not in its repertoire. We can, however, tell the TAPE HEAD to keep moving to the left until it gets to the character $A$. When it hits the $A$ we bounce one cell to the right and there we are. In doing this the TAPE HEAD passed through cell ii on its way down the TAPE. However, when we were first there we did not recognize it as our destination. Only when we bounce off of our marker, the first $A$ encountered, do we realize where we are. Half the trick in programming TM's is to know where the TAPE HEAD is by bouncing off of landmarks.

When we have located this left-most uncounted $a$ we convert it into an $A$ and begin marching up the TAPE looking for the corresponding $b$. This means that we skip over some $a$'s and over the symbol $B$, which we previously wrote, leaving them unchanged, until we get to the first uncounted $b$. Once we have located it, we have found our second pair of $a$ and $b$. We count this second $b$ by converting it into a $B$, and we march back down the TAPE looking for our next uncounted $a$. This will be in cell iii. Again, we cannot tell the TAPE HEAD, "find cell iii." We must program it to find the intended cell. The same instructions as given last time work again. Back down to the first $A$ we meet and then up one cell. As we march down we walk through a $B$ and some $a$'s until we first reach the character $A$. This will be the second $A$, the one in cell ii. We bounce off this to the right, into cell iii, and find an $a$. This we convert to $A$ and move up the TAPE to find its corresponding $b$.

This time marching up the TAPE we again skip over $a$'s and $B$'s until we find the first $b$. We convert this to $B$ and march back down looking for the first unconverted $a$. We repeat the pairing process over and over.

What happens when we have paired off all the $a$'s and $b$'s? After we have

converted our last $b$ into a $B$ and we move left looking for the next $a$ we find that after marching left back through the last of the $B$'s we encounter an $A$. We recognize that this means we are out of little $a$'s in the initial field of $a$'s at the beginning of the word.

We are about ready to accept the word, but we want to make sure that there are no more $b$'s that have not been paired off with $a$'s, or any extraneous $a$'s at the end. Therefore we move back up through the field of $B$'s to be sure that they are followed by a blank, otherwise the word initially may have been *aaabbbb* or *aaabbba*.

When we know that we have only $A$'s and $B$'s on the TAPE, in equal number, we can accept the input string.

The following is a picture of the contents of the TAPE at each step in the processing of the string *aaabbb*. Remember, in a trace the TAPE HEAD is indicated by the underlining of the letter it is about to read.

$\underline{a}\ a\ a\ b\ b\ b$

$A\ \underline{a}\ a\ b\ b\ b$

$A\ a\ \underline{a}\ b\ b\ b$

$A\ a\ a\ \underline{b}\ b\ b$

$A\ a\ \underline{a}\ B\ b\ b$

$A\ \underline{a}\ a\ B\ b\ b$

$\underline{A}\ a\ a\ B\ b\ b$

$A\ \underline{a}\ a\ B\ b\ b$

$A\ A\ \underline{a}\ B\ b\ b$

$A\ A\ a\ \underline{B}\ b\ b$

$A\ A\ a\ B\ \underline{b}\ b$

$A\ A\ a\ \underline{B}\ B\ b$

$A\ A\ \underline{a}\ B\ B\ b$

$A\ \underline{A}\ a\ B\ B\ b$

$A\ A\ \underline{a}\ B\ B\ b$

$A\ A\ A\ \underline{B}\ B\ b$

$A\ A\ A\ B\ \underline{B}\ b$

$A\ A\ A\ B\ B\ \underline{b}$

$A\ A\ A\ B\ \underline{B}\ B$

$A\ A\ A\ \underline{B}\ B\ B$

$A\ A\ \underline{A}\ B\ B\ B$

$A\ A\ A\ \underline{B}\ B\ B$

$A\ A\ A\ B\ \underline{B}\ B$

$A\ A\ A\ B\ B\ \underline{B}$

$A\ A\ A\ B\ B\ B\ \underline{\Delta}$

HALT

Based on this algorithm we can define a set of states that have the following meanings:

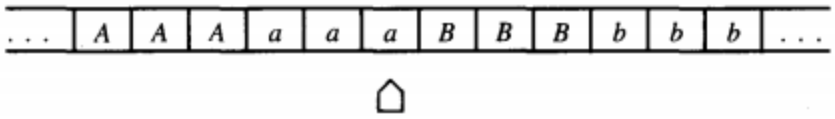
State 1 This is the start state, but it is also the state we are in whenever we are about to read the lowest unpaired $a$. In a PDA we can never return to the START state, but in a TM we can. The edges leaving from here must convert this $a$ to the character $A$ and move the TAPE HEAD right and enter state 2.

State 2 This is the state we are in when we have just converted an $a$ to an $A$ and we are looking for the matching $b$. We begin moving up the TAPE. If we read another $a$, we leave it alone and continue to march up the TAPE, moving the TAPE HEAD always to the right. If we read a $B$, we also leave it alone and continue to move the TAPE HEAD right. We cannot read an $A$ while in this state. In this algorithm all the $A$'s remain to the left of the TAPE HEAD once they are printed. If we read $\Delta$ while we are searching for the $b$ we are in trouble because we have not paired off our $a$. So we crash. The first $b$ we read, if we are lucky enough to find one, is the end of the search in this state. We convert it to $B$, move the TAPE HEAD left and enter state 3.

State 3 This is the state we are in when we have just converted a $b$ to $B$. We should now march left down the TAPE looking for the field of unpaired $a$'s. If we read a $B$, we leave it alone and keep moving left. If and when

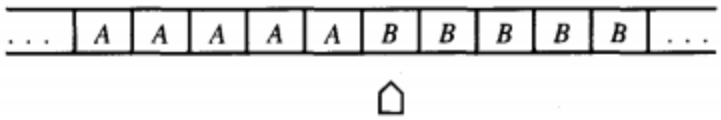
we read an $a$, we have done our job. We must then go to state 4, which will try to find the left-most unpaired $a$. If we encounter the character $b$ while moving to the left, something has gone very wrong and we should crash. If, however, we encounter the character $A$ before we hit an $a$, we know that used up the pool of unpaired $a$'s at the beginning of the input string and we may be ready to terminate execution. Therefore, we leave the $A$ alone and reverse directions to the right and move into state 5.

State 4 We get here when state 3 has located the right-most end of the field of unpaired $a$'s. The TAPE and TAPE HEAD situation looks like this:




In this state we must move left through a block of solid $a$'s (we crash if we encounter a $b$, a $B$, or a $\Delta$) until we find an $A$. When we do, we bounce off it to the right, which lands us at the left-most uncounted $a$. This means that we should next be in state 1 again.

State 5 When we get here it must be because state 3 found that there were no unpaired $a$'s left and it bounced us off the right-most $A$. We are now reading the left-most $B$ as in the picture below:




It is now our job to be sure that there are no more $a$'s or $b$'s left in this word. We want to scan through solid $B$'s until we hit the first blank. Since the program never printed any blanks, this will indicate the end of the input string. If there are no more surprises before the $\Delta$, we then accept the word by going to the state HALT. Otherwise we crash. For example, *aabba* would become *AABBa* and then crash because while searching for the $\Delta$ we find an $a$.

This explains the TM program that we began with. It corresponds to the description above state for state and edge for edge.

Let us trace the processing of the input string *aabb* by looking at its execution chain:

This explains the TM program that we began with. It corresponds to the description above state for state and edge for edge.

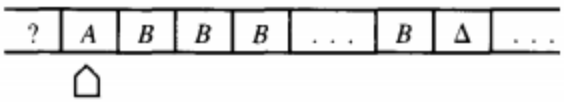
Let us trace the processing of the input string *aabb* by looking at its execution chain:

$$\underset{\underline{a}abb}{1} \rightarrow \underset{A\underline{a}bb}{2} \rightarrow \underset{Aa\underline{b}b}{2} \rightarrow \underset{A\underline{a}Bb}{3} \rightarrow \underset{\underline{A}aBb}{4} \rightarrow \underset{A\underline{a}Bb}{1}$$

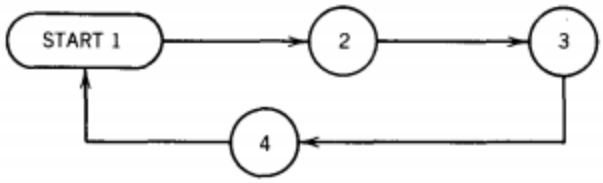
$$\rightarrow \underset{AA\underline{B}b}{2} \rightarrow \underset{AAB\underline{b}}{2} \rightarrow \underset{AA\underline{B}B}{3} \rightarrow \underset{A\underline{A}BB}{3} \rightarrow \underset{AA\underline{B}B}{5} \rightarrow \underset{AAB\underline{B}}{5}$$

$$\rightarrow \underset{AABB\underline{\Delta}}{5} \rightarrow HALT$$

It is clear that any string of the form $a^nb^n$ will reach the HALT state. To show that any string that reaches the HALT state must be of the form $a^nb^n$ we trace backward. To reach HALT we must get to state 5 and read a $\Delta$. To be in state 5 we must have come from state 3 from which we read an $A$ and some number of $B$'s while moving to the right. So at the point we are in state 3 ready to terminate, the TAPE and TAPE HEAD situation is as shown below:



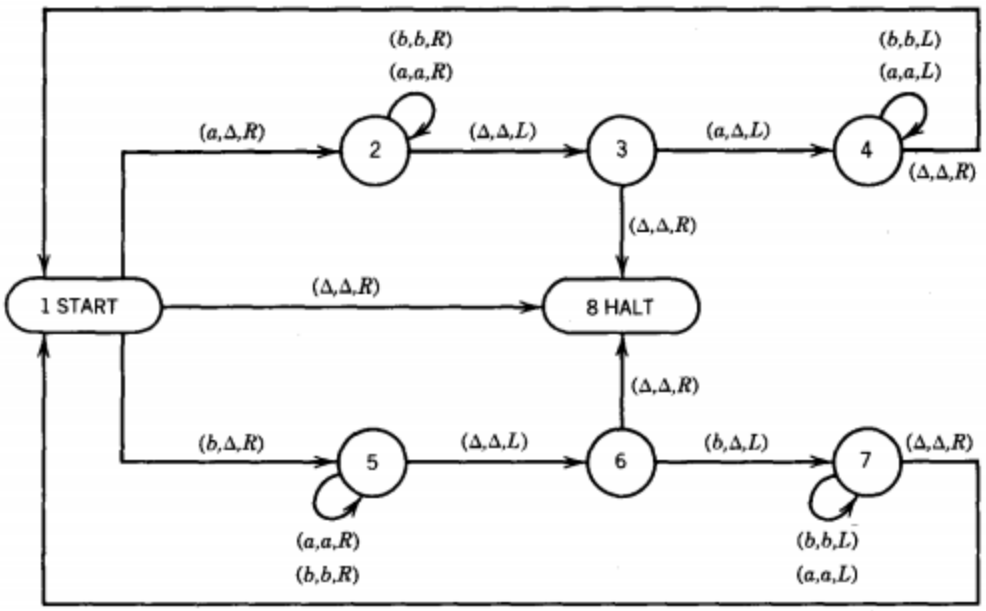

To be in state 3 means we have begun at START and circled around the loop some number of times.




Every time we go from START to state 3 we have converted an $a$ to an $A$ and a $b$ to a $B$. No other edge in the program of this TM changes the contents of any cell on the TAPE. However many $B$'s there are, there are just as many $A$'s. Examination of the movement of the TAPE HEAD shows that all the $A$'s stretch in one connected sequence of cells starting at cell i. To go from state 3 to HALT shows that the whole TAPE has been converted to $A$'s then $B$'s followed by blanks. Putting this all together, to get to HALT the input word must be $a^nb^n$ for some $n > 0$. ■

**EXAMPLE**

Consider the following TM




This looks like another monster, yet it accepts the familiar language PALINDROME and does so by a very simple deterministic algorithm.

We read the first letter of the input string and erase it, but we remember whether it was an *a* or a *b*. We go to the last letter and check to be sure it is the same as what used to be the first letter. If not, we crash, but if so, we erase it too. We then return to the front of what is left of the input string and repeat the process. If we do not crash while there are any letters left, then when we get to the condition where the whole TAPE is blank we accept the input string. This means that we reach the HALT state. Notice that the input string itself is no longer on the TAPE.

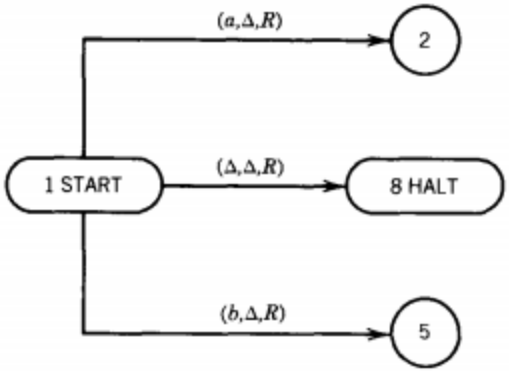
The process, briefly, works like this:

*a b b a b b a*
*b b a b b a*
*b b a b b*
*b a b b*
*b a b*
*a b*
*a*
Δ

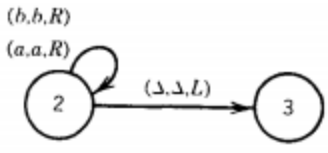
We mentioned above that when we erase the first letter we remember what it was as we march up to the last letter. Turing machines have no auxiliary memory device, like a PUSHDOWN STACK, where we could store this information, but there are ways around this. One possible method is to use some of the blank space further down the TAPE for making notes. Or, as in this case, the memory comes in by determining what path through the program the input takes. If the first letter is an $a$, we are off on the state 2—state 3—state 4 loop. If the first letter is a $b$, we are off on the state 5—state 6—state 7 loop.

All of this is clear from the descriptions of the meanings of the states below:

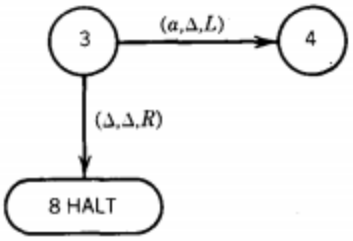
State 1 When we are in this state, we read the first letter of what is left of the input string. This could be because we are just starting and reading cell i or because we have been returned here from state 4 or state 7. If we read an $a$, we change it to a $\Delta$ (erase it), move the TAPE HEAD to the right, and progress to state 2. If we read a $b$, we erase it and move the TAPE HEAD to the right and progress to state 5. If we read a $\Delta$ where we expect the string to begin, it is because we have erased everything, or perhaps we started with the input word $\Lambda$. In either case, we accept the word and we shall see that it is in EVENPALINDROME.




State 2 We get here because we have just erased an $a$ from the front of the input string and we want to get to the last letter of the remaining input string to see if it too is an $a$. So we move to the right through all the $a$'s and $b$'s left in the input until we get to the end of the string at the first $\Delta$. When that happens we back up one cell (to the left) and move into state 3.

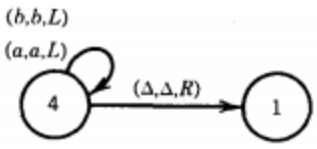


State 3 We get here only from state 2, which means that the letter we erased at the start of the string was an *a* and state 2 has requested us now to read the last letter of the string. We found the end of the string by moving to the right until we hit the first Δ. Then we bounced one cell back to the left. If this cell is also blank, then there are only blanks left on the TAPE. The letters have all been successfully erased and we can accept the word. So we go to HALT. If there is something left of the input string, but the last letter is a *b*, the input string was not a palindrome. Therefore we crash by having no labeled edge to go on. If the last non-Δ letter is an *a*, then we erase it, completing the pair, and begin moving the TAPE HEAD left, down to the beginning of the string again to pair off another set of letters. We should note that if the word is accepted by going from state 3 to HALT then the *a* that is erased in moving from state 1 to state 2 is not balanced by another erasure but was the last letter left in the erasure process. This means that it was the middle of a word in ODDPALINDROME:

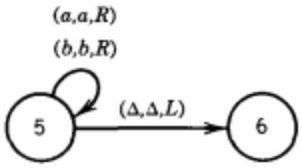



Notice that when we read the Δ and move to HALT we still need to include in the edge's label instructions to write something and move the TAPE HEAD somewhere. The label (Δ, *a*, *R*) would work just as well, or (Δ, *B*, *R*). However, (Δ, *a*, *L*) might be a disaster. We might have started with a one-letter word, say *a*. State 1 erases this *a*. Then state 2 reads the Δ in cell ii and returns us to cell i where we read the blank. If we try to move left from cell i we crash on the very verge of accepting the input string.

State 4 Like state 2, this is a travel state searching for the beginning of what is left of the input string. We keep heading left fearlessly because we know that cell i contains a Δ, so we shall not fall off the edge of the earth and crash by going left from cell i. When we hit the first Δ, we back up one position to the right, setting ourselves up in state 1 ready to read the first letter of what is left of the string:



State 5 We get to state 5 only from state 1 when the letter it has just erased was a $b$. In other words, state 5 corresponds exactly to state 2 but for strings beginning with a $b$. It too searches for the end of the string:




State 6 We get here when we have erased a $b$ in state 1 and found the end of the string in state 5. We examine the letter at hand. If it is an $a$, then the string began with $b$ and ended with $a$, so we crash since it is not in PALINDROME. If it is a $b$, we erase it and hunt for the beginning again. If it is a $\Delta$, we know that the string was an ODDPALINDROME with middle letter $b$. This is the twin of state 3.

State 7 This state is exactly the same as state 4. We try to find the beginning of the string.

Putting all these states together, we get the picture we started with.

Let us trace the running of this TM on the input string *ababa:*

$$\overset{1}{\underline{a}baba} \rightarrow \overset{2}{\Delta\underline{b}aba} \rightarrow \overset{2}{\Delta b\underline{a}ba} \rightarrow \overset{2}{\Delta ba\underline{b}a} \rightarrow \overset{2}{\Delta bab\underline{a}}$$

$$\rightarrow \overset{2}{\Delta baba\underline{\Delta}} \rightarrow \overset{3}{\Delta bab\underline{a}} \rightarrow \overset{4}{\Delta ba\underline{b}\Delta} \rightarrow \overset{4}{\Delta b\underline{a}b\Delta} \rightarrow \overset{4}{\Delta\underline{b}ab\Delta}$$

$$\rightarrow \overset{4}{\underline{\Delta}bab\Delta} \rightarrow \overset{5}{\Delta\underline{b}ab\Delta} \rightarrow \overset{5}{\Delta\Delta\underline{a}b\Delta} \rightarrow \overset{5}{\Delta\Delta a\underline{b}\Delta} \rightarrow \overset{5}{\Delta\Delta ab\underline{\Delta}}$$

$$\rightarrow \overset{6}{\Delta\Delta a\underline{b}\Delta} \rightarrow \overset{7}{\Delta\Delta\underline{a}\Delta\Delta} \rightarrow \overset{7}{\Delta\underline{\Delta}a\Delta\Delta} \rightarrow \overset{1}{\Delta\Delta\underline{a}\Delta\Delta} \rightarrow \overset{2}{\Delta\Delta\Delta\underline{\Delta}\Delta}$$

$$\rightarrow \overset{3}{\Delta\Delta\underline{\Delta}\Delta\Delta} \rightarrow \overset{8}{\text{HALT}}$$

(See Problem 7 below for comments on this machine.) ■

Our first example was no more than a converted FA, and the language it accepted was regular. The second example accepted a language that was context-free and nonregular and the TM given employed separate alphabets for

writing and reading. The third machine accepted a language that was also context-free but that could be accepted only by a nondeterministic PDA, whereas the TM that accepts it is deterministic.

We have seen that we can use the TAPE for more than a PUSHDOWN STACK. In the last two examples we ran up and down the TAPE to make observations and changes in the string at both ends and in the middle. We shall see later that the TAPE can be used for even more tasks: It can be used as work space for calculation and output.

In these three examples the TM was already assembled. In this next example we shall design the Turing machine for a specific purpose.

## NONDETERMINISTIC TURING MACHINES

A nondeterministic Turing machine is defined in the expected way. At any point in a computation the machine may proceed according to several possibilities. The transition function for a nondeterministic Turing machine has the form

$$\delta: Q \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}).$$

The computation of a nondeterministic Turing machine is a tree whose branches correspond to different possibilities for the machine. If some branch of the computation leads to the accept state, the machine accepts its input. If you feel the need to review nondeterminism, turn to Section 1.2 (page 47). Now we show that nondeterminism does not affect the power of the Turing machine model.

**THEOREM 3.16**

Every nondeterministic Turing machine has an equivalent deterministic Turing machine.

**PROOF IDEA** We can simulate any nondeterministic TM $N$ with a deterministic TM $D$. The idea behind the simulation is to have $D$ try all possible branches of $N$'s nondeterministic computation. If $D$ ever finds the accept state on one of these branches, $D$ accepts. Otherwise, $D$'s simulation will not terminate.

We view $N$'s computation on an input $w$ as a tree. Each branch of the tree represents one of the branches of the nondeterminism. Each node of the tree is a configuration of $N$. The root of the tree is the start configuration. The TM $D$ searches this tree for an accepting configuration. Conducting this search carefully is crucial lest $D$ fail to visit the entire tree. A tempting, though bad, idea is to have $D$ explore the tree by using depth-first search. The depth-first search strategy goes all the way down one branch before backing up to explore other branches. If $D$ were to explore the tree in this manner, $D$ could go forever down one infinite branch and miss an accepting configuration on some other branch. Hence we design $D$ to explore the tree by using breadth first search instead. This strategy explores all branches to the same depth before going on to explore any branch to the next depth. This method guarantees that $D$ will visit every node in the tree until it encounters an accepting configuration.

**PROOF** The simulating deterministic TM $D$ has three tapes. By Theorem 3.13 this arrangement is equivalent to having a single tape. The machine $D$ uses its three tapes in a particular way, as illustrated in the following figure. Tape 1 always contains the input string and is never altered. Tape 2 maintains a copy of $N$'s tape on some branch of its nondeterministic computation. Tape 3 keeps track of $D$'s location in $N$'s nondeterministic computation tree.

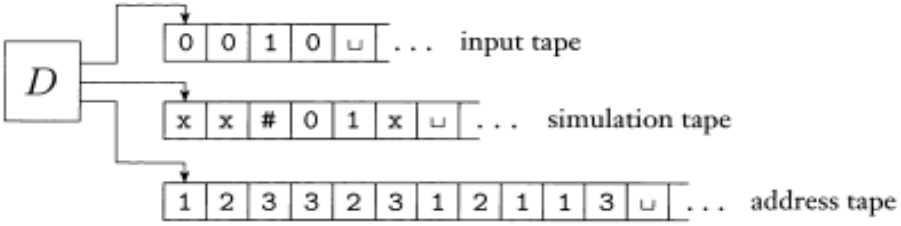



**FIGURE 3.17**
Deterministic TM $D$ simulating nondeterministic TM $N$

Let's first consider the data representation on tape 3. Every node in the tree can have at most $b$ children, where $b$ is the size of the largest set of possible choices given by $N$'s transition function. To every node in the tree we assign an address that is a string over the alphabet $\Sigma_b = \{1, 2, \ldots, b\}$. We assign the address 231 to the node we arrive at by starting at the root, going to its 2nd child, going to that node's 3rd child, and finally going to that node's 1st child. Each symbol in the string tells us which choice to make next when simulating a step in one branch in $N$'s nondeterministic computation. Sometimes a symbol may not correspond to any choice if too few choices are available for a configuration. In that case the address is invalid and doesn't correspond to any node. Tape 3 contains a string over $\Sigma_b$. It represents the branch of $N$'s computation from the root to the node addressed by that string, unless the address is invalid. The empty string is the address of the root of the tree. Now we are ready to describe $D$.

1. Initially tape 1 contains the input $w$, and tapes 2 and 3 are empty.
2. Copy tape 1 to tape 2.
3. Use tape 2 to simulate $N$ with input $w$ on one branch of its nondeterministic computation. Before each step of $N$ consult the next symbol on tape 3 to determine which choice to make among those allowed by $N$'s transition function. If no more symbols remain on tape 3 or if this nondeterministic choice is invalid, abort this branch by going to stage 4. Also go to stage 4 if a rejecting configuration is encountered. If an accepting configuration is encountered, *accept* the input.
4. Replace the string on tape 3 with the lexicographically next string. Simulate the next branch of $N$'s computation by going to stage 2.

**COROLLARY 3.18**

A language is Turing-recognizable if and only if some nondeterministic Turing machine recognizes it.

**PROOF** Any deterministic TM is automatically a nondeterministic TM, and so one direction of this theorem follows immediately. The other direction follows from Theorem 3.16.

We can modify the proof of Theorem 3.16 so that if $N$ always halts on all branches of its computation, $D$ will always halt. We call a nondeterministic Turing machine a ***decider*** if all branches halt on all inputs. Exercise 3.3 asks you to modify the proof in this way to obtain the following corollary to Theorem 3.16.

**COROLLARY 3.19**

A language is decidable if and only if some nondeterministic Turing machine decides it.

## ENUMERATORS

As we mentioned earlier, some people use the term *recursively enumerable language* for Turing-recognizable language. That term originates from a type of Turing machine variant called an ***enumerator***. Loosely defined, an enumerator is a Turing machine with an attached printer. The Turing machine can use that printer as an output device to print strings. Every time the Turing machine wants to add a string to the list, it sends the string to the printer. Exercise 3.4 asks you to give a formal definition of an enumerator. The following figure depicts a schematic of this model.

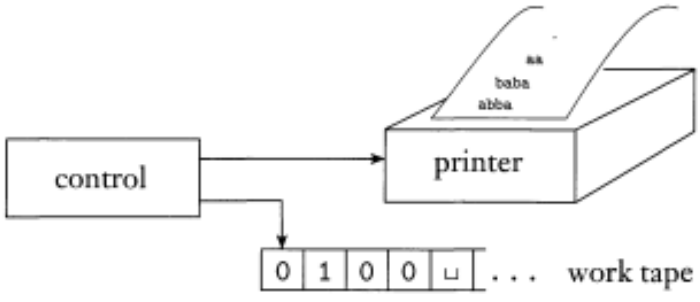



**FIGURE 3.20**
Schematic of an enumerator

An enumerator $E$ starts with a blank input tape. If the enumerator doesn't halt, it may print an infinite list of strings. The language enumerated by $E$ is the collection of all the strings that it eventually prints out. Moreover, $E$ may generate the strings of the language in any order, possibly with repetitions. Now we are ready to develop the connection between enumerators and Turing-recognizable languages.

**THEOREM 3.21**

A language is Turing-recognizable if and only if some enumerator enumerates it.

**PROOF** First we show that if we have an enumerator $E$ that enumerates a language $A$, a TM $M$ recognizes $A$. The TM $M$ works in the following way.

$M$ = "On input $w$:

1. Run $E$. Every time that $E$ outputs a string, compare it with $w$.
2. If $w$ ever appears in the output of $E$, *accept*."

Clearly, $M$ accepts those strings that appear on $E$'s list.

Now we do the other direction. If TM $M$ recognizes a language $A$, we can construct the following enumerator $E$ for $A$. Say that $s_1, s_2, s_3, \ldots$ is a list of all possible strings in $\Sigma^*$.

$E$ = "Ignore the input.

1. Repeat the following for $i = 1, 2, 3, \ldots$
2. Run $M$ for $i$ steps on each input, $s_1, s_2, \ldots, s_i$.
3. If any computations accept, print out the corresponding $s_j$."

If $M$ accepts a particular string $s$, eventually it will appear on the list generated by $E$. In fact, it will appear on the list infinitely many times because $M$ runs from the beginning on each string for each repetition of step 1. This procedure gives the effect of running $M$ in parallel on all possible input strings.

## THE DEFINITION OF ALGORITHM

Informally speaking, an *algorithm* is a collection of simple instructions for carrying out some task. Commonplace in everyday life, algorithms sometimes are called *procedures* or *recipes*. Algorithms also play an important role in mathematics. Ancient mathematical literature contains descriptions of algorithms for a variety of tasks, such as finding prime numbers and greatest common divisors. In contemporary mathematics algorithms abound.

Even though algorithms have had a long history in mathematics, the notion of algorithm itself was not defined precisely until the twentieth century. Before that, mathematicians had an intuitive notion of what algorithms were, and relied upon that notion when using and describing them. But that intuitive notion was insufficient for gaining a deeper understanding of algorithms. The following story relates how the precise definition of algorithm was crucial to one important mathematical problem.

### HILBERT'S PROBLEMS

In 1900, mathematician David Hilbert delivered a now-famous address at the International Congress of Mathematicians in Paris. In his lecture, he identified twenty-three mathematical problems and posed them as a challenge for the coming century. The tenth problem on his list concerned algorithms.

Before describing that problem, let's briefly discuss polynomials. A ***polynomial*** is a sum of terms, where each ***term*** is a product of certain variables and a

constant called a ***coefficient***. For example,

$$6 \cdot x \cdot x \cdot x \cdot y \cdot z \cdot z = 6x^3yz^2$$

is a term with coefficient 6, and

$$6x^3yz^2 + 3xy^2 - x^3 - 10$$

is a polynomial with four terms over the variables $x$, $y$, and $z$. For this discussion, we consider only coefficients that are integers. A ***root*** of a polynomial is an assignment of values to its variables so that the value of the polynomial is 0. This polynomial has a root at $x = 5$, $y = 3$, and $z = 0$. This root is an ***integral root*** because all the variables are assigned integer values. Some polynomials have an integral root and some do not.

Hilbert's tenth problem was to devise an algorithm that tests whether a polynomial has an integral root. He did not use the term *algorithm* but rather "a process according to which it can be determined by a finite number of operations."[4] Interestingly, in the way he phrased this problem, Hilbert explicitly asked that an algorithm be "devised." Thus he apparently assumed that such an algorithm must exist—someone need only find it.

As we now know, no algorithm exists for this task; it is algorithmically unsolvable. For mathematicians of that period to come to this conclusion with their intuitive concept of algorithm would have been virtually impossible. The intuitive concept may have been adequate for giving algorithms for certain tasks, but it was useless for showing that no algorithm exists for a particular task. Proving that an algorithm does not exist requires having a clear definition of algorithm. Progress on the tenth problem had to wait for that definition.

The definition came in the 1936 papers of Alonzo Church and Alan Turing. Church used a notational system called the $\lambda$-calculus to define algorithms. Turing did it with his "machines." These two definitions were shown to be equivalent. This connection between the informal notion of algorithm and the precise definition has come to be called the ***Church–Turing thesis***.

The Church–Turing thesis provides the definition of algorithm necessary to resolve Hilbert's tenth problem. In 1970, Yuri Matijasevič, building on work of Martin Davis, Hilary Putnam, and Julia Robinson, showed that no algorithm exists for testing whether a polynomial has integral roots. In Chapter 4 we develop the techniques that form the basis for proving that this and other problems are algorithmically unsolvable.

| *Intuitive notion of algorithms* | equals | *Turing machine algorithms* |
|---|---|---|

**FIGURE 3.22**
The Church–Turing Thesis

Let's phrase Hilbert's tenth problem in our terminology. Doing so helps to introduce some themes that we explore in Chapters 4 and 5. Let

$$D = \{p \mid p \text{ is a polynomial with an integral root}\}.$$

Hilbert's tenth problem asks in essence whether the set $D$ is decidable. The answer is negative. In contrast we can show that $D$ is Turing-recognizable. Before doing so, let's consider a simpler problem. It is an analog of Hilbert's tenth problem for polynomials that have only a single variable, such as $4x^3 - 2x^2 + x - 7$. Let

$$D_1 = \{p \mid p \text{ is a polynomial over } x \text{ with an integral root}\}.$$

Here is a TM $M_1$ that recognizes $D_1$:

$M_1$ = "The input is a polynomial $p$ over the variable $x$.

**1.** Evaluate $p$ with $x$ set successively to the values 0, 1, $-1$, 2, $-2$, 3, $-3$, ... If at any point the polynomial evaluates to 0, *accept*."

If $p$ has an integral root, $M_1$ eventually will find it and accept. If $p$ does not have an integral root, $M_1$ will run forever. For the multivariable case, we can present a similar TM $M$ that recognizes $D$. Here, $M$ goes through all possible settings of its variables to integral values.

Both $M_1$ and $M$ are recognizers but not deciders. We can convert $M_1$ to be a decider for $D_1$ because we can calculate bounds within which the roots of a single variable polynomial must lie and restrict the search to these bounds. In Problem 3.21 you are asked to show that the roots of such a polynomial must lie between the values

$$\pm k \frac{c_{\max}}{c_1},$$

where $k$ is the number of terms in the polynomial, $c_{\max}$ is the coefficient with largest absolute value, and $c_1$ is the coefficient of the highest order term. If a root is not found within these bounds, the machine *rejects*. Matijasevič's theorem shows that calculating such bounds for multivariable polynomials is impossible.

## 10.5 Linear Bounded Automata

While it is not possible to extend the power of the standard Turing machine by complicating the tape structure, it is possible to limit it by restricting the way in which the tape can be used. We have already seen an example of this with pushdown automata. A pushdown automaton can be regarded as a nondeterministic Turing machine with a tape that is restricted to being used like a stack. We can also restrict the tape usage in other ways; for example, we might permit only a finite part of the tape to be used as work space. It can be shown that this leads us back to finite automata (see Exercise 3 at the end of this section), so we need not pursue this. But there is a way of limiting tape use that leads to a more interesting situation: We allow the machine to use only that part of the tape occupied by the input. Thus, more space is available for long input strings than for short ones, generating another class of machines, the **linear bounded automata** (or **lba**).

A linear bounded automaton, like a standard Turing machine, has an unbounded tape, but how much of the tape can be used is a function of the input. In particular, we restrict the usable part of the tape to exactly the cells taken by the input.[1] To enforce this, we can envision the input as bracketed by two special symbols, the **left-end marker** [ and the **right-end marker** ]. For an input w, the initial configuration of the Turing machine is given by the instantaneous description $q_0\,[w]$. The end markers cannot be rewritten, and the read-write head cannot move to the left of [ or to the right of ]. We sometimes say that the read-write head "bounces" off the end markers.

**Definition 10.5**

A linear bounded automaton is a nondeterministic Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$, as in Definition 10.2, subject to the restriction that $\Sigma$ must contain two special symbols [ and ], such that $\delta(q_i, [)$ can contain only elements of the form $(q_j, [, R)$, and $\delta(q_i, ])$ can contain only elements of the form $(q_j, ], L)$.

**Definition 10.6**

A string $w$ is accepted by a linear bounded automaton if there is a possible sequence of moves

$$q_0\,[w] \overset{*}{\vdash} [x_1 q_f x_2]$$

for some $q_f \in F$, $x_1, x_2 \in \Gamma^*$. The language accepted by the lba is the set of all such accepted strings.

Note that in this definition a linear bounded automaton is assumed to be nondeterministic. This not just a matter of convenience but essential to the discussion of lba's.

**Example 10.5**

Find a linear bounded automaton that accepts the language

$$L = \{a^{n!} : n \geq 0\}.$$

One way to solve the problem is to divide the number of $a$'s successively by 2, 3, 4,..., until we can either accept or reject the string. If the input is in L, eventually there will be a single a left; if not, at some point a nonzero remainder will arise. We sketch the solution to point out one tacit implication of Definition 10.5. Since the tape of a linear bounded automaton may be multitrack, the extra tracks can be used as work space. For this problem, we can use a two-track tape. The first track contains the number of $a$'s left during the process of division, and the second track contains the current divisor (Figure 10.18). The actual solution is fairly simple. Using the divisor on the second track, we divide the number of a's on the first track, say by removing all symbols except those at multiples of the divisor. After this, we increment the divisor by one, and continue until we either find a nonzero remainder or are left with a single $a$.

**Figure 10.18**


The last two examples suggest that linear bounded automata are more powerful than pushdown automata, since neither of the languages is context-free. To prove such a conjecture, we still have to

show that any context-free language can be accepted by a linear bounded automaton. We will do this later in a somewhat roundabout way; a more direct approach is suggested in Exercises 6 and 7 at the end of this section. It is not so easy to make a conjecture on the relation between Turing machines and linear bounded automata. Problems like Example 10.5 are invariably solvable by a linear bounded automaton, since an amount of scratch space proportional to the length of the input is available. In fact, it is quite difficult to come up with a concrete and explicitly defined language that cannot be accepted by any linear bounded automaton. In Chapter 11 we will show that the class of linear bounded automata is less powerful than the class of unrestricted Turing machines, but a demonstration of this requires a lot more work.

## 11.3 Context-Sensitive Grammars and Languages

Between the restricted, context-free grammars and the general, unrestricted grammars, a great variety of "somewhat restricted" grammars can be defined. Not all cases yield interesting results; among the ones that do, the context-sensitive grammars have received considerable attention. These grammars generate languages associated with a restricted class of Turing machines, linear bounded automata, which we introduced in Section 10.5.

**Definition 11.4**

A grammar $G = (V, T, S, P)$ is said to be **context-sensitive** if all productions are of the form

$$x \rightarrow y,$$

where $x, y \in (V \cup T)^+$ and

$$|x| \leq |y|. \tag{11.15}$$

This definition shows clearly one aspect of this type of grammar; it is **noncontracting**, in the sense that the length of successive sentential forms can never decrease. It is less obvious why such grammars should be called context-sensitive, but it can be shown (see, for example, Salomaa 1973) that all such grammars can be rewritten in a normal form in which all productions are of the form

$$xAy \rightarrow x\upsilon y.$$

This is equivalent to saying that the production

$$A \rightarrow \upsilon$$

can be applied only in the situation where $A$ occurs in a context of the string $x$ on the left and the string $y$ on the right. While we use the terminology arising from this particular interpretation, the form itself

is of little interest to us here, and we will rely entirely on Definition 11.4.

## Context-Sensitive Languages and Linear Bounded Automata

As the terminology suggests, context-sensitive grammars are associated with a language family with the same name.

**Definition 11.5**

A language $L$ is said to be context-sensitive if there exists a context-sensitive grammar $G$, such that $L = L(G)$ or $L = L(G) \cup \{\lambda\}$.

In this definition, we reintroduce the empty string. Definition 11.4 implies that $x \rightarrow \lambda$ is not allowed, so that a context-sensitive grammar can never generate a language containing the empty string. Yet, every context-free language without $\lambda$ can be generated by a special case of a context-sensitive grammar, say by one in Chomsky or Greibach normal form, both of which satisfy the conditions of Definition 11.4. By including the empty string in the definition of a context-sensitive language (but not in the grammar), we can claim that the family of context-free languages is a subset of the family of context-sensitive languages.

**Example 11.2**

The language $L = \{a^n b^n c^n : n \geq 1\}$ is a context-sensitive language. We show this by exhibiting a context-sensitive grammar for the language. One such grammar is

$$\begin{aligned} S &\rightarrow abc|aAbc, \\ Ab &\rightarrow bA, \\ Ac &\rightarrow Bbcc, \\ bB &\rightarrow Bb, \\ aB &\rightarrow aa|aaA. \end{aligned}$$

We can see how this works by looking at a derivation of $a^3b^3c^3$.

$$\begin{aligned} S &\Rightarrow aAbc \Rightarrow abAc \Rightarrow abBbcc \\ &\Rightarrow aBbbcc \Rightarrow aaAbbcc \Rightarrow aabAbcc \\ &\Rightarrow aabbAcc \Rightarrow aabbBbccc \\ &\Rightarrow aabBbbccc \Rightarrow aaBbbbccc \\ &\Rightarrow aaabbbccc. \end{aligned}$$

The solution effectively uses the variables $A$ and $B$ as messengers. An $A$ is created on the left, travels

to the right to the first c, where it creates another $b$ and $c$. It then sends the messenger $B$ back to the left in order to create the corresponding $a$. The process is very similar to the way one might program a Turing machine to accept the language $L$.

---

Since the language in the previous example is not context-free, we see that the family of context-free languages is a proper subset of the family of context-sensitive languages. Example 11.2 also shows that it is not an easy matter to find a context-sensitive grammar even for relatively simple examples. Often the solution is most easily obtained by starting with a Turing machine program, then finding an equivalent grammar for it. A few examples will show that, whenever the language is context-sensitive, the corresponding Turing machine has predictable space requirements; in particular, it can be viewed as a linear bounded automaton.

## Theorem 11.8

For every context-sensitive language $L$ not including $\lambda$, there exists some linear bounded automaton $M$ such that $L = L(M)$.

**Proof:** If $L$ is context-sensitive, then there exists a context-sensitive grammar for $L - \{\lambda\}$. We show that derivations in this grammar can be simulated by a linear bounded automaton. The linear bounded automaton will have two tracks, one containing the input string $w$, the other containing the sentential forms derived using $G$. A key point of this argument is that no possible sentential form can have length greater than $|w|$. Another point to notice is that a linear bounded automaton is, by definition, non deterministic. This is necessary in the argument, since we can claim that the correct production can always be guessed and that no unproductive alternatives have to be pursued. Therefore, the computation described in Theorem 11.6 can be carried out without using space except that originally occupied by $w$; that is, it can be done by a linear bounded automaton. ■

## Theorem 11.9

If a language $L$ is accepted by some linear bounded automaton $M$, then there exists a context-sensitive grammar that generates $L$.

**Proof:** The construction here is similar to that in Theorem 11.7. All productions generated in Theorem 11.7 are non contracting except (11.13),

$$\square \rightarrow \lambda.$$

But this production can be omitted. It is necessary only when the Turing machine moves outside the bounds of the original input, which is not the case here. The grammar obtained by the construction without this unnecessary production is non contracting, completing the argument. ■