8 Post Machine

LEARNING OBJECTIVES

After completing this chapter, the reader will be able to understand the following:

- Introduction to Post machine and its finite control model
- Pictorial representation of Post machine
- Comparison of powers of pushdown stack-memory machine (PDM) and Post machine (PM)
- Comparison of powers of finite state machine (FSM) and Post machine (PM)
- Equivalence of Post machine (PM) and Turing machine (TM)

8.1 INTRODUCTION

In Chapter 6, we have seen the elements and working of a pushdown stack-memory machine (PDM). The PDM uses a stack as an external memory, which is an improvement over the finite state machine (FSM). We also know that the power of the PDM is intermediate between the FSM and the Turing machine (TM).

There is one more machine comparable to the TM, which is called the Post machine (PM). This machine can accept the set of languages accepted by the FSM, PDM, as well as the TM. The major difference between the PDM and the PM, apart from the fact that the latter is more powerful, is that the PM uses queue data structure as memory, while the PDM uses external stack.

The power of the PM is equivalent to that of the TM; the only difference is in the representation.

8.2 ELEMENTS OF POST MACHINE

A PM is collectively defined by the following elements:

1. A tape, which is bounded on one end and unbounded (or infinite) on the other. Initially, the input string is written onto the tape, using a special character '#' to indicate the end; the rest of the tape is blank ($\not b$)

2. A finite set of allowable tape symbols (Γ) including '#' and blank character $\not b$
3. A finite input alphabet Σ, which is a subset of the set of allowable tape symbols, that is, $\Sigma \subset \Gamma$
4. A start (or initial) state
5. Halt states: 'accept' and 'reject'
6. A non-branching state: 'add'
7. A branching state: 'read'
8. A tape implemented as a queue data structure

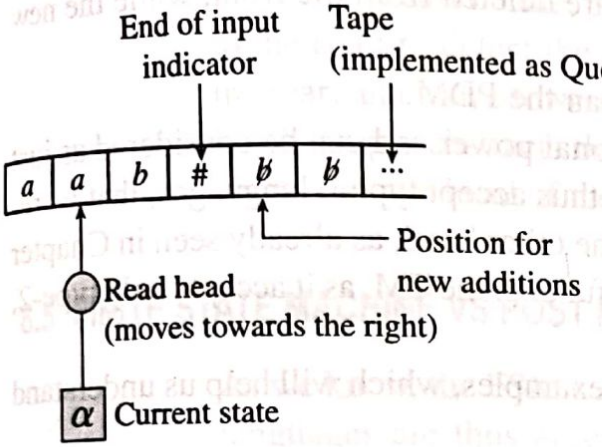


Figure 8.1 Finite control representation for PM

Thus, in a PM the input is written onto its tape; the machine reads the tape symbols and, if necessary, adds symbols to the end of the input string (rear). This means that the PM can write symbols onto its tape just as the TM does. However, the major difference between the TM and the PM is that the head of the PM is allowed to move towards the right only. It cannot move towards the left, or remain in the same position like in the TM (refer to Fig. 8.1).

Note: Though the head of the PM can only move in one direction; it can add symbols to the rear of the tape. This means that the mechanism of moving to the right till the end of the input, adding symbols to the end, and coming back to read the next input symbol, is kept implicit. Addition of the symbol is thus implemented as a subroutine called 'add'. However, we know that internally, this requires moving in both the directions. Hence, the PM is simply an alternative visualization of the TM that implements its tape as a queue of symbols.

8.3 PUSHDOWN STACK-MEMORY MACHINE VS POST MACHINE

As we have already discussed, the PM uses queue data structure, while the PDM uses stack data structure as its external memory. Since the stack is external to the PDM, the stack and tape are two distinct entities. On the other hand, the PM uses its tape as queue, and can add new symbols at the end of the input string, which is indicated by the '#' symbol; hence, in a PM, the tape and queue are not distinct entities—the tape is implemented as its queue. Essentially, the PM consumes its own output (symbols added to the queue) as an input for use a later instance just as a TM does. The PDM, however, cannot consume the stack as an input, as it is external to its tape.

Since the PM uses its tape as queue, it has the ability to add symbols to the tape while executing. This means that the PM can write symbols onto its tape. On the other hand, the PDM cannot write anything onto its tape; it can only read from the tape, because it has access to an external stack for storing symbols. The PDM, thus, has a read head only, whereas, the PM has a read/write head.

In the PM, the head is perceived to read from left to right and traverse only in one direction. However, in reality, the addition of more symbols to the rear of the tape is kept implicit.

This means that the head can traverse to the extreme right till end of input (#) is received, add some symbols, and traverse back from right to left to point to the new symbol that is to be read from the tape. Thus, the head of the PM head can move in any direction in reality, while the head of the PDM can only move from left to right. The only restriction the PM has is that it is not allowed to read the symbols, which are already read. The symbols that are read are considered as deleted, as the tape is implemented as a queue—first-in-first-out (FIFO)—data structure. In queue, the elements are deleted from the front, while the new elements are added at the rear.

Thus, we see that the PM is more powerful than the PDM.

The PM is equivalent to the TM in computational power and can be considered as just another implementation of the TM. The PM can thus accept type-0 languages, that is, recursively enumerable languages. The PDM, on the other hand, as already seen in Chapter 6, is more powerful than the FSM, but less powerful than the TM, as it accepts only type-2, that is, context-free languages (CFLs).

In the subsequent sections, we shall see some examples, which will help us understand this better.

8.4 PICTORIAL REPRESENTATION OF POST MACHINE ELEMENTS

We can represent the PM with the help of a flow-chart-like notation similar to the PDM. The pictorial representation of the different PM elements is shown in Fig. 8.2.

We see that the start block denotes the initial state of the PM, that is, the entry point.

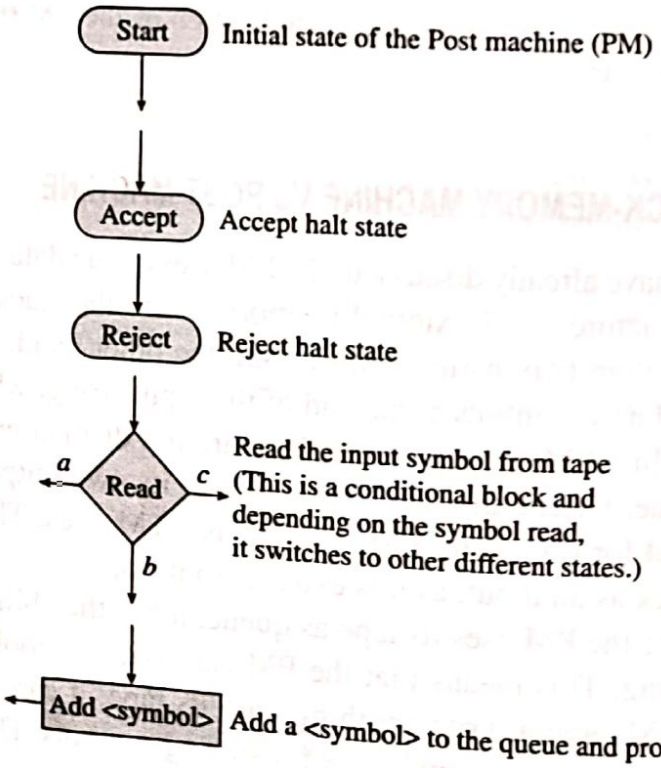


Figure 8.2 Pictorial representation of PM elements

The 'accept' and 'reject' blocks respectively represent the accept halt and reject halt states, depending on whether the machine has accepted or rejected the input string.

The 'read' block is a conditional block, and, depending on the symbol read, the machine makes transitions to different states.

The 'add' block is a process block and must be provided along with a symbol. This symbol gets added to the end of the queue, that is, after the '#' symbol. The add block is more like a subroutine, as discussed earlier, which abstracts the following steps—move to the right to detect the end of input (#); write the symbol to be added onto the tape at the rear; and move towards the left to point to the next symbol that is to be read. The add block simply abstracts the head movement in the direction—from right to left, though it seems that the head of the PM can only move in one direction, that is from left to right.

8.5 FINITE STATE MACHINE VS POST MACHINE

As we know, the FSM can only accept regular languages (or type-3 languages). Regular languages are thus acceptable by machines such as the FSM, which does not have a memory. The same set of regular languages can also be accepted using the PM. However, the PM is more powerful than the FSM, as the former has infinite memory in the form of its unbounded tape. The PM can remember any large string of symbols with the help of the internal queue (tape), which is a first-in-first-out (FIFO) type of data structure. Anyway, the queue (or memory) is not required for accepting or recognizing any regular language.

Thus, we see that the PM, being equivalent to the TM, is much powerful machine than the FSM.

Let us discuss some examples of PMs that accept regular languages.

Example 8.1 Construct a PM that recognizes the language accepted by the DFA shown in Fig. 8.3.

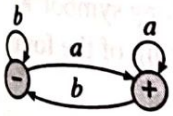


Figure 8.3 An example DFA

Solution The PM equivalent to DFA in Fig. 8.3 can be constructed as shown in Fig. 8.4.

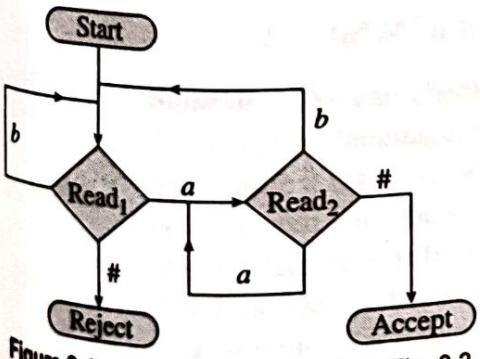


Figure 8.4 PM equivalent to DFA in Fig. 8.3

We see that state $Read_1$ is analogous to the initial state of the given DFA, and state $Read_2$ is analogous to the final state of the given DFA. The only difference is that acceptance and rejection are clearly specified in the PM, which is not made explicit in the given DFA.

In state $Read_1$, the PM consumes any number of b's and does not change the state till it reads the symbol a. Upon reading a, it makes a transition to state $Read_2$. In this state, the PM keeps consuming the a's till it reads symbol b. Upon reading b, it makes a transition back to state $Read_1$. If the end of input indicator '#' is received in the final state, that is, $Read_2$, then the input is accepted; else, it is rejected.

Note: The DFA shown in Fig. 8.3 and the PM in Fig. 8.4 are equivalent machines, since they accept the same language. In this example, the PM does not use queue to store anything, as the language being accepted is regular. As we know, regular languages are accepted by the DFA, as they require no memory. Thus, Fig. 8.4 is more like a DFA expressed in the form of a PM. Further, if we compare the PM in Fig. 8.4 with the PDA in Fig. 6.4 (refer to Chapter 6, Section 6.4.1), we observe that they are the same. This is because neither of the two machines have used a stack or a queue, while accepting the given regular language. We can also say that, the diagrams drawn in Fig. 8.4 and Fig. 6.3 are only DFAs that are represented using the pictorial representation scheme.

Obviously, if we delete stack from the PDA and queue from the PM, we will get the machine without an external memory, which is nothing but the FA.

Example 8.2 Construct a PM that accepts the following language:

$$L = \{a^n\, b^m \mid n \geq 0, m \geq 0\}$$

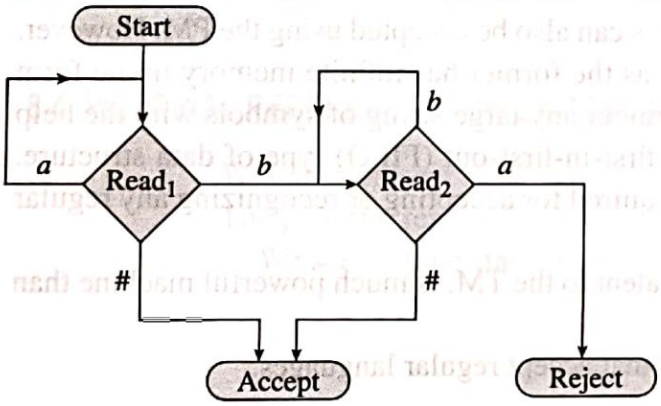


Figure 8.5 PM accepting $\{a^n\, b^m \mid n, m \geq 0\}$

Solution We see that the given language is a regular language and can be represented by the regular expression $a^* \cdot b^*$. The language contains all the strings with any number of a's, followed by any number of b's. Thus, the PM that accepts the given language is a simple machine that does not require memory, as shown in Fig. 8.5.

We see from Fig. 8.5 that the PM also accepts the minimal length string—for $n = 0$, $m = 0$, which is nothing but an empty string. An empty string is accepted by the machine when it reads '#' in the state $Read_1$. This state consumes all the a's before it transits to state $Read_2$ on reading symbol b. State $Read_2$ consumes all the remaining b's before it accepts the string on reading symbol '#'.

If a follows b in the input, then the PM rejects the input string, as the string of the form $a^n b^m$ cannot have a b followed by an a.

Simulation

1. Let us simulate the acceptance of the string '*aab*'.

Current state	Tape and head position
Start ↓	a a b # ∅ ∅ … ↑
$Read_1$ a ↓	a a b # ∅ ∅ … ↑
$Read_1$ a ↓	a a b # ∅ ∅ … ↑
$Read_1$ b ↓	a a b # ∅ ∅ … ↑
$Read_2$ # ↓	a a b # ∅ ∅ … ↑
Accept	

2. Let us simulate the working of the machine for the input string '*b*' (i.e., for $n = 0$ and $m = 1$).

Current state	Tape and head position
Start ↓	*b* # ɸ ɸ … ↑
Read$_1$ *b*↓	*b* # ɸ ɸ … ↑
Read$_2$ #↓	*b* # ɸ ɸ … ↑
Accept	

3. Let us simulate the rejection of the string '*aba*'.

Current state	Tape and head position
Start ↓	*a b a* # ɸ ɸ … ↑
Read$_1$ *a*↓	*a b a* # ɸ ɸ … ↑
Read$_1$ *b*↓	*a b a* # ɸ ɸ … ↑
Read$_2$ *a*↓	*a b a* # ɸ ɸ … ↑
Reject	

8.6 PM ACCEPTING CFLs

As already mentioned, the PM can accept all regular languages that are accepted by any FSM. However, since the PM has *unlimited memory* in the form of queue, it can be used to remember arbitrarily long strings of symbols. Hence, it can also solve problems such as checking if the given parentheses are well-formed, and accept most CFLs.

Example 8.3 Construct a PM that recognizes the following CFL:

$$L = \{a^n b^n \mid n \geq 0\}$$

Solution We can check using the pumping lemma, whether or not the given language L is a regular language, and hence, whether or not we can have an FA that recognizes it. We require a machine, which is capable of remembering the string of a's so that they can be compared with the string of b's to check if the number of a's and b's are equal. Recall that in Chapter 6 (refer to Section 6.5), we already have solved this problem using PDA, which uses stack data structure. Let us now construct a PM, which uses its tape as a queue data structure, for remembering arbitrarily long strings.

We further know that the PM can write onto its tape in a restricted fashion—it can only add more symbols to the end of the input string and cannot overwrite or delete the previous contents as the TM does.

Algorithm

1. After reading the string of a's, skip the first a and add all the remaining a's to the queue. Similarly, skip the first b and add the remaining b's to the queue.
2. While reading, if you get '#', representing the end of the input, then add '#' to the end of newly added string of a's and b's, which will be in the form:

$$a^1 a^2 \ldots a^{n-1} b^1 b^2 \ldots b^{n-1}$$

3. Thus, we are deleting one a and one b to compare or match the number of a's with b's, as we want to check equality of the number of a's and b's.
4. Repeat the aforementioned procedure for the newly added string with $(n-1)$ number of a's and $(n-1)$ number of b's. Keep cancelling one a and one b each time. If the number of a's is equal to the number of b's, then all of them will be finally exhausted.

The PM following this algorithm is as shown in Fig. 8.6.

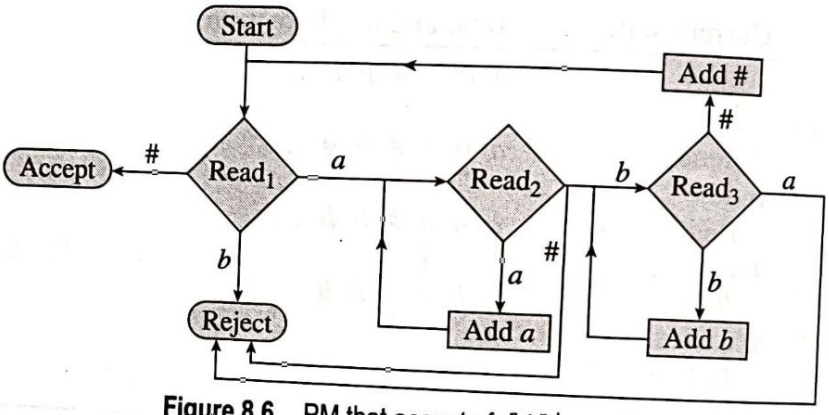


Figure 8.6 PM that accepts $\{a^n b^n \mid n \geq 0\}$

The various decision boxes function as follows:

$Read_1$: Skip one a.
If # is read, then go to 'Accept'.
If b is a starting symbol, then go to 'Reject'.

$Read_2$: Add all remaining a's to the queue.
Skip one b.

$Read_3$: Add remaining b's to the queue.
If '#' is read, then add '#' to the queue and repeat.
If a follows b in the input string, then go to 'Reject'.

We see that state $Read_1$ is associated with three different functions: It reads the first a and moves to state $Read_2$ without adding it to the queue. Thus, it skips the first a read. On reading '#' from the input tape, it accepts the input. Thus, it also accepts an empty input with zero number of a's and zero number of b's (i.e., $n = 0$), along with the correct input for $n > 0$.

If state $Read_1$ reads b first, it rejects the input, because a string of the form $a^n b^n$ cannot start with b.

State $Read_2$ skips the first b to match with the a skipped by $Read_1$. It queues the remaining a's to be processed later.

State $Read_3$ is also associated with three different functionalities, depending on the input symbol read. It adds the remaining b's to the queue for later processing. If it reads

'#', then it adds '#' (end marker to the newly added string) to the queue. On reading *a*, it rejects the input because in a string of the form $a^n b^n$, *a* cannot follow *b*.

Simulation

1. Let us simulate the acceptance of the string '*aabb*'.

Current state	Tape as queue with head
Start ↓	*a a b b # ∅ ∅ …* ↑
$Read_1$ *a* ↓	*a a b b # ∅ ∅ …* ↑
	(skip one *a*)
$Read_2$ *a* ↓	*a a b b # ∅ ∅ …* ↑
Add *a* ↓	*a a b b # a ∅ …* ↑
	(add remaining *a* to the queue)
$Read_2$ *b* ↓	*a a b b # a ∅ …* ↑
	(skip one *b*)
$Read_3$ *b* ↓	*a a b b # a ∅ …* ↑
Add *b* ↓	*a a b b # a b ∅ …* ↑
	(add remaining *b* to the queue)
$Read_3$ # ↓	*a a b b # a b ∅ …* ↑
Add # ↓	*a a b b # a b # ∅ …* ↑
	(add # to the queue)
	(repeat the procedure for the newly added string)
$Read_1$ *a* ↓	*a a b b # a b # ∅ …* ↑
	(skip one *a*)
$Read_2$ *b* ↓	*a a b b # a b # ∅ …* ↑
	(skip one *b*)
$Read_3$ # ↓	*a a b b # a b # ∅ …* ↑
Add # ↓	*a a b b # a b # # ∅ …* ↑
	(add '#' to the queue)
	(repeat the procedure for the newly added string)
$Read_1$ # ↓	*a a b b # a b # # ∅ …* ↑
Accept	

Note: The arrows on the left-hand side indicate the transitions from one state to another on reading some symbol. All the states are shown in *italics*. Add is not a state but a process and hence is not in italics.

2. Let us simulate the rejection of the input string '*aab*'.

Current state	Tape as queue with head
Start ↓	*a a b # ∅ ∅ …* ↑
Read$_1$ *a* ↓	*a a b # ∅ ∅ …* ↑
	(skip one *a*)
Read$_2$ *a* ↓	*a a b # ∅ ∅ …* ↑
Add *a* ↓	*a a b # a ∅ …* ↑
	(add the remaining *a* to the queue)
Read$_2$ *b* ↓	*a a b # a ∅ …* ↑
	(skip one *b*)
Read$_3$ # ↓	*a a b # a ∅ …* ↑
Add # ↓	*a a b # a # ∅ …* ↑
	(repeat the procedure for the newly added string)
Read$_1$ *a* ↓	*a a b # a # ∅ …* ↑
Read$_2$ # ↓	*a a b # a # ∅ …* ↑
Reject	

3. Simulation of rejection of the input string '*abb*':

Current state	Tape as queue with head
Start ↓	*a b b # ∅ ∅ …* ↑
Read$_1$ *a* ↓	*a b b # ∅ ∅ …* ↑
	(skip one *a*)
Read$_2$ *b* ↓	*a b b # ∅ ∅ …* ↑
	(skip one *b*)
Read$_3$ *b* ↓	*a b b # ∅ ∅ …* ↑
Add *b* ↓	*a b b # b ∅ …* ↑

(add the remaining b to the queue)

$Read_3$	$a\ b\ b\ \#\ b\ \not{b}\ \ldots$
$\#\downarrow$	$\uparrow$
Add #	$a\ b\ b\ \#\ b\ \#\ \not{b}\ \ldots$
$\downarrow$	$\uparrow$

(add '#' to the queue)

(repeat the procedure for the newly added string)

$Read_1$	$a\ b\ b\ \#\ b\ \#\ \not{b}\ \ldots$
$b\downarrow$	$\uparrow$
Reject	

4. Let us simulate the rejection of the string '*aba*'.

Current state	Tape as queue with head
Start	$a\ b\ a\ \#\ \not{b}\ \not{b}\ \ldots$
$\downarrow$	$\uparrow$
$Read_1$	$a\ b\ a\ \#\ \not{b}\ \not{b}\ \ldots$
$a\downarrow$	$\uparrow$
	(skip one a)
$Read_2$	$a\ b\ a\ \#\ \not{b}\ \not{b}\ \ldots$
$b\downarrow$	$\uparrow$
	(skip one b)
$Read_3$	$a\ b\ a\ \#\ \not{b}\ \not{b}\ \ldots$
$a\downarrow$	$\uparrow$
Reject	

Example 8.4 Design a PM that checks if the given string contains well-formed parentheses.

Solution As we know, every string of well-formed parentheses should start with a '(' and should end with a ')'. Let us apply a similar logic as that in the previous example (Example 8.3).

Algorithm

When the machine reads an opening parenthesis '(', it skips the first one and adds the remaining opening parentheses '(' to the queue. Similarly, when it reads a closing parenthesis ')', it skips the first one and adds the remaining to the queue. Thus, the machine matches and deletes each opening parenthesis with one closing parenthesis. This process is repeated until there are no more symbols in the queue. The required PM is constructed as shown in Fig. 8.7.

Every state is associated with some functionality, which is stated as follows:

$Read_1$: Skip one '('.
If string starts with ')', then go to 'Reject'.
If symbol read is '#', then go to 'Accept'.

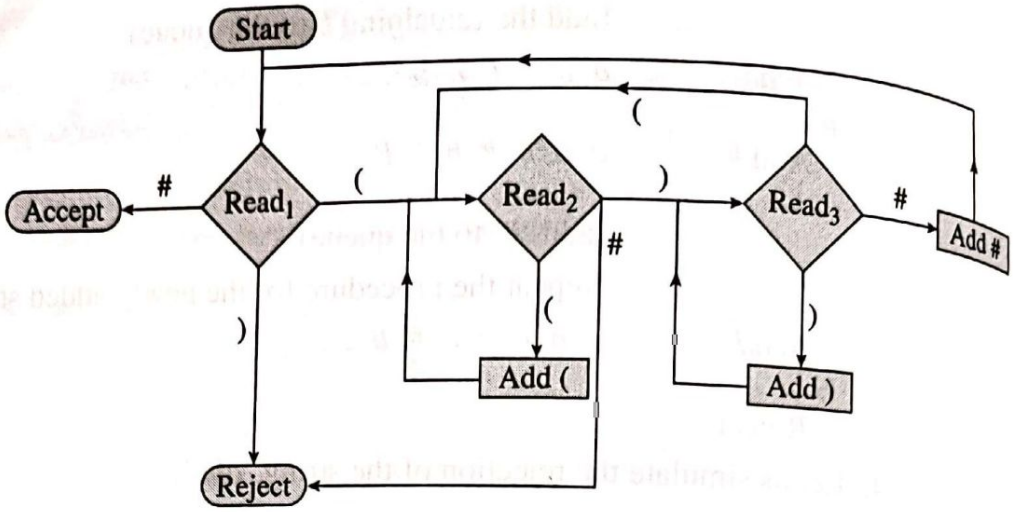


Figure 8.7 PM that checks for well-formed parentheses

Read$_2$: Add the remaining '('s.
Skip one ')'.
If the symbol read is '#', then go to 'Reject'.
Read$_3$: Add the remaining ')'s.
Skip one '('.
If symbol read is '#, then add '#' to the queue and repeat the procedure.

Simulation

1. Let us trace the acceptance of the string '(( )) ( )'.

Current state	Tape as queue and head
Start	(( )) ( ) # ß ß ...
↓	↑
Read$_1$	(( )) ( ) # ß ß ...
(↓	↑
	[skip one '(']
Read$_2$	(( )) ( ) # ß ß ...
(↓	↑
Add (	(( )) ( ) # (ß ...
↓	↑
	[add the remaining ('s to the queue]
Read$_2$	(( )) ( ) # (ß ...
) ↓	↑
	[skip one ')']
Read$_3$	(( )) ( ) # (ß ...
) ↓	↑
Add)	(( )) ( ) # ( ) ß ...
↓	↑
	[add the remaining)'s to the queue]
Read$_3$	(( )) ( ) # ( ) ß ...
(↓	↑

	[skip one '(']
$Read_2$	(( )) ( ) # ( ) ß ...
)↓	↑
	[skip one ')']
$Read_3$	(( )) ( ) # ( ) ß ...
#↓	↑
Add #	(( )) ( ) # ( ) # ß ...
↓	↑
	[add end marker to the newly added string and repeat the procedure]
$Read_1$	(( )) ( ) # ( ) # ß ...
(↓	↑
	[skip one ')']
$Read_2$	(( )) ( ) # ( ) # ß ...
)↓	↑
	[skip one ')']
$Read_3$	(( )) ( ) # ( ) # ß ...
#↓	↑
Add #	(( )) ( ) # ( ) # # ß ...
↓	↑
	[add end marker to the newly added string and repeat the procedure]
$Read_1$	(( )) ( ) # ( ) # # ß ...
#↓	↑
Accept	

2. Let us simulate the acceptance of the string '( ) ( )'.

Current state	Tape as queue and head
Start	( ) ( ) # ß ß ...
↓	↑
$Read_1$	( ) ( ) # ß ß ...
(↓	↑
	[skip one '(']
$Read_2$	( ) ( ) # ß ß ...
)↓	↑
	[skip one ')']
$Read_3$	( ) ( ) # ß ß ...
(↓	↑

	[skip one '(']
$Read_2$	( ) ( ) # ⊬ ⊬ …
)↓	↑
	[skip one ')']
$Read_3$	( ) ( ) # ⊬ ⊬ …
#↓	↑
Add #	( ) ( ) # # ⊬ …
↓	↑
	[add end marker to the newly added string and repeat the procedure]
$Read_1$	( ) ( ) # # ⊬ …
#↓	↑
Accept	

We see that only when more than one consecutive (or) occurs in the input string, they get added to the queue. Otherwise, only '#' is added to the queue, and the rest of the pairs are skipped in the process.

3. Let us simulate the rejection of the string '(( )'.

Current state	Tape as queue and head
Start	(( ) # ⊬ ⊬ …
↓	↑
$Read_1$	(( ) # ⊬ ⊬ …
(↓	↑
	[skip one '(']
$Read_2$	(( ) # ⊬ ⊬ …
(↓	↑
Add (	(( ) # (⊬ …
↓	↑
	[add remaining '(' to the queue]
$Read_2$	(( ) # (⊬ …
)↓	↑
	[skip one ')']
$Read_3$	(( ) # (⊬ …
#↓	↑
Add #	(( ) # (# ⊬ …
↓	↑

[add end marker to the newly added string
and repeat the procedure]

$Read_1$	(( ) # (# ∅ …
(↓	↑
	[skip one '(']
$Read_2$	(( ) # (# ∅ …
#↓	↑
Reject	

In the given string '(( )', one ')' is less; so the string is not well-formed, and is hence, rejected.

4. Let us trace the rejection of the string '( ))'. This string has one less opening parenthesis '('.

Current state	Tape as queue and head
Start	( )) # ∅ ∅ …
↓	↑
$Read_1$	( )) # ∅ ∅ …
(↓	↑
	[skip one '(']
$Read_2$	( )) # ∅ ∅ …
)↓	↑
	[skip one ')']
$Read_3$	( )) # ∅ ∅ …
)↓	↑
Add)	( )) #) ∅ …
↓	↑
	[add remaining '(' to the queue]
$Read_3$	( )) #) ∅ …
#↓	↑
Add #	(( ) #) # ∅ …
↓	↑
	[add end marker to the newly added string and repeat the procedure]
$Read_1$	(( ) #) # ∅ …
)↓	↑
Reject	

5. An empty input, that is, string with length zero, is a special case of the input that is well-formed. Let us trace the acceptance of the empty string. In this case, the tape will contain only '#', the end marker, as the start symbol; the tape head initially points to it.

Current state	Tape as queue and head
Start	# ♭ ♭ ...
↓	↑
	(empty input)
$Read_1$	# ♭ ♭ ...
#↓	↑
Accept	

Thus, the empty string ϵ is also considered to be well-formed and is accepted by the PM in Fig. 8.7.

***Note*:** As we have already discussed in Chapter 6, the context-free grammar (CFG) for this language—with strings of well-formed parentheses—can be defined as follows:

$$S \rightarrow (S)\, S \mid \epsilon$$

We see that ϵ can be derived from the start symbol S. Hence, the PM also accepts CFLs.

Similarly, we have seen that the PM in Example 8.3 accepted the language, $L = \{a^n b^n \mid n \geq 0\}$, which is also a context-free language; the CFG for this can be written as:

$$S \rightarrow a\, S\, b \mid \epsilon$$

Thus, we have demonstrated that PMs can accept regular languages as well as CFLs. Let us now discuss more complex examples.

8.7 NON-DETERMINISTIC POST MACHINE

A *non-deterministic PM* (NPM), after reading a single symbol, makes a transition to any of the multiple possible next states. The NPM is thus a possibilistic machine; and unless executed, one cannot really predict the behaviour of such machines. Let us illustrate this concept through the following example.

Example 8.5 Design a PM to accept palindrome strings over the alphabet, $\Sigma = \{a, b\}$.

Solution We can design a non-deterministic PM (NPM) for this problem. Each time, we check for the starting and ending symbols of the input string. If these are equal, then the algorithm proceeds with the rest of the symbols; else, an error is produced.

Recall that in Chapter 6 (refer to Section 6.6, Example 6.6) we have already constructed an NPDA for the same language.

Algorithm

As the input string is defined over $\Sigma = \{a, b\}$, it can either start with symbol a or with symbol b. If it starts with a, then it should end in a, in order to be a palindrome string.

Similarly, if it starts with b, it should end in b. The algorithm steps can be stated as follows:

1. Skip the first symbol, which is to be matched with the ending symbol, and add all the intermediate symbols to the queue.
2. Check the last symbol; if it is same as the first symbol, skip it as well.
3. Repeat the procedure for the newly added intermediate symbols till you get a string with zero symbols.

For example, consider the string '*aabbaa*'. In the first iteration, the first and last *a*'s are cancelled, and the remaining intermediate characters, that is, '*abba*', are added to the queue. In the second iteration, again the first and last *a*'s are cancelled and the remaining string, that is, '*bb*', is added to the queue. In the next iteration, the string '*bb*' is also cancelled, leaving an empty string. Finally, the machine accepts it as a palindrome string.

The NPM is constructed as shown in Fig. 8.8. We see from the figure that $Read_1$ makes a transition to $Read_2$ and $Read_4$, based on whether the beginning symbol is *a* or *b*, respectively. State $Read_6$ is reached only when there is a single symbol in the string—typically, a scenario for odd-length palindromes. Let us simulate the working of the PM to understand it better.

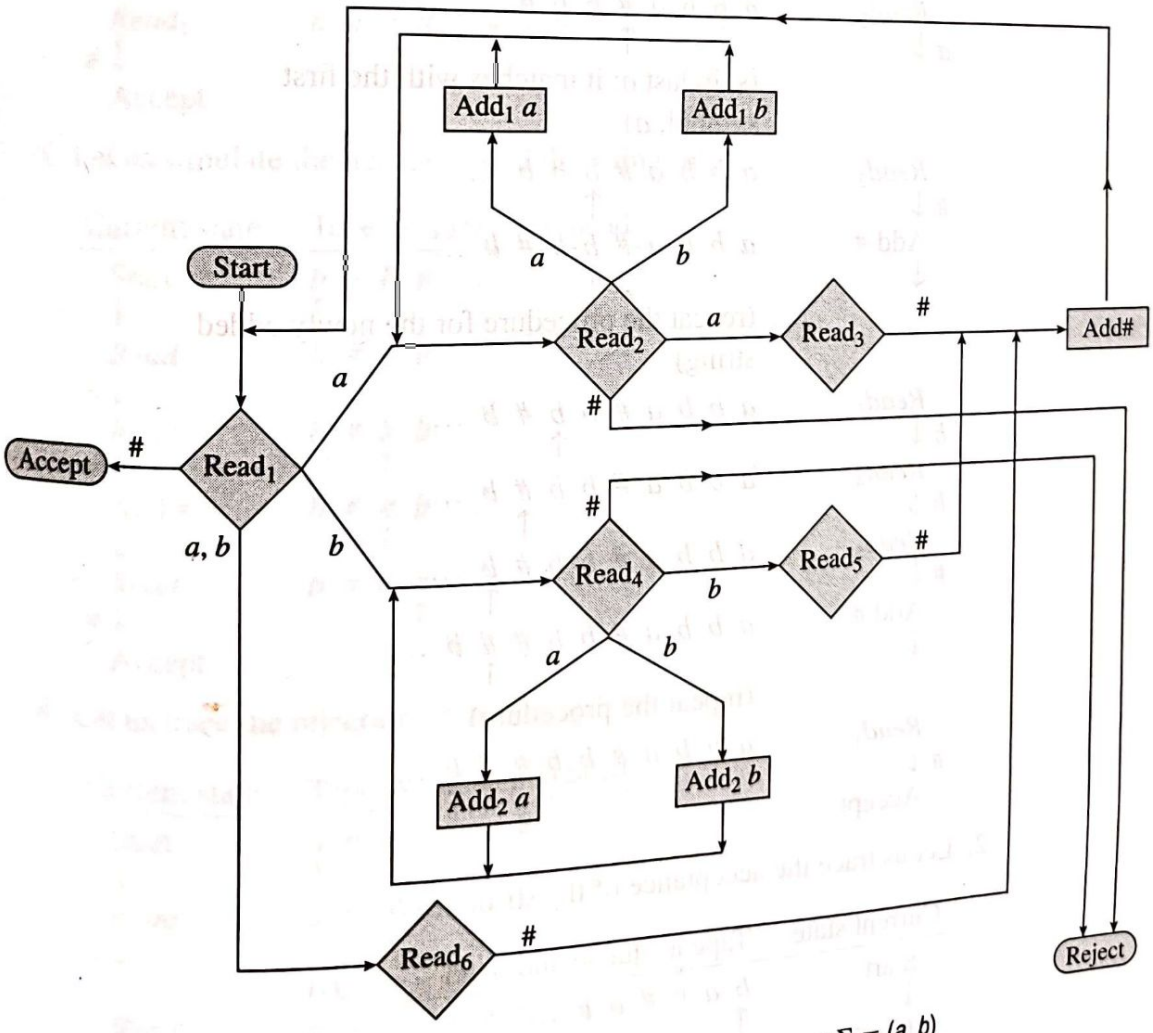


Figure 8.8 NPM that accepts palindrome strings over $\Sigma = (a, b)$

Simulation

1. Let us trace the acceptance of the input string '*abba*'.

Current state	Tape as queue and head
Start ↓	*a b b a # ∅ ∅ ∅ …* ↑
Read$_1$ *a* ↓	*a b b a # ∅ ∅ ∅ …* ↑
	(skip first *a*)
Read$_2$ *b* ↓	*a b b a # ∅ ∅ …* ↑
Add$_1$ *b* ↓	*a b b a # b ∅ …* ↑
Read$_2$ *b* ↓	*a b b a # b ∅ …* ↑
Add$_1$ *b* ↓	*a b b a # b b ∅ …* ↑
	(add the remaining intermediate string '*bb*')
Read$_2$ *a* ↓	*a b b a # b b ∅ …* ↑
	(skip last *a*; it matches with the first symbol, *a*)
Read$_3$ # ↓	*a b b a # b b ∅ …* ↑
Add # ↓	*a b b a # b b # ∅ …* ↑
	(repeat the procedure for the newly added string)
Read$_1$ *b* ↓	*a b b a # b b # ∅ …* ↑
Read$_4$ *b* ↓	*a b b a # b b # ∅ …* ↑
Read$_5$ # ↓	*a b b a # b b # ∅ …* ↑
Add # ↓	*a b b a # b b # # ∅ …* ↑
	(repeat the procedure)
Read$_1$ # ↓	*a b b a # b b # # ∅ …*
Accept	↑

2. Let us trace the acceptance of the string '*bab*'.

Current state	Tape as queue and head
Start ↓	*b a b # ∅ ∅ …* ↑
Read$_1$ *b* ↓	*b a b # ∅ ∅ …* ↑

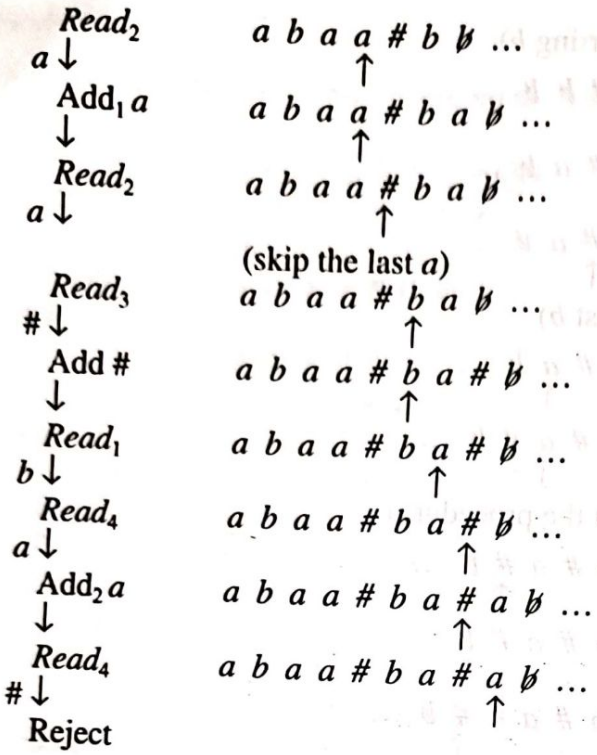
	(skip starting b)
$Read_4$	$b\ a\ b\ \#\ \not b\ \not b\ \ldots$
$a\downarrow$	
$Add_2\ a$	$b\ a\ b\ \#\ a\ \not b\ \ldots$
$\downarrow$	
$Read_4$	$b\ a\ b\ \#\ a\ \not b\ \ldots$
$b\downarrow$	
	(skip last b)
$Read_5$	$b\ a\ b\ \#\ a\ \not b\ \ldots$
$\#\downarrow$	
Add #	$b\ a\ b\ \#\ a\ \#\ \not b\ \ldots$
$\downarrow$	
	(repeat the procedure)
$Read_1$	$b\ a\ b\ \#\ a\ \#\ \not b\ \ldots$
$a\downarrow$	
$Read_6$	$b\ a\ b\ \#\ a\ \#\ \not b\ \ldots$
$\#\downarrow$	
Add #	$b\ a\ b\ \#\ a\ \#\ \#\ \not b\ \ldots$
$\downarrow$	
$Read_1$	$b\ a\ b\ \#\ a\ \#\ \#\ \not b\ \ldots$
$\#\downarrow$	
Accept	

3. Let us simulate the acceptance of the string 'b'.

Current state	Tape as queue and head
Start	$b\ \#\ \not b\ \not b\ \ldots$
$\downarrow$	
$Read_1$	$b\ \#\ \not b\ \not b\ \ldots$
$b\downarrow$	
$Read_6$	$b\ \#\ \not b\ \not b\ \ldots$
$\#\downarrow$	
Add #	$b\ \#\ \#\ \not b\ \ldots$
$\downarrow$	
$Read_1$	$b\ \#\ \#\ \not b\ \ldots$
$\#\downarrow$	
Accept	

4. Let us trace the rejection of the string '$abaa$'.

Current state	Tape as queue and head
Start	$a\ b\ a\ a\ \#\ \not b\ \not b\ \ldots$
$\downarrow$	
$Read_1$	$a\ b\ a\ a\ \#\ \not b\ \not b\ \ldots$
$a\downarrow$	
	(skip the first a)
$Read_2$	$a\ b\ a\ a\ \#\ \not b\ \not b\ \ldots$
$b\downarrow$	
$Add_1\ b$	$a\ b\ a\ a\ \#\ b\ \not b\ \ldots$
$\downarrow$	



Note: Observe that the aforementioned PM is non-deterministic. In every non-deterministic machine, there are multiple possible paths for every input string, beginning with the start state. If the input is valid, then out of these possible paths only one leads to acceptance of the string, while the others lead to rejection of the string. Such a machine is called a *possibilistic machine.*

8.8 POST MACHINE ACCEPTING NON-CFLs

In the previous section, we constructed PMs that accept regular languages and CFLs such as, $\{a^n b^n \mid n \geq 0\}$ and well-formed parentheses. In this section, let us demonstrate through some complex examples that PMs also accept non-CFLs. We also demonstrate that the PM can solve complex problems such as multiplication, proving that the PM is equivalent in power to the TM.

Example 8.6 Construct a PM that accepts the following language:

$$L_1 = \{a^n b^n a^n \mid n \geq 0\}$$

Solution The given language is not a CFL. Therefore, we cannot write a CFG for this language; we also cannot construct a PDA, because they accept only CFLs. However, we can construct a PM that accepts the given language $L_1 = \{a^n b^n a^n \mid n \geq 0\}$, as shown in Fig. 8.9.

We see that state $Read_1$ skips one a and makes a transition to state $Read_2$, which adds the remaining a's to the queue till it reads the first b. State $Read_2$ skips the first b to match with the skipped a and makes a transition to state $Read_3$. State $Read_3$ adds the remaining