6.2 ELEMENTS OF A PDM

A PDM is collectively described with the help of the following elements:

1. An input alphabet, (Σ), that is, a finite set of input symbols.
2. An unbounded input tape, bounded only by the input length. The input string *is written* onto the tape, which is initially assumed to contain blank characters ($\not{b}$). The *left end* of the input tape is fixed and the tape is unbounded towards the right end.
3. An alphabet of stack symbols (Γ).
4. A pushdown stack. Initially the stack is empty and is assumed to contain a blank character $\not{b}$ at the bottom to represent the stack empty position.
5. Start, accept, and reject indicators.
6. Branching state—read.
7. Stack operations, namely push and pop.

The input tape of a PDM is visualized to be divided into cells having one input symbol from Σ written into each—a finite control (head assembly that moves one cell at a time towards the right upon reading a symbol from the tape) and an external stack, which is an external memory. Figure 6.1 shows a pictorial representation of the memory of a PDM.

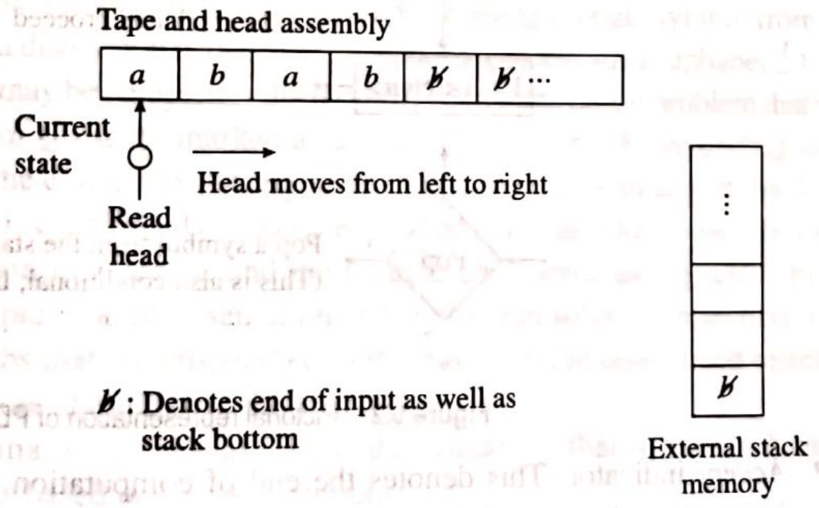


Figure 6.1 Memory of PDM

A PDM can only read from the tape and cannot write onto it; the read head always moves in one direction, that is, from left to right. The stack is external to the reading assembly and acts as an auxiliary memory. Thus, a PDM can be considered as an FSM having an external stack memory.

6.2.1 Pictorial Representation of PDM Elements

Instead of using a state transition diagram that is the usual notation for describing machines, a PDM is represented using flowcharts.

Figure 6.2 depicts the different elements that are used in the flowchart representation of a PDM. The typical elements are as follows:

- *Start* indicator: This is a very common element found in flowcharts. It denotes the beginning of the computation.

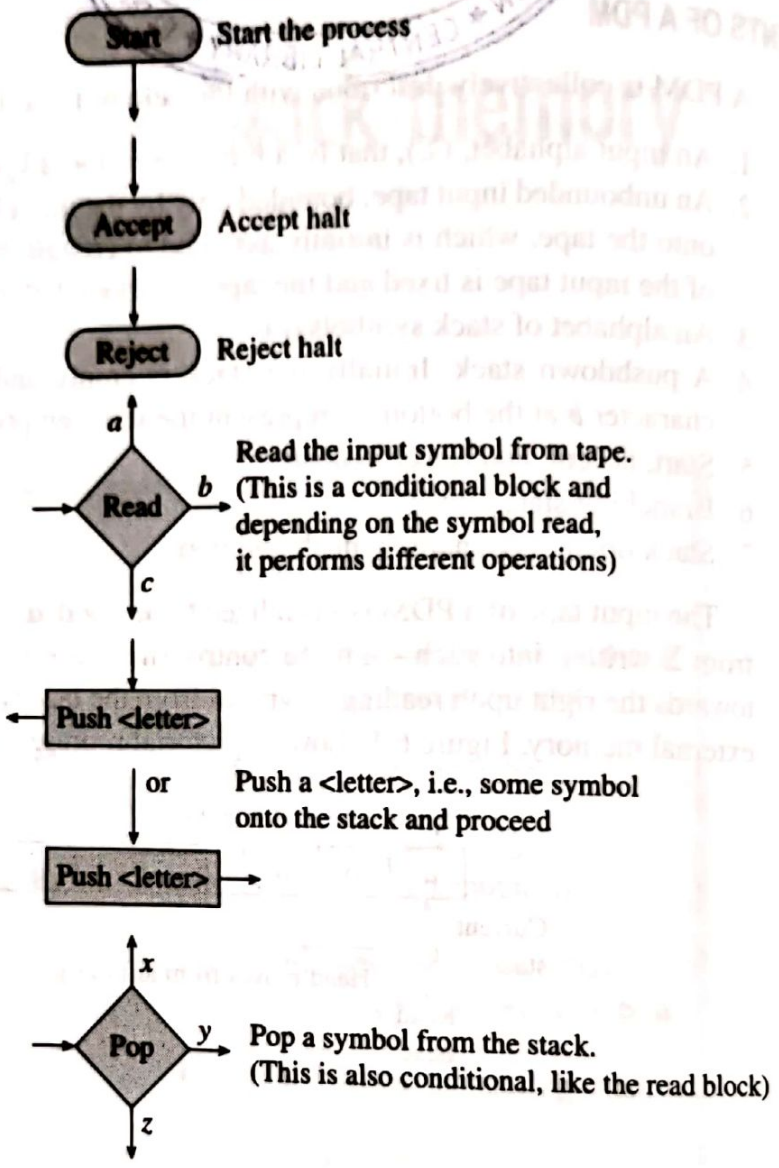

Start
Start the process
Accept
Accept halt
Reject
Reject halt
a
Read
b
c
Read the input symbol from tape. (This is a conditional block and depending on the symbol read, it performs different operations)
Push <letter>
or
Push a <letter>, i.e., some symbol onto the stack and proceed
Push <letter>
x
Pop
y
z
Pop a symbol from the stack. (This is also conditional, like the read block)

Figure 6.2 Pictorial representation of PDA elements

- ***Accept*** **indicator:** This denotes the end of computation that results in accepting the input string. It means that the input string has a valid pattern as defined by the context-free grammar (CFG). We know that a PDM accepts or recognizes CFLs.
- ***Reject*** **indicator:** This denotes the end of computation that results in the rejection of the input string. This means that the input is invalid with respect to the CFG rules.
- ***Push*** **action:** Push is an operation that is carried over the external stack and is denoted by a rectangular block. One symbol can be pushed at a time.
- ***Pop*** **action:** Pop is an operation that results in a decision block. The program flow is specified based on the symbol that is retrieved from the stack.
- ***Read*** **state:** Read state is represented as a decision block. The program flow is specified based on the input symbol that is read.

6.3 PUSHDOWN AUTOMATA

Pushdown automaton (plural—automata; abbreviated as PDA) is the mathematical model (formalism) of the PDM.

Formal Definition

A pushdown automaton M is denoted as

$$M = \{Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

where,

Q: Finite set of states
Σ: Input alphabet (input strings are composed of symbols from Σ)
Γ: Stack alphabet
q_0: Initial state q_0, which is a member of Q
Z_0: Z_0, which is a member of Γ, is a particular stack symbol called the start symbol. It indicates the bottom of the stack, and is considered as $\not{b}$ (blank character) in many designs
F: Set of final states, $F \subseteq Q$
δ: $Q \times \Sigma \times \Gamma \rightarrow Q \times \Gamma^*$ (This defines the transition function δ for a *deterministic PDA* or *DPDA*. The transition function for a non-deterministic PDA or NPDA is different, and is stated later)

The stack is an external storage system. Hence, one can store any symbol onto the stack, though it may not really be the symbol that has been read from the input tape. Moreover, one can push multiple symbols onto the stack, upon reading a single symbol from the input tape. Hence, there is a distinction between input alphabet (Σ) and stack alphabet (Γ). These two sets may overlap or may be completely distinct sets, depending on the problem that we are solving.

Any subset of Q can be marked as the set of final states F, depending on the solution. Upon reading the entire input string, if the machine resides in any of the final states, then the input string is considered as 'accepted' by the machine. Otherwise, the machine resides in any of the non-final states, and the input is considered as 'rejected' by the machine. Usually, in the pictorial representation of the problem solution, rejection is not explicitly shown. The paths that are unspecified in the flowchart are considered rejection paths. We shall discuss more about this in the examples.

A stack is initially assumed to contain the symbol Z_0, that is, the blank character $\not{b}$. Any PDM solution is based on this assumption.

In case of a DPDA, $\delta(q, a, Z)$ does not contain more than one element for any state q in Q, any symbol Z in Γ on top of the stack, and input symbol a in Σ. Thus, the transition would be of the form

$$\delta(q, a, Z) = (p, \gamma)$$

where, p is the unique next state in Q to which the machine makes the transition (p may or may not be equal to q), and γ is a member of Γ^*(zero or more occurrences of symbols from Γ). The symbol γ can be one of the following:

- Empty, that is, ϵ, if the stack operation performed is pop. This means that Z is popped out of the stack.
- Z, if the stack is not updated—only a state transition is performed.
- $xx\ldots xxZ$, if multiple symbols $xx\ldots xx$ are pushed onto the stack.
- $xx\ldots xx$, if Z is popped out of the stack and multiple symbols $xx\ldots xx$ are pushed onto the stack.

Therefore, if the DPDA is in state q, it reads symbol a from the tape cell, and if Z is on top of the stack, it changes the state to p, replaces Z on the stack top by γ (considering the aforementioned four scenarios), and moves the head one cell position to the right on the tape.

For an NPDA, the transition function δ is defined as follows:

$$\delta: Q \times \{\Sigma \cup (\epsilon)\} \times \Gamma \rightarrow \text{finite subsets of } Q \times \Gamma^*$$

For example, consider the following transition:

$$\delta(q, a, Z) = \{(p_1, \gamma_1), (p_2, \gamma_2), \ldots, (p_m, \gamma_m)\},$$

where, $q, p_1, p_2, \ldots, p_m$ are states from Q, a is any input symbol in Σ, Z is any stack symbol in Γ, and all γ_i for $1 \leq i \leq m$ are members of Γ^*.

The interpretation of this definition is that the NPDA in state q reads input symbol a while Z is the topmost symbol on the stack, possibly makes transition to any state p_i, replaces symbol Z on the stack by any string γ_i (as per the aforementioned four scenarios), and advances the read head one cell position to the right on the input tape. The NPDA, hence, is considered as an unpredictable (non-deterministic or possibilistic) machine as it can perform any of the possible transitions during its execution, though the current machine state, current input symbol being read, and the stack state are the same.

6.4 FINITE AUTOMATA VS PDA

As we have discussed, a PDA can be visualized as an FA having an external stack. Thus, a PDA has infinite storage in the form of a stack, which is missing in the FA. Hence, the FA has lesser computational power compared to the PDA. Furthermore, an FA cannot solve any problem that requires to store intermediate results in the memory for further computation.

As we know, FA are capable of accepting (or recognizing) regular languages (RLs), while PDA can accept CFLs. Since the set of all RLs is a subset of the class of CFLs, we can say that every RL is also a CFL; however, the vice versa may not be true. Hence, PDAs can accept all RLs. This is demonstrated through the following examples.

6.4.1 Examples of PDA Accepting Regular Languages

Example 6.1 Construct a PDA that recognizes the language accepted by the DFA shown in Fig. 6.3.

Solution The DFA given in Fig. 6.3 accepts the regular language represented by the following regular expression:

$$b^* \, a \, a^* \, (b \, b^* \, a \, a^*)^*$$

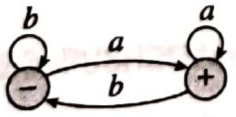


Figure 6.3 Example DFA

We can construct a DPDA equivalent to the given DFA, as shown in Fig. 6.4.

In Fig. 6.4, we see that $Read_1$ state of the DPDA is analogous to the initial state of the DFA and $Read_2$ state is analogous to the final state of the given DFA. Since $Read_2$ state is analogous to the final state, if the input ends in $Read_1$, that is, if we get a blank

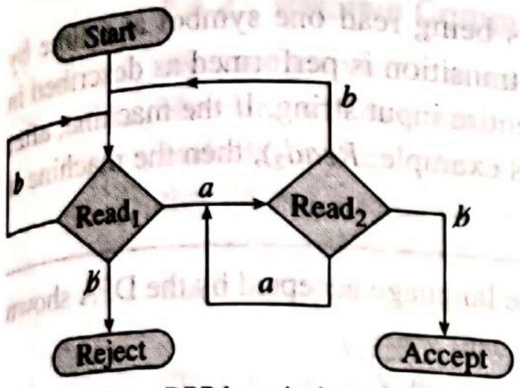


Figure 6.4 DPDA equivalent to DFA in Fig. 6.3

character b̸ on the tape in *Read*$_1$, the machine rejects the input string; else it transits to 'accept' state as shown in the figure.

Please note that the external stack is not required for this example as the language being accepted is a regular language. Hence, the DFA and the DPDA are equivalent machines and the changes are only notational.

In this example, we have demonstrated the PDA as a regular language acceptor.

Let us simulate the working of the PDA for the strings '*bbaaba*' and '*baaabab*'.

1. Simulation for string '*bbaaba*':

Current state	Input symbol	Next state	Head point position
Start	—	Read$_1$	$\underset{\uparrow}{b}baaba\not{b}\ldots$
Read$_1$	*b*	Read$_1$	$b\underset{\uparrow}{b}aaba\not{b}\ldots$
Read$_1$	*b*	Read$_1$	$bb\underset{\uparrow}{a}aba\not{b}\ldots$
Read$_1$	*a*	Read$_2$	$bba\underset{\uparrow}{a}ba\not{b}\ldots$
Read$_2$	*a*	Read$_2$	$bbaa\underset{\uparrow}{b}a\not{b}\ldots$
Read$_2$	*b*	Read$_1$	$bbaab\underset{\uparrow}{a}\not{b}\ldots$
Read$_1$	*a*	Read$_2$	$bbaaba\underset{\uparrow}{\not{b}}\ldots$
Read$_2$	*b*	Accept	$bbaabab\underset{\uparrow}{\not{b}}\ldots$

Thus, the string '*bbaaba*' is accepted by the PDA.

2. Simulation for string '*baaabab*':

Current state	Input symbol	Next state	Head point position
Start	—	Read$_1$	$\underset{\uparrow}{b}aaabab\not{b}\ldots$
Read$_1$	*b*	Read$_1$	$b\underset{\uparrow}{a}aabab\not{b}\ldots$
Read$_1$	*a*	Read$_2$	$ba\underset{\uparrow}{a}abab\not{b}\ldots$
Read$_2$	*a*	Read$_2$	$baa\underset{\uparrow}{a}bab\not{b}\ldots$
Read$_2$	*a*	Read$_2$	$baaa\underset{\uparrow}{b}ab\not{b}\ldots$
Read$_2$	*b*	Read$_1$	$baaab\underset{\uparrow}{a}b\not{b}\ldots$
Read$_1$	*a*	Read$_2$	$baaaba\underset{\uparrow}{b}\not{b}\ldots$
Read$_2$	*b*	Read$_1$	$baaabab\underset{\uparrow}{\not{b}}\ldots$
Read$_1$	b̸	Reject	$baaabab\not{b}\underset{\uparrow}{\not{b}}\ldots$

Thus, the string '*baaabab*' is rejected by the PDA

This simulation depicts how the input string is being read one symbol at a time by the machine. Upon reading every symbol, a state transition is performed as described in Fig. 6.4. The machine stops after consuming the entire input string. If the machine, after reading the entire input, is in the final state (in this example, *Read*$_2$), then the machine is said to accept the string; else it rejects the string.

Example 6.2 Construct a PDA that recognizes the language accepted by the DFA shown in Fig. 6.5.

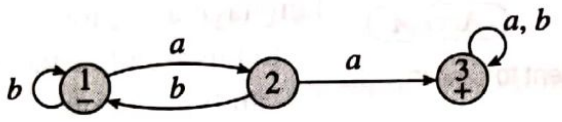


Figure 6.5 Example DFA

Solution The equivalent DPDA can be constructed as shown in Fig. 6.6.

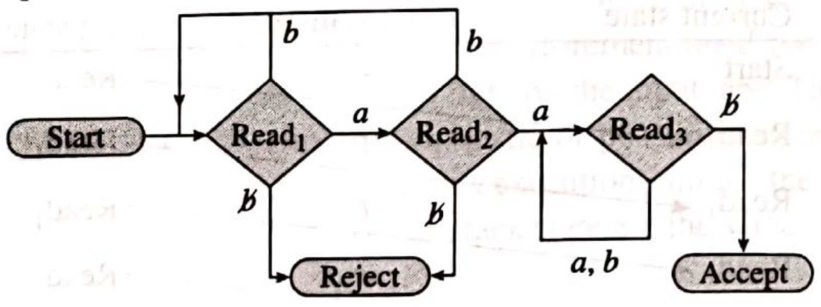


Figure 6.6 DPDA equivalent to DFA in Fig. 6.5

We see that the *Read*$_1$ state in the PDA is analogous to state 1 in the given DFA. Similarly, *Read*$_2$ is analogous to state 2, and *Read*$_3$ is analogous to state 3 in the given DFA. Hence, *Read*$_1$ and *Read*$_2$ are non-final states. If the PDA reads a blank character ƀ indicating the end of the input string while in these states, the machine rejects the input string. Upon reading the entire input string, if the PDA reaches the final state, that is, *Read*$_3$, then the input string is accepted by the PDA.

Let us simulate the working of the PDA for the input given as '*abaab*'. The acceptance of the input can be shown as follows:

Current state	Input symbol	Next state	Head point position
Start	—	Read$_1$	$\underset{\uparrow}{a}\,b\,a\,a\,b\,ƀ\ldots$
Read$_1$	*a*	Read$_2$	$a\,\underset{\uparrow}{b}\,a\,a\,b\,ƀ\ldots$
Read$_2$	*b*	Read$_1$	$a\,b\,\underset{\uparrow}{a}\,a\,b\,ƀ\ldots$
Read$_1$	*a*	Read$_2$	$a\,b\,a\,\underset{\uparrow}{a}\,b\,ƀ\ldots$
Read$_2$	*a*	Read$_3$	$a\,b\,a\,a\,\underset{\uparrow}{b}\,ƀ\ldots$
Read$_3$	*b*	Read$_3$	$a\,b\,a\,a\,b\,\underset{\uparrow}{ƀ}\,ƀ\ldots$
Read$_3$	ƀ	Accept	$a\,b\,a\,a\,b\,b\,\underset{\uparrow}{ƀ}\ldots$

6.4.2 Relative Computational Powers of PDA and FA

We have seen in the previous subsection that PDA can accept regular languages just as FA. However, a PDA is much more powerful when compared to the FA, as it has *infinite memory* in the form of an external stack that is absent in the case of FA. In other words, *we can say that FA is a special case of PDA—it is a PDA without an external stack*.

For every FA, we can construct an equivalent PDA that accepts the same regular language. Such a PDA does not require a stack and its operations such as push and pop.

We have already discussed earlier that PDA can accept CFLs that are the superset of the class of RLs. In the following section, we shall look at some examples of PDA that accept CFLs.

6.5 PDA ACCEPTING CFLs

We have seen that a PDA accepts not only RLs, but also CFLs. Further, we also know that building a DPDA or an NPDA that accepts RLs does require an external stack. However, this is not possible while designing a PDA solution for the CFLs.

The following examples will illustrate this further.

Example 6.3 Construct a PDA that accepts the following language:

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

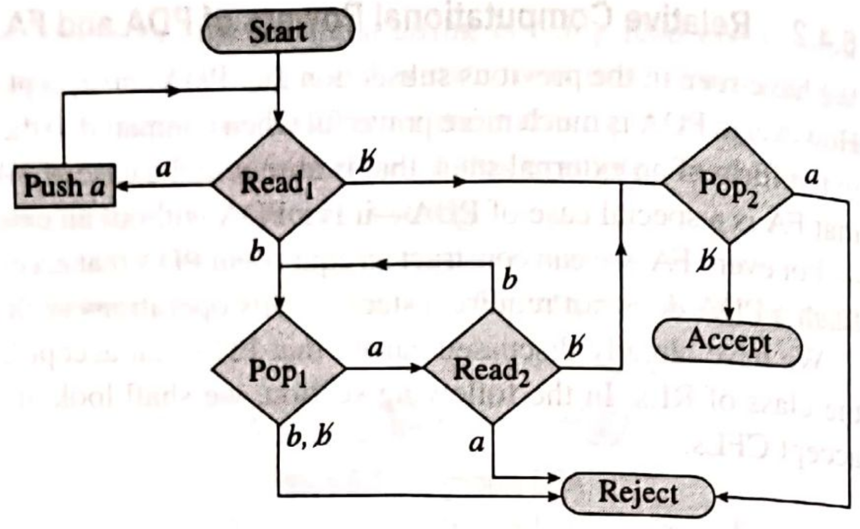
Solution As per the definition, every string in the language contains n number of a's followed by the same number of b's; also, it contains an empty string ϵ, whenever $n = 0$.

Algorithm

1. Keep on pushing all a's onto the stack till the machine reads the first b.
2. Each time it reads b, it pops one a from the stack.
3. If the number of a's are equal to the number of b's, then at the end of the input string, that is, when blank character $\not{b}$ is read from the tape, the top of the stack should also contain the blank character $\not{b}$, indicating that the stack is empty. The string is accepted only in this case; else it is rejected.

The required DPDA is constructed as shown in Fig. 6.7.

We see from Fig. 6.7 that $Read_1$ state pushes all a's onto the stack. On reading the first b, it performs the pop operation; $Read_2$ state checks whether or not the same number of b's are following the a's, by repeatedly popping the a's from the stack for every matching b that is read.



Read$_1$: Keep on pushing *a*'s till you get the first *b*
Pop$_1$: Pop an *a* when you read one *b*
Read$_2$: When you get *b* while reading *b*'s go to 'Pop$_2$'
Pop$_1$: Input is finished, so check whether the stack is empty or not. If empty go to 'Accept'

Figure 6.7 PDA that accepts $\{a^n b^n \mid n \geq 0\}$

Simulation

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

Current state	Stack contents	Tape and head
Start	$\not b$	$\underset{\uparrow}{a}\ a\ b\ b\ \not b\ \not b \dots$
$\downarrow$		
Read$_1$	$\not b$	$a\ \underset{\uparrow}{a}\ b\ b\ \not b\ \not b \dots$
$a \downarrow$		
Push *a*	$\not b\ a$	$a\ \underset{\uparrow}{a}\ b\ b\ \not b\ \not b \dots$
$\downarrow$		
Read$_1$	$\not b\ a$	$a\ a\ \underset{\uparrow}{b}\ b\ \not b\ \not b \dots$
$a \downarrow$		
Push *a*	$\not b aa$	$a\ a\ \underset{\uparrow}{b}\ b\ \not b\ \not b \dots$
$\downarrow$		
Read$_1$	$\not b aa$	$a\ a\ b\ \underset{\uparrow}{b}\ \not b\ \not b \dots$
$b \downarrow$		
Pop$_1$	$\not b a$	$a\ a\ b\ \underset{\uparrow}{b}\ \not b\ \not b \dots$
$a \downarrow$		
Read$_2$	$\not b a$	$a\ a\ b\ b\ \underset{\uparrow}{\not b}\ \not b \dots$
$b \downarrow$		
Pop$_1$	$\not b$	$a\ a\ b\ b\ \underset{\uparrow}{\not b}\ \not b \dots$
$a \downarrow$		
Read$_2$	$\not b$	$a\ a\ b\ b\ \not b\ \underset{\uparrow}{\not b} \dots$
$\not b \downarrow$		
Pop$_2$	—	$a\ a\ b\ b\ \not b\ \underset{\uparrow}{\not b} \dots$
$\not b \downarrow$		
Accept		

The arrows on the left-hand side indicate transitions from one state to another on reading the same symbol. The operations 'push' and 'pop' are performed while processing the input.

In the beginning, the stack is assumed to contain the blank character $\not{b}$ to indicate the bottom of the stack. In the aforementioned simulation, the first two characters in the input string are pushed onto the stack as both are a's. Then, for every b read, an a is popped out of the stack. Once the blank character $\not{b}$ is read, indicating the end of the input, 'pop' operation is performed to check whether the stack is empty as well. If the stack is empty, that is, if pop operation returns the bottommost element as $\not{b}$, then the input string is accepted by the PDA. Thus, the input string '*aabb*' is found to match the expected pattern, $a^n b^n$. This is also referred to as *acceptance by empty stack* (refer to Section 6.5.2).

2. Let us now simulate the rejection of '*abbba*':

Current state	Stack contents	Tape and head
Start	$\not{b}$	$\underset{\uparrow}{a}\ b\ b\ b\ a\ \not{b}\ \not{b}\ \ldots$
$\downarrow$		
Read$_1$	$\not{b}$	$a\ \underset{\uparrow}{b}\ b\ b\ a\ \not{b}\ \not{b}\ \ldots$
$a\downarrow$		
Push a	$\not{b}a$	$a\ \underset{\uparrow}{b}\ b\ b\ a\ \not{b}\ \not{b}\ \ldots$
$\downarrow$		
Read$_1$	$\not{b}a$	$a\ b\ \underset{\uparrow}{b}\ b\ a\ \not{b}\ \not{b}\ \ldots$
$b\downarrow$		
Pop$_1$	$\not{b}$	$a\ b\ \underset{\uparrow}{b}\ b\ a\ \not{b}\ \not{b}\ \ldots$
$a\downarrow$		
Read$_2$	$\not{b}$	$a\ b\ b\ \underset{\uparrow}{b}\ a\ \not{b}\ \not{b}\ \ldots$
$b\downarrow$		
Pop$_1$	—	$a\ b\ b\ \underset{\uparrow}{b}\ a\ \not{b}\ \not{b}\ \ldots$
$\not{b}\downarrow$		
Reject		

In this simulation, the first a gets pushed onto the stack. When the second input symbol b is read, the topmost stack symbol a is popped to match the read symbol b. The stack becomes empty after the first pop. The third symbol b is read but the stack is found to be empty—it does not contain any more a's to match with the b that is read. Hence, the input string is rejected.

Notes:

1. **The given language $L = \{a^n b^n \mid n \geq 0\}$, is not a regular language; therefore, it cannot be recognized by an FA. It is actually a CFL and the CFG for the same can be written as follows:**

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

2. We observe that the PDA in Fig. 6.7 is a DPDA because from every state there is a unique transition for a symbol that is read.
3. We need not restrict ourselves to using the same alphabet for the input strings as well as the stack. Since the stack is an external memory component, the stack alphabet (Γ) is considered to be distinct from the input alphabet (Σ). In this example, it is possible to read an a from the tape and push x onto the stack. In this case, the number of x's represents the count of the number of a's present, as shown in Fig. 6.8.

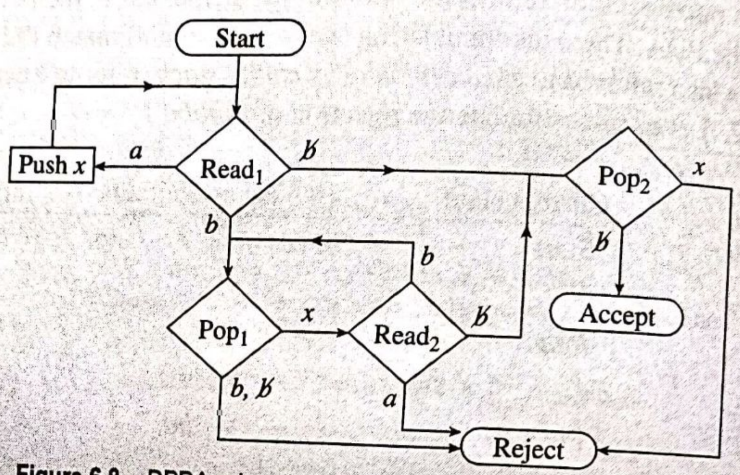


Figure 6.8 DPDA using distinct sets of input and stack alphabets

The DPDA in Fig. 6.7 can also be specified using the following set of equations, which are based on the formal notations discussed in Section 6.3.

The following two equations denote the pushing of symbol a onto the stack if the symbol read is a; the stack is either empty or contains a as its topmost symbol. These two equations collectively signify the process of pushing all the a's onto the stack (till the first b is read). Refer to program block 1 in Fig. 6.9.

$$\delta\,(Read_1, a, \cancel{b}) = (Read_1, a\cancel{b})$$
$$\delta\,(Read_1, a, a) = (Read_1, aa)$$

The following two equations denote the pop operation of a out of the stack for every b that is read. This is indicated by the stack state, which changes from a to ϵ (refer to program block 2 in Fig. 6.9). The two transitions indicate moving from $Read_1$ to $Read_2$ for the first b that is read and $Read_2$ to itself for the subsequent b's that are read.

$$\delta\,(Read_1, b, a) = (Read_2, \epsilon)$$
$$\delta\,(Read_2, b, a) = (Read_2, \epsilon)$$

The following two equations denote the two program paths that lead to the acceptance of the input string, if it matches the required pattern, a^nb^n. The first equation represents the case $n = 0$, that is, when the input string is empty (ϵ). The second equation represents the case $n > 0$.

$$\delta\,(Read_1, \cancel{b}, \cancel{b}) = \text{Accept}$$
$$\delta\,(Read_2, \cancel{b}, \cancel{b}) = \text{Accept}$$

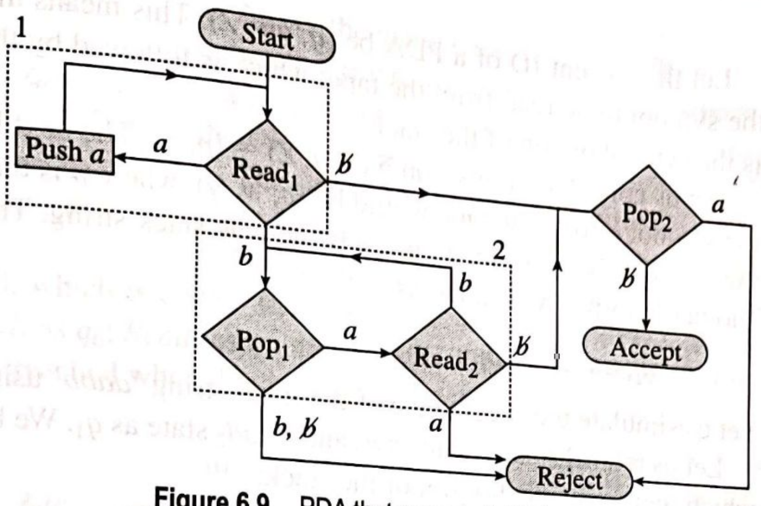


Figure 6.9 PDA that accepts $\{a^nb^n \mid n \geq 0\}$

In principle, the aforementioned six equations are sufficient to denote the DPDA that accepts the given CFL. The following six are additional equations that denote rejections and may not be specified—since anyway, what is not specified is always rejected.

The following two rejection equations represent the presence of additional number of a's.

$$\delta\ (Read_1, \not b, a) = \text{Reject}$$
$$\delta\ (Read_2, \not b, a) = \text{Reject}$$

The following two rejections represent that the number of b's is more than the number of a's.

$$\delta\ (Read_1, b, \not b) = \text{Reject}$$
$$\delta\ (Read_2, b, \not b) = \text{Reject}$$

These two equations are reached when the DPDA reads a while in $Read_2$ state, no matter what is on the top of the stack. This is something that is against the expected pattern. These equations represent the case where there are some a's even after all the b's have been read. Hence, they are considered as rejection equations.

$$\delta\ (Read_2, a, \not b) = \text{Reject}$$
$$\delta\ (Read_2, a, a) = \text{Reject}$$

6.5.1 Instantaneous Description of PDA

Instantaneous description (ID) of a PDA, as the name suggests, is its description at a given instance, while computing a given input string. It is described with the help of a triple (q, w, z), where q denotes the current state of the machine; w denotes the input string remaining to be read from the tape; and z denotes the topmost stack symbol.

While consuming any given input string (symbol by symbol), the PDA moves from one ID to another. This process continues as long as the input lasts. This means that the simulations we have seen in the examples can also be described more formally with the help of a series of IDs.

Let the current ID of a PDA be (q, aw, Z). This means that q is the current state; a is the symbol to be read from the tape, which is followed by the remaining string w; and Z is the symbol on top of the stack.

Let us consider a transition $\delta(q, a, Z) = (p, \gamma)$. The next ID for the PDA after reading the symbol a from the tape would be (p, w, γ), where p is the next state; w is the remaining string to be consumed; and γ is the new stack string. This transition from one ID to another is formally denoted as

$$(q, aw, Z) \vdash (p, w, \gamma)$$

Let us simulate the acceptance of the input string '*aabb*' using the series of IDs.

Let us relabel *Read*$_1$ state as q_0 and *Read*$_2$ state as q_1. We have already labelled Z_0 as $\not{b}$, which represents the bottom of the stack.

$$\begin{aligned}(q_0, aabb\not{b}, \not{b}) &\vdash (q_0, abb\not{b}, a\not{b})\\ &\vdash (q_0, bb\not{b}, aa\not{b})\\ &\vdash (q_1, b\not{b}, a\not{b})\\ &\vdash (q_1, \not{b}, \not{b})\\ &\vdash (\text{Accept}, \epsilon, \epsilon)\end{aligned}$$

6.5.2 Acceptance of CFL by Empty Stack

The acceptance of an input string '*aabb*' as depicted in the previous section is an example of acceptance by an empty stack. Observe that neither *Read*$_1$ nor *Read*$_2$ is designated as a final state in the DPDA shown in Fig. 6.7. In this chapter, we have implemented acceptance by empty stack for all the solved examples, as it is the most common way of implementation. Readers may attempt to build the acceptance by final state for these examples, if interested.

A string w is said to be accepted by a PDA, if

$$(q_0, w, Z_0) \vdash^* (p, \epsilon, \epsilon)$$

where, p is any state in Q, and '$\vdash^*$' denotes an ID after reading the entire input string. '$\vdash^*$' denotes multiple steps in reading the string, while '$\vdash$' denotes one step at a time (refer to Fig. 6.10).

6.5.3 Acceptance of CFL by Final State

Acceptance by final state is not very commonly implemented for PDA and is not very different from acceptance by empty stack.

A string w is said to be accepted by a PDA if:

$$(q_0, w, Z_0) \vdash^* (p, \epsilon, \gamma)$$

where, p is a member of F, which is a designated set of final states (refer to Section 6.3), and γ is any stack string. This means that upon reading the entire string w, if the PDA transits to a final state, then the string w is said to be accepted by the PDA (refer to Fig. 6.11).

6.5.4 State Transition Diagram for PDA

We can also represent PDAs using the state transition diagram, just as we did in the case of FA and Turing machines. However, there is a slight change in the notation, based on whether a given PDA accepts a CFL with an empty stack or a final state.

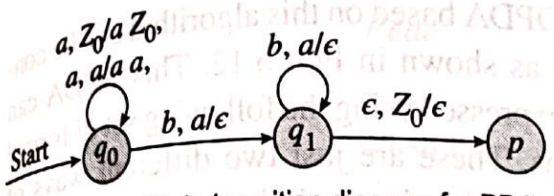


Figure 6.10 State transition diagram for PDA that accepts $\{a^n b^n \mid n \geq 0\}$ by empty stack

As per the usual notations, the state transition diagram is a directed graph, where each node represents a state and an edge represents a transition. The state transition diagram for the DPDA in Fig. 6.7 is drawn as shown in Fig. 6.10.

The edges are labelled in the form 'a, Z/γ' where, a is the input symbol read; Z denotes the topmost stack symbol; and γ denotes the stack string after transition is complete.

In Fig. 6.10, which is a state transition diagram for the DPDA in Fig. 6.7, we have relabelled $Read_1$ as q_0; $Read_2$ as q_1; and accept indicator as p. Note that though p is not a final state, it is reached when the input ends and the stack becomes empty.

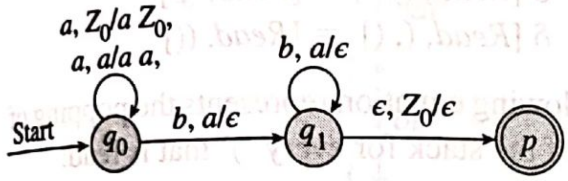


Figure 6.11 State transition diagram for PDA that accepts $\{a^n b^n \mid n \geq 0\}$ by final state

The state transition diagram in Fig. 6.10 can be modified to depict the DPDA accepting the same CFL by the final state, as shown in Fig. 6.11.

We observe that the only change in Fig. 6.11 is that p is now marked with double circles to indicate that it is a final state; and the stack string Z_0 remains unchanged while approaching the state p.

Example 6.4 Design a PDA that checks for well-formed parentheses.

Solution As we know, a string of parentheses that is well-formed should start with the opening bracket '(' and must end with the closing bracket ')'. This is a CFL and can be described with the help of the following grammar:

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

The language can be expressed as

$$L = \{\epsilon, (\,), ((\,)), (\,)(\,), (\,)((\,)), ((\,))(\,), \ldots\}$$

Algorithm

The problem is similar to that in the previous example, in which we have seen that the string should begin with a, and is matched with the subsequent b's; in this example, all '('s are pushed onto the stack to be matched later with the subsequent ')'s. The only difference is that the machine can read opening parenthesis '(' even after reading the closing parenthesis ')', rather than all '('s being read first followed by all ')'s. In other words, the language is not of the form $(^n\,)^n$; for example, there may be strings of the form '( ) ( )' or '( ) (( ))' as well. Due to this, the algorithm and the PDA will be a little different, as follows:

1. If you read symbol '(', push '(' onto the stack.
2. On reading every ')' pop one '(' from the stack.
3. Repeat the aforementioned two steps till you finish with all the parentheses in the string.
4. When the input ends, that is, on reading $\not{b}$, the top of the stack should also contain $\not{b}$; this indicates that the string consists of well-formed parentheses.

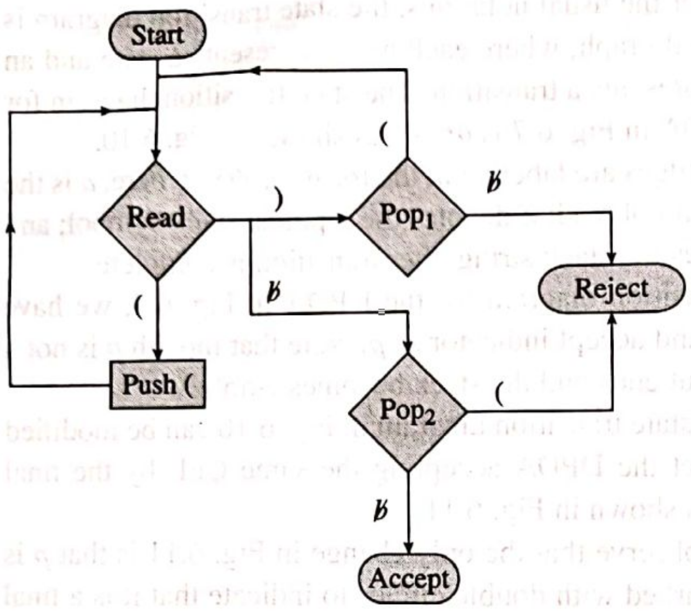


Figure 6.12 DPDA that checks for well-formed parentheses

The DPDA based on this algorithm can be constructed as shown in Fig. 6.12. This DPDA can also be expressed using the following set of formal equations. These are just two different ways of representing the given solution and are equivalent to each other.

When the symbol '(' is read, push it onto the stack. This is expressed using the following two equations:

$$\delta\,[Read, (, \not b] = [Read, (\not b]$$
$$\delta\,[Read, (, (] = [Read, ((]$$

The following equation represents the popping of '(' out of the stack for every ')' that is read:

$$\delta\,[Read,), (] = [Read, \epsilon]$$

The following equation is the only path to acceptance. When the input ends, the stack must be empty as well.

$$\delta\,(Read, \not b, \not b) = \text{Accept}$$

Rejection paths are expressed with the help of the following two equations:

$\delta\,[Read,), \not b] = \text{Reject}$ … number of ')'s is more
$\delta\,[Read, \not b, (] = \text{Reject}$ … number of '('s is more

Simulation

Let us simulate the acceptance of the string '(( ) ( ))' by an empty stack.

Current state	Stack contents	Tape and head
Start	$\not b$	(()())$\not b\not b$…
↓		↑
Read	$\not b$	(()())$\not b\not b$…
(↓		↑
Push (	$\not b$(	(()())$\not b\not b$…
↓		↑
Read	$\not b$(	(()())$\not b\not b$…
(↓		↑
Push (	$\not b$((	(()())$\not b\not b$…
↓		↑
Read	$\not b$((	(()())$\not b\not b$…
)↓		↑
Pop$_1$	$\not b$(	(()())$\not b\not b$…
(↓		↑

Read	ƀ(	(()())ƀƀ…
(↓		↑
Push (	ƀ((	(()())ƀƀ…
↓		↑
Read	ƀ((	(()())ƀƀ…
) ↓		↑
Pop$_1$	ƀ(	(()())ƀƀ…
(↓		↑
Read	ƀ(	(()())ƀƀ…
) ↓		↑
Pop$_1$	ƀ	(()())ƀƀ…
(↓		↑
Read	ƀ	(()())ƀƀ…
ƀ ↓		↑
Pop$_2$	—	(()())ƀƀ…
ƀ ↓		↑
Accept		

Example 6.5 Construct a PDA that accepts the following language:

$$L = \{X, aXa, bXb, aaXaa, abXba, baXab, bbXbb, aaaXaaa, \ldots\}$$

Solution We see that the language consists of all odd length palindrome strings over $\Sigma = \{a, b\}$ having X as their middle symbol. This is a CFL with the grammar expressed as

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

Algorithm

1. Push all symbols onto the stack till X is read by the machine.
2. Beyond X, for every input symbol that is read, pop a symbol from the stack and check for the equality with the recently-read symbol.
3. If the symbol on the stack matches with the symbol read, then continue step 2; else, reject the string.
4. If the end of the input is reached, that is, the symbol read is ƀ, pop the symbol onto the top of the stack. If the symbol popped is also ƀ (indicates stack empty), then accept the input string; else, reject the string.

The DPDA based on this algorithm is constructed as shown in Fig. 6.13. The set of equations equivalent to the DPDA are given here:

The following six equations are responsible for pushing all a's and b's till the symbol X has been read.

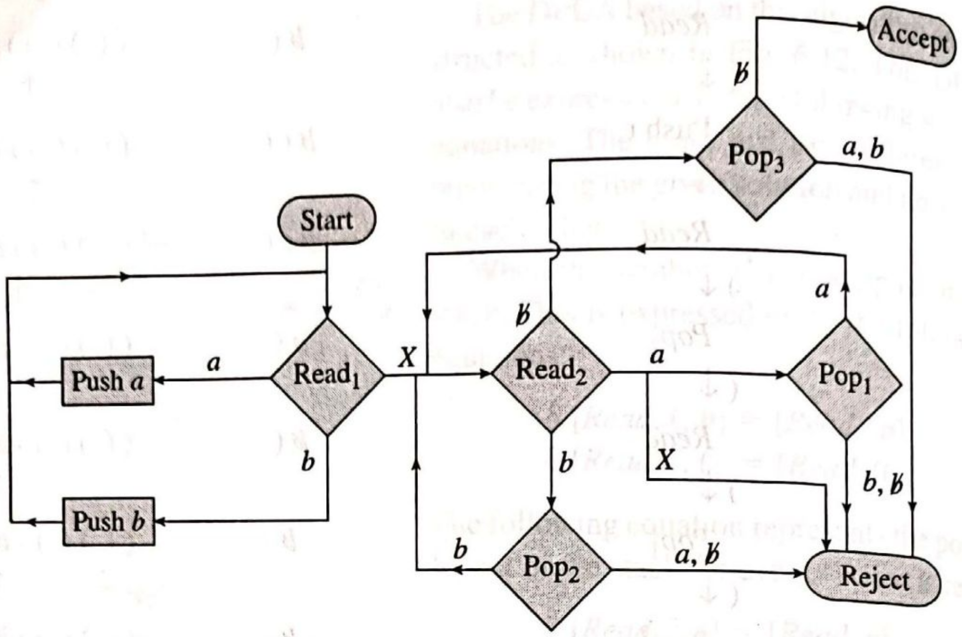


Figure 6.13 PDA that accepts language *L*

$\delta\,(Read_1, a, \not{b}) = (Read_1, a\not{b})$
$\delta\,(Read_1, a, a) = (Read_1, aa)$
$\delta\,(Read_1, a, b) = (Read_1, ab)$

$\delta\,(Read_1, b, \not{b}) = (Read_1, b\not{b})$
$\delta\,(Read_1, b, b) = (Read_1, bb)$
$\delta\,(Read_1, b, a) = (Read_1, ba)$

The following three equations represent the case when the middle symbol *X* is read. A transition is made from $Read_1$ state to the next state, $Read_2$, and the stack remains unchanged.

$\delta\,(Read_1, X, a) = (Read_2, a)$
$\delta\,(Read_1, X, b) = (Read_2, b)$
$\delta\,(Read_1, X, \not{b}) = (Read_2, \not{b})$

Once the symbol *X* is read, the steps are followed to match the remaining half of the input string to the first half, which is already present on the stack. This is expressed with the help of the following three equations:

If the symbol read is the same as the symbol on top of the stack, then it gets popped off.

$\delta\,(Read_2, b, b) = (Read_2, \epsilon)$
$\delta\,(Read_2, a, a) = (Read_2, \epsilon)$
$\delta\,(Read_2, \not{b}, \not{b}) = \text{Accept}$

The invalid input strings are rejected by the DPDA. This is expressed through the following equations:

$\delta\,(Read_2, a, \not{b}) = \text{Reject}$
$\delta\,(Read_2, a, b) = \text{Reject}$

$\delta\,(Read_2, b, \not{b}) = \text{Reject}$
$\delta\,(Read_2, b, a) = \text{Reject}$

$\delta\,(Read_2, \not{b}, b) = \text{Reject}$
$\delta\,(Read_2, \not{b}, a) = \text{Reject}$

$\delta\,(Read_2, X, b) = \text{Reject}$
$\delta\,(Read_2, X, a) = \text{Reject}$
$\delta\,(Read_2, X, \not{b}) = \text{Reject}$

Simulation

Let us simulate the acceptance of the string '*abXba*'.

Current state	Stack contents	Tape and head
Start	$\not{b}$	$a\ \underset{\uparrow}{b}\ X\ b\ a\ \not{b}\ \not{b}\ \ldots$
↓		
$Read_1$	$\not{b}$	$a\ \underset{\uparrow}{b}\ X\ b\ a\ \not{b}\ \not{b}\ \ldots$
a ↓		
Push a	$\not{b}\ a$	$a\ \underset{\uparrow}{b}\ X\ b\ a\ \not{b}\ \not{b}\ \ldots$
↓		
$Read_1$	$\not{b}\ a$	$a\ b\ \underset{\uparrow}{X}\ b\ a\ \not{b}\ \not{b}\ \ldots$
b ↓		
Push b	$\not{b}\ a\ b$	$a\ b\ \underset{\uparrow}{X}\ b\ a\ \not{b}\ \not{b}\ \ldots$
↓		
$Read_1$	$\not{b}\ a\ b$	$a\ b\ X\ \underset{\uparrow}{b}\ a\ \not{b}\ \not{b}\ \ldots$
X ↓		
$Read_2$	$\not{b}\ a\ b$	$a\ b\ X\ b\ \underset{\uparrow}{a}\ \not{b}\ \not{b}\ \ldots$
b ↓		
Pop_2	$\not{b}\ a$	$a\ b\ X\ b\ \underset{\uparrow}{a}\ \not{b}\ \not{b}\ \ldots$
b ↓		
$Read_2$	$\not{b}\ a$	$a\ b\ X\ b\ a\ \underset{\uparrow}{\not{b}}\ \not{b}\ \ldots$
a ↓		
Pop_1	$\not{b}$	$a\ b\ X\ b\ a\ \underset{\uparrow}{\not{b}}\ \not{b}\ \ldots$
a ↓		
$Read_2$	$\not{b}$	$a\ b\ X\ b\ a\ \not{b}\ \underset{\uparrow}{\not{b}}\ \ldots$
$\not{b}$ ↓		
Pop_3	—	$a\ b\ X\ b\ a\ \not{b}\ \underset{\uparrow}{\not{b}}\ \ldots$
$\not{b}$ ↓		
Accept		

6.6 DPDA VS NPDA

Consider a CFL that consists of all possible palindrome strings over $\Sigma = \{a, b\}$. Such a CFL can be expressed with the help of the following CFG:

$$S \rightarrow a\,S\,a \mid b\,S\,b \mid a \mid b \mid \epsilon \qquad (G)$$

Odd length palindrome strings over $\Sigma = \{a, b\}$ can be generated using the following grammar rules:

$$S \rightarrow a\,S\,a \mid b\,S\,b \mid a \mid b \qquad (G1)$$

This is obtained by replacing the middle symbol X in the CFG we have seen in Example 6.5 by a or b.

Similarly, all even length palindrome strings over $\Sigma = \{a, b\}$ can be generated using the following grammar rules:

$$S \rightarrow a\,S\,a \mid b\,S\,b \mid \epsilon \qquad (G2)$$

Here, the middle symbol X in Example 6.5 is replaced by ϵ.

We see that the grammar G is a combination of the rules for odd as well as even length palindrome strings over $\Sigma = \{a, b\}$.

Using this grammar, and with the help of the following example, let us attempt to compare DPDA and NPDA.

Example 6.6 Construct a PDA, which accepts the language denoted by the following grammar:

$$S \rightarrow a\,S\,a \mid b\,S\,b \mid a \mid b \mid \epsilon \qquad (G)$$

Solution The given language consists of all possible palindrome strings over $\Sigma = \{a, b\}$.

The required PDA can be obtained by making the following changes in the DPDA shown in Fig. 6.13.

In Fig. 6.13, the middle symbol X separates the first half of the string that is pushed onto the stack, from the second half that is used for comparison. If we wish to accept all odd length palindrome strings over $\Sigma = \{a, b\}$, this middle symbol X must be replaced by either a or b, as shown in grammar $G1$. Hence, the transition from $Read_1$ state to $Read_2$ state is changed either to symbol a or b (refer to Fig. 6.14).

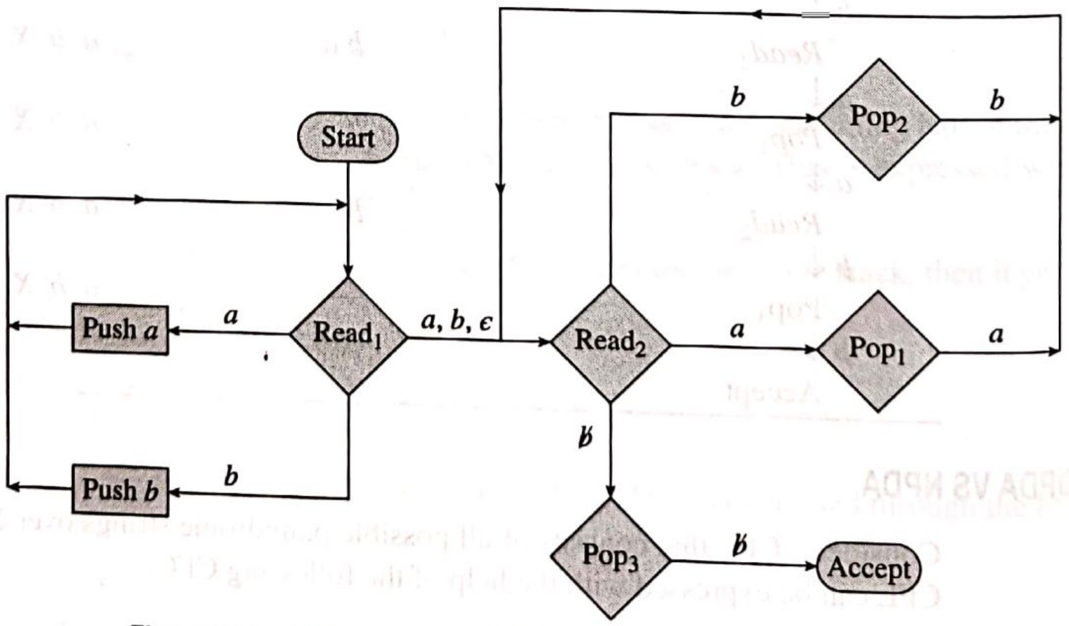


Figure 6.14 NPDA that accepts all palindrome strings over $\Sigma = \{a, b\}$

Similarly, as per grammar $G2$, the middle symbol X becomes ϵ for even length palindrome strings; hence, the transition from $Read_1$ state to $Read_2$ state. Therefore, in the required PDA (refer to Fig. 6.14), we change the symbol X to ϵ.

Thus, there are now three possible transitions from $Read_1$ to $Read_2$ depending on the input symbol read—a, b, or ϵ.

Please note that the PDA that we have obtained in Fig. 6.14 is non-deterministic. Further, the rejection paths are not shown in the figure for further simplification. The NPDA in Fig. 6.14 can also be expressed using the equations mentioned here.

The following set of equations denotes the operation of pushing a's or b's onto the stack, till the mid-point is reached:

$$\left.\begin{aligned}
\delta\,(Read_1, a, \not{b}) &= (Read_1, a\not{b})\\
\delta\,(Read_1, a, a) &= (Read_1, aa)\\
\delta\,(Read_1, a, b) &= (Read_1, ab)\\
\delta\,(Read_1, b, \not{b}) &= (Read_1, b\not{b})\\
\delta\,(Read_1, b, b) &= (Read_1, bb)\\
\delta\,(Read_1, b, a) &= (Read_1, ba)
\end{aligned}\right\} \tag{6.1}$$

The following set of six equations ignores the middle symbol. The transition from $Read_1$ state to $Read_2$ state is made without changing the stack string.

$$\left.\begin{aligned}
\delta\,(Read_1, a, a) &= (Read_2, a)\\
\delta\,(Read_1, a, b) &= (Read_2, b)\\
\delta\,(Read_1, b, a) &= (Read_2, a)\\
\delta\,(Read_1, b, b) &= (Read_2, b)\\
\delta\,(Read_1, \epsilon, a) &= (Read_2, a)\\
\delta\,(Read_1, \epsilon, b) &= (Read_2, b)
\end{aligned}\right\} \tag{6.2}$$

The following three equations denote the process of matching the second half of the input string with the stacked symbols in order to check whether or not the string is a palindrome, and accept the string in case it is a palindrome.

$$\begin{aligned}
\delta\,(Read_2, b, b) &= (Read_2, \epsilon)\\
\delta\,(Read_2, a, a) &= (Read_2, \epsilon)\\
\delta\,(Read_2, \not{b}, \not{b}) &= \text{Accept}
\end{aligned}$$

We notice that Eq. sets (6.1) and (6.2) have some conflicting entries, which are mentioned here:

$$\begin{aligned}
&\delta\,(Read_1, a, a) = (Read_1, aa) \text{ from Eqs (6.1)}\\
&\delta\,(Read_1, a, a) = (Read_2, a) \text{ from Eqs (6.2)}
\end{aligned}$$

This is because the PDA is non-deterministic.

We have already discussed in Section 6.3, the differences between the transition function δ in a DPDA and an NPDA.

The conflicting entries can be explained using the transition function δ of the NPDA, which is defined as

$$\delta\text{: } Q \times \{\Sigma \cup (\epsilon)\} \times \Gamma \rightarrow \text{finite subsets of } Q \times \Gamma^*$$

The aforementioned example of conflicting entries can formally be written as

$$\delta\,(Read_1, a, a) = \{(Read_1, aa), (Read_2, a)\}$$

Thus, the NPDA in Fig. 6.14 is capable of accepting the language of all palindrome strings over $= \{a, b\}$. We see that it is however not possible to build a DPDA for the same CFL, as we cannot really decide when the mid-point is reached, in order to match the two halves of the input string using a single stack.

Simulation

Let us see a possible simulation for the acceptance of the string '*aba*'.

Current state	Stack contents	Tape and head
Start	$\not b$	$\underset{\uparrow}{a}\ b\ a\ \not b\ \not b\ \ldots$
$\downarrow$		
$Read_1$	$\not b$	$a\ \underset{\uparrow}{b}\ a\ \not b\ \not b\ \ldots$
$a\downarrow$		
Push a	$\not b\ a$	$a\ \underset{\uparrow}{b}\ a\ \not b\ \not b\ \ldots$
$\downarrow$		
$Read_1$	$\not b\ a$	$a\ b\ \underset{\uparrow}{a}\ \not b\ \not b\ \ldots$
$b\downarrow$		
$Read_2$	$\not b\ a$	$a\ b\ a\ \underset{\uparrow}{\not b}\ \not b\ \ldots$
$a\downarrow$		
Pop_1	$\not b$	$a\ b\ a\ \underset{\uparrow}{\not b}\ \not b\ \ldots$
$a\downarrow$		
$Read_2$	$\not b$	$a\ b\ a\ \not b\ \underset{\uparrow}{\not b}\ \ldots$
$\not b\downarrow$		
Pop_3	—	$a\ b\ a\ \not b\ \underset{\uparrow}{\not b}\ \ldots$
$\not b\downarrow$		
Accept		

Since NPDA is a possibilistic machine, the aforementioned simulation is one possible machine transition that leads to acceptance. This is interpreted considering the fact that b is a mid-point of '*aba*'.

Note: **No DPDA exists, which can accept the CFL expressed by the given grammar G. Thus, we can say that NPDA is more powerful than DPDA; NPDA accepts many more CFLs for which no DPDA can be built. In other words, the class of CFLs accepted by DPDA is a proper subset of the CFLs accepted by NPDA.**

6.6.1 Relative Powers of DPDA/NPDA and NFA/DFA

From the aforementioned discussion, we can conclude that a language accepted by an NPDA may not be accepted by a DPDA. As a result, for every NPDA, there may not exist an equivalent DPDA. This is comparatively different from the NFA/DFA scenario.

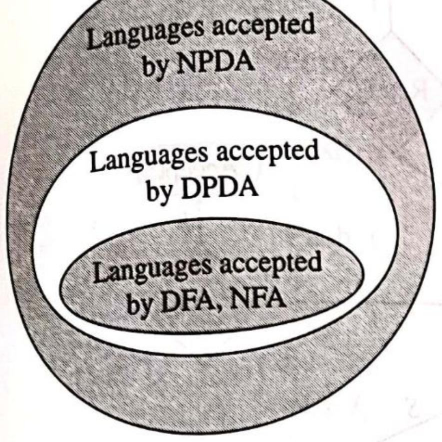

Languages accepted by NPDA
Languages accepted by DPDA
Languages accepted by DFA, NFA

Figure 6.15 Venn diagram explaining relative powers of DPDA/NPDA and DFA/NFA

Refer to the Venn diagram shown in Fig. 6.15 that depicts the relative powers of DFA, NFA, DPDA, and NPDA. Since, for every NFA there exists an equivalent DFA accepting the same regular language, we can say that DFA and NFA have equal powers. However, this does not hold true for PDAs—NPDA and DPDA have different capabilities. The NPDA can accept any CFL, while DPDA is a special case of NPDA that accepts only a subset of the CFLs accepted by the NPDA. Thus, DPDA is less powerful than NPDA.

6.7 EQUIVALENCE OF CFG AND PDA

So far in the examples, we have not directly used the CFG for building the PDA. Instead, we have used the language properties and a simple algorithm that uses a single stack to achieve the acceptance.

In this section, we shall discuss the algorithm to build an NPDA directly, based on the given CFG. This algorithm treats the NPDA mainly as a syntax analyser (or parser) that validates the input string based on the given grammar or CFG. Precisely, the NPDA is considered as a top-down parser, which uses leftmost derivation to generate the input string to be validated. The NPDA thus constructed uses the CFG to apply one grammar rule at a time and generates the input string, symbol by symbol. Each generated symbol is then matched with the input symbol read. If all the input symbols are matched, then the NPDA considers the input string as valid.

Example 6.7 Construct a PDA that accepts the language generated by the following CFG:

$$S \to SS \mid (S) \mid (\ )$$

Solution The required NPDA is constructed as shown in Fig. 6.16.

The algorithm is based on the leftmost derivation process as mentioned earlier. As we are aware, the leftmost derivation process involves replacing the leftmost non-terminal at every step by the right-hand side of the grammar rule. For example, if the provided grammar rule is of the form '$A \to \alpha$', then A is replaced by α. Further, whenever a terminal symbol is generated on the stack, an input symbol is read to match it.

Algorithm

1. Start with pushing the start symbol onto the stack.
2. After popping the start symbol, push the right-hand side of the production rule that is applicable in the leftmost derivation of the string onto the stack, but in the reverse way. Please note that the right-hand side of the production rule is pushed in the reverse order so as to receive the leftmost non-terminal symbol onto the top of the stack. In this way, the leftmost non-terminal symbol can be popped out of the stack and expanded further to achieve the leftmost derivation.
3. Whenever a terminal symbol is generated after popping from the stack, read an input symbol from the tape and check whether it matches the currently popped symbol.

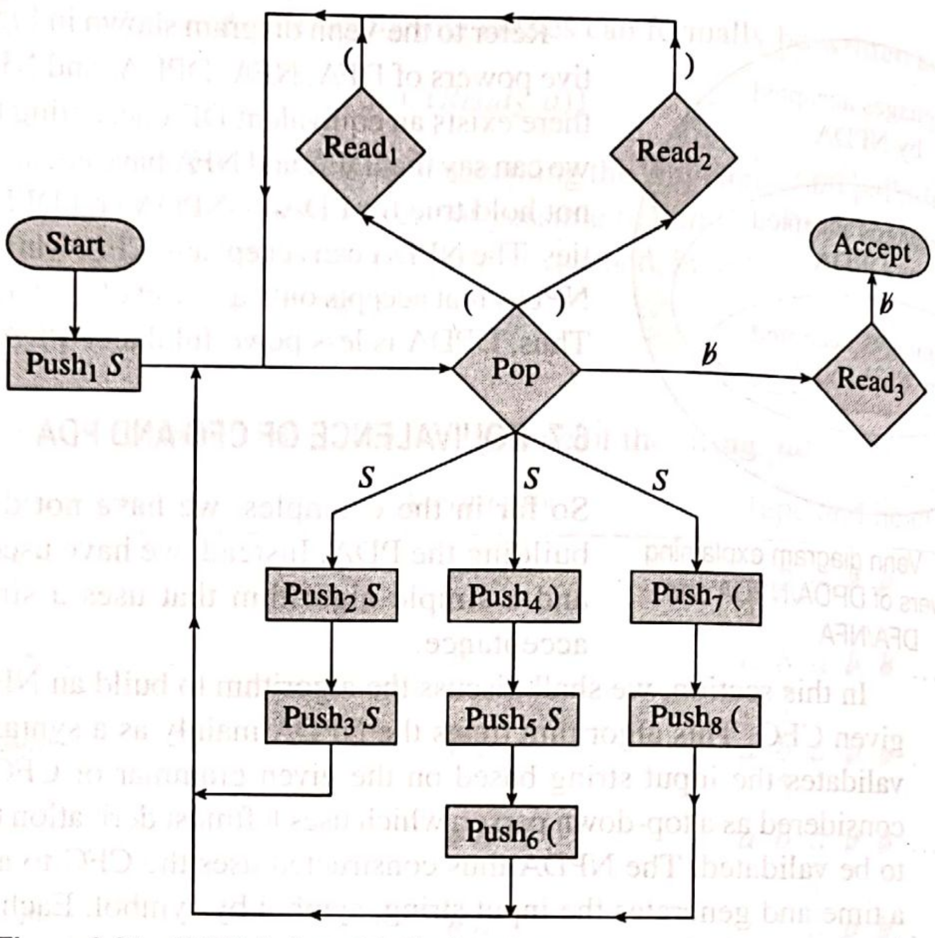


Figure 6.16 NPDA that accepts the language generated by $S \rightarrow SS \mid (S) \mid (\,)$

4. Continue the process till the entire input string is consumed.
5. When the stack is empty, that is, the popped symbol is b̸, read the input. If the symbol read from the input stream is b̸ (indicates end of input), then declare acceptance. This also denotes that the input string is entirely derived using the NPDA and that the input string is syntactically valid.

Simulation

Let us simulate the working of the NPDA in Fig. 6.16 for the string '(( ) ( ))'.

Leftmost derivation for the string '(( ) ( ))' is:

$$\begin{aligned} S &\Rightarrow (S) \\ &\Rightarrow (SS) \\ &\Rightarrow ((\,)S) \\ &\Rightarrow ((\,)(\,)) \end{aligned}$$

We shall use this derivation for the simulation, as follows:

Current state	Stack contents	Tape and head
Start ↓	b̸	(( ) ( )) b̸ b̸ ... ↑
Push$_1$ S ↓	b̸ S	(( ) ( )) b̸ b̸ ... ↑

Pop	$\not b$	$\underset{\Uparrow}{(}(\ )(\ ))\not b\not b\ldots$
$S\downarrow$		
Push_4)	$\not b\,)$	$\underset{\Uparrow}{(}(\ )(\ ))\not b\not b\ldots$
$\downarrow$		
$\text{Push}_5\,S$	$\not b\,)\,S$	$\underset{\Uparrow}{(}(\ )(\ ))\not b\not b\ldots$
$\downarrow$		
Push_6 (	$\not b\,)\,S\,($	$\underset{\uparrow}{(}(\ )(\ ))\not b\not b\ldots$
$\downarrow$		
Pop	$\not b\,)\,S$	$\underset{\uparrow}{(}(\ )(\ ))\not b\not b\ldots$
$(\downarrow$		
Read$_1$	$\not b\,)\,S$	$(\underset{\uparrow}{(}\ )(\ ))\not b\not b\ldots$
$(\downarrow$		
Pop	$\not b\,)$	$(\underset{\uparrow}{(}\ )(\ ))\not b\not b\ldots$
$S\downarrow$		
$\text{Push}_2\,S$	$\not b\,)\,S$	$(\underset{\uparrow}{(}\ )(\ ))\not b\not b\ldots$
$S\downarrow$		
$\text{Push}_3\,S$	$\not b\,)\,SS$	$(\underset{\uparrow}{(}\ )(\ ))\not b\not b\ldots$
$\downarrow$		
Pop	$\not b\,)\,S$	$(\underset{\uparrow}{(}\ )(\ ))\not b\not b\ldots$
$S\downarrow$		
Push_7)	$\not b\,)\,S\,)$	$(\underset{\uparrow}{(}\ )(\ ))\not b\not b\ldots$
$\downarrow$		
Push_8 (	$\not b\,)\,S\,)\,($	$(\underset{\uparrow}{(}\ )(\ ))\not b\not b\ldots$
$\downarrow$		
Pop	$\not b\,)\,S\,)$	$(\underset{\uparrow}{(}\ )(\ ))\not b\not b\ldots$
$(\downarrow$		
Read$_1$	$\not b\,)\,S\,)$	$((\ \underset{\uparrow}{)}(\ ))\not b\not b\ldots$
$(\downarrow$		
Pop	$\not b\,)\,S$	$((\ \underset{\uparrow}{)}(\ ))\not b\not b\ldots$
$)\downarrow$		
Read$_2$	$\not b\,)\,S$	$((\ )\underset{\uparrow}{(}\ ))\not b\not b\ldots$
$)\downarrow$		
Pop	$\not b\,)$	$((\ )\underset{\uparrow}{(}\ ))\not b\not b\ldots$
$S\downarrow$		
Push_7)	$\not b\,))$	$((\ )\underset{\uparrow}{(}\ ))\not b\not b\ldots$
$\downarrow$		
Push_8 (	$\not b\,))($	$((\ )\underset{\uparrow}{(}\ ))\not b\not b\ldots$
$\downarrow$		
Pop	$\not b\,))$	$((\ )\underset{\uparrow}{(}\ ))\not b\not b\ldots$
$(\downarrow$		
Read$_1$	$\not b\,))$	$((\ )(\ \underset{\uparrow}{)})\not b\not b\ldots$
$(\downarrow$		
Pop	$\not b\,)$	$((\ )(\ \underset{\uparrow}{)})\not b\not b\ldots$
$)\downarrow$		

Read$_2$	$\not b\,)$	$(()())\not b\not b\ldots$ ↑
$)\downarrow$		
Pop	$\not b$	$(()())\not b\not b\ldots$ ↑
$)\downarrow$		
Read$_2$	$\not b$	$(()())\not b\not b\ldots$ ↑
$)\downarrow$		
Pop	—	$(()())\not b\not b\ldots$ ↑
$\not b\downarrow$		
Read$_3$	—	$(()())\not b\not b\ldots$ ↑
$\not b\downarrow$		
Accept		

Example 6.8 Construct a PDA that accepts the language generated by the grammar

$$S \rightarrow S + S \mid S * S \mid 4$$

Solution The required NPDA is constructed as shown in Fig. 6.17.

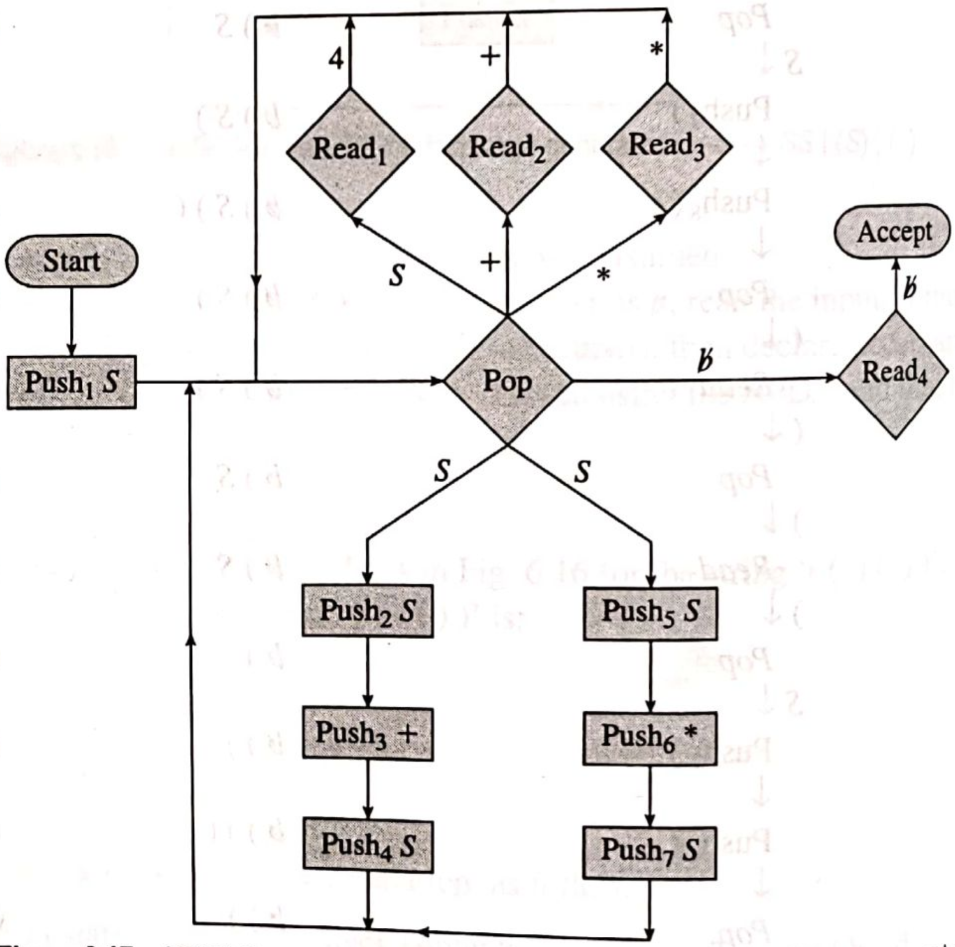


Figure 6.17 NPDA that accepts the language generated by $S \rightarrow S + S \mid S * S \mid 4$

Simulation

Let us simulate the working of the NPDA that we have constructed for the string '4 + 4*4'

Leftmost derivation for the string '4 + 4 * 4' is as follows:

$$
\begin{aligned}
S &\Rightarrow S + S \\
&\Rightarrow 4 + S \\
&\Rightarrow 4 + S * S \\
&\Rightarrow 4 + 4 * S \\
&\Rightarrow 4 + 4 * 4
\end{aligned}
$$

Based on the leftmost derivation, the simulation for the acceptance of the string '4 + 4 * 4' is as follows:

Current state	Stack contents	Tape and head
Start	ƀ	$\underset{\uparrow}{4}$ + 4 * 4 ƀ ƀ …
↓		
Push$_1$ S	ƀ S	$\underset{\uparrow}{4}$ + 4 * 4 ƀ ƀ …
↓		
Pop	ƀ	$\underset{\uparrow}{4}$ + 4 * 4 ƀ ƀ …
S ↓		
Push$_2$ S	ƀ S	$\underset{\uparrow}{4}$ + 4 * 4 ƀ ƀ …
S ↓		
Push$_3$ +	ƀ S +	$\underset{\uparrow}{4}$ + 4 * 4 ƀ ƀ …
↓		
Push$_4$ S	ƀ S + S	$\underset{\uparrow}{4}$ + 4 * 4 ƀ ƀ …
↓		
Pop	ƀ S +	$\underset{\uparrow}{4}$ + 4 * 4 ƀ ƀ …
S ↓		
Read$_1$	ƀ S +	4 $\underset{\uparrow}{+}$ 4 * 4 ƀ ƀ …
4 ↓		
Pop	ƀ S	4 $\underset{\uparrow}{+}$ 4 * 4 ƀ ƀ …
+ ↓		
Read$_2$	ƀ S	4 + $\underset{\uparrow}{4}$ * 4 ƀ ƀ …
+ ↓		
Pop	ƀ	4 + $\underset{\uparrow}{4}$ * 4 ƀ ƀ …
S ↓		
Push$_5$ S	ƀ S	4 + $\underset{\uparrow}{4}$ * 4 ƀ ƀ …
↓		
Push$_6$ *	ƀ S*	4 + $\underset{\uparrow}{4}$ * 4 ƀ ƀ …
↓		
Push$_7$ S	ƀ S * S	4 + $\underset{\uparrow}{4}$ * 4 ƀ ƀ …
↓		
Pop	ƀ S *	4 + $\underset{\uparrow}{4}$ * 4 ƀ ƀ …
S ↓		
Read$_1$	ƀ S *	4 + 4 $\underset{\uparrow}{*}$ 4 ƀ ƀ …
4 ↓		

Pop	ƀ*S*	4 + 4 * 4 ƀ ƀ ... ↑
* ↓		
Read$_3$	ƀ*S*	4 + 4 * 4 ƀ ƀ ... ↑
* ↓		
Pop	ƀ	4 + 4 * 4 ƀ ƀ ... ↑
S ↓		
Read$_1$	ƀ	4 + 4 * 4 ƀ ƀ ... ↑
4 ↓		
Pop	—	4 + 4 * 4 ƀ ƀ ... ↑
ƀ ↓		
Read$_4$	—	4 + 4 * 4 ƀ ƀ ... ↑
ƀ ↓		
Accept		

6.7.1 NPDA Construction using Chomsky Normal Form

As we know, Chomsky normal form (CNF) mandates only two forms of production rules:

1. $S \rightarrow AB$
2. $S \rightarrow a$

Production rule of type 1 has exactly two non-terminals on the right-hand side, while the production rule of type 2 contains a single terminal symbol at the right-hand side.

In the previous two examples, we observe that even the terminal symbols are pushed onto the stack and again popped to match the input symbols. The complexity of this algorithm can be reduced using CNF. If we express the grammar in CNF, then the production rule of type 1 is used to push the right-hand side onto the stack, while the rule of type 2 is directly used to match the input symbols from the tap. This has two benefits:

1. Terminals are not pushed onto the stack.
2. At most, two non-terminal symbols are pushed onto the stack at every step in the derivation process.

This in turn helps control the stack growth, which as we have seen, is much more without CNF. On the other hand, grammar without CNF may have multiple symbols on the right-hand side of production rules.

Example 6.9 Construct a PDA that accepts the following language:

$$L = \{a^{2n} \mid n > 0\}$$

Solution The CFG for the given language can be written as

$$S \rightarrow SS \mid aa$$

Let us express the grammar in CNF:

$$S \rightarrow SS \mid AA$$
$$A \rightarrow a$$

The NPDA is constructed using the CFG expressed in CNF as shown in Fig. 6.18.

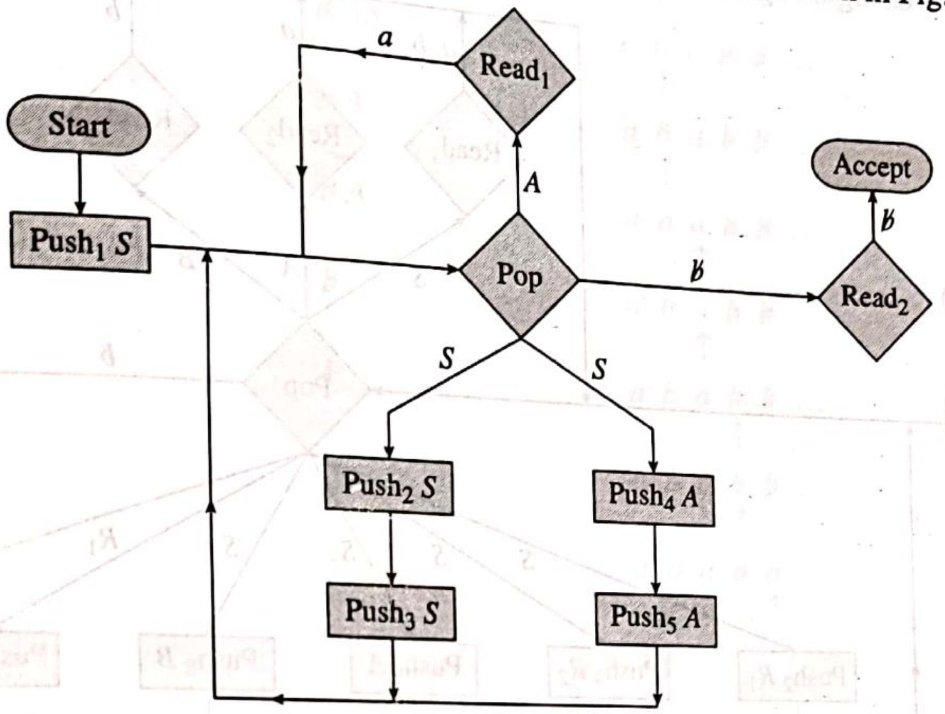


Figure 6.18 NPDA that accepts $\{a^{2n} \mid n > 0\}$

We see from Fig. 6.18 that in the grammar expressed in CNF, no terminal gets pushed onto the stack.

Example 6.10 Construct a PDA that accepts all palindrome strings over $\Sigma = \{a, b\}$.

Solution This is the same as Example 6.6 that we solved earlier. Let us apply the CNF and then compare the efficiency of the resultant PDA.

The CFG for the language is defined as

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

Removing the ϵ-production, the grammar can be written as

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

Now, let us convert this grammar to CNF:

$$S \rightarrow A\,S\,A \mid B\,S\,B \mid A\,A \mid B\,B \mid a \mid b$$
$$A \rightarrow a$$
$$B \rightarrow b$$

Therefore, the final CFG expressed in the CNF, which can be considered for constructing the required PDA is as follows:

$$S \rightarrow A\,R_1 \mid B\,R_2 \mid A\,A \mid B\,B \mid a \mid b$$
$$R_1 \rightarrow SA$$
$$R_2 \rightarrow SB$$
$$A \rightarrow a$$
$$B \rightarrow b$$

Using this grammar, the NPDA can be constructed as shown in Fig. 6.19.

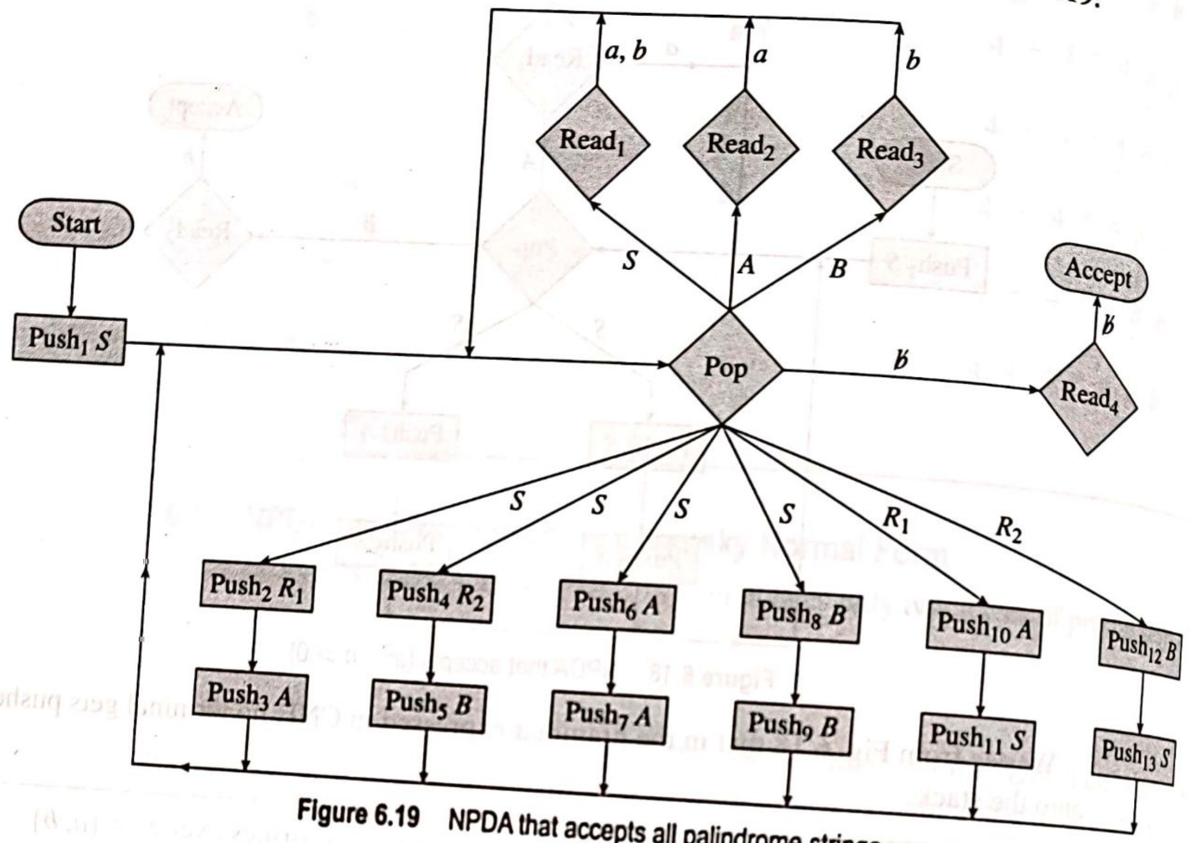


Figure 6.19 NPDA that accepts all palindrome strings

Simulation

Let us simulate the working of the NPDA in Fig. 6.19 for the string '*aba*'.

Current state	Stack contents	Tape and head
Start ↓	$\not b$	$\underset{\uparrow}{a}\ b\ a\ \not b\ \not b \ldots$
Push$_1$ S ↓	$\not b\, S$	$\underset{\uparrow}{a}\ b\ a\ \not b\ \not b \ldots$
Pop S ↓	$\not b$	$\underset{\uparrow}{a}\ b\ a\ \not b\ \not b \ldots$
Push$_2$ R_1 ↓	$\not b\, R_1$	$\underset{\uparrow}{a}\ b\ a\ \not b\ \not b \ldots$
Push$_3$ A ↓	$\not b\, R_1 A$	$\underset{\uparrow}{a}\ b\ a\ \not b\ \not b \ldots$
Pop A ↓	$\not b\, R_1$	$\underset{\uparrow}{a}\ b\ a\ \not b\ \not b \ldots$
Read$_2$ a ↓	$\not b\, R_1$	$a\ \underset{\uparrow}{b}\ a\ \not b\ \not b \ldots$
Pop R_1 ↓	$\not b$	$a\ \underset{\uparrow}{b}\ a\ \not b\ \not b \ldots$

$\text{Push}_{10}\,A$	$\not b A$	
$\downarrow$		
$\text{Push}_{11}\,S$	$\not b AS$	$a\ \underset{\uparrow}{b}\ a\ \not b\ \not b \ldots$
$\downarrow$		
Pop	$\not b A$	$a\ \underset{\uparrow}{b}\ a\ \not b\ \not b \ldots$
$S \downarrow$		
Read$_1$	$\not b A$	$a\ \underset{\uparrow}{b}\ a\ \not b\ \not b \ldots$
$b \downarrow$		
Pop	$\not b$	$a\ b\ \underset{\uparrow}{a}\ \not b\ \not b \ldots$
$A \downarrow$		
Read$_2$	$\not b$	$a\ b\ \underset{\uparrow}{a}\ \not b\ \not b \ldots$
$a \downarrow$		
Pop	—	$a\ b\ a\ \underset{\uparrow}{\not b}\ \not b \ldots$
$\not b \downarrow$		
Read$_4$	—	$a\ b\ a\ \underset{\uparrow}{\not b}\ \not b \ldots$
$\not b \downarrow$		
Accept		$a\ b\ a\ \not b\ \underset{\uparrow}{\not b} \ldots$

This simulation is based on the following leftmost derivation:

$$\begin{aligned} S &\Rightarrow \underline{A}\,R_1 \\ &\Rightarrow a\,\underline{R_1} \\ &\Rightarrow a\,\underline{S}\,A \\ &\Rightarrow a\,b\,\underline{A} \\ &\Rightarrow a\,b\,a \end{aligned}$$

6.8 CLOSURE PROPERTIES OF CFLs

Context-free languages (CFLs), like RLs, are also closed under union, concatenation, and Kleene closure. Let us attempt to prove these facts using CFG and PDA concepts.

Theorem 6.1

If L_1 and L_2 are CFLs, then their union '$L_1 \cup L_2$' is also a CFL. In other words, *CFLs are closed under union.*

Proof

Let us consider two context-free languages, L_1 and L_2. This means that these languages can be represented using context-free grammars (CFGs).

Let us assume that L_1 is represented by the CFG G_1, which has S_1 as its starting non-terminal (or start symbol). Hence, G_1 begins with a production of the form

$$S_1 \rightarrow \alpha_1$$

$$\ldots$$

6.9 ADDITIONAL PDA EXAMPLES

In this section, let us discuss some more examples on the construction of PDA.

Example 6.11 Construct a PDA that recognizes the following language:

$$\{a^n x \mid n \geq 0, x \in \{a, b\}^* \text{ and } |x| \leq n\}$$

Solution We see that for the given language, it is not possible to determine where the string of a's—that is, a^n—ends, and the string x starts. This is because x also consists of a's. Hence, we design an NPDA as shown in Fig. 6.20.

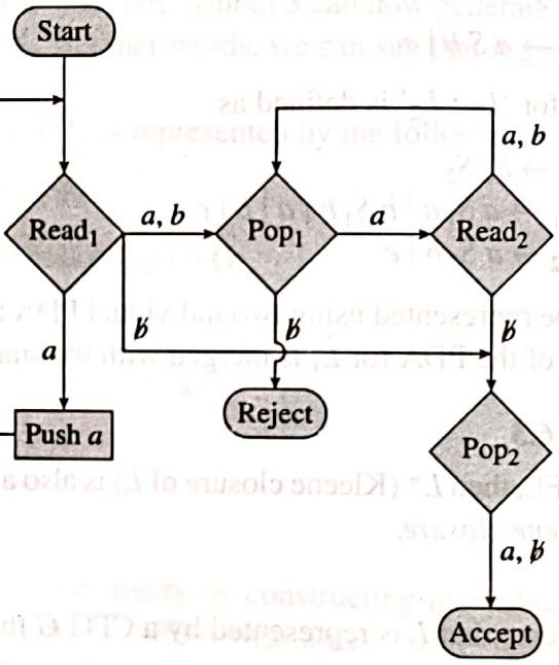


Figure 6.20 NPDA that accepts $\{a^n x \mid n \geq 0, x \in \{a, b\}^* \text{ and } |x| \leq n\}$

Algorithm

1. Push all a's onto the stack till we reach the beginning of x. (Remember that x may start with either a or b. Determinism is impossible to achieve if x starts with a).
2. Read one symbol from x—either a or b—and pop one a from the stack.
3. Continue the process till you reach the end of the input.
4. When the input ends, that is, when you read $\not{b}$ on the tape, the stack may either be empty (if $|x| = n$) or may have a few a's left (if $|x| < n$).
5. In either case, the stack cannot be empty before the input ends, as $|x| > n$ is not allowed. Hence, in such a case, the input string should be rejected.

Example 6.12 Construct a PDA for the language described as follows:
The set of all strings over alphabet $\{a, b\}$ with exactly equal number of a's and b's.

Solution We need to consider the fact that the string might begin with either a or b. Therefore, pushing only a's or only b's will not work. We need to push whenever the stack is empty or the top of the stack carries the same symbol as the one just read. Refer to the algorithm that follows. The required DPDA is depicted in Fig. 6.21.

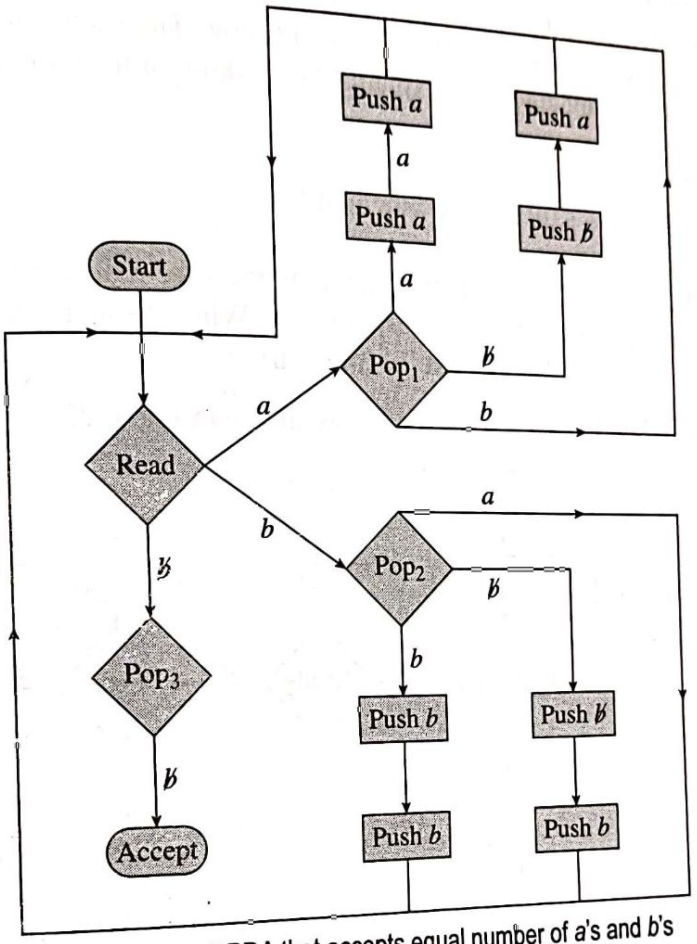


Figure 6.21 DPDA that accepts equal number of a's and b's

Algorithm

1. Read a symbol, either *a* or *b*.
2. If the stack is empty, that is, the top of the stack top contains *b̸*, push the symbol that has been read.
3. If the top of the stack contains the same symbol as the symbol that has been read, then also push this symbol.
4. If the top of the stack carries a symbol that is different from what has been read—for example, if the symbol read is *a* and the top of the stack contains *b*, or vice versa—pop the symbol onto the top of the stack. This contributes to matching *a*'s with *b*'s, or vice versa.
5. Continue with the aforementioned four steps until the input string ends, that is, until you read *b̸* on the tape.
6. When the input string ends, then the top of this stack should be *b̸*. If this holds true, then accept the input string.

Example 6.13 Construct a PDA that accepts the language $L = \{a^n b^m a^n \mid m, n \geq 1\}$.

Solution For this language, we need to match the number of a's at the beginning and th end of any input string. This means that we must push all the a's before the string of b' and match them later with the a's that follow. The b's in between can be read and ignored. The following algorithm explains the working of the required PDA.

Algorithm

1. Push all a's till you read the first b.
2. Read and ignore all b's.
3. For every a you read after that, pop one a from the stack.
4. Continue step three till the input ends. When the input ends, the stack should be empty if this holds true, accept the input string.

The DPDA can be constructed as shown in Fig. 6.22.

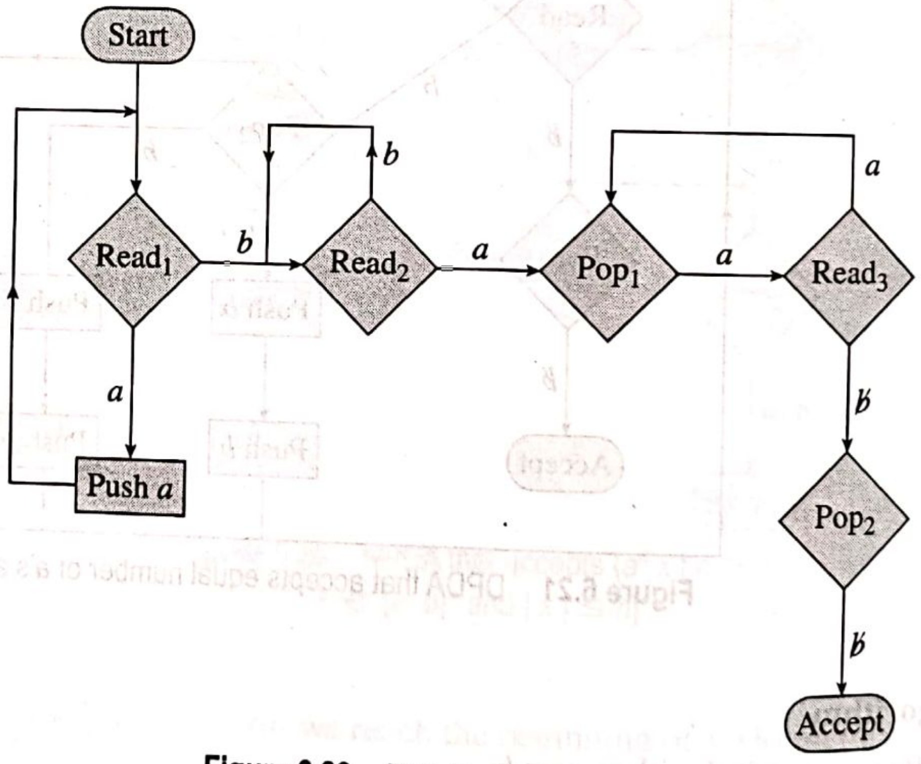


Figure 6.22 PDA that accepts $a^n b^m a^n$

SUMMARY

A pushdown stack-memory machine (PDM) is a computational model that we can use to solve any problem that has an algorithmic solution and requires a single stack memory.

A PDM accepts or recognizes regular languages (RLs) as well as context-free languages (CFLs). Since the set of RLs is a proper subset of the class of CFLs, for every finite state machine (FSM), there exists an equivalent PDM, but not vice versa. Hence, a PDM is more powerful than the FSM due to its infinite memory in the form of an external stack.