Chapter 3

Pushdown Automata and Context Free Languages

As we have seen, Finite Automata are somewhat limited in the languages they can recognize. Pushdown Automata are another type of machine that can recognize a wider class of languages. Context Free Grammars, which we can think of as loosely analogous to regular expressions, provide a method for describing a class of languages called Context Free Languages. As we will see, the Context Free Languages are exactly the languages recognized by Pushdown Automata.

Pushdown Automata can be used to recognize certain types of structured input. In particular, they are an important part of the front end of compilers. They are used to determine the organization of the programs being processed by a compiler; tasks they handle include forming expression trees from input arithmetic expressions and determining the scope of variables.

Context Free Languages, and an elaboration called Probabilistic Context Free Languages, are widely used to help determine sentence organization in computer-based text understanding (e.g., what is the subject, the object, the verb, etc.).

3.1 Pushdown Automata

A Pushdown Automata, or PDA for short, is simply an NFA equipped with a single stack. As with an NFA, it moves from vertex to vertex as it reads its input, with the additional possibility of also pushing to and popping from its stack as part of a move from one vertex to another. As with an NFA, there may be several viable computation paths. In addition, as with an NFA, to recognize an input string \( w \), a PDA \( M \) needs to have a recognizing path, from its start vertex to a final vertex, which it can traverse on input \( w \).

Recall that a stack is an unbounded store which one can think of as holding the items it stores in a tower (or stack) with new items being placed (written) at the top of the tower and items being read and removed (in one operation) from the top of the tower. The first operation is called a Push and the second a Pop. For example, if we perform the sequence
of operations Push(A), Push(B), Pop, Push(C), Pop, Pop, the 3 successive pops will read the items B, C, A, respectively. The successive states of the stack are shown in Figure 3.1.

![Figure 3.1: Stack Behavior.](image)

Let’s see how a stack allows us to recognize the following language \( L_1 = \{a^ib^i \mid i \geq 0 \} \). We start by explaining how to process a string \( w = a^ib^i \in L \). As the PDA reads the initial string of \( a \)'s in its input, it pushes a corresponding equal length string of \( A \)'s onto its stack (one \( A \) for each \( a \) read). Then, as \( M \) reads the \( b \)'s, it seeks to match them one by one against the \( A \)'s on the stack (by popping one \( A \) for each \( b \) it reads). \( M \) recognizes its input exactly if the stack becomes empty on reading the last \( b \).

In fact, PDAs are not allowed to use a Stack Empty test. We use a standard technique, which we call \( \$\)-shielding, to simulate this test. Given a PDA \( M \) on which we want to perform Stack Empty tests, we create a new PDA \( \overline{M} \) which is identical to \( M \) apart from the following small changes. \( \overline{M} \) uses a new, additional symbol on its stack, which we name \( \$ \). Then at the very start of the computation, \( \overline{M} \) pushes a \( \$ \) onto its stack. This will be the only occurrence of \( \$ \) on its stack. Subsequently, \( \overline{M} \) performs the same steps as \( M \) except that when \( M \) seeks to perform a Stack Empty test, \( \overline{M} \) pops the stack and then immediately pushes the popped symbol back on its stack. The simulated stack is empty exactly if the popped symbol was a \( \$ \).

Next, we explain what happens with strings outside the language \( L_1 \). We do this by looking at several categories of strings in turn.

1. \( a^ib^j, h < i \).
   After the last \( b \) is read, there will still be one or more \( A \)'s on the stack, indicating the input is not in \( L_1 \).

2. \( a^ib^j, j > i \).
   On reading the \((i + 1)\)st \( b \), there is an attempt to pop the now empty stack to find a matching \( A \); this attempt fails, and again this indicates the input is not in \( L_1 \).

3. The only other possibility for the input is that it contains the substring \( ba \); as already described, the processing consists of an \( a \)-reading phase, followed by a \( b \)-reading phase. The \( a \) in the substring \( ba \) is being encountered in the \( b \)-reading phase and once more this input is easily recognized as being outside \( L_1 \).

As with an NFA, we can specify the computation using a directed graph, with the edge labels indicating the actions to be performed when traversing the given edge. To recognize an input \( w \), the PDA needs to be able to follow a path from its start vertex to a final vertex.
starting with an empty stack, where the path’s read labels spell out the input, and the stack operations on the path are consistent with the stack’s ongoing contents as the path is traversed. To emphasize the fact that a path traversal involves both input reading and stack operations, we will often call the traversed path a computation Path. Also, as before, we say that a vertex \( v \) is a destination vertex for string \( s \) if the PDA can end up at vertex \( v \) after reading all of string \( s \).

A PDA \( M_1 \) recognizing \( L_1 \) is shown in Figure 3.2. Because the descriptions of the vertices are quite long, we have given them the shorter names \( p_1 - p_4 \). These descriptions specify exactly those strings and the corresponding stack contents for which the vertex in question is a destination vertex. We will verify the correctness of these specifications by looking at what happens on the possible inputs. To do this effectively, we need to divide the strings into appropriate categories.

An initial understanding of what \( M_1 \) recognizes can be obtained by ignoring what the stack does and viewing the machine as just an NFA (i.e. using the same graph but with just the reads labeling the edges). See Figure 3.3 for the graph of the NFA \( N_1 \) derived from \( M_1 \). The significant point is that if \( M_1 \) can reach vertex \( p \) on input \( w \) using computation path

\[
\begin{align*}
\text{Read } b & \quad \text{Pop } A \\
\text{Pop } \$$ & \quad \text{Read } \lambda \\
\text{Read } a & \quad \text{Push } A \\
\text{Push } \$$ & \quad \text{Read } \lambda
\end{align*}
\]

Figure 3.2: PDA \( M_1 \) recognizing \( L_1 = \{ a^i b^i | i \leq 0 \} \).

\[
\begin{align*}
\text{Read } a & \quad \text{Push } A \\
\text{Push } \$$ & \quad \text{Read } \lambda \\
p_1: & \quad \lambda \text{ read, } \lambda \text{ read} \\
p_2: & \quad a^i \text{ read, } i \geq 0 \\
p_3: & \quad a^i b^h \text{ read, } i \geq h \geq 0 \\
p_4: & \quad a^i b^h \text{ read, } i \geq h \geq 0
\end{align*}
\]

Figure 3.3: The NFA \( N_1 \) derived from \( M_1 \).
recognized by $M_1$ is also recognized by $N_1$: $L(M_1) \subseteq L(N_1)$.

It is not hard to see that $N_1$ recognizes $a^*b^*$. If follows that $M_1$ recognizes a subset of $a^*b^*$. So to explain the behavior of $M_1$ in full it suffices to look at what happens on inputs of the form $a^ib^j$, $i, j \geq 0$, which we do by examining five subcases that account for all such strings.

1. $\lambda$.
   $M_1$ starts at $p_1$. On pushing $\$, $p_2$ and $p_3$ can be reached. Then on popping the $\$, $p_4$ can be reached. Note that the specification of $p_2$ holds with $i = 0$, that of $p_3$ with $i = h = 0$, and that of $p_4$ with $i = 0$. Thus the specification at each vertex includes the case that the input is $\lambda$.

2. $a^i$, $i \geq 1$.
   To read $a^i$, $M_1$ needs to push $\$, then follow edge $(p_1, p_2)$, and then follow edge $(p_2, p_2)$ $i$ times. This puts $\lambda A^i$ on the stack. Thus on input $a^i$, $p_2$ can be reached and its specification is correct. In addition, the edge to $p_3$ can be traversed without any additional reads or stack operations, and so the specification for $p_3$ with $h = 0$ is correct for this input.

3. $a^ib^h$, $1 \leq h < i$.
   The only place to read $b$ is on edge $(p_3, p_3)$. Thus, for this input, $M_1$ reads $a^i$ to bring it to $p_3$ and then follows $(p_3, p_3)$ $h$ times. This leaves $\lambda A^{i-h}$ on the stack, and consequently the specification of $p_3$ is correct for this input. Note that as $h < i$, edge $(p_3, p_4)$ cannot be followed as $\$ is not on the stack top.

4. $a^ib^i$, $i \geq 1$.
   After reading the $i b$’s, $M_1$ can be at vertex $p_3$ as explained in (3). Now, in addition, edge $(p_3, p_4)$ can be traversed and this pops the $\$ from the stack, leaving it empty. So the specification of $p_4$ is correct for this input.

5. $a^ib^j$, $j > i$.
   On reading $a^ib^j$, $M_1$ can reach $p_3$ with the stack holding $\$ or reach $p_4$ with an empty stack, as described in (4). From $p_3$ the only available move is to $p_4$, without reading anything further. At $p_4$ there is no move, so the rest of the input cannot be read, and thus no vertex can be reached on this input.

This is a very elaborate description which we certainly don’t wish to repeat for each similar PDA. We can describe $M_1$’s functioning more briefly as follows.

$M_1$ checks that its input has the form $a^*b^*$ (i.e. all the $a$’s precede all the $b$’s) using its underlying NFA (i.e. without using the stack). The underlying NFA is often called its finite control. In tandem with this, $M_1$ uses its $\$-shielded stack to match the $a$’s against the $b$’s, first pushing the $a$’s on the stack (it is understood that in fact $A$’s are being pushed) and then popping them off, one for one, as the $b$’s are read, confirming that the numbers of $a$’s and $b$’s are equal.
The detailed argument we gave above is understood, but not spelled out.

Now we are ready to define a PDA more precisely. As with an NFA, a PDA consists of a directed graph with one vertex, start, designated as the start vertex, and a (possibly empty) subset of vertices designated as the final set, \( F \), of vertices. As before, in drawing the graph, we show final vertices using double circles and indicate the start vertex with a double arrow. Each edge is labeled with the actions the PDA performs on following that edge.

For example, the label on edge \( e \) might be: Pop \( A \), read \( b \), Push \( C \), meaning that the PDA pops the stack, reads the next input character, and if the pop returns an \( A \) and the character read is a \( b \), then the PDA can traverse \( e \), which entails it pushing \( C \) onto the stack.

Some or all of these values may be \( \lambda \): Pop \( \lambda \) means that no Pop is performed, read \( \lambda \) that no read occurs, and Push \( \lambda \) that no Push happens. To avoid clutter, we usually omit the \( \lambda \)-labeled terms; for example, instead of Pop \( \lambda \), read \( \lambda \), Push \( C \), we write Push \( C \). Also, to avoid confusion in the figures, if there are multiple triples of actions that take the PDA from a vertex \( u \) to a vertex \( v \), we use multiple edges from \( u \) to \( v \), one for each triple.

In sum, a label, which specifies the actions accompanying a move from vertex \( u \) to vertex \( v \), has up to three parts.

1. Pop the stack and check that the returned character has a specified value (in our example this is the value \( A \)).

2. Read the next character of input and check that it has a specified value (in our example, the value \( b \)).

3. Push a specified character onto the stack (in our example, the character \( C \)).

From an implementation perspective it may be helpful to think in terms of being able to peek ahead, so that one can see the top item on the stack without actually popping it, and one can see the next input character (or that one is at the end of the input) without actually reading forward.

One further rule is that an empty stack may not be popped.

A PDA also comes with an input alphabet \( \Sigma \) and a stack alphabet \( \Gamma \) (these are the symbols that can be written on the stack). It is customary for \( \Sigma \) and \( \Gamma \) to be disjoint, in part to avoid confusion. To emphasize this disjointness, we write the characters of \( \Sigma \) using lowercase letters and those of \( \Gamma \) using uppercase letters.

Because the stack contents make it difficult to describe the condition of the PDA after multiple moves, we use the transition function here to describe possible out edges from single vertices only. Accordingly, \( \delta(p,A,b) = \{(q_1,C_1),(q_2,C_2),\ldots,(q_l,C_l)\} \) indicates that the edges exiting vertex \( p \) and having both Pop \( A \) and read \( b \) in their label are the edges going to vertices \( q_1,q_2,\ldots,q_l \) where the rest of the label for edge \( (p,q_i) \) includes Push \( C_i \), for \( 1 \leq i \leq l \). That is \( \delta(p,A,b) \) specifies the possible moves out of vertex \( p \) on popping character \( A \) and reading \( b \). (Recall that one or both of \( A \) and \( b \) might be \( \lambda \), as might some or all of the \( C_i \)'s.)

In sum, a PDA \( M \) consists of a 6-tuple: \( M = (\Sigma,\Gamma,V,start,F,\delta) \), where
3.1. PUSHDOWN AUTOMATA

1. Σ is the input alphabet,
2. Γ is the stack alphabet,
3. \( V \) is the vertex or state set,
4. \( F \subseteq V \) is the final vertex set,
5. \( \text{start} \) is the start vertex, and
6. \( \delta \) is the transition function, which specifies the edges and their labels.

Recognition is defined as for an NFA, that is, PDA \( M \) recognizes input \( w \) if there is a path that \( M \) can follow on input \( w \) that takes \( M \) from its start vertex to a final vertex. We call this path a \( w \)-recognizing computation path to emphasize that stack operations may occur in tandem with the reading of input \( w \). More formally, \( M \) recognizes \( w \) if there is a path \( \text{start} = p_0, p_1, \ldots, p_m \), where \( p_m \) is a final vertex, the label on edge \((p_{i-1}, p_i)\) is \((\text{Read } a_i, \text{Pop } B_i, \text{Push } C_i)\), for \( 1 \leq i \leq m \), and the stack contents at vertex \( p_i \) is \( \sigma_i \), for \( 0 \leq i \leq m \), where

1. \( a_1a_2\cdots a_m = w \),
2. \( \sigma_0 = \lambda \),
3. and \( \text{Pop } B_i, \text{Push } C_i \) applied to \( \sigma_{i-1} \) produces \( \sigma_i \) for \( 1 \leq i \leq m \).

We write \( L(M) \) for the language, or set of strings, recognized by \( M \).

Next, we show some more examples of languages that can be recognized by PDAs.

**Example 3.1.1.** \( L_2 = \{ a^i b^i \mid i \geq 0 \} \).

PDA \( M_2 \) recognizing \( L_2 \) is shown in Figure 3.4. The processing by \( M_2 \) is similar to that of \( M_1 \). \( M_2 \) checks that its input has the form \( a^i b^i \) using its finite control. In tandem, \( M_2 \) uses its $-shielded stack to match the \( a \)'s against the \( b \)'s, first pushing the \( a \)'s on the stack (actually \( A \)'s are being pushed), then reads the \( c \) without touching the stack, and finally pops the \( a \)'s off, one for one, as the \( b \)'s are read, confirming that the numbers of \( a \)'s and \( b \)'s are equal.

**Example 3.1.2.** \( L_3 = \{ wcw^R \mid w \in \{a, b\}^* \} \).

PDA \( M_3 \) recognizing \( L_3 \) is shown in Figure 3.5. \( M_3 \) uses its $-shielded stack to match the \( w \) against the \( w^R \), as follows. It pushes \( w \) on the stack (the end of the substring \( w \) being indicated by reaching the \( c \)). At this point, the stack content read from the top is \( w^R \$, so popping down to the $ outputs the string \( w^R \). This stack contents is readily compared to the string following the \( c \). The input is recognized exactly if they match.

**Example 3.1.3.** \( L_4 = \{ ww^R \mid w \in \{a, b\}^* \} \).

PDA \( M_4 \) recognizing \( L_4 \) is shown in Figure 3.6. This is similar to Example 3.1.2. The one difference is that the PDA \( M_4 \) can decide at any point to stop reading \( w \) and begin reading \( w^R \). Of course there is only one correct switching location, at most, but as \( M_4 \) does not know where it is, \( M_4 \) considers all possibilities by means of its nondeterminism.
Figure 3.4: PDA $M_2$ recognizing $L_2 = \{a^i cb^i \mid i \geq 0\}$.

Figure 3.5: PDA $M_3$ recognizing $L_3 = \{wcw^R \mid w \in \{a, b\}^*\}$. Z denotes string $z$ in capital letters.
3.2 CLOSURE PROPERTIES

Example 3.1.4. \(L_5 = \{a^i b^j c^k \mid i = j \text{ or } i = k\}\).

PDA \(M_5\) recognizing \(L_5\) is shown in Figure 3.7.

This is the union of languages \(L_6 = \{a^i b^j c^k \mid i, k \geq 0\}\) and \(L_7 = \{a^i b^j c^i \mid i, j \geq 0\}\), each of which is similar to \(L_2\). \(M_5\), the PDA recognizing \(L_5\) uses submachines to recognize each of \(L_6\) and \(L_7\). \(M_5\)'s first move from its start vertex is to traverse (Push \$)-edges to the start vertices of the submachines. The net effect is that \(M_5\) recognizes the union of the languages recognized by the submachines. As the submachines are similar to \(M_2\) they are not explained further.

3.2 Closure Properties

Lemma 3.2.1. Let \(A\) and \(B\) be languages recognized by PDAs \(M_A\) and \(M_B\), respectively. Then \(A \cup B\) is also recognized by a PDA called \(M_{A \cup B}\).

Proof. The graph of \(M_{A \cup B}\) consists of the graphs of \(M_A\) and \(M_B\) plus a new start vertex \(start_{A \cup B}\), which is joined by \(\lambda\)-edges to the start vertices \(start_A\) and \(start_B\) of \(M_A\) and \(M_B\), respectively. Its final vertices are the final vertices of \(M_A\) and \(M_B\). The graph is shown in figure 3.8.

While it is clear that \(L(M_{A \cup B}) = L(M_A) \cup L(M_B)\), we present the argument for completeness.

First, we show that \(L(M_{A \cup B}) \subseteq L(M_A) \cup L(M_B)\), i.e. if \(w \in L(M_{A \cup B})\) then \(w \in L(M_A) \cup L(M_B)\). So suppose that \(w \in L(M_{A \cup B})\). Then there is a \(w\)-recognizing computation path from \(start_{A \cup B}\) to a final vertex \(f\). If \(f\) lies in \(M_A\), then removing the first edge of \(P\) leaves a path \(P'\) from \(start_A\) to \(f\). Further, at the start of \(P'\), the stack is empty and nothing has
Figure 3.7: PDA $M_5$ recognizing $L_5 = \{a^i b^j c^k \mid i = j \text{ or } i = k\}$.

Figure 3.8: PDA $M_{A \cup B}$. 
been read, so $P'$ is a $w$-recognizing path in $M_A$. That is, $w \in L(M_A)$. Similarly, if $f$ lies in $M_B$, then $w \in L(M_B)$. In either case, $w \in L(M_A) \cup L(M_B)$.

Second, we show that $L(M_A) \cup L(M_B) \subseteq L(M_{A\cup B})$, i.e. if $w \in L(M_A) \cup L(M_B)$ then $w \in L(M_{A\cup B})$. Suppose that $w \in L(M_A)$. Then there is a $w$-recognizing computation path $P'$ from $start_A$ to a final vertex $f$ in $M_A$. Adding the $\lambda$-edge $(start_{A\cup B}, start_A)$ to the beginning of $P'$ creates a $w$-recognizing computation path in $M_{A\cup B}$, showing that $L(M_A) \subseteq L(M_{A\cup B})$. Similarly, if $w \in L(M_B)$, then $w \in L(M_{A\cup B})$. Thus if $w \in L(M_A) \cup L(M_B)$ then $w \in L(M_{A\cup B})$.

Our next construction is simplified by the following technical lemma.

**Lemma 3.2.2.** Let PDA $M$ recognize $L$. There is an $L$-recognizing PDA $M'$ with the following properties: $M'$ has only one final vertex, final$_{M'}$, and $M'$ will always have an empty stack when it reaches final$_{M'}$. $M'$ is called a single-destination, empty-stack PDA.

**Proof.** The idea is quite simple. $M'$ simulates $M$ using a $\$-$shielded stack. When $M$'s computation is complete, $M'$ moves to a new stack-emptying vertex, stack-emptier, at which $M'$ empties its stack of everything apart from the $\$-$shield. To then move to final$_{M'}$, $M'$ pops the $\$, thus ensuring it has an empty stack when it reaches final$_{M'}$. $M'$ is illustrated in Figure 3.9.

More precisely, $M'$ consists of the graph of $M$ plus three new vertices: start$_{M'}$, stack-emptier, and final$_{M'}$. The following edges are also added: (start$_{M'}$, start$_M$) labeled Push $\$, $\lambda$-edges from each of $M'$'s final vertices to stack-emptier, self-loops (stack-emptier, stack-emptier) labeled Pop $X$ for each $X \in \Gamma$, where $\Gamma$ is $M$'s stack alphabet (so $\not= X$), and edge (stack-emptier, final$_{M'}$) labeled Pop $\$.

It is clear that $L(M) = L(M')$. Nonetheless, we present the argument for completeness.

First, we show that $L(M') \subseteq L(M)$, i.e. if $w \in L(M')$ then $w \in L(M)$ also. So let $w \in L(M')$. Let $P'$ be a $w$-recognizing path in $M'$ and let $f$ be the final vertex of $M$ preceding stack-emptier on the path $P'$. Removing the first edge in $P'$ and every edge including and after $(f, stack-emptier)$, leaves a path $P$ which is a $w$-recognizing path in $M$. Thus $w \in L(M)$ and consequently $L(M') \subseteq L(M)$.
Now we show \( L(M) \subseteq L(M') \), i.e. if \( w \in L(M) \) then \( w \in L(M') \) also. Let \( w \in L(M) \) and let \( P \) be a \( w \)-recognizing path in \( M \). Suppose that \( P \) ends with string \( s \) on the stack at final vertex \( f \). We add the edges \((start_M', start_M), (f, stack-emptier), |s| \) self-loops at stack-emptier, and \((stack-emptier, final_M')\) to \( P \), yielding path \( P' \) in \( M' \). By choosing the self-loops to be labeled with the characters of \( s^R \) in this order, we cause \( P' \) to be a \( w \)-recognizing path in \( M' \). Thus \( w \in L(M') \) and consequently \( L(M) \subseteq L(M') \).

**Lemma 3.2.3.** Let \( A \) and \( B \) be languages recognized by PDAs. Then there is a PDA recognizing \( A \circ B \).

**Proof.** Let \( M_A \) and \( M_B \) be single-destination, empty-stack PDAs recognizing \( A \) and \( B \), respectively.

We will construct PDA \( M_{A \circ B} \) that recognizes \( A \circ B \). It consists of \( M_A, M_B \) plus one \( \lambda \)-edge \((final_A, start_B)\). Its start vertex is \( start_A \) and its final vertex is \( final_B \).

To see that \( L(M_{A \circ B}) = A \circ B \) is straightforward.

First, we show that \( L(M_{A \circ B}) \subseteq A \circ B \), i.e. if \( w \in L(M_{A \circ B}) \) then \( w \in A \circ B \) also. So suppose that \( w \in L(M_{A \circ B}) \). Then there is a \( w \)-recognizing path \( P \) in \( M_{A \circ B} \): \( P \) is formed from a path \( P_A \) in \( M_A \) going from \( start_A \) to \( final_A \) (and which therefore ends with an empty stack), \( \lambda \)-edge \((final_A, start_B)\), and a path \( P_B \) in \( M_B \) starting from \( start_B \) with an empty stack and going to \( final_B \). Let \( u \) be the sequence of reads labeling \( P_A \) and \( v \) those labeling \( P_B \). It follows that \( P_A \) is \( u \)-recognizing, and \( P_B \) is \( v \)-recognizing (see Figure 3.10) and thus \( u \in A \) and \( v \in B \). In addition, \( w = u\lambda v = uv \), which implies that \( w = uv \in A \circ B \).

![Figure 3.10: PDA L(M_{A \circ B}).](image)

Next, we show that \( A \circ B \subseteq L(M_{A \circ B}) \). So suppose that \( w = uv \in A \circ B \), where \( u \in A \) and \( v \in B \). Then there is a \( u \)-recognizing path \( P_A \) in \( M_A \) and a \( v \)-recognizing path \( P_B \) in \( M_B \). We argue that the following path \( P \) is \( w \)-recognizing: \( P_A \) followed by the \( \lambda \)-edge \((final_A, start_B)\), followed by \( P_B \). Note that the stack is empty when at node \( final_A \) by construction and hence it is also empty at node \( start_B \). Consequently, computation path \( P \) in \( M_{A \circ B} \) recognizes \( u\lambda v = uv = w \), as claimed. It follows that \( w = uv \in L(M_{A \circ B}) \).

**Lemma 3.2.4.** Suppose that \( L \) is recognized by a PDA and suppose that \( R \) is a regular language. Then there is a PDA recognizing \( L \cap R \).

**Proof.** Let \( M_L = (\Sigma, \Gamma_L, V_L, start_L, F_L, \delta_L) \) be a single-destination, empty-stack PDA recognizing \( M_L \), and let \( R \) be recognized by DFA \( M_R = (\Sigma, V_R, start_R, F_R, \delta_R) \). We will construct...
3.2. CLOSURE PROPERTIES

PDA $M_{L \cap R}$ that recognizes $L \cap R$. The vertices of $M_{L \cap R}$ will be 2-tuples, the first component corresponding to a vertex of $M_L$ and the second component to a vertex of $M_R$. The computation of $M_{L \cap R}$, when looking at the first components along with the stack will be exactly the computation of $M_L$, and when looking at the second components, but without the stack, it will be exactly the computation of $M_R$. This leads to the following edges in $M_{L \cap R}$.

1. If $M_L$ has an edge $(u_L, v_L)$ with label (Pop $A$, Read $b$, Push $C$) and $M_R$ has an edge $(u_R, v_R)$ with label $b$, then $M_{L \cap R}$ has an edge $((u_L, u_R), (v_L, v_R))$ with label (Pop $A$, Read $b$, Push $C$).

2. If $M_L$ has an edge $(u_L, v_L)$ with label (Pop $A$, Read $\lambda$, Push $C$) then $M_{L \cap R}$ has an edge $((u_L, u_R), (v_L, u_R))$ with label (Pop $A$, Read $\lambda$, Push $C$) for every $u_R \in V_R$.

The start vertex for $M_{L \cap R}$ is $(\text{start}_L, \text{start}_R)$ and its set of final vertices is $F_L \times F_R$, the pairs of final vertices, one from $M_L$ and one from $M_R$, respectively.

**Assertion.** $M_{L \cap R}$ can reach vertex $(v_L, v_R)$ on input $w$ if and only if $M_L$ can reach vertex $v_L$ and $M_R$ can reach vertex $v_R$ on input $w$.

Next, we argue that the assertion is true. For suppose that on input $w$, $M_{L \cap R}$ can reach vertex $(v_L, v_R)$ by computation path $P_{L \cap R}$. If we consider the first components of the vertices in $P_{L \cap R}$, we see that it is a computation path of $M_L$ on input $w$ reaching vertex $v_L$. Likewise, if we consider the second components of the vertices of $M_{L \cap R}$, we obtain a path $P'_R$. The only difficulty is that this path may contain repetitions of a vertex $u_R$ corresponding to reads of $\lambda$ by $M_{L \cap R}$. Eliminating such repetitions creates a path $P_R$ in $M_R$ reaching $v_R$ and having the same label $w$ as path $P'_R$.

Conversely, suppose that $M_L$ can reach $v_L$ by computation path $P_L$ and $M_R$ can reach $v_R$ by computation path $P_R$. Combining these paths, with care, gives a computation path $P$ which reaches $(v_L, v_R)$ on input $w$. We proceed as follows. The first vertex is $(\text{start}_L, \text{start}_R)$. Then we traverse $P_L$ and $P_R$ in tandem. Either the next edges in $P_L$ and $P_R$ are both labeled by a Read $b$ (simply a $b$ on $P_R$) in which case we use Rule (1) above to give the edge to add to $P$, or the next edge on $P_L$ is labeled by Read $\lambda$ (together with a Pop and a Push possibly) and then we use Rule (2) to give the edge to add to $P$. In the first case we advance one edge on both $P_L$ and $P_R$, in the second case we only advance on $P_L$. Clearly, the path ends at vertex $(v_L, v_R)$ on input $w$.

It is now easy to see that $L(M_{L \cap R}) = L \cap R$. For on input $w$, $M_{L \cap R}$ can reach a final vertex $v \in F = F_L \times F_R$ if and only if on input $w$, $M_L$ reaches a vertex $v_L \in F_L$ and $M_R$ reaches a vertex $v_R \in F_R$. That is, $w \in L(M_{L \cap R})$ if and only if $w \in L(M_L) = L$ and $w \in L(M_R) = R$, or in other words $w \in L(M_{L \cap R})$ if and only if $w \in L \cap R$.

**Exercise.** Show that if $A$ is recognized by PDA $M_A$ then there is a PDA $M_{A^*}$ recognizing $A^*$.
3.3 Context Free Languages

Context Free Languages (CFLs) provide a way of specifying certain recursively defined languages. Let’s begin by giving a recursive method for generating integers in decimal form. We will call this representation of integers *decimal strings*. A decimal string is defined to be either a single digit (one of 0–9) or a single digit followed by another decimal string. This can be expressed more succinctly as follows.

\[
\langle \text{decimal string} \rangle \rightarrow \langle \text{digit} \rangle \mid \langle \text{digit} \rangle \langle \text{decimal string} \rangle \quad (3.1)
\]

\[
\langle \text{digit} \rangle \rightarrow 0 \mid 1 \mid 2 \mid \ldots \mid 9
\]

We can also view this as providing a way of generating decimal strings. To generate a particular decimal string we perform a series of replacements or substitutions starting from the “variable” \( \langle \text{decimal string} \rangle \). The sequence of replacements generating 147 is shown below.

\[
\begin{align*}
\langle \text{decimal string} \rangle & \Rightarrow \langle \text{digit} \rangle \langle \text{decimal string} \rangle \\
& \Rightarrow 1 \langle \text{decimal string} \rangle \\
& \Rightarrow 1 \langle \text{digit} \rangle \langle \text{decimal string} \rangle \\
& \Rightarrow 14 \langle \text{decimal string} \rangle \\
& \Rightarrow 14 \langle \text{digit} \rangle \\
& \Rightarrow 147
\end{align*}
\]

We write \( \sigma \Rightarrow \tau \) if the string \( \tau \) is the result of a single replacement in \( \sigma \). The possible replacements are those given in Equation (3.1). Each replacement takes one occurrence of an item on the left-hand side of an arrow and replaces it with one of the items on the left-hand side; these are the items separated by vertical bars. Specifically, the possible replacements are to take one occurrence of one of:

- \( \langle \text{decimal string} \rangle \) and replace it with one of \( \langle \text{digit} \rangle \) or the sequence \( \langle \text{digit} \rangle \langle \text{decimal string} \rangle \).

- \( \langle \text{digit} \rangle \) and replace it with one of 0–9.

An easier way to understand this is by viewing the generation using a tree, called a *derivation* or *parse* tree, as shown in Figure 3.11. Clearly, were we patient enough, in principle we could generate any number.

The above generation rules are relatively simple. Let’s look at the slightly more-elaborate example of arithmetic expressions such as \( x + x \times x \) or \( (x + x) \times x \). For simplicity, we limit the expressions to those built from a single variable \( x \), the “+” and “\( \times \)” operators, and parentheses. We also would like the generation rules to follow the precedence order of the operators “+” and “\( \times \)”, in a sense that will become clear.

The generation rules, being recursive, generate arithmetic expressions top-down. Let’s use the example \( x + x \times x \) as motivation. To guide us, it is helpful to look at the expression tree representation, shown in Figure 3.12a. The root of the tree holds the operator “+”;
3.3. CONTEXT FREE LANGUAGES

Figure 3.11: Parse Tree Generating 147.

Figure 3.12: Parse Tree Generating $x + x \times x$. 
CHAPTER 3. PUSHDOWN AUTOMATA AND CONTEXT FREE LANGUAGES

correspondingly, the first substitution we apply needs to create a “+”; the remaining parts of the expression will then be generated recursively. This is implemented with the following variables: ⟨expr⟩, which can generate any arithmetic expression; ⟨term⟩, which can generate any arithmetic expression whose top level operator is times (×) or matched parentheses (“(“ and “)“); and ⟨factor⟩, which can generate any arithmetic expression whose top level operator is a pair of matched parentheses. This organization is used to enforce the usual operator precedence. This leads us to the following substitution rules:

- ⟨expr⟩ → ⟨expr⟩ + ⟨term⟩ | ⟨term⟩.
  
  This rule implies that the top-level additions are generated in right to left order and hence performed in left to right order (for the expression generated by the left ⟨expr⟩ is evaluated before being added to the expression generated by the ⟨term⟩ to its right). So eventually the initial ⟨expr⟩ is replaced by ⟨term⟩ + ⟨term⟩ + · · · + ⟨term⟩, each term being an operand for the “+” operator. If there is no addition, ⟨expr⟩ is simply replaced by ⟨term⟩.

- ⟨term⟩ → ⟨term⟩ × ⟨factor⟩ | ⟨factor⟩.
  
  Similarly, this rule implies that the multiplications are performed in left to right order. Eventually the initial ⟨term⟩ is replaced by ⟨factor⟩ × ⟨factor⟩ × · · · × ⟨factor⟩, each factor being an operand to the “×” operator. If there is no multiplication, ⟨term⟩ is simply replaced by ⟨factor⟩.

- ⟨factor⟩ → x | (expr).
  
  Since “×” has the highest precedence each of its operands must be either a simple variable (x) or a parenthesized expression, which is what we have here.

The derivation tree for the example of Figure 3.12a is shown in Figure 3.12b. Note that the left-to-right order of evaluation is an arbitrary choice for the “+” and the “×” operators, but were we to introduce the “−” and “÷” operators it ceases to be arbitrary; left-to-right is then the correct rule.

Let’s look at one more example.

Example 3.3.1. \( L = \{ a^i b^i \mid i \geq 0 \} \). Here are a set of rules to generate the strings in \( L \), to generate \( L \) for short, starting from the term ⟨Balanced⟩.

\[
\langle \text{Balanced} \rangle \Rightarrow \lambda \mid a\langle \text{Balanced} \rangle b.
\]

Notice how a string \( s \in L \) is generated: from the outside in. First the outermost \( a \) and \( b \) are created, together with a ⟨Balanced⟩ term between them; this ⟨Balanced⟩ term will be used to generate \( a^{i-1}b^{i-1} \). Then the second outermost \( a \) and \( b \) are generated, etc. The bottom of the recursion, the base case, is the generation of string \( \lambda \).

Now we are ready to define Context Free Grammars (CFGs), \( G \), (which have nothing to do with graphs). A Context Free Grammar has four parts:
3.3. CONTEXT FREE LANGUAGES

1. A set $V$ of variables (such as ⟨factor⟩ or ⟨Balanced⟩); note that $V$ is not a vertex set here.
   The individual variables are usually written using single capital letters, often from the end of the alphabet, e.g. X, Y, Z; no angle brackets are used here. This has little mnemonic value, but it is easier to write. If you do want to use longer variable names, I advise using the angle brackets to delimit them.

2. An alphabet $T$ of terminals: these are the characters used to write the strings being generated. Usually, they are written with small letters.

3. $S \in V$ is the start variable, the variable from which the string generation begins.

4. A set of rules $R$. Each rule has the form $X \rightarrow \sigma$, where $X \in V$ is a variable and $\sigma \in (T \cup V)^*$ is a string of variables and terminals, which could be $\lambda$, the empty string.

   If we have several rules with the same left-hand side, for example $X \rightarrow \sigma_1, X \rightarrow \sigma_2, \ldots, X \rightarrow \sigma_k$, they can be written as $X \rightarrow \sigma_1 | \sigma_2 | \cdots | \sigma_k$ for short. The meaning is that an occurrence of $X$ in a generated string can be replaced by any one of $\sigma_1, \sigma_2, \ldots, \sigma_k$. Different occurrences of $X$ can be replaced by distinct $\sigma_i$, of course.

The generation of a string proceeds by a series of replacements starting from the string $s_0 = S$, and which, for $1 \leq i \leq k$, obtains $s_i$ from $s_{i-1}$ by replacing some variable $X$ in $s_{i-1}$ by one of the replacements $\sigma_1, \sigma_2, \ldots, \sigma_k$ for $X$, as provided by the rule collection $R$. We will write this as

$$S = s_0 \Rightarrow s_1 \Rightarrow s_2 \Rightarrow \cdots \Rightarrow s_k \text{ or } S \Rightarrow^* s_k \text{ for short.}$$

The language generated by grammar $G$, $L(G)$, is the set of strings of terminals that can be generated from $G$’s start variable $S$:

$$L(G) = \{ w \mid S \Rightarrow^* w \text{ and } w \in T^* \}.$$

**Example 3.3.2.** Grammar $G_2$, the grammar generating the language of properly nested parentheses. It has:

- Variable set: \{S\}.
- Terminal set: \{(,\).\}.
- Rules: $S \Rightarrow (S) \mid SS \mid \lambda$.
- Start variable: $S$.

Here are some example derivations.

$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()(S) \Rightarrow ()(()).$
$S \Rightarrow (S) \Rightarrow (SS) \Rightarrow ((S)S) \Rightarrow ()(S) \Rightarrow ()(())$. 
3.3.1 Closure Properties

Lemma 3.3.3. Let $G_A$ and $G_B$ be CFGs generating languages $A$ and $B$, respectively. Then there are CFGs generating $A \cup B$, $A \circ B$, $A^*$.

Proof. Let $G_A = (V_A, \Sigma_A, R_A, S_A)$ and $G_B = (V_B, \Sigma_B, R_B, S_B)$. By renaming variables if needed, we can ensure that $V_A$ and $V_B$ are disjoint.

First, we show that $A \cup B$ is generated by the following grammar $G_{A \cup B}$.

$G_{A \cup B}$ has variable set $\{S_{A \cup B}\} \cup V_A \cup V_B$, terminal set $T_A \cup T_B$, start variable $S_{A \cup B}$, rules $R_A \cup R_B$ plus the rules $S_{A \cup B} \rightarrow S_A \mid S_B$.

To generate a string $w \in A$, $G_{A \cup B}$ performs a derivation with first step $S_{A \cup B} \Rightarrow S_A$, and then follows this with a derivation of $w$ in $G_A$: $S_A \Rightarrow^* w$. So if $w \in A$, $w \in L(G_{A \cup B})$. Likewise, if $w \in B$, then $w \in L(G_{A \cup B})$ also. Thus $A \cup B \subseteq L(G_{A \cup B})$.

To show $L(G_{A \cup B}) \subseteq A \cup B$ is also straightforward. For if $w \in L(G_{A \cup B})$, then there is a derivation $S_{A \cup B} \Rightarrow^* w$. Its first step is either $S_{A \cup B} \Rightarrow S_A$ or $S_{A \cup B} \Rightarrow S_B$. Suppose it is $S_{A \cup B} \Rightarrow S_A$. Then the remainder of the derivation is $S_A \Rightarrow^* w$; this says that $w \in A$. Similarly, if the first step is $S_{A \cup B} \Rightarrow S_B$, then $w \in B$. Thus $L(G_{A \cup B}) \subseteq A \cup B$.

Next, we show that $A \circ B$ is generated by the following grammar $G_{A \circ B}$.

$G_{A \circ B}$ has variable set $S_{A \circ B} \cup V_A \cup V_B$, terminal set $T_A \cup T_B$, start variable $S_{A \circ B}$, and rules $R_A \cup R_B$ plus the rule $S_{A \circ B} \rightarrow S_A S_B$.

If $w \in A \circ B$, then $w = uv$ for some $u \in A$ and $v \in B$. So to generate $w$, $G_{A \circ B}$ performs the following derivation. The first step is $S_{A \circ B} \Rightarrow S_A S_B$; this is followed by a derivation $S_A \Rightarrow^* u$, which yields the string $u S_B$; this is then followed by a derivation $S_B \Rightarrow^* v$, which yields the string $uv = w$. Thus $A \circ B \subseteq L(G_{A \circ B})$.

To show $L(G_{A \circ B}) \subseteq A \circ B$ is also straightforward. For if $w \in L(G_{A \circ B})$, then there is a derivation $S_{A \circ B} \Rightarrow^* w$. The first step can only be $S_{A \circ B} \Rightarrow S_A S_B$. Let $u$ be the string of terminals derived from $S_A$, and $v$ the string of terminals derived from $S_B$, in the full derivation. So $uv = w$, $u \in A$ and $v \in B$. Thus $L(G_{A \circ B}) \subseteq A \circ B$.

The fact that there is a grammar $G_A^*$ generating $A^*$ we leave as an exercise for the reader. 

Lemma 3.3.4. $\{\lambda\}$, $\phi$, $\{a\}$ are all context free languages.

Proof. The grammar with the single rule $S \rightarrow \lambda$ generates $\{\lambda\}$, the grammar with no rules generates $\phi$, and the grammar with the single rule $S \rightarrow a$ generates $\{a\}$, where, in each case, $S$ is the start variable. 

Corollary 3.3.5. All regular languages have CFGs.

Proof. It suffices to show that for any language represented by a regular expression $r$ there is a CFG $G_r$ generating the same language. This is done by means of a proof by induction on the number of operators in $r$. As this is identical in structure to the proof of Lemma 2.4.1, the details are left to the reader.
3.4 Converting CFGs to Chomsky Normal Form (CNF)

In this section, we show that every CFL can be generated by a Chomsky Normal Form grammar, CNF for short, a Context Free Grammar with a helpfully simple form. More precisely, a CNF grammar is a CFG with rules restricted as follows.

The right-hand side of a rule consists of:

i. Either a single terminal, e.g. \( A \rightarrow a \).

ii. Or two variables, e.g. \( A \rightarrow BC \).

iii. Or the rule \( S \rightarrow \lambda \), if \( \lambda \) is in the language.

iv. The start symbol \( S \) may appear only on the left-hand side of rules.

Given a CFG \( G \), we show how to convert it to a CNF grammar \( G' \) generating the same language.

We use a grammar \( G \) with the following rules as a running example.

\[
S \rightarrow AAS \mid aA; \quad A \rightarrow BB \mid aA \mid S; \quad B \rightarrow b \mid \lambda
\]

We proceed in a series of steps which gradually enforce the above CNF criteria; each step leaves the generated language unchanged.

**Step 1** For each terminal \( a \), we introduce a new variable, \( U_a \) say, add a rule \( U_a \rightarrow a \), and for each occurrence of \( a \) in a string of length 2 or more on the right-hand side of a rule, replace \( a \) by \( U_a \). Clearly, the generated language is unchanged.

Example: If we have the rule \( A \rightarrow Ba \), this is replaced by \( U_a \rightarrow a \), \( A \rightarrow BU_a \).

This ensures that terminals on the right-hand sides of rules obey criteria (i) above.

This step changes our example grammar \( G \) to have the rules:

\[
S \rightarrow AAS \mid U_a A; \quad A \rightarrow BB \mid U_a A \mid S; \quad B \rightarrow b \mid \lambda; \quad U_a \rightarrow a
\]

**Step 2** For each rule with 3 or more variables on the right-hand side, we replace it with a new collection of rules obeying criteria (ii) above. Suppose there is a rule \( U \rightarrow W_1 W_2 \cdots W_k \), for some \( k \geq 3 \). Then we create new variables \( X_2, X_3, \cdots, X_{k-1} \), and replace the prior rule with the rules:

\[
U \rightarrow W_1 X_2; \quad X_2 \rightarrow W_2 X_3; \quad \cdots; \quad X_{k-2} \rightarrow W_{k-2} X_{k-1}; \quad X_{k-1} \rightarrow W_{k-1} W_k
\]

Clearly, the use of the new rules one after another, which is the only way they can be used, has the same effect as using the old rule \( U \rightarrow W_1 W_2 \cdots W_k \). Thus the generated language is unchanged.

This ensures, for criteria (ii) above, that no right-hand side has more than 2 variables.

We have yet to eliminate right-hand sides of one variable or of the form \( \lambda \).
This step changes our example grammar $G$ to have the rules:

$$S \rightarrow AX \mid U_a A; \quad X \rightarrow AS; \quad A \rightarrow BB \mid U_a A \mid S; \quad B \rightarrow b \mid \lambda; \quad U_a \rightarrow a$$

**Step 3** We replace each occurrence of the start symbol $S$ with the variable $S'$ and add the rule $S \rightarrow S'$. This ensures criteria (iv) above.

This step changes our example grammar $G$ to have the rules:

$$S \rightarrow S'; \quad S' \rightarrow AX \mid U_a A; \quad X \rightarrow AS'; \quad A \rightarrow BB \mid U_a A \mid S'; \quad B \rightarrow b \mid \lambda; \quad U_a \rightarrow a$$

**Step 4** This step removes rules of the form $A \rightarrow \lambda$.

To understand what needs to be done it is helpful to consider a derivation tree for a string $w$. If the tree use a rule of the form $A \rightarrow \lambda$, we label the resulting leaf with $\lambda$. We will be focussing on subtrees in which all the leaves have $\lambda$-labels; we call such subtrees $\lambda$-subtrees. Now imagine pruning all the $\lambda$-subtrees, creating a reduced derivation tree for $w$. Our goal is to create a modified grammar which can form the reduced derivation tree. A derivation tree, and its reduced form is shown in Figure 3.13.

![Diagram](image_url)

Figure 3.13: (a) A Derivation Tree for string $ab$. (b) The Reduced Derivation Tree.

We need to change the grammar as follows. Whenever there is a rule $A \rightarrow BC$ and $B$ can generate $\lambda$, we need to add the rule $A \rightarrow C$ to the grammar (note that this does not allow any new strings to be generated); similarly, if $C$ can generate $\lambda$, we need to add the rule $A \rightarrow B$; also, note that if there is a rule $A \rightarrow BB$ and $B$ can generate $\lambda$, we need to add the rule $A \rightarrow B$. 
Next, we remove all rules of the form \( A \rightarrow \lambda \). We argue that any previously generatable string \( w \neq \lambda \) remains generatable. For given a derivation tree for \( w \) using the old rules, using the new rules we can create the reduced derivation tree, which is a derivation tree for \( w \) in the new grammar. To see this, consider a maximal sized \( \lambda \)-subtree (that is a \( \lambda \)-subtree whose parent is not part of a \( \lambda \)-subtree). Then its root \( v \) must have a sibling \( w \) and parent \( u \) (these are the names of nodes, not strings). Suppose that \( u \) has variable label \( A \), \( v \) has label \( B \) and \( w \) has label \( C \). Then node \( v \) was generated by applying either the rule \( A \rightarrow BC \) or the rule \( A \rightarrow CB \) at node \( u \) (depending on whether \( v \) is the left or right child of \( u \)). In the reduced tree, applying the rule \( A \rightarrow C \) generates \( w \) and omits \( v \) and its subtree.

Finally, we take care of the case that, under the old rules, \( S \) can generate \( \lambda \). In this situation, we simply add the rule \( S \rightarrow \lambda \), which then allows \( \lambda \) to be generated by the new rules also.

To find the variables that can generate \( \lambda \), we use an iterative rule reduction procedure. First, we make a copy of all the rules. We then create new reduced rules by removing from the right-hand sides all instances of variables \( A \) for which there is a rule \( A \rightarrow \lambda \). We keep iterating this procedure so long as it creates additional new reduced rules with \( \lambda \) on the right-hand side.

For our example grammar we start with the rules:
\[
S \rightarrow S'; \quad S' \rightarrow AX | U_a A; \quad X \rightarrow AS'; \quad A \rightarrow BB | U_a A | S'; \quad B \rightarrow b | \lambda; \quad U_a \rightarrow a
\]
As \( B \rightarrow \lambda \) is a rule, we obtain the reduced rules:
\[
S \rightarrow S'; \quad S' \rightarrow AX | U_a A; \quad X \rightarrow AS'; \quad A \rightarrow BB | U_a A | S' | B | \lambda; \quad B \rightarrow b | \lambda; \quad U_a \rightarrow a
\]
As \( A \rightarrow \lambda \) is now a rule, we next obtain:
\[
S \rightarrow S'; \quad S' \rightarrow AX | X | U_a A | U_a; \quad X \rightarrow AS' | S'; \quad A \rightarrow BB | U_a A | S' | B | U_a | \lambda; \quad B \rightarrow b | \lambda; \quad U_a \rightarrow a
\]
There are no new rules with \( \lambda \) on the right-hand side. So the procedure is now complete and this yields the new collection of rules:
\[
S \rightarrow S'; \quad S' \rightarrow AX | X | U_a A | U_a; \quad X \rightarrow AS' | S'; \quad A \rightarrow BB | U_a A | S' | B | U_a; \quad B \rightarrow b; \quad U_a \rightarrow a
\]

An efficient implementation keeps track of the lengths of each right-hand side, and a list of the locations of each variable; the new rules with \( \lambda \) on the right-hand side are those which have newly obtained length 0. It is not hard to have this procedure run in time linear in the sum of the lengths of the rules.

**Step 5** This step removes rules of the form \( A \rightarrow B \), which we call unit rules.

What is needed is to replace derivations of the form \( A_1 \Rightarrow A_2 \Rightarrow \cdots \Rightarrow A_k \Rightarrow BC \) with a new derivation of the form \( A \Rightarrow BC \); this is achieved with a new rule \( A \rightarrow BC \). Similarly,
derivations of the form \( A_1 \Rightarrow A_2 \Rightarrow \cdots \Rightarrow A_k \Rightarrow a \) need to be replaced with a new derivation of the form \( A \Rightarrow a \); this is achieved with a new rule \( A \rightarrow a \). We proceed in two substeps.

**Substep 5.1.** This substep identifies variables that are equivalent, i.e. collections \( B_1, B_2, \ldots, B_l \) such that for each pair \( B_i \) and \( B_j \), \( 1 \leq i < j \leq l \), \( B_i \) can generate \( B_j \), and \( B_j \) can generate \( B_i \). We then replace all of \( B_1, B_2, \ldots, B_l \) with a single variable, \( B_1 \) say. Clearly, this does not change the language that is generated.

To do this we form a directed graph based on the unit rules. For each variable, we create a vertex in the graph, and for each unit rule \( A \rightarrow B \) we create an edge \( (A, B) \). Figure 3.14(a) shows the graph for our example grammar. The vertices in each strong component of the graph correspond to a collection of equivalent variables.

For the example grammar, the one non-trivial strong component contains the variables \( \{S', X\} \). We replace \( S' \) with \( X \) yielding the rules:

\[
S \rightarrow X; \quad X \rightarrow AX | X | U_aA | U_a; \quad A \rightarrow BB | U_aA | B | X | U_a; \quad B \rightarrow b; \quad U_a \rightarrow a
\]

We can remove the useless rule \( X \rightarrow X \) also.

**Substep 5.2.** In this substep, we add rules \( A \rightarrow BC \) and \( A \rightarrow a \), as described above, so as to shortcut derivations that were using unit rules.

To this end, we use the graph formed from the unit rules remaining after Substep 5.1, which we call the *reduced graph*. It is readily seen that this is an acyclic graph.

In processing \( A \rightarrow B \), we will add appropriate non-unit rules that allow the shortcutting of all uses of \( A \rightarrow B \), and hence allow the rule \( A \rightarrow B \) to be discarded. If there are no unit rules with \( B \) on the left-hand side it suffices to add a rule \( A \rightarrow CD \) for each rule \( B \rightarrow CD \), and a rule \( A \rightarrow b \) for each rule \( B \rightarrow b \).

To be able to do this, we just have to process the unit rules in a suitable order. Recall that each unit rule is associated with a distinct edge in the reduced graph. As this graph will be used to determine the order in which to process the unit rules, it will be convenient to write "processing an edge" when we mean "processing the associated rule". It suffices to ensure that each edge is processed only after any descendant edges have been processed. So it suffices to start at vertices with no outedges and to work backward through the graph.

![Figure 3.14: (a) Graph showing the unit rules. (b) The reduced graph.](image-url)
This is called a reverse topological traversal. (This traversal can be implemented via a depth first search on the acyclic reduced graph.)

For each traversed edge \((E, F)\), which corresponds to a rule \(E \rightarrow F\), for each rule \(F \rightarrow CD\), we add the rule \(E \rightarrow CD\), and for each rule \(F \rightarrow f\), we add the rule \(E \rightarrow f\); then we remove the rule \(E \rightarrow F\). Any derivation which had used the rules \(E \rightarrow F\) followed by \(F \rightarrow CD\) or \(F \rightarrow f\) can now use the rule \(E \rightarrow CD\) or \(E \rightarrow f\) instead. So the same strings are derived with the new set of rules.

This step changes our example grammar \(G\) as follows (see Figure 3.14(b)):

First, we traverse edge \((A, B)\). This changes the rules as follows:

- Add \(A \rightarrow b\)
- Remove \(A \rightarrow B\).

Next, we traverse edge \((X, U_a)\). This changes the rules as follows:

- Add \(X \rightarrow a\)
- Remove \(X \rightarrow U_a\).

Now, we traverse edge \((A, X)\). This changes the rules as follows:

- Add \(A \rightarrow a\)
- Remove \(X \rightarrow U_a\).

Finally, we traverse edge \((S, X)\). This changes the rules as follows:

- Add \(S \rightarrow AX | U_a A | a\)
- Remove \(S \rightarrow X\).

The net effect is that our grammar now has the rules

\[
S \rightarrow AX | U_a A | a; \ X \rightarrow AX | U_a A | a; \ A \rightarrow BB | AX | U_a A | b | a; \ B \rightarrow b; \ U_a \rightarrow a
\]

Steps 4 and 5 complete the attainment of criteria (ii), and thereby create a CNF grammar generating the same language as the original grammar.

3.5 Showing Languages are not Context Free

We will do this with the help of a Pumping Lemma for Context Free Languages. To prove this lemma we need two results relating the height of derivation trees and the length of the derived strings, when using CNF grammars.

**Lemma 3.5.1.** Let \(T\) be a derivation tree of height \(h\) for string \(w \neq \lambda\) using CNF grammar \(G\). Then \(w \leq 2^{h-1}\).

**Proof.** The result is easily confirmed by strong induction on \(h\). Recall that the height of the tree is the length, in edges, of the longest root to leaf path.

The base case, \(h = 1\), occurs with a tree of two nodes, the root and a leaf child. Here, \(w\) is the one terminal character labeling the leaf, so \(|w| = 1 = 2^{h-1}\); thus the claim is true in this case.
Suppose that the result is true for trees of height \( k \) or less. We show that it is also true for trees of height \( k + 1 \). To see this, note that the root of \( T \) has two children, each one being the root of a subtree of height \( k \) or less. Thus, by the inductive hypothesis, each subtree derives a string of length at most \( 2^{k-1} \), yielding that \( T \) derives a string of length at most \( 2 \cdot 2^{k-1} = 2^k \). This shows the inductive claim for \( h = k + 1 \).

It follows that the result holds for all \( h \geq 1 \).

**Corollary 3.5.2.** Let \( w \) be the string derived by derivation tree \( T \) using CNF grammar \( G \). If \( |w| > 2^{h-1} \), then \( T \) has height at least \( h + 1 \) and hence has a root to leaf path with at least \( h + 1 \) edges and at least \( h + 1 \) internal nodes.

Now let’s consider the language \( L = \{ a^i b^i c^i \mid i \geq 0 \} \) which is not a CFL as we shall proceed to show. Let’s suppose for a contradiction that \( L \) were a CFL. Then it would have a CNF grammar \( G \), with \( m \) variables say. Let \( p = 2^m \).

Let’s consider string \( s = a^p b^p c^p \in L \), and look at the derivation tree \( T \) for \( s \). As \( |s| > 2^{m-1} \), by Corollary 3.5.2, \( T \) has a root to leaf path with at least \( m + 1 \) internal nodes. Let \( P \) be a longest such path. Each internal node on \( P \) is labeled by a variable, and as \( P \) has at least \( m + 1 \) internal nodes, some variable must be used at least twice.

![Derivation Tree for \( s \in L \).](image)

**Observation 1.** The height of \( c \) is at most \( m + 1 \). Hence, by Lemma 3.5.1, \(|vwx| \leq 2^m = p\). This follows because \( P \) is a longest root to leaf path and because no variable label is repeated on the path below node \( c \).
Observation 2. Either $|v| \geq 1$ or $|x| \geq 1$ (or both). We abbreviate this as $|vx| \geq 1$.

For node $c$ has two children, one on path $P$, and a second child, which we name $e$, that is not on path $P$. This is illustrated in Figure 3.16. Suppose that $e$ is $c$'s right child. Let $x_2$ be the string derived by the subtree rooted at $e$. Then, as $e$ is not a leaf, $|x_2| \geq 1$. Clearly $x_2$ is the right end portion of $x$ (it could be that $x = x_2$); thus $|x| \geq 1$. Similarly if $e$ is $c$’s left child, $|v| \geq 1$.

Let’s consider replicating the middle portion of the derivation tree, namely the wedge $W$ formed by taking the subtree $C$ rooted at $c$ and removing the subtree $D$ rooted at $d$, to create a derivation tree for a longer string, as shown in Figure 3.17. We can do this because the root of the wedge is labeled by $A$ and hence $W$ plus a nested subtree $D$ is a legitimate replacement for subtree $D$. The resulting tree, with two copies of $W$, one nested inside the other, is a derivation tree for $uvvwxxy$. Thus $uvvwxxy \in L$. 
Clearly, we could duplicate $W$ more than once, or remove it entirely, showing that all the strings $uv^iwx^iy \in L$, for any integer $i \geq 0$.

Now let’s see why $uvvwxxy \not\in L$. Note that we know that $|vx| \geq 1$ and $|vwx| \leq p$, by Observations 1 and 2. Further recall that $a^pb^pc^p = uvwxy$. As $|vwx| \leq p$, it is contained entirely in either one or two adjacent blocks of letters, as illustrated in Figure 3.18. Therefore,

```
a^p
   ┊ ┊ ┊
b^p
   ┊ ┊
c^p
```

Key: └── substring $vwx$, possible locations

Figure 3.18: Possible Locations of $vwx$ in $a^pb^pc^p$.

when $v$ and $x$ are duplicated, as $|vx| \geq 1$, the number of occurrences of one or two of the characters increases, but not of all three. Consequently, in $uvvwxxy$ there are not equal numbers of $a$’s, $b$’s, and $c$’s, and so $uvvwxxy \not\in L$.

We have shown both that $uvvwxxy \in L$ and $uvvwxxy \not\in L$. This contradiction means that the original assumption (that $L$ is a CFL) is mistaken. We conclude that $L$ is not a CFL.

We are now ready to prove the Pumping Lemma for Context Free Languages, which will provide a tool to show many languages are not Context Free, in the style of the above argument.

**Lemma 3.5.3.** (Pumping Lemma for Context Free Languages.) Let $L$ be a Context Free Language. Then there is a constant $p = p_L$ such that if $s \in L$ and $|s| \geq p$ then $s$ is pumpable, that is $s$ can be written in the form $s = uvwxy$ with

1. $|vx| \geq 1$.
2. $|vwx| \leq p$.
3. For every integer $i$, $i \geq 0$, $uv^iwx^iy \in L$.

**Proof.** Let $G$ be a CNF grammar for $L$, with $m$ variables say. Let $p = 2^m$. Let $s \in L$, where $|s| \geq p$, and let $T$ be a derivation tree for $s$. As $|s| > 2^{m-1}$, by Corollary 3.5.2, $T$ has a root to leaf path with at least $m+1$ internal nodes. Let $P$ be a longest such path. Each internal node on $P$ is labeled by a variable, and as $P$ has at least $m+1$ internal nodes, some variable must be used at least twice.

Working up from the bottom of $P$, let $c$ be the first node to repeat a variable label. So on the portion of $P$ below $c$ each variable is used at most one. Thus $c$ has height at most $m+1$. The derivation tree is shown in Figure 3.15. Let $d$ be the descendant of $c$ on $P$ having the same variable label as $c$, $A$ say. Let $w$ be the substring derived by the subtree rooted at
3.5. SHOWING LANGUAGES ARE NOT CONTEXT FREE

Let $vwx$ be the substring derived by the subtree rooted at $c$. Let $uvwxy$ be the string derived by the whole tree.

By Observation 1, $c$ has height at most $m + 1$; hence, by Lemma 3.5.1, $vwx$, the string $c$ derives, has length at most $2^m = p$. This shows Property (2). Property (1) is shown in Observation 2, above.

Finally, we show property (3). Let's replicate the middle portion of the derivation tree, namely the wedge $W$ formed by taking the subtree $C$ rooted at $c$ and removing the subtree $D$ rooted at $d$, to create a derivation tree for a longer string, as shown in Figure 3.17.

We can do this because the root of the wedge is labeled by $A$ and hence $W$ plus a nested subtree $D$ is a legitimate replacement for subtree $D$. The resulting tree, with two copies of $W$, one nested inside the other, is a derivation tree for $uvwxy$. Thus $uvwxy \in L$.

Clearly, we could duplicate $W$ more than once, or remove it entirely, showing that all the strings $uv^iwx^iy \in L$, for any integer $i \geq 0$.

Next, we demonstrate by example how to use the Pumping Lemma to show languages are not context free. The argument structure is identical to that used in applying the Pumping Lemma to regular languages.

**Example 3.5.4.** $J = \{ww \mid w \in \{a, b\}^*\}$. We will show that $J$ is not context free.

**Step 1.** Suppose, for a contradiction, that $J$ were context free. Then, by the Pumping Lemma, there is a constant $p = p_J$ such that for any $s \in J$ with $|s| \geq p$, $s$ is pumpable.

**Step 2.** Choose $s = a^{p+1}b^{p+1}a^{p+1}b^{p+1}$. Clearly $s \in J$ and $|s| \geq p$, so $s$ is pumpable.

**Step 3.** As $s$ is pumpable we can write $s$ as $s = uvwx$ with $|vwx| \leq p$, $|vx| \geq 1$ and $uv^iwx^iy \in J$ for all integers $i \geq 0$. Also, by condition (3) with $i = 0$, $s' = uwy \in J$. We argue next that in fact $s' \notin J$. As $|vwx| \leq p$, $vwx$ can overlap one or two adjacent blocks of characters in $s$ but no more. Now, to obtain $s'$ from $s$, $v$ and $x$ are removed. This takes away characters from one or two adjacent blocks in $s$, but at most $p$ characters in all (as $|vx| \leq p$). Thus $s'$ has four blocks of characters, with either one of the blocks of $a$'s shorter than the other, or one of the blocks of $b$'s shorter than the other, or possibly both of these. In every case $s' \notin J$. We have shown that both $s' \in J$ and $s' \notin J$. This is a contradiction.

**Step 4.** The contradiction shows that the initial assumption was mistaken. Consequently, $J$ is not context free.

**Comment.** Suppose, by way of example, that $vwx$ overlaps the first two blocks of characters. It would be incorrect to assume that $v$ is completely contained in the block of $a$'s and $x$ in the block of $b$'s. Further, it may be that $v = \lambda$ or $x = \lambda$ (but not both). All you know is that $|vwx| \leq p$ and that one of $|v| \geq 1$ or $|x| \geq 1$. Don’t assume more than this.

**Example 3.5.5.** $K = \{a^i b^j c^k \mid i < j < k\}$. We show that $K$ is not context free.

**Step 1.** Suppose, for a contradiction, that $K$ were context free. Then, by the Pumping Lemma, there is a constant $p = p_K$ such that for any $s \in K$ with $|s| \geq p$, $s$ is pumpable.

**Step 2.** Choose $s = a^{p}b^{p+1}c^{p+2}$. Clearly, $s \in K$ and $|s| \geq p$, so $s$ is pumpable.
Step 3. As \( s \) is pumpable we can write \( s = uvwxy \) with \(|vwx| \leq p\), \(|vx| \geq 1\) and \(uw'wx'y \in K\) for all integers \( i \geq 0\).

As \(|vwx| \leq p\), \(vwx\) can overlap one or two blocks of the characters in \( s \), but not all three. Our argument for obtaining a contradiction depends on the position of \(vwx\).

Case 1. \(vx\) does not overlap the block of \(c\)'s. Then consider \(s' = uvvwxxy\). As \( s \) is pumpable, by Condition (3) with \( i = 2\), \( s' \in L\). We argue next that in fact \( s' \notin K\). As \( v \) and \( x \) have been duplicated in \( s' \), the number of \( a \)'s or the number of \( b \)'s is larger than in \( s \) (or possibly both numbers are larger); but the number of \( c \)'s does not change. If the number of \( b \)'s has increased, then \( s' \) has at least as many \( b \)'s as \( c \)'s, and then \( s' \notin K\). Otherwise, the number of \( a \)'s increases, and the number of \( b \)'s is unchanged, so \( s' \) has at least as many \( a \)'s as \( b \)'s, and again \( s' \notin K\).

Case 2. \(vwx\) does not overlap the block of \(a\)'s. Then consider \(s' = uwy\). Again, as \( s \) is pumpable, by Condition (3) with \( i = 0\), \( s' \in s\). Again, we show that in fact \( s' \notin K\). To obtain \( s' \) from \( s \), the \( v \) and the \( x \) are removed. So in \( s' \) either the number of \( c \)'s is smaller than in \( s \), or the number of \( b \)'s is smaller (or both). But the number of \( a \)'s is unchanged. If the number of \( b \)'s is reduced, then \( s' \) has at least as many \( a \)'s as \( b \)'s, and so \( s' \notin K\). Otherwise, the number of \( c \)'s decreases and the number of \( b \)'s is unchanged; but then \( s' \) has at least as many \( b \)'s as \( c \)'s, and again \( s' \notin K\).

In either case, a pumped string \( s' \) has been shown to be both in \( K \) and not in \( K \). This is a contradiction.

Step 4. The contradiction shows that the initial assumption was mistaken. Consequently, \( K \) is not context free.

The following example uses the yet to be proven result that if \( L \) is context free and \( R \) is regular then \( L \cap R \) is also context free.

Example 3.5.6. \( H = \{w \mid w \in \{a, b, c\}^* \) and \( w \) has equal numbers of \( a \)'s, \( b \)'s and \( c \)'s\).

Consider \( H \cap a^ib^ic^i = L = \{a^ib^ic^i\} \). If \( H \) were context free, then \( L \) would be context free too. But we have already seen that \( L \) is not context free. Consequently, \( H \) is not context free either.

This could also be shown directly by pumping on string \( s = a^pb^pc^p \).

When applying the Pumping Lemma, it seems a nuisance to have to handle the cases where one of \( v \) or \( x \) may be the empty string, and fortunately, we can prove a variant of the Pumping Lemma in which both \(|v| \geq 1\) and \(|x| \geq 1\).

Lemma 3.5.7. (Variant of the Pumping Lemma for Context Free Languages.) Let \( L \) be a Context Free Language. Then there is a constant \( p = p_L \) such that if \( s \in L \) and \(|s| \geq p\) then \( s \) is pumpable, that is \( s \) can be written in the form \( s = uvwxy \) with

1. \(|v|, |x| \geq 1\).
2. \(|vwx| \leq p\).
3. For every integer \( i, i \geq 0, uv^iwx^iy \in L \).

Proof. Let \( \tilde{p}_L \) be the pumping constant for the standard pumping lemma applied to \( L \). We will chose \( p_L = 2\tilde{p}_L \).

Now let \( s \in L \) be any string of length at least \( p = p_L \).

We apply the standard pumping lemma to \( s \) and conclude that we can write \( s \) as \( s = \tilde{u}\tilde{v}\tilde{w}\tilde{x}\tilde{y} \) with

1. \( |\tilde{v}\tilde{x}| \geq 1 \).
2. \( |\tilde{v}\tilde{w}\tilde{x}| \leq \tilde{p} \).
3. For every integer \( i, i \geq 0, \tilde{u}\tilde{v}^i\tilde{w}\tilde{x}^i\tilde{y} \in L \).

If both \( |\tilde{v}| \geq 1 \) and \( |\tilde{x}| \geq 1 \) then the new result follows on setting \( u = \tilde{u}, v = \tilde{v}, w = \tilde{w}, x = \tilde{x}, y = \tilde{y} \).

Otherwise, if \( |\tilde{v}| \geq 1 \) and \( |\tilde{x}| = 0 \), then we set \( u = \tilde{u}, v = \tilde{v}, w = \lambda, x = \tilde{v}, y = \tilde{w}\tilde{y} \).

We observe that for all \( i, uv^iwx^iy = \tilde{u}\tilde{v}^i\lambda\tilde{v}^i\tilde{w}\tilde{y} = \tilde{u}\tilde{v}^{2i}\tilde{w}\tilde{x}^{2i}\tilde{y} \in L \); we also observe that \( |vwx| = 2|\tilde{v}| \leq 2|\tilde{v}\tilde{w}\tilde{x}| \leq 2\tilde{p} = p \). Likewise, if \( |\tilde{v}| = 0 \) and \( |\tilde{x}| \geq 1 \), then we set \( u = \tilde{u}\tilde{w}, v = \tilde{x}, w = \lambda, x = \tilde{x}, y = \tilde{y} \).

Again, \( uv^iwx^iy \in L \) for all integer \( i \geq 0 \) and \( |vwx| \leq p \).

3.6 PDAs Recognize exactly the Context Free Languages

It will be helpful to have some notation to specify the stack contents on a PDA following the reading of an input string \( s \).

Definition 3.6.1. \((s, \sigma)\) is called a data-configuration of PDA \( M \) at vertex \( p \) if \( M \) can end up at vertex \( p \) having \( \sigma \) on its stack and having read input string \( s \) (when starting the computation at its start vertex with an empty stack).

Lemma 3.6.2. Let \( L \) be a CFL. Then there is a PDA \( M_L \) that recognizes \( L \).

Proof. Let \( L \) be generated by CNF grammar \( G_L = (V_L, T, R_L, S_L) \). The corresponding \( M_L \) is illustrated in Figure 3.19. The computation centers on vertex \( Main \). Each return visit to \( Main \) corresponds to the simulation of one step of a leftmost derivation in \( G_L \). Specifically:

Claim. Let \( s \in T^* \) and \( \sigma \in V^* \). Then

\[ G_L \text{ generates string } s\sigma \text{ exactly if } M_L \text{ can reach vertex } Main \text{ with data configuration } (s, \$\sigma^R). \]
Note that the derivation is always replacing the leftmost symbol in $\sigma$. In order to simulate the derivation’s use of rule $A \rightarrow a$, $M_L$ has a self-loop labeled (Pop $A$, Read $a$). To simulate the use of rule $A \rightarrow BC$, $M_L$ will execute the sequence Pop $A$, Push $C$, Push $B$ (remember, $S\sigma_R$ is on the stack). To achieve this, $M_L$ has an additional vertex called “Sim $A \rightarrow BC$” and edges (Main, “Sim $A \rightarrow BC$”) and (“Sim $A \rightarrow BC$, Main), labeled (Pop $A$, Push $C$) and Push $B$, respectively. It will be helpful to refer to these actions, that take $M_L$ from vertex Main back to itself, as supermoves. So each supermove of $M_L$ corresponds to one derivation step in $G_L$.

A derivation of a terminal string $s$ occurs if $\sigma = \lambda$. To allow this to be recognized, $M_L$ uses a $S$-shielded stack. Then if $M_L$ is at vertex Main with data-configuration ($s$, $\lambda$), it can pop its stack and move to its final vertex. Thus if we can show the claim it is immediate that $G_L \Rightarrow^* w$ exactly if $M_L$ can reach vertex Main with data configuration ($w$, $\lambda$), which is the case exactly if $M_L$ can reach its final vertex on input $w$, i.e. exactly if $w \in L(M_L)$.

We show the claim in two steps. First, suppose that $G_L \Rightarrow^* w$. Then there is a leftmost derivation $S = s_1\sigma_1 \Rightarrow s_2\sigma_2 \Rightarrow \cdots \Rightarrow s_{k+1}\sigma_{k+1} = w$. (In a leftmost derivation, at step $i$, the leftmost variable in $\sigma_i$, for $1 \leq i < k$, is always the one to be replaced using a rule of the grammar. It corresponds to a Depth First Traversal of the derivation tree.) The corresponding $k$ supermove computation by $M_L$ starts by moving to vertex Main with $\lambda$ read and $ SS$ on its stack, i.e. it has data configuration $C_1 = (\lambda, SS) = (s_1, S\sigma_R)$. It then proceeds through the following $k$ data configurations at vertex Main: $C_2 = (s_2, S\sigma_R^2), \cdots C_{k+1} = (s_{k+1}, S\sigma_R^{k+1}) = (w, \lambda)$, and it goes from $C_i$ to $C_{i+1}$, for $1 \leq i \leq k$, by means of the supermove corresponding to the application of the rule taking $s_i\sigma_i$ to $s_{i+1}\sigma_{i+1}$. Finally $M_L$ moves from vertex Main to its final vertex, popping $S$ from its stack, and thereby recognizing $w$; i.e. $w \in L(M_L)$.

Next suppose that $M_L$ recognizes $w$. It does so by means of a computation using $k$ supermoves, for some $k \geq 0$. The computation begins by moving to vertex Main, while reading $\lambda$ and pushing $SS$ on the stack, i.e. it is at configuration $C_1 = (\lambda, \lambda) = (s_1, S\sigma_R^1)$. 

Figure 3.19: PDA $M_L$ simulating CFL $G_L$. 

![Diagram of PDA $M_L$ simulating CFL $G_L$.](image-url)
It then moves through the following series of data configurations at vertex Main: $C_2 = (s_2, \$s_2^R), \cdots, C_{k+1} = (s_{k+1}, \$s_{k+1}^R)$, where, for $1 \leq i \leq k$, $C_{i+1}$ is reached from $C_i$ by means of a supermove. By definition, the $i$th supermove corresponds to the application of the rule that takes string $s_i \sigma_i$ to $s_{i+1} \sigma_{i+1}$. Thus, the following is a derivation in grammar $G_L$: $S = s_1 \sigma_1 \Rightarrow s_2 \sigma_2 \Rightarrow \cdots \Rightarrow s_{k+1} \sigma_{k+1}$. Following the $k$th supermove, $M_L$ moves to its final vertex, which entails popping $\$ from its stack. At this point, $M_L$ has data configuration $(s_{k+1}, \lambda)$, and it accepts $s_{k+1}$, the string read. By assumption, $M_L$ was recognizing $w$, thus $s_{k+1} = w$. We conclude that $S \Rightarrow^* s_{k+1} \sigma_{k+1} = w$; i.e. $w \in L$. 

**Lemma 3.6.3.** Let $L \subseteq \Sigma^*$ be context-free and let $R \subseteq \Sigma^*$ be regular. Then $L \cap R$ is context free.

**Proof.** Let $G_L = (V_L, \Sigma, R_L, S_L)$ be a CNF grammar generating $L$ and let $M_R = (V_R, \text{start}, F_R, \delta_R)$ be a DFA recognizing $R$. We will build a grammar $G_{L \cap R} = (V_{L \cap R}, \Sigma, R_{L \cap R}, S_{L \cap R})$ to generate $L \cap R$. Let $V_R = \{q_1, q_2, \cdots, q_m\}$. For each variable $U \in V_L$, we create $m^2$ variables $U_{ij}$ in $V_{L \cap R}$. The rules we create will ensure that:

$$U_{ij} \Rightarrow^* w \in \Sigma^*$$

exactly if

$$U \Rightarrow^* w \text{ and there is a path labeled } w \text{ in } M_R \text{ going from } v_i \text{ to } v_j.$$ 

Thus a variable in $G_{L \cap R}$ records the name of the corresponding variable in $G_L$ and also records a “start” and a “finish” vertex in $M_R$. The constraint we are imposing is that $U_{ij}$ can generate $w$ exactly if both $U$ can generate $w$ and $M_R$ when started at $q_i$ will end up at $q_j$ on input $w$. It follows that if $q_f$ is a final vertex of $M_R$ and if $q_1 = \text{start}$, then

$$(S_L)_{1f} \Rightarrow^* w \text{ for some } q_f \in F_R$$

exactly if

$$w \in L \cap R.$$ 

If there is more than one $q_f \in F_R$ we cannot use $(S_L)_{1f}$ as the start variable in $G_{L \cap R}$, as this is not a single variable. Instead, we introduce an additional start variable $S_{R \cap L}$ and add the rules

- $S_{R \cap L} \rightarrow (S_L)_{1f}$ for all $q_f \in F_R$.

To simulate the use of a single rule in $G_L$ together with the corresponding move in $M_R$ in the case that the rule generates a terminal, we create the following rules.

- $U_{ij} \rightarrow a$ if $A \rightarrow a$ is a rule in $G_L$ and $\delta_R(q_i, a) = q_j$.

- $A_{ik} \rightarrow B_{ij}C_{jk}$ if $A \rightarrow BC$ is a rule in $G_L$, for all $i, j, k$, $1 \leq i, j, k \leq m$.

Finally, to handle the case that $\lambda \in L \cap R$, i.e. that $S_L \rightarrow \lambda$ is a rule in $G_L$ and $\lambda \in R$, we add the following rule.

- $S_{L \cap R} \rightarrow \lambda$ if $\lambda \in L \cap R$. 

To see why this works, we consider any non-empty string \( w \in L \cap R \) and look at a derivation tree \( T \) for \( w \) with respect to \( G_L \). At the same time we look at a \( w \)-recognizing path \( P \) in \( M_R \).

We label each leaf of \( T \) with the names of two vertices in \( M_R \): the vertices that \( M_R \) is at right before and right after reading the input character corresponding to the character labeling the leaf. If we read across the leaves from left to right, recording vertex and character labels, we obtain a sequence \( p_1w_1p_2w_2p_3, \ldots, p_nw_np_{n+1} \), where \( p_1 = \text{start}, p_{n+1} \in F_R \), and \( \delta_R(p_i, w_i) = p_{i+1} \) for \( 1 \leq i \leq n \).

Next, we provide vertex labels to the internal nodes in \( T \). A node receives as its first label the first label of its leftmost leaf and as its second label the second label of its rightmost leaf. Suppose that \( A \) is the variable label at an internal node with children having variable labels \( B \) and \( C \). To obtain the derivation in \( G_L \cap R \), we simply replace \( A \Rightarrow BC \) by \( A \Rightarrow B \) where \( p \) and \( q \) are the vertex labels on the leaf (and on its parent). Clearly, this is a derivation in \( G_L \cap R \) and further it derives \( w \). Thus if \( w \in L \cap R \), then \( S_{L \cap R} \Rightarrow^* w \), i.e. \( w \in (G_{L \cap R}) \).

On the other hand, suppose that \( S_{L \cap R} \Rightarrow^* w \). Then consider the derivation tree for \( w \) in \( G_{L \cap R} \). On removing the root and replacing each variable \( U_{ij} \) by \( U \) we obtain a derivation tree for \( w \) in \( G_L \) (for recall that the root has one child labeled by \( (S_L)_{1f} \) for some \( q_f \in F_R \); the variable replacement process turns this child’s label to \( S_L \)). Thus \( S_L \Rightarrow^* w \) also.

On looking at the leaf level, and labeling each leaf with the vertex indices on the variable at its parent, we obtain a sequence \( p_1w_1p_2w_2p_3, \ldots, p_nw_np_{n+1} \), where \( w = w_1w_2 \cdots w_n, p_1 = \text{start} \) and \( p_{n+1} \in F_R \). As \( \delta(p_i, w_i) = p_{i+1} \), for \( 1 \leq i \leq n \), by the first rule definition for \( G_{L \cap R} \), we see that \( p_1p_2p_3 \cdots p_{n+1} \) is a \( w \)-recognizing path in \( M_R \), and so \( w \in R \). This shows that if \( S_{L \cap R} \Rightarrow^* w \) then \( w \in L \cap R \). \( \square \)

3.6.1 Constructing a CFG Generating a PDA-Recognized Language

Our final construction will show that if \( L \) is recognized by a PDA, then there is a CFG generating \( L \). The first step in the construction it to represent the computation of the PDA in terms of matching pushes and pops to the stack.
A little more terminology will be helpful.

**Definition 3.6.4.** A computation of $M$ that goes from vertex $p$ to vertex $q$ and begins and ends with $\sigma$ on the stack is called $\sigma$-preserving if throughout the computation $\sigma$ remains on the stack (possibly with other characters pushed on top of some of the time); i.e. none of $\sigma$ is popped during this part of the computation.

**Lemma 3.6.5.** Let $P$ be a computation path going from vertex $p$ to vertex $q$ that reads input $w$ and that is $\sigma$-preserving for some $\sigma$. Then the same computation is also $\tau$-preserving, for any $\tau$.

**Proof.** The computation never seeks to pop any of the characters forming $\sigma$ from the stack, so it does not matter what is on this portion of the stack. All that matters is what is pushed once the computation starts. Thus if the $\sigma$ is replaced by $\tau$ it has no effect on the computation.

In other words, the specific $\sigma$ on the stack when a $\sigma$-preserving computation begins is irrelevant. Accordingly, we also call a $\sigma$-preserving computation a *stack-preserving* computation.

**Matching Pushes and Pops: The Trapezoidal Decomposition**

Let $M_L = (Q, T, \Gamma, F, s, \delta)$ be a $\$$-shielded PDA recognizing $L$ that has a single final vertex $f$ which is reachable only with an empty stack. Let us further suppose that on each move $M_L$ does either a Pop or a Push but not both (if there is a move in which neither a Pop nor a Push occurs, it can be replaced by two moves, the first being an unnecessary Push and the second being a Pop; likewise, a move which has both a Pop and a Push can be replaced by two moves: a Pop followed by a Push). Finally, we suppose that the first and last steps of $M_L$’s computation must go from vertex $s$ to $s'$ and from $f'$ to $f$, respectively, and do not do any reads, but just push and pop the $\$$ shield.

We begin with an example that illustrates the general construction. Consider the following PDA $M_L$ which recognizes the set $L$ of strings with equal numbers of $a$’s and $b$’s. The diagram for $M_L$ is shown in Figure 3.21. Now consider a possible computation of $M_L$ on input $a(ab)(b(aa)(ba)b$ (the brackets indicate one way of matching the $a$’s and $b$’s). It is helpful to view $M_L$’s computation in terms of a *Stack Contents Diagram*, as shown in Figure 3.22. It shows the height of the stack evolving as the computation proceeds. The successive nodes in the diagram represent the successive vertices reached by $M_L$ over the course of its computation. For clarity in our example, so as to distinguish the multiple instances of vertex $p$, we name them $p^1, p^2$, etc. In general, though, we will label a node in the diagram with the name of the corresponding vertex in $M_L$; thus multiple nodes could have the same label. When no confusion will result, we will often name a node by using its label (even when this is not a unique identifier). Also, each read is shown as a label on an edge; so for example in going from $p^1$ to $p^2$ an $a$ is read (i.e. in going from the first visit to vertex $p$ to the second
visit to $p$). In addition, the stack contents at any point in the computation can be obtained by reading vertically up the series of trapezoids; e.g. at $p^6$, the stack contents are $\lambda ABB$.

Notice that the string recognized by this computation can be obtained by concatenating the series of edge labels in left to right order. Our goal is to create a derivation tree that has one leaf for each edge, in the same left to right order, and with each leaf having the same label as the corresponding edge. One way to do this is to introduce one internal node for each trapezoid and one leaf for each edge in the Stack Contents Diagram. Clearly, this derivation tree derives the string recognized by the computation of $M_L$ represented in the Stack Contents Diagram. Of course, we have yet to specify the variables and rules of the grammar that would actually let us generate this derivation tree. But the goal is clear: we want to generate the derivation tree shown in Figure 3.23.

Now we are ready to describe how to create the grammar $G_L$ generating $L$.

**Forming the trapezoids** We associate each node in the Stack Contents Diagram with a matching successor or predecessor node (or possibly both), as follows.

**Definition 3.6.6.** Nodes $p$ and $q$ are matched if $q$ is the first vertex following $p$ for which the computation path in $M_L$ from $p$ to $q$ is $\sigma$-preserving.

If we draw the edges connecting matched vertices in the Stack Contents Diagram, this naturally partitions the diagram into trapezoids. We call this the *Trapezoidal Decomposition*. In our example, the first and last vertices are $s$ and $f$, and they are matched. All the other vertices are instances of vertex $p$. Note that $p^2$ is matched with $p^4$, and $p^4$ is also matched with $p^8$. (The vertices at a point in the computation where $M_L$ switches from doing Pops to Pushes will be the ones that are double matched.)

We give each trapezoid $T$ a name that exactly specifies the two computation steps it represents. Each such name has the form $T_{Aab}^{pqrs}$, where the first computation step comprises “Read($a$), Push($A$)” operations and goes from vertex $p$ to vertex $q$, and the second computation step comprises “Read($b$), Pop” operations and goes from vertex $r$ to vertex $s$. Notice
that either or both of $a$ and $b$ could equal $\lambda$; indeed, in our example the bottommost trapezoid is $T_{\lambda\lambda\lambda}$. We will only allow values for the parameters $A, a, b, p, q, r, s$ that correspond to possible computation steps of $M_L$ and to emphasize this point we refer to such trapezoids $T_{pqrs}$ as realizable trapezoids. In our example, $T_{Aab}$ is realizable but $T_{Aaa}$ is not, for opposite characters must be read by the two computation steps represented by a trapezoid.

We also note that the intermediate computation that takes $M_L$ from $q$ to $r$ must be stack-preserving. The consequence is that the Pop performed in $T$’s second step pops the very $A$ pushed in its first step. And so the full computation going from $p$ to $s$ is also stack-preserving. We call this computation a phase. Note that the phase starting at node $p$ is simply the shortest stack-preserving computation beginning at node $p$.

Let $P$ be the phase associated with trapezoid $T$. We call $T$ the base trapezoid for $P$. If $T = T_{pqrs}$, then $P$ consists of $T$ plus a stack-preserving computation that goes from $q$ to $r$. As this computation is stack-preserving, it consists of one or more (sub)-phases, $P_1, P_2, \ldots, P_k$, say, where $k \geq 1$. In our example, the phase $P$ that begins at $p^1$ ends at $p^{11}$. $p^1p^2p^{10}p^{11}$ is its base trapezoid. $P$ contains three subphases that begin and end at, respectively, $p^2$ and $p^4$, $p^4$ and $p^8$, and $p^8$ and $p^{10}$.

If a phase consists of just two steps, we call its base trapezoid a triangle. In our example, the subphase which goes from $p^2$ to $p^4$ is a triangle.

A key property of the trapezoids (and phases) is their nesting. Suppose that realizable trapezoid $T$ has $k$ realizable trapezoids $T_1, T_2, \ldots, T_k$ immediately above it in the trapezoidal decomposition. We will say that $T_{i+1}$ follows $T_i$, for $1 \leq i < k$, meaning that the bottom right-hand corner of $T_i$ has the same label as the bottom left-hand corner of $T_{i+1}$. We will also say that $T_1, T_2, \ldots, T_k$ are nested in $T$, meaning that $T$’s top left-hand corner has the same label as the bottom left-hand corner of $T_1$, and $T$’s top right-hand corner has the same
Figure 3.23: Derivation Tree for string $a(ab)(bbaa)(ab)b$ recognized by $M_L$

label as the bottom right-hand corner of $T_k$, and that $T_{i+1}$ follows $T_i$, for $1 \leq i < k$. In terms of the trapezoid names, we have that $T_i = T_{q_i q_{i+1}^o}$, for $1 \leq i \leq k$, and $T = T_{q_1 q_{k+1}^o}$, for some $q_1, q_2, \cdots, q_{k+1}$, where the $^o$ instead of parameters indicate unspecified values.

**The Context Free Grammar**  For each realizable trapezoid $T = T_{pqr}^{Aab}$ in the trapezoidal decomposition we will create a node in the derivation tree labeled by a variable $U = U_{pqr}^{Aab}$. In our example, this yields the following three variables:

$$U^{\$\lambda}_s p f, U^{Aab}_{ppp}, \text{ and } U^{Bba}_{ppp}.$$  

The start variable in our example is $U^{\$\lambda}_s p f$; it corresponds to the initial and final moves of a recognizing computation, which are respectively, a move from vertex $s$ to $p$ pushing $\$$, and a move from vertex $p$ to $f$ popping this same $\$.$

Let $T$ be a realizable trapezoid and suppose that realizable trapezoids $T_1, T_2, \cdots, T_k$ are nested inside $T$. Let the corresponding nodes in the derivation tree be named $v$ and $v_1, v_2, \cdots, v_k$, respectively, and let them be labeled by $U$ and $U_1, U_2, \cdots, U_k$, respectively. We will say that $U_1, U_2, \cdots, U_k$ are nested inside $U$ and that $U_{i+1}$ follows $U_i$, for $1 \leq i < k$.

**A Difficulty: High Degree Nodes**  However, there is one difficulty: we would like $v_1, v_2, \cdots, v_k$ to be the children of $v$ in left to right order. But nodes in a derivation tree can have only bounded degree, so for large $k$, $v$ cannot have $k$ children. Instead, we replace each high degree node with an equivalent subtree of low degree nodes, which we do using the standard construction for representing a tree with high degree nodes using a binary tree. We illustrate the solution for our example in Figure 3.24. In detail, if $k > 1$, we introduce $k - 1$ vertices $x_1, \cdots, x_{k-1}$. $x_1$ is made the middle child of $v$ ($v$’s other two children being the leaves labeled by $a$ and $b$, where $T = T_{o o o o o}^{o o o o o}$). For $1 \leq i \leq k - 2$, $x_i$ is given left child $v_i$ and right child $x_{i+1}$, while $x_{k-1}$ is given left child $v_{k-1}$ and right child $v_k$.

**The Grammar**  All that remains is to specify the rules of our grammar.
Suppose that realizable trapezoids $T_1, T_2, \ldots, T_k$ are nested inside realizable trapezoid $T$, where $T$ begins with a “Read a” operation and ends with a “Read b”. Then we need to be able to perform the corresponding derivation, namely:

$$U \Rightarrow^* aU_1U_2 \cdots U_k b.$$ 

If $k = 1$ we add the rule

$$U \rightarrow aU_1 b.$$ 

For larger values of $k$, first recall that $U_i = U_{q_i q_{k+1}}$ for $1 \leq i \leq k$ and $U = U_{q_1 q_{k+1}}$. We will introduce $k - 1$ intermediate variables $X_i = X_{q_i q_{k+1}}$ for $1 \leq i < k$, where $X_i$ will be able to derive $U_i \cdots U_k$. This derivation will use the following rules:

- $U \rightarrow aX_1 b$
- $X_i \rightarrow U_i X_{i+1}$, for $1 \leq i \leq k - 2$
- $X_{k-1} \rightarrow U_{k-1}U_k$

To ensure these rules are available we create the following (possibly larger) collection of rules.

- $A_{pqq} \rightarrow \lambda$, for all variables $U_{pqq}^{A_{pqq}}$, i.e. for all realizable triangles $T_{pqq}^{A_{pqq}}$.
- $A_{pqx} \rightarrow aU_{pqx}^{C_{qst}} b$, for all variables $U_{pqx}^{A_{pqx}}$ and $U_{pqx}^{C_{qst}}$, i.e. for all realizable $T_{pqx}^{A_{pqx}}$ and $T_{pqx}^{C_{qst}}$.
- $A_{pqrs} \rightarrow aX_{q}, b$, for all variables $U_{pqrs}^{A_{pqrs}}$.
- $X_{pt} \rightarrow U_{pqrs}^{A_{pqrs}}U_{st}^{C_{qst}}$, for all variables $U_{pqrs}^{A_{pqrs}}$ and all $t \in Q$.
- $X_{py} \rightarrow U_{pqrs}^{A_{pqrs}}U_{st}^{C_{qst}}$, for all variables $U_{pqrs}^{A_{pqrs}}$ and $U_{pqrs}^{C_{qst}}$.

For our example, this creates the following rules.

$$U_{sppf}^{A_{sppf}} \rightarrow U_{ppp}^{A_{ppp}} | U_{ppp}^{B_{ppp}}$$

$$U_{ppp}^{A_{ppp}} \rightarrow aU_{ppp}^{A_{ppp}} b | aU_{ppp}^{B_{ppp}} b | aX_{pp} b | ab$$

$$U_{ppp}^{B_{ppp}} \rightarrow bU_{ppp}^{A_{ppp}} a | bU_{ppp}^{B_{ppp}} a | bX_{pp} a | ba$$

$$X_{pp} \rightarrow U_{ppp}^{A_{ppp}}U_{ppp}^{A_{ppp}} | U_{ppp}^{A_{ppp}}U_{ppp}^{B_{ppp}} | U_{ppp}^{B_{ppp}} X_{pp} | U_{ppp}^{B_{ppp}} X_{pp}$$

$$X_{pp} \rightarrow U_{ppp}^{A_{ppp}} X_{pp} | U_{ppp}^{B_{ppp}} X_{pp}$$
We let \( G_L \) denote the grammar created by the above construction.

It should be clear that if \( M_L \) recognizes \( w \) then we can build a derivation tree for \( w \) in \( G_L \) as described above, though we will prove this formally in Lemma 3.6.8. We also need to confirm that the only strings having derivation trees are those recognized by \( M_L \), which we do in Lemma 3.6.9. First, we give some preliminary lemmas regarding the grammar.

**Lemma 3.6.7.** Suppose that \( U_1, U_2, \ldots, U_k \) are nested inside \( U = U_{pqrs}^{Aab} \). Then there is a derivation \( U \Rightarrow^* a U_1 U_2 \cdots U_k b \).

*Proof.** Note that for \( 1 \leq i \leq k \), \( U_i = U_{q_i oo q_i+1}^{oo} \), for some \( q_1, q_2, \ldots, q_{k+1} \), with \( q_1 = q \) and \( q_{k+1} = r \), where \( o \) denotes an unspecified value.

If \( k = 1 \), the derivation uses the rule \( U_{pqrs}^{Aab} = U_{pqq_2}^{Aab} a U_{q_1 oo q_2}^{oo} b = a U_1 b \) which is a rule because \( U_{pqrs}^{Aab} \) is a variable.

For \( k > 1 \), we use the following series of rules to derive \( a U_1 U_2 \cdots U_k b \).

i. \( U \Rightarrow a X_{q_1} b = a X_{q_1 q_{k+1}} b \).

ii. In turn, for \( 1 \leq i \leq k - 2 \), \( X_{q_i q_{k+1}} \Rightarrow U_{q_i oo q_{i+1}}^{oo} X_{q_{i+1} q_{k+1}} = U_i X_{q_{i+1} q_{k+1}} \).

iii. Finally, \( X_{q_{k-1} q_{k+1}} \Rightarrow U_{q_{k-1} oo q_k}^{oo} U_{q_k oo q_{k+1}}^{oo} = U_{k-1} U_k \).

It is readily confirmed that these are all rules of the grammar, and the net effect of applying this series of rules is to make the derivation \( U \Rightarrow^* a U_1 U_2 \cdots U_k b \). \( \square \)

**Lemma 3.6.8.** Let \( \sigma \) be read by a phase of \( M_L \)'s computation that goes from vertex \( p \) to vertex \( s \) and has base trapezoid \( T_{pqrs}^{Aab} \). Then, in grammar \( G_L \), \( U_{pqrs}^{Aab} \) can derive \( \sigma \).

*Proof.** We create the derivation tree, as described above.

We now prove by induction that if a computation corresponding to a Phase \( \mathcal{P} \) with base trapezoid \( T_{pqrs}^{Aab} \) reads string \( \sigma \), then the corresponding derivation tree derives string \( \sigma \). If \( \mathcal{P} \) corresponds to a triangle, \( T_{pqrs}^{Aab} \), then \( \mathcal{P} \) reads string \( \sigma = ab \) (\( r = q \) in this case); also \( U_{pqrs}^{Aab} \rightarrow a b \) is a rule, so \( U_{pqrs}^{Aab} \) can derive \( ab \).

Otherwise, suppose that \( \mathcal{P} \) has base trapezoid \( T_{pqrs}^{Aab} \) and subphases \( \mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_k \), with base trapezoids \( T_1, T_2, \ldots, T_k \). Let \( U_i \) be the variable corresponding to \( T_i \), for \( 1 \leq i \leq k \). \( T_1, T_2, \ldots, T_k \) are nested inside \( T_{pqrs}^{Aab} \). Correspondingly, \( U_1, U_2, \ldots, U_k \) are nested inside \( U_{pqrs}^{Aab} \).

By Lemma 3.6.7, \( U \Rightarrow^* a U_1 U_2 \cdots U_k b \). If the computation corresponding to phase \( \mathcal{P}_i \) reads \( \sigma_i \) then phase \( \mathcal{P} \) reads \( ab \sigma_2 \cdots \sigma_k b = \sigma \). By induction, \( U_i \) can derive \( \sigma_i \). It follows that \( U_{pqrs}^{Aab} \) can derive \( a \sigma_2 \cdots \sigma_k b = \sigma \). \( \square \)

**Lemma 3.6.9.** Suppose that, in the grammar \( G_L \), \( U_{pqrs}^{Aab} \) can derive \( \sigma \); then there is a phase of \( M_L \)'s computation that reads \( \sigma \) while going from vertex \( p \) to vertex \( s \), and which has base trapezoid \( T_{pqrs}^{Aab} \).

*Proof.** Consider a derivation tree for \( w \) starting at a root node labeled by variable \( U_{pqrs}^{Aab} \). For each node labeled by variable \( U_{wxyz}^{Ccd} \) create a Phase \( \mathcal{P} \) with base trapezoid \( T_{wxyz}^{Ccd} \). Suppose that variables \( U_1, U_2, \ldots, U_k \) are nested inside variable \( U \). Let \( \mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_k \) and \( \mathcal{P} \) be the
corresponding phases and let $P_i$ have base trapezoid $T_i$, for $1 \leq i \leq k$; then $T_1, T_2, \ldots, T_k$ are nested inside $T$.

We argue by induction that the resulting computation is a phase which reads string $\sigma$, and has base trapezoid $T_{Aab}^{pqrs}$. If the derivation tree has two leaf nodes, then the derivation has the form $U_{pqrs}^{Aab} \Rightarrow ab$ ($r = q$ in this case). The corresponding computation comprises a single trapezoid $T_{pqrs}^{Aab}$, which forms a phase, and this computation reads $ab$. Otherwise, suppose that $U_1, U_2, \ldots, U_k$ are nested inside $U_{pqrs}^{Aab}$. Then, by Lemma 3.6.7, $U_{pqrs}^{Aab}$ can derive the sequence $aU_1U_2\cdots U_kb$. For $1 \leq i \leq k$, let $P_i$ be the subphase created corresponding to the node labeled by variable $U_i$; $P_i$ has base trapezoid $T_i = T_{q_i^{q_{i+1}}}$, for some $q_1, q_2, \ldots, q_{k+1}$, with $q = q_1$ and $r = q_{k+1}$. Suppose that $U_i$ derives $\sigma_i$ in the derivation tree; then by induction $P_i$ reads string $\sigma_i$, goes from vertex $q_i$ to $q_{i+1}$ and is a phase. It follows that $P$ is a phase (for it comprises a base trapezoid along with a sequence of subphases), and it reads string $a\sigma_1\sigma_2\cdots\sigma_kb = \sigma$, and finally it goes from vertex $p$ to vertex $s$. \hfill \Box

The main result now follows readily.

**Theorem 3.6.10.** Let PDA $M_L$ recognize language $L$. Then there is a CFL generating $L$.

**Proof.** WLOG, assume that $M_L$ has the form assumed in the current section (a single start vertex, a single final vertex, each move performs either a Pop or a Push but not both etc.). Let $G_L$ be the grammar built by construction of this section. Then, if $M_L$ recognizes string $\sigma$, in must be via a computation with base trapezoid $T_{ss'f'f}$, and by Lemma 3.6.8, $U_{ss'f'f}^{s\lambda\lambda}$ generates $\sigma$, i.e. $\sigma$ is generated by $G_L$. While if $\sigma \in L(G_L)$, $U_{ss'f'f}^{s\lambda\lambda}$ generates $\sigma$, and by Lemma 3.6.9 there is a computation of $M_L$ that reads $\sigma$ while going from $s$ to $f$, i.e. $M_L$ recognizes $\sigma$. \hfill \Box

**Exercises**

1. For the PDA in Figure 3.25 answer the following questions.

![Figure 3.25: Figure for Problem 1](image_url)
i. What is its start vertex?
ii. What are its final vertices?
iii. Give the sequence of vertices and associated data configurations the PDA goes through in an accepting computation on input \( abba \). Are there any other computation paths that can be followed on this input, and if yes, give the sequence of vertices such a computation goes through.
iv. What is the language accepted by this PDA?

2. Give PDAs to recognize the following languages. You are to give both an explanation in English of what each PDA does plus a graph of the PDA; seek to label the graph vertices accurately.
   i. \( A = \{ w \mid w \text{ has odd length, } w \in \{a, b\}^* \text{ and the middle symbol of } w \text{ is an } a \} \).
   ii. \( B = \{ w \mid w \in \{a, b\}^* \text{ and } w \neq a^i b^i \text{ for any integer } i \} \).
   iii. \( C = \{ w \mid w \in \{a, b\}^* \text{ and } w \text{ contains equal numbers of } a\text{'s and } b\text{'s} \} \).
   iv. \( D = \{ w w^R x \mid w, x \in \{a, b\}^* \} \).
   v. \( E = \{ w c w^R x \mid w, x \in \{a, b\}^* \} \).

3. Suppose that \( A \) is recognized by PDA \( M \). Give a PDA to recognize \( A^* \).

4. Let \( C \) be a language over the alphabet \( \{a, b\} \).
   i. Let \( \text{Suffix}(C) = \{ w \mid \text{there is a } u \in \{a, b\}^* \text{ with } uw \in C \} \). Show that if \( C \) is recognized by a PDA then so is \( \text{Suffix}(C) \).
   ii. Similarly, let \( \text{Prefix}(C) = \{ w \mid \text{there is an } x \in \{a, b\}^* \text{ with } wx \in C \} \). Show that if \( C \) is recognized by a PDA then so is \( \text{Prefix}(C) \).

5. This problem aims to show that there is a PDA recognizing the following language: \( \{ s\#t \mid s, t \in \{a, b\}^* \text{ and } s \neq t \} \).
   i. Let \( A = \{ uav\#xby \mid u, v, x, y \in \{a, b\}^* \text{ and } (|u| - |v|) = (|x| - |y|) \} \). Give a PDA to recognize \( A \).
   ii. Let \( B = \{ w\#z \mid w, z \in \{a, b\}^* \text{ and } |w| \neq |z| \} \). Give a PDA to recognize \( B \).
   iii. Show that \( A \cup B = \{ s\#t \mid s, t \in \{a, b\}^* \text{ and } s \neq t \} \).

6. Consider the PDA in Figure 3.25. Add descriptors to the vertices. What language does this PDA recognize?

7. Consider the following context free grammar: \( S \leftarrow (S)|SS|( )|[] \)
   i. What are its terminals?
ii. What are its variables?

iii. What are its rules?

iv. Show the derivation tree for string ([ ] ( )).

v. Describe in English the set of strings generated by this grammar.

8. Give CFG’s to generate the following languages.

   i. \( A = \{ w \mid w \text{ has odd length, } w \in \{a, b\}^* \text{ and the middle symbol of } w \text{ is an } a \} \).

   ii. \( B = \{ w \mid w \in \{a, b\}^* \text{ and } w = w^R \} \). \( A \) is the language of palindromes, strings that read the same forward and backward. 
       Hint: Be sure to handle strings of all possible lengths.

   iii. \( C = \{ w w^R x \mid w, x \in \{a, b\}^* \} \).

   iv. \( D = \{ w \mid w \in \{a, b\}^* \text{ and } w \text{ contains an equal number of } a \text{'s and } b \text{'s} \} \).
       Hint: suppose that the first character in \( w \) is an \( a \). Let \( x \) be the shortest initial substring of \( w \) having an equal number of \( a \text{'s and } b \text{'s. If } |x| < |w|, \text{ then } w \text{ can be written as } w = xy; \text{ what can you say about } y? \text{ Otherwise, } x = w \text{ and } w \text{ can be written as } w = azb; \text{ what can you say about } z? \)

   v. \( E = \{ w \# x \mid w, x \in \{a, b\}^* \text{ and } w^R \text{ is an initial substring of } x \} \).
       Hint: \( x \) can be written as \( x = w^R y \) for some \( x \in \{a, b\}^* \).

9. i. Let \( E = \{ a^i b^j \mid i < j \} \). Give a CFL to generate \( E \).

   ii. Let \( F = \{ a^i b^j \mid 2i > j \} \). Give a CFL to generate \( F \).

   iii. Let \( J = \{ a^i b^j \mid i < j < 2i \} \). Give a CFL to generate \( J \).
       Hint: Let \( i = h + l \) and \( j = h + 2l \). what can you say about \( h \) and \( l? \)

10. i. Give a context free grammar to generate the following language: \( L_1 = \{ a^i \# b^{i+j} \$ a^j \mid i, j \geq 0 \} \).

   ii. Give a context free grammar to generate the following language: \( L_2 = \{ w \# x \$ y \mid w, x, y \in \{a, b\}^* \text{ and } |x| = |w| + |y| \} \).

   iii. Hence give a context free grammar to generate the following language: \( L_3 = \{ uv \mid |u| = |v| \text{ but } u \neq v \} \). Hint: think of the \( \# \) and the \( \$ \) from part (2) as a pair of aligned yet unequal characters in \( u \) and \( v \); what is the relation among the lengths of the remaining pieces of \( u \) and \( v? \)

11. Let \( A \) be a CFL generated by a CFG \( G_A \). Give a CFG grammar \( G_{A^*} \), based on \( G_A \), to generate \( A^* \). Show that \( L(G_{A^*}) = A^* \).

12. Convert the following CFGs to CNF form.

   i. \( G_1 \) has start variable \( S \), terminal set \( \{a, b\} \) and rule \( S \rightarrow ab \).
This exercise concerns a variant of the Pumping Lemma, called Ogden’s Lemma. Ogden’s Lemma is useful for it can be used to show certain languages are not CFLs, languages for which the standard pumping lemma does not suffice.

13. Show that the following languages are not context free.

i. \( A = \{a^ib^ic^j \mid i, j, k \geq 0\} \).

ii. \( B = \{a^mb^n c^m d^n \mid m, n \geq 0\} \).

iii. \( C = \{w \mid w \in \{a,b,c\}^* \text{ and the number of } a \text{’s, } b \text{’s and } c \text{’s in } w \text{ are all equal}\} \).

iv. \( D = \{u#wv#w \mid u, v \in \{a,b\}^*, \text{ the number of } a \text{’s in } u \text{ equals } |v|, \text{ and the number of } b \text{’s in } v \text{ equals } |w|\} \).

v. \( E = \{w \mid w \in \{a,b,c,d\}^* \text{ and } w \text{ contains equal numbers of } a \text{’s and } b \text{’s, and equal numbers of } c \text{’s and } d \text{’s}\} \).

vi. \( F = \{a^i a^j \mid i, j \geq 1\} \).

Comment. Any CFL over a 1-character alphabet is a regular language. Give a proof without using this fact.

vii. \( H = \{a^i \mid i \geq 0\} \).

Comment. Any CFL over a 1-character alphabet is a regular language. Give a proof without using this fact.

viii. \( J = \{x_1 #x_2 # \cdots #x_l \mid x_h \in \{a,b\}^*, 1 \leq h \leq l, \text{ and for some } i, j, k, 1 \leq i < j < k, \mid x_i = \mid x_j = \mid x_k\}\} \).

ix. \( K = \{x_1 #x_2 # \cdots #x_k \mid x_h \in \{a,b\}^*, 1 \leq h \leq k, \text{ and for some } i, j, 1 \leq i < j \leq k, x_i = x_j\} \).

x. \( L = \{a^ib^j \mid i \text{ is an integer multiple of } j\} \).

xi. \( M = \{wxw^R \mid w, x \in \{a,b\}^* \text{ and } |w| = |x|\} \).

xii. \( N \) is the language consisting of all palindromes over the alphabet \( \Sigma = \{a, b, c\} \) having equal numbers of \( a \)'s and \( b \)'s.

xiii. \( P = \{a^ib^j c^j \mid j > i\} \).

xiv. \( Q = \{a^ib^j c^k \mid k = \max\{i, j\}\} \).

14. This exercise concerns a variant of the Pumping Lemma, called Ogden’s Lemma. Ogden’s Lemma is useful for it can be used to show certain languages are not CFLs, languages for which the standard pumping lemma does not suffice.
i. Let $L$ be a CFL and let $s \in L$ be a string in which some characters have been marked. Ogden’s Lemma states that there is a constant $p = p_L$ such that if $s \in L$ is a string with at least $p$ marked characters then $s$ can be written as $s = uvwxy$, where

   a. $vx$ contains at least one marked character.
   b. $vwx$ contains at most $p$ marked characters.
   c. For all $i \geq 0$, $uv^iwx^iy \in L$.

Prove Ogden’s Lemma.

Hint. Proceed in the same way as for the proof of the standard pumping lemma. In order to obtain analogs of Observations 1 and 2 you will need to count just the marked characters; you will also need to redefine the notion of “height” to count just those internal nodes both of whose subtrees have marked characters.

ii. Prove a variant of Ogden’s Lemma in which both $v$ and $x$ contain at least one marked character.

iii. Using Ogden’s Lemma prove that the following languages are not context free.

   a. $L = \{a^ib^jc^kd^l \mid i = 0 \text{ or } j = k = l\}$.
   Why will the ordinary Pumping Lemma fail to prove that $L$ is not a CFL, i.e. show that any string $s \in L$ can be pumped so that all the resulting strings are in $L$.

   b. $K = \{a^ib^jc^j \mid j \neq i\}$.
   Hint. Consider the string $s = a^pb^ps^p$.
   Again, why will the ordinary Pumping Lemma fail to prove that $L$ is not a CFL, i.e. show that any string $s \in L$ can be pumped so that all the resulting strings are in $L$.

15. Consider the following CNF context free grammar.

   $S \rightarrow AB, A \rightarrow AA, A \rightarrow a, B \rightarrow b$.

Show the PDA generated by applying the construction of Lemma 3.6.2 to this grammar.

16. Consider the PDA shown in Figure 3.2.

   i. Draw the trapezoidal diagram for the computation recognizing input $aabb$.
   ii. Give the CFL generated by applying the construction of Section 3.6.1 to this PDA.

17. For each of the language transformations $T$ defined in the parts below, answer the following two questions.

   a. Suppose that $L$ is a CFL. Show that $T(L)$ is also a CFL by giving a CFG to generate $T(L)$. Remember to explain why your solution is correct.
   b. Now suppose that $L$ is recognized by a PDA. Show that $T(L)$ is also recognized by a PDA. Again, remember to explain why your solution is correct.

Comment: The two parts are equivalent; nonetheless, you are being asked for a separate construction for each part.
CHAPTER 3. PUSHDOWN AUTOMATA AND CONTEXT FREE LANGUAGES

i. Let \( w \in \{a, b, c\}^* \). Define \( \text{Sbst}(w, a, b) \) to be the string obtained by replacing all instances of the character \( a \) in \( w \) with \( b \). e.g. \( \text{Sbst(ac, a, b)} = bc \), \( \text{Sbst(cc, a, b)} = cc \), \( \text{Sbst(abc, a, b)} = bbc \), \( \text{Sbst(acacac, a, b)} = bcbcbc \).

Let \( L \) be a language over the alphabet \( \{a, b, c\} \). Define \( T(L) = \text{Sbst}(L, a, b) = \{ x \mid x = \text{Sbst}(w, a, b) \text{ for some } w \in L \} \).

ii. Let \( w \in \{a, b, c\}^* \). Define \( \text{OneSubst}(w, a, b) \), or OS\((w, a, b)\) for short, to be the set of strings obtained by replacing one instance of the character \( a \) from \( w \) with \( b \). e.g. \( \text{OS(acacac, a, b)} = \{bcacac, acbcac, acacbc \} \).

Let \( L \) be a language over the alphabet \( \{a, b, c\} \). Define \( T(L) = \text{OS}(L, a, b) = \{ x \mid x = \text{OS}(w, a, b) \text{ for some } w \in L \} \).

iii. Let \( w \in \{a, b, c\}^* \). Define \( \text{Remove-c}(w) \) to be the string obtained by deleting all instances of the character \( c \) from \( w \). e.g. \( \text{Remove-c(ab)} = ab \), \( \text{Remove-c(cc)} = \lambda \), \( \text{Remove-c(abc)} = ab \), \( \text{Remove-c(acacac)} = aaaa \).

Let \( L \) be a language over the alphabet \( \{a, b, c\} \). Define \( T(L) = \text{Remove-c}(L) = \{ x \mid x = \text{Remove-c}(w) \text{ for some } w \in L \} \).

iv. Let \( w \in \{a, b, c\}^* \). Define \( \text{Remove-One-c}(w) \) to be the set of strings obtained by deleting one instance of the character \( c \) from \( w \). e.g. \( \text{Remove-One-c(acacac)} = \{aacac, acaac, acaca \} \).

Let \( L \) be a language over the alphabet \( \{a, b, c\} \). Define \( \text{L}(T) = \text{Remove-One-c}(L) = \{ x \mid x = \text{Remove-One-c}(w) \text{ for some } w \in L \} \).

v. Let \( h \) be a mapping from \( \Sigma \) to \( \Sigma^* \), that is \( h \) maps each character in \( \Sigma \) to a string of characters. Define \( h(s) \) for a string \( s = s_1s_2 \cdots s_k \) to be the string \( h(s_1)h(s_2) \cdots h(s_k) \).

Define \( T(L) = \{ h(w) \mid w \in L \} \).

vi. Let \( h \) be a mapping from \( \Sigma \) to \( R \) where \( R \) is the set of regular expressions over alphabet \( \Sigma \); i.e. \( h(a) \) is a regular expression for each \( a \in \Sigma \). Define \( h(s) \) for a string \( s = s_1s_2 \cdots s_k \) to be the set of strings specified by the regular expression \( h(s_1)h(s_2) \cdots h(s_k) \).

Define \( T(K) = \{ L(h(w)) \mid w \in K \} \).

18. See Problem 17v. Using a suitable mapping \( h \) from \( \Sigma \) to \( \Sigma^* \) and the standard Pumping Lemma, show that \( L = \{a^ib^jc^kd^l \mid i = 0 \text{ or } j = k = l \} \) is not a CFL.

Comment. See Problem 14.iii.a. Note that Ogden’s Lemma is not needed to show \( L \) is not a CFL.