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.

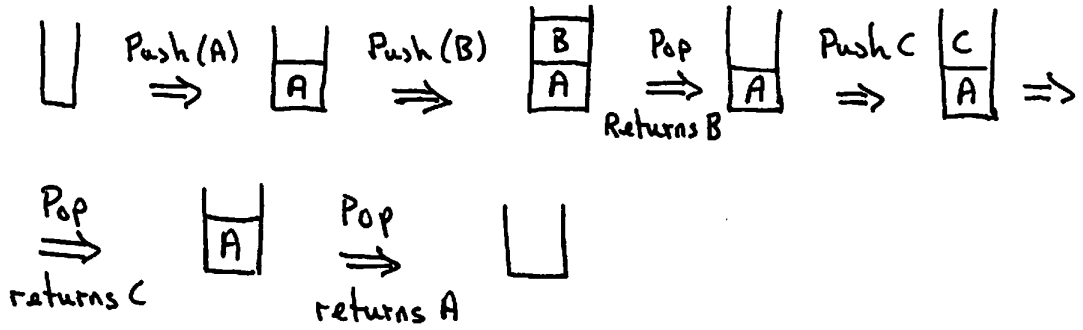

Figure 3.1: Stack Behavior.

Let's see how a stack allows us to recognize the following language $L_1 = \{a^i b^i \mid i \geq 0\}$. We start by explaining how to process a string $w = a^i b^i \in L$. 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. We do this by looking at several categories of strings in turn.

1. $a^i b^h$, $h < i$.
   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. $a^i b^j$, $j > i$.
   On reading the $(i + i)$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.

A PDA M_1 recognizing L_1 is shown in Figure 3.2. Because the descriptions of the

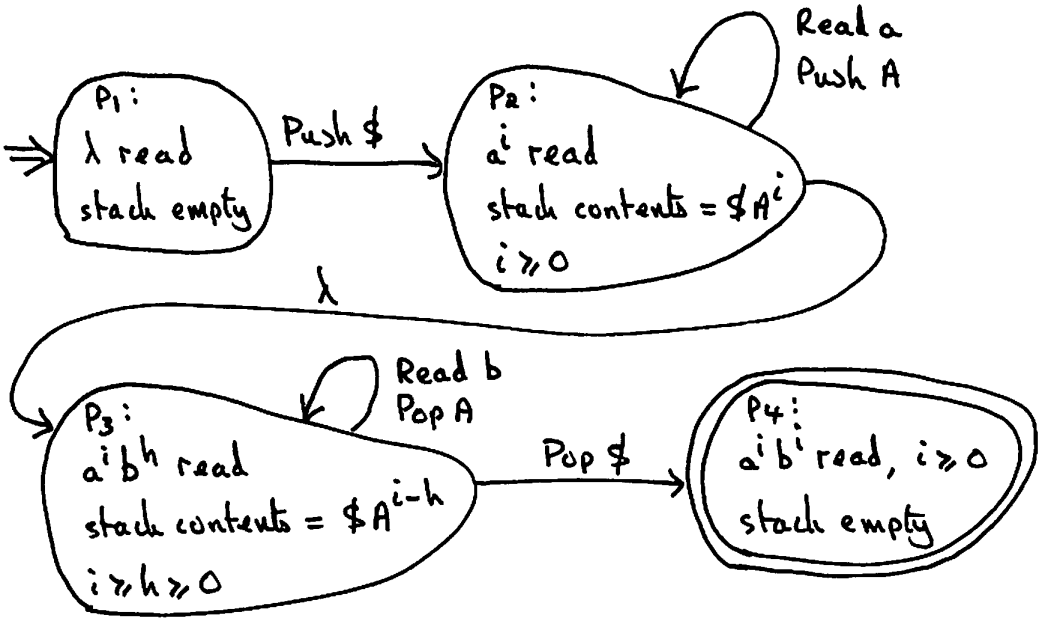

Figure 3.2: PDA M_1 recognizing $L_1 = \{a^i b^i \mid i \leq 0\}$.

vertices are quite long, we have given them the shorter names p_1–p_4. These descriptions specify exactly the strings that can be read and the corresponding stack contents to reach the vertex in question. 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 P then so can N_1 (for the same reads label P in both machines). It follows that any string recognized by M_1 is also recognized by N_1: $L(M_1) \subseteq L(N_1)$.

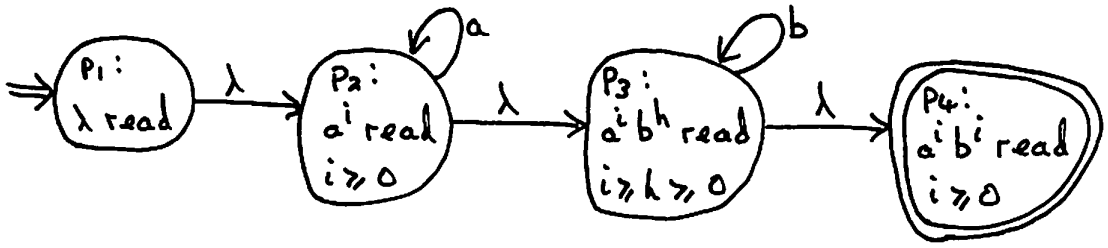

Figure 3.3: The NFA N_1 derived from M_1.

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 the form a^ib^j, $i, j \geq 0$, which we do by examining five subcases that account for all such strings.

1. λ.
   M_1 starts at p_1. On pushing \$, p_2 and p_3 can be reached. Them 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 λ.

2. a^i, $i \geq 1$.
   To read a^i, M_1 needs to push \$ and then follow edge (p_1, p_2) i times. This puts $\$A^i$ 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 $\$A^{i-h}$ 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^i, 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 read a b, then the PDA can traverse e, which entails it pushing C onto the stack. Some or all of these values may be λ: Pop λ means that no Pop is performed, read λ that no read occurs, and Push λ that no Push happens. To avoid clutter, we usually omit the λ-labeled terms; for example, instead of Pop λ, read λ, 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 Σ and a stack alphabet Γ (these are the symbols that can be written on the stack). It is customary for Σ and Γ to be disjoint, in part to avoid confusion. To emphasize this disjointness, we write the characters of Σ using lowercase letters and those of Γ using uppercase letters.

Became 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, $\delta(p, A, b) = \{(q_1, C_1), (q_2, C_2), \cdots, (q_l, C_l)\}$ indicates that the edges exiting vertex p and having both Pop A and read b in their label are the edges going to vertices $q_1, q_2, \cdots, q_l$ where the rest of the label for edge (p, q_i) includes action 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 λ.)

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

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. *start* is the start vertex, and
6. δ 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 $start = p_0, p_1, \cdots, p_m$, where p_m is a final vertex, the label on edge (p_{i-i}, p_i) is (Read a_i, Pop B_i, Push C_i), for $1 \leq i \leq m$, and the stack contents at vertex p_i is σ_i, for $0 \leq i \leq m$, where

1. $a_1 a_2 \cdots a_m = w$,
2. $\sigma_0 = \lambda$,
3. and Pop B_i, Push C_i applied to σ_{i-i} produces σ_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. $L_2 = \{a^i c b^i \mid i \geq 0\}$.

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 $a^* c b^*$ 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.

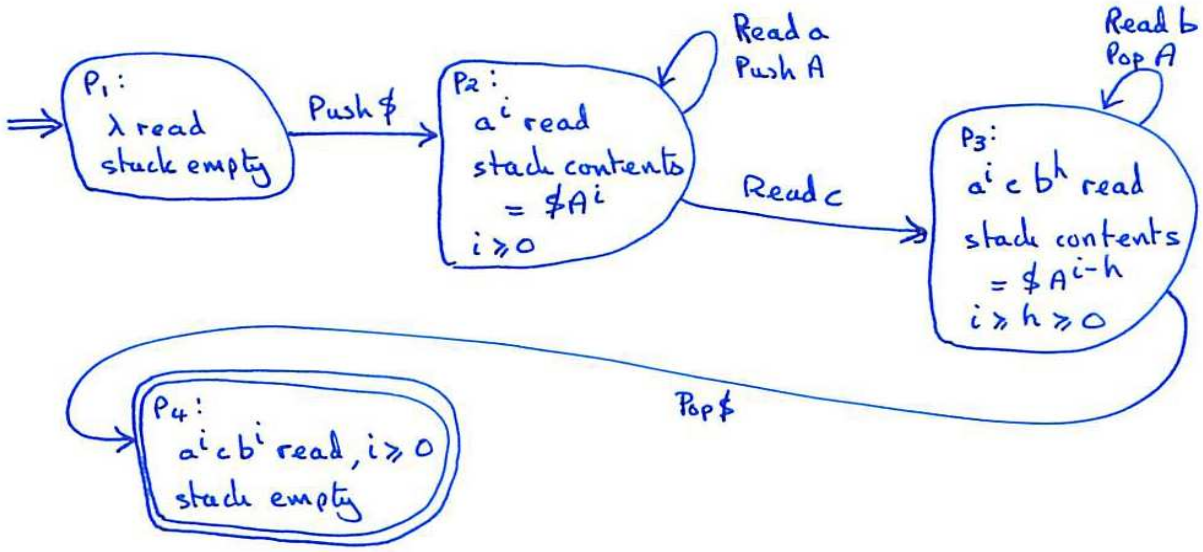

Figure 3.4: PDA M_2 recognizing $L_2 = \{a^i c b^i \mid i \geq 0\}$.

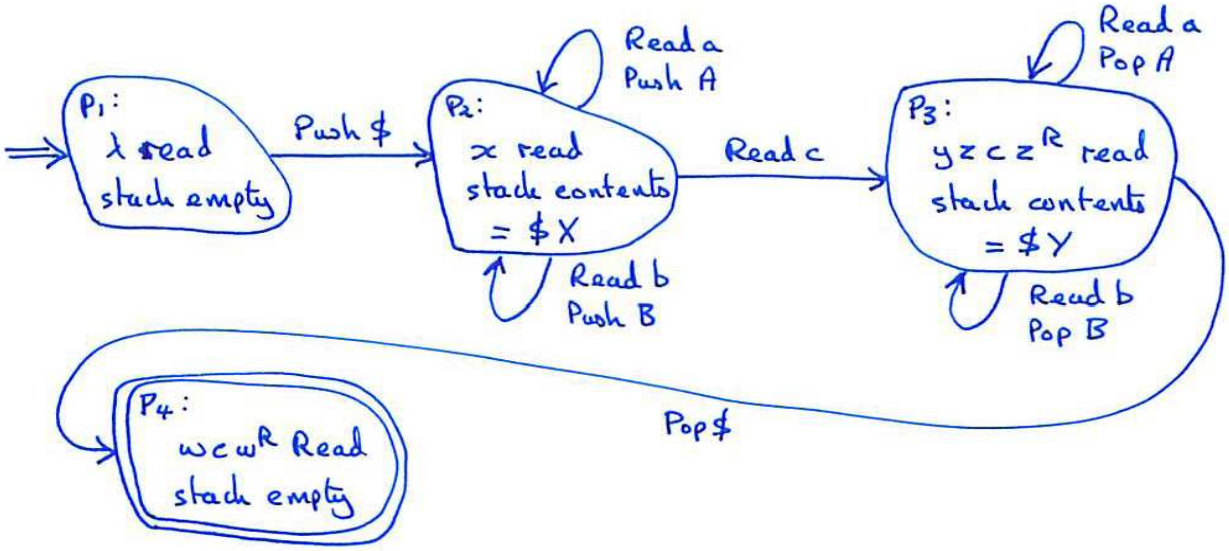

Figure 3.5: PDA M_3 recognizing $L_3 = \{wcw^R \mid w \in \{a, b\}^*\}$. Z denotes string z in capital letters.

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

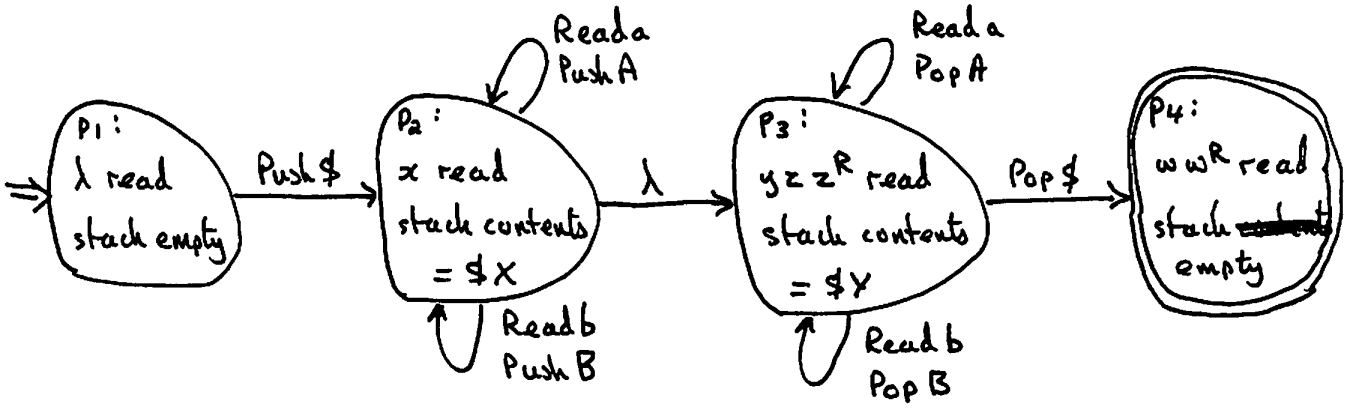

Figure 3.6: PDA M_4 recognizing $L_4 = \{wcw^R \mid w \in \{a, b\}^*\}$. Z denotes string z in capital letters.

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.

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 $L_6 = \{a^i b^i c^k \mid i, k \geq 0\}$ 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 $M_{A \cup B}$.*

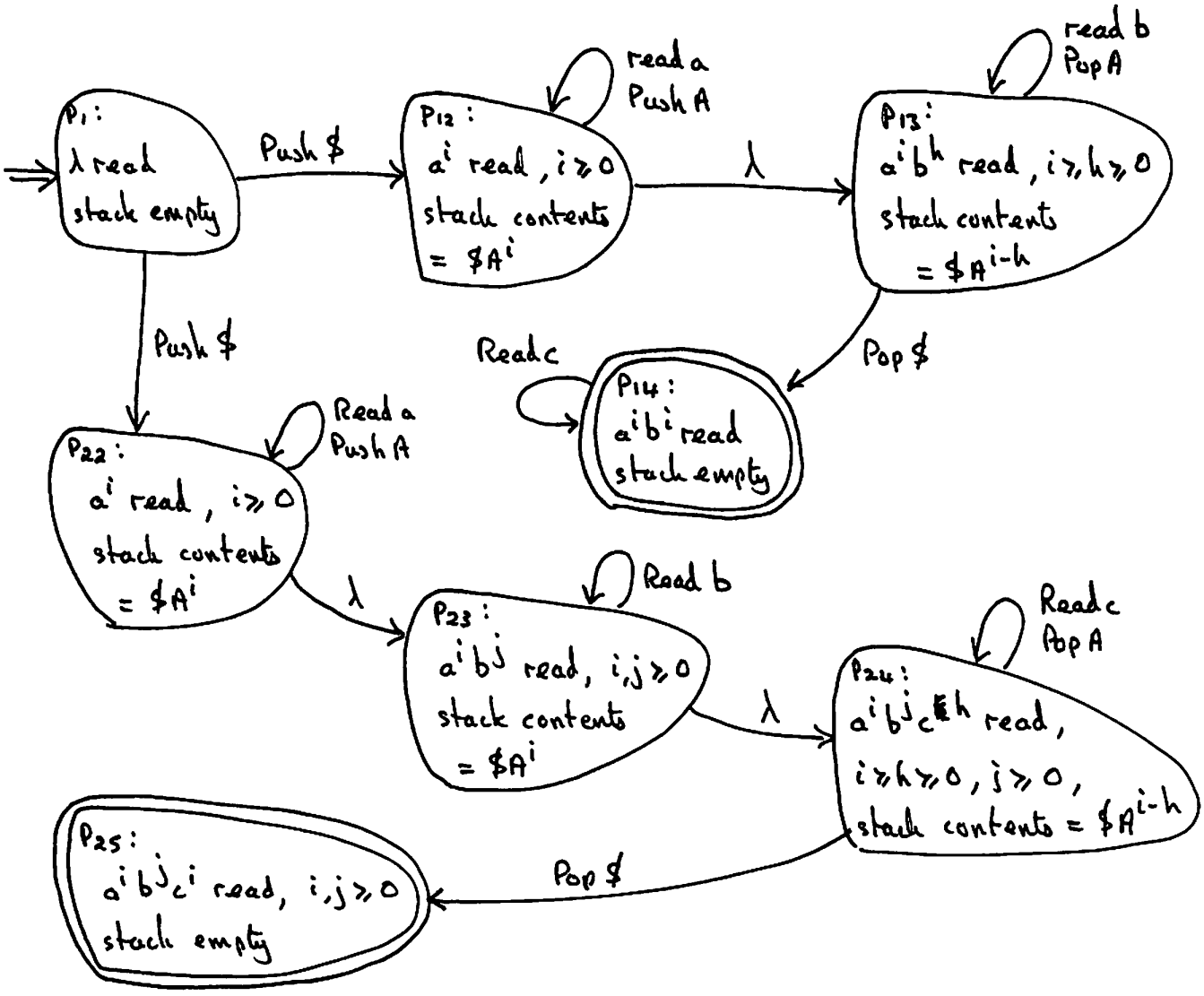

Figure 3.7: PDA M_5 recognizing $L_5 = \{a^i b^j c^k \mid i = j \text{ or } i = k\}$. Z denotes string z in capital letters.

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 λ-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.

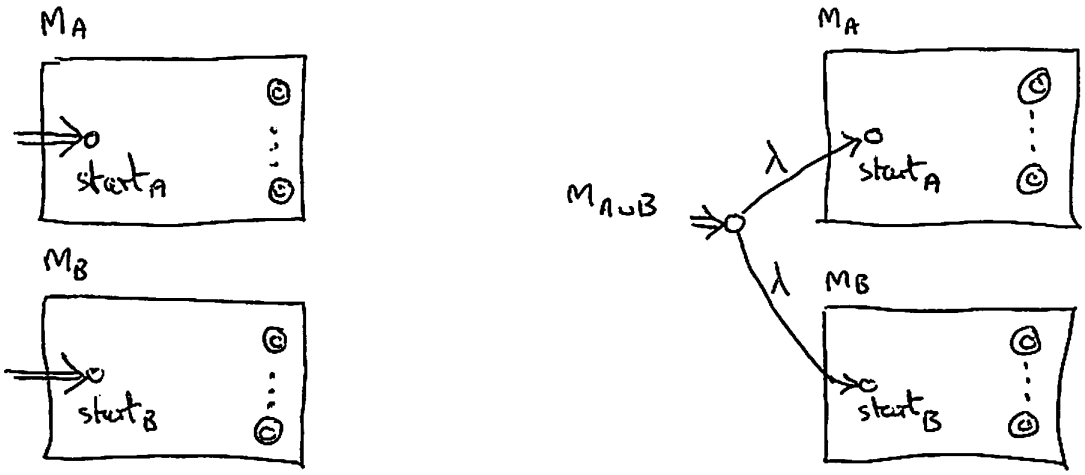

Figure 3.8: PDA $M_{A\cup B}$.

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 $L(M_{A\cup B}) \subseteq L(M_A) \cup L(M_B)$. 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 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 $w \in L(M_B)$. Either way, $w \in L(M_A) \cup L(M_B)$.

Second, we show that $L(M_A) \cup L(M_B) \subseteq L(M_{A\cup B})$. 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 λ-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 $L(M_B) \subseteq L(M_{A\cup B})$. □

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 $final_{M'}$.*

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-empty*, 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-*

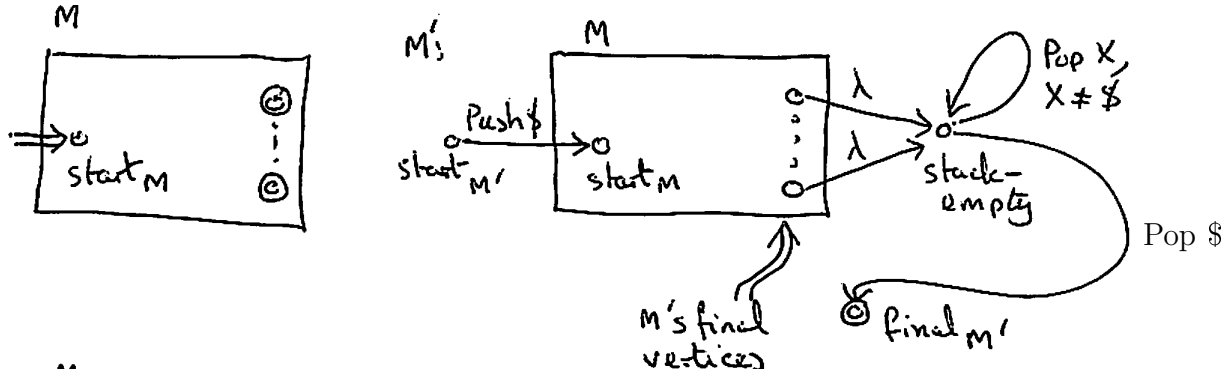

Figure 3.9: PDA M' for Lemma 3.2.2.

empty, and $final_{M'}$. The following edges are also added: $(start_{M'}, start_M)$ labeled Push \$, λ-edges from each of M's final vertices to *stack-empty*, self-loops (*stack-empty*, *stack-empty*) labeled Pop X for each $X \in \Gamma$, where Γ is M's stack alphabet (so $\$ \neq X$), and edge $(\textit{stack-empty}, final_{M'})$ labeled Pop \$.

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

First, we show that $L(M') \subseteq L(M)$. 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-empty* on the path P'. Removing the first edge in P' and every edge including and after $(f, \textit{stack-empty})$, leaves a path P which is a w-recognizing path in M. Thus $L(M') \subseteq L(M)$.

Now we show $L(M) \subseteq L(M')$. 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 $(start_{M'}, start_M)$, $(f, \textit{stack-empty})$, $|s|$ self-loops at *stack-empty*, and $(\textit{stack-empty}, final_{M'})$ 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 $L(M) \subseteq L(M')$. □

Lemma 3.2.3. *Let A and B be languages recognized by PDAS M_A and M_B, respectively, Then $A \circ B$ is also recognized by a PDA called $M_{A \circ B}$.*

Proof. Let M_A and M_B be PDAs recognizing A and B, respectively, where they each have just one final vertex that can be reached only with an empty stack.

$M_{A \circ B}$ consists of M_A, M_B plus one λ-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. For $w \in L(M_{A \circ B})$ if and only if 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), λ-edge $(final_A, start_B)$, and a path P_B in M_B going from $start_B$ to $final_B$. Let u be the sequence of reads labeling P_A and v

those labeling P_B. Then $w = uv$, P_A is u-recognizing, and P_B is v-recognizing (see Figure 3.10). So $w \in L(M_{A\circ B})$ if and only if $w = uv$, where $u \in L(M_A) = A$ and $v \in L(M_B) = B$.

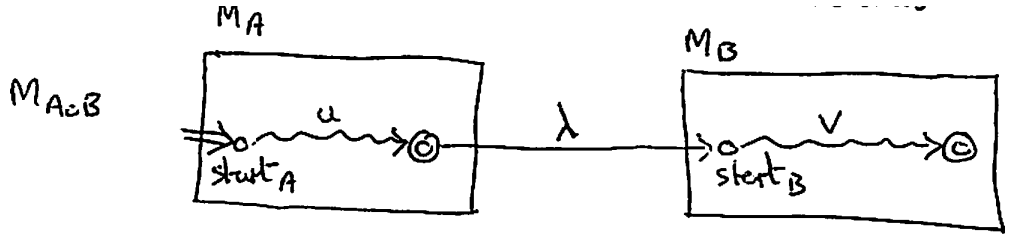

Figure 3.10: PDA $L(M_{A\circ B})$.

In other words $w \in L(M_{A\circ B})$ if and only if $w \in A \circ B$. □

Lemma 3.2.4. *Suppose that L is recognized by PDA M_L and suppose that R is a regular language. Then $L \cap R$ is recognized by a PDA called $M_{L\cap R}$.*

Proof. Let $M_L = (\Sigma, \Gamma_L, V_L, start_L, F_L, \delta_L)$ and let R be recognized by DFA $M_R = (\Sigma, V_R, start_R, F_R, \delta_R)$. We will construct $M_{L\cap R}$. 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 λ, Push C) then $M_{L\cap R}$ has an edge $((u_L, u_R), (v_L, u_R))$ with label (Pop A, Read λ, Push C) for every $u_R \in V_R$.

The start vertex for $M_{L\cap R}$ is $(start_L, start_R)$ 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 λ 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 $(start_L, start_R)$. 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 λ (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 $w \in L \cap R$.

□

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.

$$\begin{aligned} \langle\text{decimal string}\rangle &\rightarrow \langle\text{digit}\rangle \mid \langle\text{digit}\rangle\langle\text{decimal string}\rangle \\ \langle\text{digit}\rangle &\rightarrow 0 \mid 1 \mid 2 \mid \cdots \mid 9 \end{aligned} \tag{3.1}$$

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" ⟨decimal string⟩. The sequence of replacements generating 147 is shown below.

$$\begin{aligned} \langle\text{decimal string}\rangle &\Rightarrow \langle\text{digit}\rangle\langle\text{decimal string}\rangle \\ &\Rightarrow 1\langle\text{decimal string}\rangle \\ &\Rightarrow 1\langle\text{digit}\rangle\langle\text{decimal string}\rangle \\ &\Rightarrow 14\langle\text{decimal string}\rangle \\ &\Rightarrow 14\langle\text{digit}\rangle \\ &\Rightarrow 147 \end{aligned}$$

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

- ⟨decimal string⟩ and replaces it with one of ⟨digit⟩ or the sequence ⟨digit⟩⟨decimal string⟩.

- ⟨digit⟩ and replaces 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

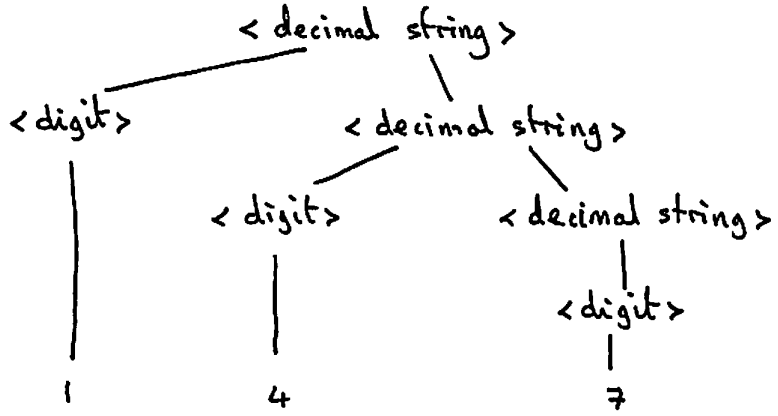

Figure 3.11: Parse Tree Generating 147.

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 "×" operators, and parentheses. We also would like the generation rules to follow the precedence order of the operators "+" and "×", 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 "+"; 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

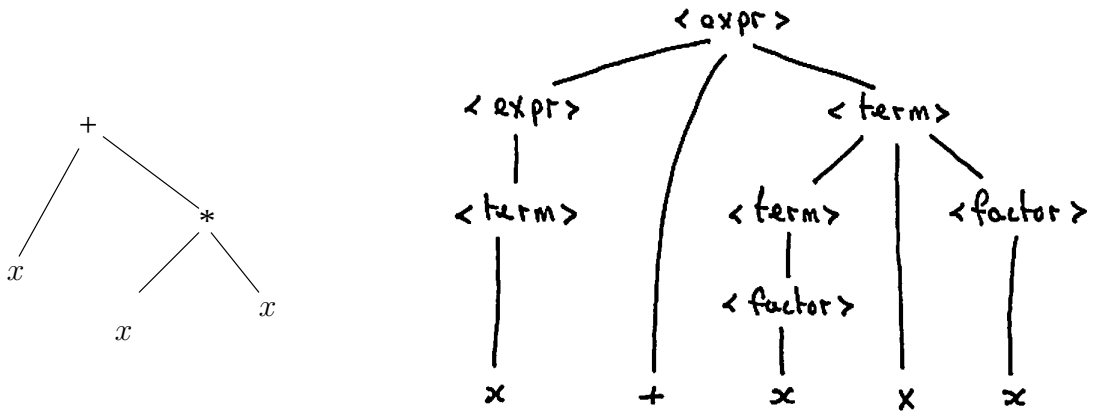

Figure 3.12: Parse Tree Generating $x + x \times x$.

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 "×" orator. 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 $\langle Balanced \rangle$.

$$\langle Balanced \rangle \Rightarrow \lambda \mid a\langle Balanced \rangle b.$$

Notice how a string $s \in L$ is generated: from the outside in. First the outermost a and b are created, together with a $\langle Balanced \rangle$ term between them; this $\langle Balanced \rangle$ 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 λ.

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:

1. A set V of variables (such as $\langle factor \rangle$ or $\langle Balanced \rangle$); 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 $X \to \sigma$, where $X \in V$ is a variable and $\sigma \in (T \cup V)^*$ is a string of variables and terminals, which could be λ, the empty string.
   If we have several rules with left the same lefthand side, for example $X \to \sigma_1, X \to \sigma_2, \cdots, X \to \sigma_k$, they can be written as $X \to \sigma_1 \mid \sigma_2 \mid \cdots \mid \sigma_k$ for short. The meaning is that an occurrence of X in a generated string can be replaced by any one of $\sigma_1, \sigma_2, \cdots, \sigma_k$. Different occurrences of X can be replaced by distinct σ_i, of course.

The generation of a string proceeds by a series of replacements which start from the string $s_0 = S$, and which, for $1 \leq i \leq k$, obtain s_i from s_{i-1} by replacing some variable X in s_{i-1} by one of the replacements $\sigma_1, \sigma_2, \cdots, \sigma_k$ 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.
$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()(S) \Rightarrow ()((S)) \Rightarrow ()(())$.
$S \Rightarrow (S) \Rightarrow (SS) \Rightarrow ((S)S) \Rightarrow (()S) \Rightarrow (()(S)) \Rightarrow (()())$.

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 $S_{A \cup B} \to S_A \mid S_B$.

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 $S_{A \circ B} \to S_A S_B$.

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 uS_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. *$\{\lambda\}$, ϕ, $\{a\}$ are all context free languages.*

Proof. The grammar with the single rule $S \to \lambda$ generates $\{\lambda\}$, the grammar with no rules generates ϕ, and the grammar with the single rule $S \to a$ 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)

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. $A \to a$.

ii. Or two variables, e.g. $A \to BC$.

iii. Or the rule $S \to \lambda$, if λ 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.

$$S \to ASA \mid aB;\ A \to B \mid S;\ B \to b \mid \lambda$$

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 $U_a \to a$, 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 $A \to Ba$, this is replaced by $U_a \to a$, $A \to BU_a$.

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:

$$S \to ASA \mid U_aB;\ A \to B \mid S;\ B \to b \mid \lambda;\ U_a \to a$$

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 $U \to W_1W_2 \cdots W_k$, 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:

$$U \to W_1X_2;\ X_2 \to W_2X_3;\ \cdots;\ X_{k-2} \to W_{k-2}X_{k-1};\ X_{k-1} \to W_{k-1}W_k$$

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 $U \to W_1W_2 \cdots W_k$. 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 λ.

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

$$S \to AX \mid U_aB; X \to SA;\ A \to B \mid S;\ B \to b \mid \lambda;\ U_a \to a$$

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

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

$$S \to S';\ S' \to AX \mid U_aB; X \to S'A;\ A \to B \mid S';\ B \to b \mid \lambda;\ U_a \to a$$

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

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 $A \to \lambda$, we label the resulting leaf with λ. We will be focussing on subtrees in which all the leaves have λ-labels; we call such subtrees λ-*subtrees*. Now imagine pruning all the λ-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.

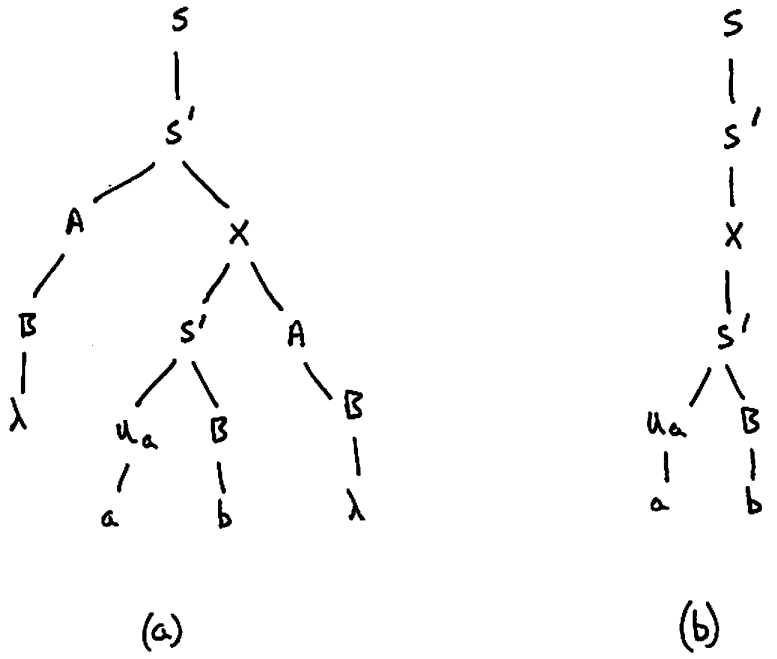

Figure 3.13: (a) A Derivation Tree for string ab. (b) The Reduced Derivation Tree.

We need to change the grammar as follows. Whenever there is a rule $A \to BC$ and B can generate λ, we need to add the rule $A \to C$ to the grammar (note that this does not allow any new strings to be generated); similarly, if there is is rule $A \to DE$ and E can generate λ, we need to add the rule $A \to D$; likewise, if there is a rule $A \to BB$ and B can generate λ, we need to add the rule $A \to B$.

Next, we remove all rules of the form $A \to \lambda$. 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 λ-subtree (that is a λ-subtree whose parent is not part of a λ-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 $A \to BC$ or the rule $A \to CB$ at node u (depending on whether v is the left or right child of u). In the reduced tree, applying the rule $A \to C$ generates w and omits v and its subtree.

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

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

For our example grammar we start with the rules

$$S \to S';\ S' \to AX \mid U_aB; X \to S'A;\ A \to B \mid S';\ B \to b \mid \lambda;\ U_a \to a$$

As $B \to \lambda$ is a rule, we obtain the reduced rules

$$S \to S';\ S' \to AX \mid U_aB \mid U_a; X \to S'A;\ A \to B \mid \lambda \mid S';\ B \to b \mid \lambda;\ U_a \to a$$

As $A \to \lambda$ is now a rule, we next obtain

$$S \to S';\ S' \to AX \mid X \mid U_aB \mid U_a;\ X \to S'A \mid S';\ A \to B \mid \lambda \mid S';\ B \to b \mid \lambda;\ U_a \to a$$

There are no new rules with λ on the right-hand side. So the procedure is now complete and this yields the new collection of rules:

$$S \to S';\ S' \to AX \mid X \mid U_aB \mid U_a; X \to S'A \mid S';\ A \to B \mid S';\ B \to b;\ U_a \to a$$

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 λ 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 $A \to B$, 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 $A \to BC$. 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 $A \to a$. We proceed in two substeps.

Substep 5.1. This substep identifies variables that are equivalent, i.e. collections $B_1, B_2, \cdots, B_l$ such that for each pair B_i and B_j, $1 \le i < j \le l$, B_i can generate B_j, and B_j can generate

B_i. We then replace all of $B_1, B_2, \cdots, B_l$ 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 $A \to B$ 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

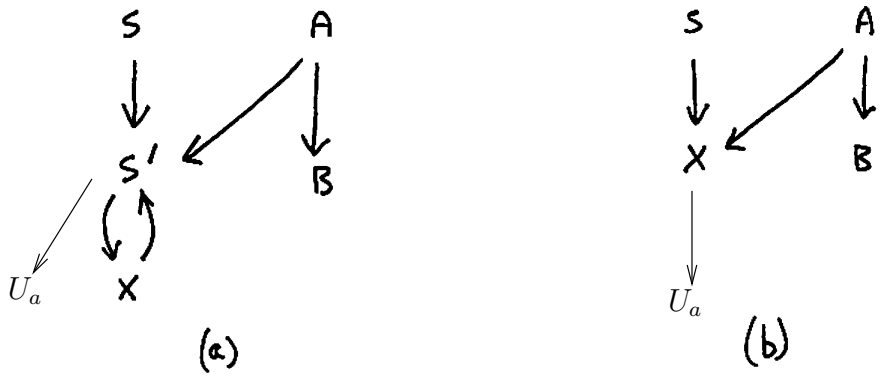

Figure 3.14: (a) Graph showing the unit rules. (b) The reduced graph.

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:

$$S \to X;\ X \to AX \mid X \mid U_aB \mid U_a;\ X \to XA \mid X;\ A \to B \mid X;\ B \to b;\ U_a \to a$$

We can remove the useless rule $X \to X$ also.

Substep 5.2. In this substep, we add rules $A \to BC$ and $A \to a$, 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 $A \to B$, we will add appropriate non-unit rules that allow the shortcutting of all uses of $A \to B$, and hence allow the rule $A \to B$ to be discarded. If there are no unit rules with B on the left-hand side it suffices to add a rule $A \to CD$ for each rule $B \to CD$, and a rule $A \to b$ for each rule $B \to b$.

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. 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 $E \to F$, for each rule $F \to CD$, we add the rule $E \to CD$, and for each rule $F \to f$, we add the rule $E \to f$; then we remove the rule $E \to F$. Any derivation which had used the rules $E \to F$ followed by $F \to CD$ or $F \to f$ can now use the rule $E \to CD$ or $E \to f$ 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 $A \to b$
Remove $A \to B$.
Next, we traverse edge (X, U_a). This changes the rules as follows:
Add $X \to a$
Remove $X \to U_a$.
Now, we traverse edge (A, X). This changes the rules as follows:
Add $A \to AX \mid XA \mid U_aB \mid a$.
Remove $A \to X$.
Finally, we traverse edge (S, X). This changes the rules as follows:
Add $S \to AX \mid XA \mid U_aB \mid a$.
Remove $S \to X$.
The net effect is that our grammar now has the rules

$$S \to AX \mid U_aB \mid XA \mid a;\ X \to AX \mid U_aB \mid XA \mid a;\ A \to b \mid AX \mid U_aB \mid XA \mid a;\ B \to b;\ U_a \to a$$

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 leat $h + 1$ 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.

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. 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.

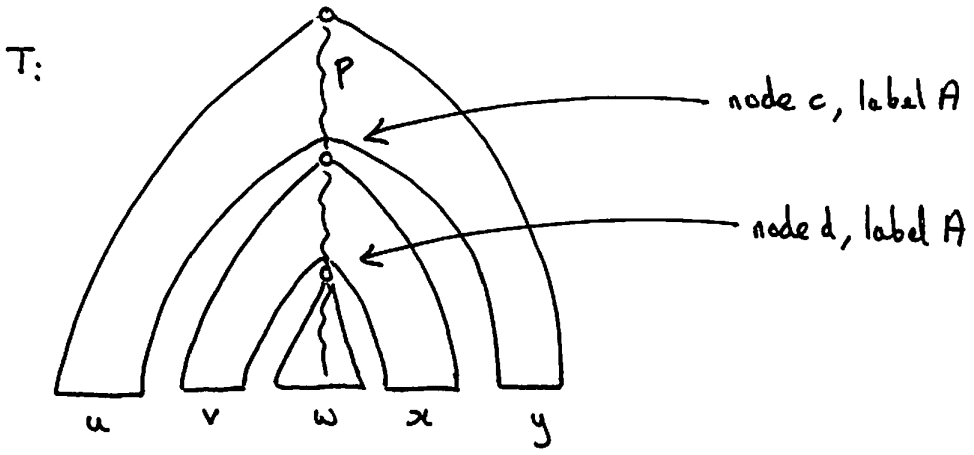

Figure 3.15: Derivation Tree for $s \in L$.

Let w be the substring derived by the subtree rooted at d. Let vwx be the substring derived by the subtree rooted at c (so v, for example, is derived by the subtrees hanging from P to its left side on the portion of P starting at c and ending at d's parent). Let $uvwxy$ be the string derived by the whole tree.

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

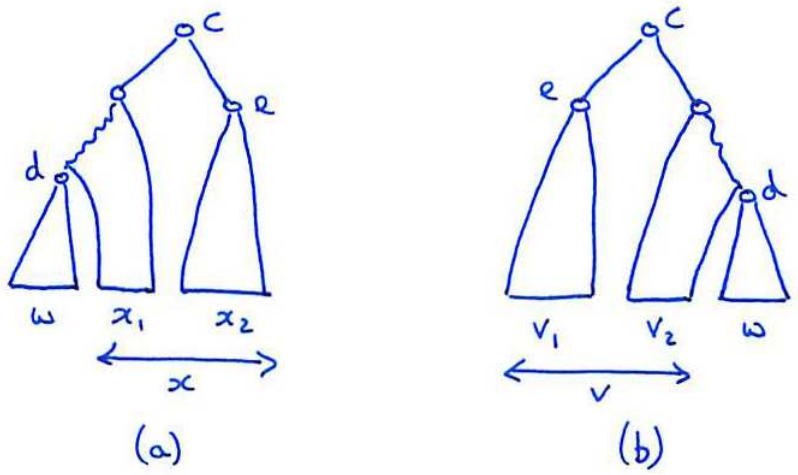

Figure 3.16: Possible Locations of e, the Off Path Child of node c.

be the string derived by the subtree rooted at e. Then $|x_2| \geq 1$. 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, $i \geq 0$.

Now let's see why $uvvwxxy \notin L$. 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, 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 $uvvwxx \notin L$.

We have shown both that $uvvwxxy \in L$ and $uvvwxxy \notin L$. 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

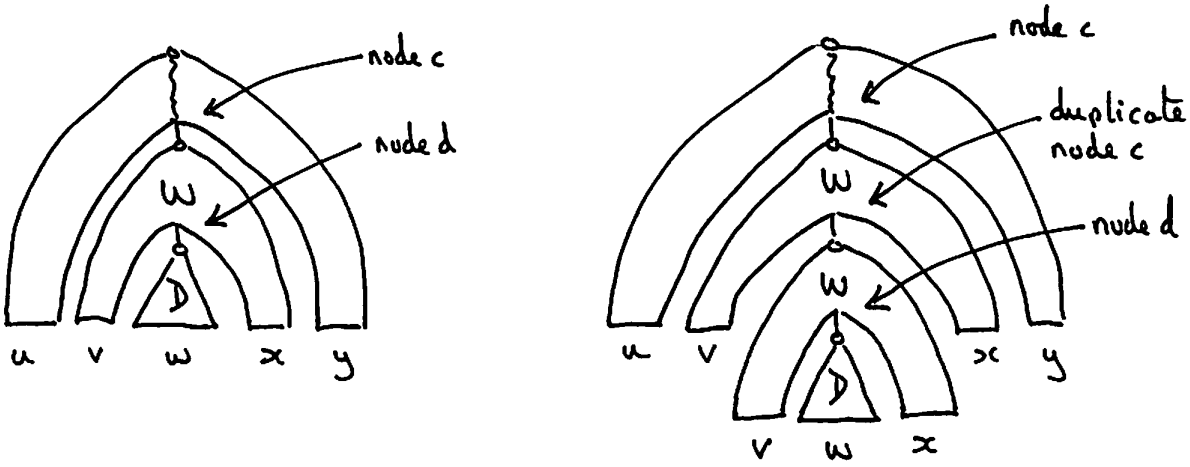

Figure 3.17: Duplicating wedge W.

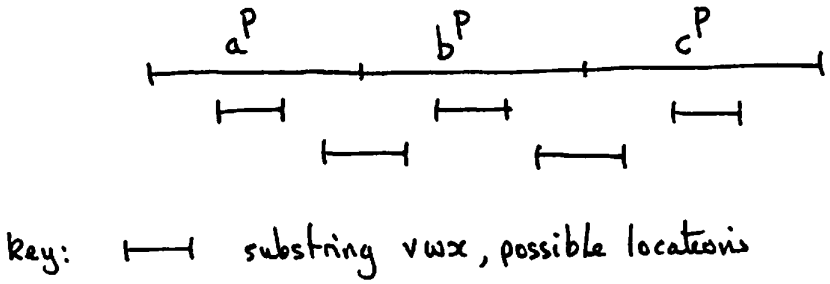

Figure 3.18: Possible Locations of vwx in $a^p b^p c^p$.

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 d. 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 $uvvwxxy$. 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, $i \geq 0$. □

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 $s = uvwxy$ 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. $K = \{a^ib^jc^k \mid i < j < k\}$. 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 $s = a^pb^{p+1}c^{p+2}$. Clearly, $s \in K$ and $|s| \geq p$, so s is pumpable.

Step 3. As s is pumpable we can write s as $s = uvwxy$ with $|vwx| \leq p$, $|vx| \geq 1$ and $uv^iwx^iy \in K$ 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 $H \cap a^*b^*c^* = L = \{a^i b^i c^i\}$. 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 $s = a^p b^p c^p$.