Chapter 6

Grammars and Pushdown Automata

In this chapter we will examine the concept of a *grammar*, which is a very powerful tool for generating strings and thereby defining languages. The languages generated by (most) grammars go well beyond what is expressible via regular expressions. We will begin by defining a general grammar, then examine several restricted forms of grammars: context-free and right linear. We will then describe a new type of automaton, known as a *pushdown automata*, that is intimately related to context-free grammars, in essentially the same manner that finite automata are related to regular expressions. We will also see that context-free languages, that is, languages generated by context-free grammars, all exhibit a fundamental pattern that is analogous to that exhibited by regular languages. This will lead to a version of the pumping lemma for context-free languages, which can be used to show that a language is *not* context-free.

6.1 Grammars

A *grammar* is a practical and powerful mechanism for *deriving* or *generating* languages; that is, a mechanism for directly constructing the strings of a language, just as regular expressions provide a mechanism to construct the strings of a regular language. We begin with a formal set-theoretic definition of a grammar.

Definition 16 A grammar G is a 4-tuple $(\Sigma, \Gamma, \gamma, \Pi)$, where

1. Σ is a finite non-empty set called the *terminal alphabet*
2. Γ is a finite non-empty set called the the *non-terminal alphabet*
3. γ is an element of Γ called the *start symbol*

4. Π is a finite subset of $(\Sigma \cup \Gamma)^* \circ \Gamma \circ (\Sigma \cup \Gamma)^* \times (\Sigma \cup \Gamma)^*$ known as the *productions*[1]

where the only additional constraint on these sets is that $\Sigma \cap \Gamma = \emptyset$.

The elements of Σ are called *terminals*, and the elements of Γ are called *non-terminals*. Formally, the elements of Π are ordered pairs of strings (α, β), where both α and β may consist of terminals and non-terminals intermixed in any order; the only restriction is that α must contain at least one non-terminal. This restriction on α makes the formal definition above somewhat cumbersome, requiring the concatenation of three sets of strings, the middle one being Γ. Note that β is allowed to be ε, the empty string, as it is simply an element of $(\Sigma \cup \Gamma)^*$. It is customary to denote a production $(\alpha, \beta) \in \Pi$ as

$$\alpha \longrightarrow \beta. \tag{6.1}$$

Thus, the productions of the grammar

$$G = \{\{a, b\}, \{A, B\}, A, \{(A, aA), (aA, AaB), (B, abb)\}\} \tag{6.2}$$

would be written as

$$\begin{aligned} A &\longrightarrow aAb \\ aA &\longrightarrow AaB \\ B &\longrightarrow abb. \end{aligned}$$

We say that a string v can be *derived in one step* from a string u, according to the grammar G, if and only if there exists a production $\alpha \longrightarrow \beta$ in Π such that α occurs as a substring one or more times in u, and replacing *some* occurrence of α in u by β results in the string v. In this case, we then write $u \Longrightarrow v$, or $u \Longrightarrow_G v$ when we wish to emphasize the grammar we are utilizing. For example, it follows that

$$abxax \Longrightarrow abzzzax \tag{6.3}$$

if the production $x \longrightarrow zzz$ is in Π. Observe that only one x in the string $abxax$ is replaced by zzz. The above derivation would also hold if either the production $bxa \longrightarrow bzzza$, or the production $abxax \longrightarrow abzzzax$ were in Π. The symbol $\Longrightarrow$ technically denotes a binary relation on Σ^*, which is analogous to the $\vdash$ relation on DFA configurations. Moreover, as with the yields relation of DFAs, we extend the "derives in one step" relation to its reflexive transitive closure, which means "derives in zero or more steps," as follows:

1. if $u \Longrightarrow v$, then $u \overset{*}{\Longrightarrow} v$ *(Extension).*
2. if $u \overset{*}{\Longrightarrow} u$ for all $u \in \Sigma^*$ *(Reflexivity).*
3. if $u \Longrightarrow v$ and $v \overset{*}{\Longrightarrow} w$, then $u \overset{*}{\Longrightarrow} w$ *(Transitivity).*

[1] Productions are also frequently referred to as *rewrite rules*.

Definition 17 The language L *generated* by a grammar $G = (\Sigma, \Gamma, \gamma, \Pi)$ is the set of all strings in Σ^* that can be generated by G starting from the symbol $\gamma \in \Gamma$. That is,

$$\mathcal{L}(G) \;=\; \left\{ w \in \Sigma^* \mid \gamma \stackrel{*}{\Longrightarrow} w \right\}.$$

Note carefully that the language generated by a grammar consists of strings of *terminals* only; for any $w \in \mathcal{L}(G)$, $w \in \Sigma^*$. Thus, non-terminal symbols are used only as intermediate symbols in the derivation of the final strings; they never appear in the final strings.

As an example of a grammar, let $G_1 = (\Sigma, \Gamma, \gamma, \Pi)$, where $\Sigma = \{a, b\}$, $\Gamma = \{S\}$, $\gamma = S$, and $\Pi = \{(S, aSa), (S, bSb), (S, a), (S, b), (S, \varepsilon)\}$. Thus, using the more intuitive notation for the productions, the grammar G_1 can be expressed as

$$\begin{array}{lcl} S & \longrightarrow & aSa \\ S & \longrightarrow & bSb \\ S & \longrightarrow & a \\ S & \longrightarrow & b \\ S & \longrightarrow & \varepsilon \end{array}$$

where S is understood to be the start symbol. Thus, $ababa \in \mathcal{L}(G_1)$ because $S \Longrightarrow aSa \Longrightarrow abSba \Longrightarrow ababa$. In fact, G_1 generates all strings in the language $\left\{ w \in \Sigma^* \mid w = w^R \right\}$, which is the set of all *palindromes* over Σ; a palindrome is a word whose spelling is the same forwards and backwards, like "radar." The above list of productions can be written more compactly as

$$\begin{array}{lcl} S & \longrightarrow & aSa \mid bSb \\ S & \longrightarrow & a \mid b \mid \varepsilon \end{array}$$

where the vertical bar represents a choice of right-hand sides. In fact, in this example we can fold all five productions into a single line, since there is only one non-terminal symbol.

A *context-free* grammar is a grammar in which the left-hand side of each production is restricted to consist of a single non-terminal. More formally, the grammar $G = (\Sigma, \Gamma, \gamma, \Pi)$ is context-free if and only if

$$\Pi \subset \Gamma \times (\Sigma \cup \Gamma)^*,$$

in addition to meeting all the other constraints for general grammars. In particular, Π must be finite, $\gamma \in \Gamma$, and $\Sigma \cap \Gamma = \emptyset$. An example of a context-free grammar is

$$\begin{array}{lcl} S & \longrightarrow & AB \\ A & \longrightarrow & aAb \mid \varepsilon \\ B & \longrightarrow & Bc \mid \varepsilon \end{array}$$

which generates the language $\left\{ a^n b^n c^k \mid n \geq 0, k \geq 0 \right\}$. Another example of a context-free grammar is the one for palindromes given previously. The term "context-free" means that the string substitutions are carried out independent of the context in which they are found. In contrast, a general grammar can include productions such as $aAb \longrightarrow aBBb$, which replaces A with BB, but *only when the A has an a on the left and a b on the right*; that is, only when the A is in the context aAb.

We call a language context-free if there exists some context-free grammar that generates it. We shall see that the context-free languages are exactly the languages that are accepted by (nondeterministic) pushdown automata. Context-free languages are extremely important, as nearly all programming languages are context-free; that is, the *syntax* of most programming languages can be completely described by a context-free grammar.

A *right linear* grammar is a context-free grammar in which the right-hand side of each production is restricted to consist of a string of terminals followed by zero or one non-terminal. More formally, the grammar $G = (\Sigma, \Gamma, \gamma, \Pi)$ is right linear if and only if

$$\Pi \subset \Gamma \times (\Sigma^* \circ (\Gamma \cup \{\varepsilon\}))$$

in addition to meeting all the other constraints for context-free grammars. In particular, the left-hand side of every production consists of a single non-terminal. An example of a right-linear grammar is

$$\begin{aligned} S &\longrightarrow aS \mid \varepsilon \\ S &\longrightarrow A \\ A &\longrightarrow bA \mid \varepsilon \end{aligned}$$

which generates the language $\left\{a^n b^k \mid n \geq 0, k \geq 0\right\}$. We call a language right-linear if there exists some right-linear grammar that generates it. We have already encountered the class of right-linear languages, but by a different name, as the next theorem shows.

Theorem 23 *A language is right linear if and only if it is regular.*

Proof: (To be supplied) □

6.2 The Pumping Lemma for Context-Free Languages

Just as with regular languages, context-free languages possess symmetries that result from limitations of the mechanisms that generate them. In the case of regular languages all sufficiently long strings would force the machine back into a state it had already visited; this leads to the first pumping property, a distinguishing feature of all regular languages. While context-free languages are somewhat more complex than regular languages, they also possess a similar property, which we will call the *second pumping property.*

Definition 18 Let k be a natural number and let L be a language. Then we shall say that $w \in L$ has the *second pumping property of length* k (with respect to the language L) if and only if there exist strings u, v, x, y, and z (not necessarily in the language L) such that $w = uvxyz$, $|vxy| \leq k$, $|vy| \geq 1$, and $uv^n xy^n z \in L$ for all $n \geq 0$.

The second pumping property is somewhat more complicated than the first pumping property, but is directly analogous. Here, instead of partitioning a string into three pieces, we partition it into five

pieces. We constrain the middle three pieces to be no longer than k, and we do not allow *both* the second and fourth pieces to be empty (although one of them may be). With this definition, we may now state the Second Pumping Lemma, which is identical in form to the First Pumping Lemma.

Lemma 5 (Second Pumping Lemma) *Let L be a context-free language. Then for some $k \in \mathbb{N}$, every $w \in L$ with $|w| \geq k$ has the second pumping property of length k.*

Proof: See, for example, Kozen [15, page 148]. □

Example 1 The language $L_1 = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ is not Context-Free.

Let $k \in \mathbb{N}$ be given. Let $w = a^m b^m c^m$ where $m = k + 1$. Suppose that $w = uvxyz$ with $|vxy| \leq k$, and $|vy| \geq 1$. We will consider several possible cases. **Case I:** If either v or y contains two distinct symbols, then the string uv^2xy^2z will contain an a that is preceded by a b, or a b that is preceded by a c; neither of these can occur in any of the strings of L_1. **Case II:** If both v and y consist of a single (possibly repeated) letter, then the string uv^2xy^2z will be $a^p b^q c^r$, where p, q, and r are not all equal; this is another pattern that does not occur in the strings of L_1. Therefore, in either case, the string uv^2xy^2z fails to be in L_1, so w does not have the second pumping property of length k. Since k was arbitrary, we conclude that L_1 is not context-free. □

Example 2 The language $L_2 = \{a^n \mid n \in \mathbb{N} \wedge n \text{ is prime}\}$ is not context-free.

Proving this is almost identical to the proof that L_2 is not regular. Let $k \in \mathbb{N}$ be given, and let $w = a^p$ where p is prime and $p \geq k$. If $w = uvxyz$ with $|vy| \geq 1$, then

$$\begin{aligned} |uv^n xy^n z| &= |xyzuv| + (n-1)\,|vy| \\ &= p + (n-1)m, \end{aligned}$$

where $m = |vy|$. We must now show that for some $n \in \mathbb{N}$, $p + (n-1)m$ is not prime, from which it follows that $uv^n xy^n z \notin L_2$. Letting $n = p + 1$, the proof follows exactly as it did in showing that $a^n b^n$ is non-regular. We have

$$|uv^n xy^n z| \;=\; p + pm \;=\; p(m+1).$$

But since both p and $m+1$ are greater than 1, $p(m+1)$ is a composite number. Therefore, for any $k \in \mathbb{N}$ and any partition $w = uvxyz$ where $w \in L_2$, $|w| \geq 0$, and $|vy| > 0$, we can find an n such that $uv^n xy^n z \notin L_2$. Therefore, L_2 is not context-free. □

6.3 Closure Properties of Context-Free Languages

Previously we showed that regular languages are closed under all the common set operations. We shall now show that context-free languages are also closed under certain set operations, but not all of them.

Theorem 24 *Context-Free languages are closed under union, concatenation, and Kleene star.*

Proof: Closure under union and concatenation can be easily demonstrated by combining two context-free grammars to obtain a new context-free grammar. Let L_1 and L_2 be arbitrary context-free languages over the same alphabet Σ. Then there exist two context-free grammars,

$$\begin{aligned} G_1 &= (\ \Sigma,\ \Gamma_1,\ \gamma_1,\ \Pi_1\ ) \\ G_2 &= (\ \Sigma,\ \Gamma_2,\ \gamma_2,\ \Pi_2\ ) \end{aligned}$$

that generate L_1 and L_2, respectively. Without loss of generality, let us assume that $\Gamma_1 \cap \Gamma_2 = \emptyset$, since we can always relabel the non-terminals of a grammar without changing the language it generates. Now, let S denote some symbol that is not in Γ_1 or Γ_2 and define the three following grammars:

$$\begin{aligned} G_3 &= (\ \Sigma,\ \Gamma_1 \cup \Gamma_2 \cup \{S\}\ ,\ S,\ \Pi_1 \cup \Pi_2 \cup \{S \longrightarrow \gamma_1, S \longrightarrow \gamma_2\}\ ) \\ G_4 &= (\ \Sigma,\ \Gamma_1 \cup \Gamma_2 \cup \{S\}\ ,\ S,\ \Pi_1 \cup \Pi_2 \cup \{S \longrightarrow \gamma_1\gamma_2\}\ ) \\ G_5 &= (\ \Sigma,\ \Gamma_1 \cup \{S\}\ ,\ S,\ \Pi_1 \cup \{S \longrightarrow S\gamma_1, S \longrightarrow \varepsilon\}\ ) \end{aligned}$$

It is easy to see that G_3, G_4, and G_5 are all context-free, as the only new productions are of the correct form; i.e. they all have exactly one non-terminal on the left. Furthermore, it is also easy to see that

$$\begin{aligned} \mathcal{L}(G_3) &= L_1 \cup L_2 \\ \mathcal{L}(G_4) &= L_1 \circ L_2 \\ \mathcal{L}(G_5) &= (L_1)^*, \end{aligned}$$

consequently, these languages are context-free, since there is a context-free grammar that generates them. It follows that context-free languages are closed under all three of these operations. □

However, unlike regular languages, context-free languages are not closed under all of the typical set operations. In particular, both intersection and complementation of context-free languages can result in languages that are not-context free.

Theorem 25 *Context-Free languages are not closed under intersection or complementation.*

Proof: That context-free languages are not closed under intersection follows easily by observing that both $L_1 = \left\{a^n b^n c^k \mid n \geq 0, k \geq 0\right\}$ and $L_2 = \left\{a^k b^n c^n \mid n \geq 0, k \geq 0\right\}$ are context free. However, the language $L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 0\}$ is not context free. That context-free languages are not closed under complementation now follows from the fact that

$$L_1 \cap L_2 = \overline{\left(\overline{L_1} \cup \overline{L_2}\right)}.$$

Thus, if context-free languages were closed under complementation, they would also be closed under intersection, which we have just shown is not the case. □

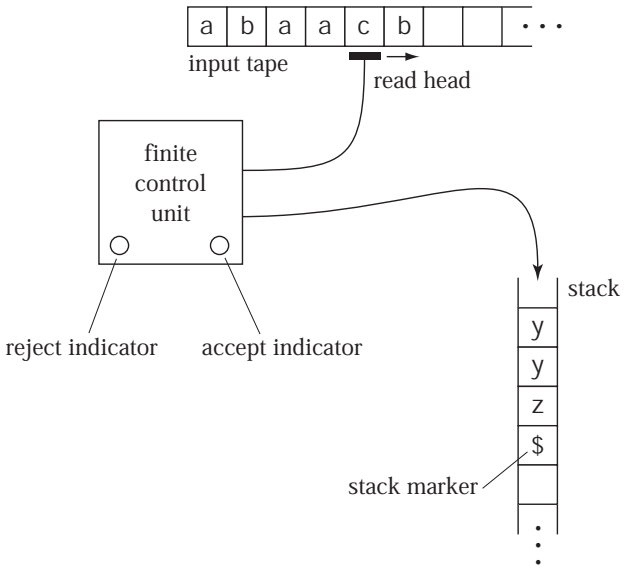

Figure 6.1: *Conceptually, a pushdown automaton (PDA) is very similar to a finite automaton; the difference is that it is also equipped with an infinite capacity stack. At each transition the machine uses the current state, the symbol under the read head, and the symbol on top of the stack to determine its next action. Each transition (optionally) moves the read head one cell to the right, pops the top symbol from the stack, and pushes zero or more symbols back onto the stack.*

6.4 Pushdown Automata

As with DFAs and NFAs, the machine "reads" the symbols of a string, which we imagine to be printed on a "tape" that extends from left to right, beginning in its "initial state." The difference is that the Pushdown Automaton also "pops" symbols from its stack at each step, and these symbols influence the action of the machine. Moreover, the machine may "push" any finite number of symbols onto the stack at each step. To summarize, when the machines starts

1. The current state is q_0.
2. The read head is positioned over the left-most (first) symbol of the string on the tape.
3. The symbol γ is the only symbol on the stack.

Each transition of the machine is determined by

1. The current state.
2. The symbol currently under the read head (this is optional).
3. The symbol currently on top of the stack.

At each transition, the following changes occur:

1. The machine is put into a new state (possibly the same as before).
2. The read head either stays fixed, or moves exactly one cell to the right.
3. The top symbol is removed (i.e. "popped") from the stack.
4. A string of zero or more symbols is added (i.e. "pushed") onto the stack.

This process continues until one of two events occurs:

1. The last symbol on the tape has been read, and there are no more null transitions to follow.
2. An undefined transition is encountered.

After reading the last symbol on the tape, the read head is positioned over the "blank" symbol, just past the end of the string. When an undefined transition is encountered, the string is rejected, by definition, which is the same policy used in NFAs.

Definition 19 A Pushdown Automaton (PDA) is formally specified by the 6-tuple $(\Sigma, \Gamma, \gamma, Q, q_0, A, \Delta)$, where most of the elements retain the meanings they were given in the context of finite automata:

1. Σ is a finite non-empty set called the *input alphabet*,
2. Γ is a finite non-empty set called the *stack alphabet*,
3. γ is the *stack marker*,
4. Q is a finite non-empty set of *states*,
5. q_0 is the state of the machine before reading each string, or the *initial state*,
6. $A \subseteq Q$ is a special set of states called the *accept* states, and
7. Δ is the *transition relation*, which is a finite subset of $(Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma) \times (Q \times \Gamma^*)$, where ε in this context denotes the null symbol.

The ordered pairs $((p, a, s), (q, w))$ in Δ determine the new state to move to and the symbols to push onto the stack as a function of the current state, the current symbol being read, and the symbol on the top of the stack[2]. The tape symbol may also be substituted by "ε," which indicates that no symbol is read and the read head remains fixed.

Let M denote the machine $(\Sigma, \gamma, \Gamma, Q, q_0, A, \Delta)$, and let $w \in \Sigma^*$ be some string. We shall use the notation $\mathcal{S}_{\text{M}}(w)$ to denote the *set of states* reachable by machine M after reading string w, which is

[2]Here we have retained the structure of the Cartesian product rather than denoting the elements of Δ as 5-tuples (i.e. as in NFAs, where we denoted the elements of Δ by 3-tuples). This is nothing more than a stylistic choice.

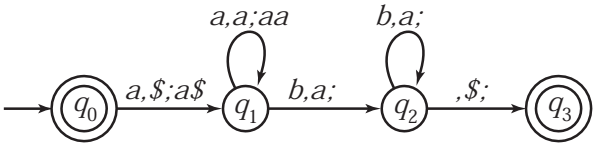

Figure 6.2: *This is a deterministic PDA that accepts the language* $\{a^n b^n \mid n \geq 0\}$. *The stack marker is "*\$*" and* ε *is the empty string. Note that state* q_3 *will be entered for all strings* $a^n b^k$ *where* $k \geq n$*; however, if* $k > n$*, then the machine will reject while in state* q_3*, since there are no transitions defined for that case.*

consistent with our previous usage of this notation. As before, if $\mathcal{S}_{\text{M}}(w) \cap A \neq \emptyset$, we say that the machine *accepts* or *recognizes* the string w. The set of all recognized strings constitutes a language associated with M. More precisely, we define

$$\mathcal{L}(M) \stackrel{\text{def}}{=} \{w \in \Sigma^* \mid \mathcal{S}_{\text{M}}(w) \cap A \neq \emptyset\} \tag{6.4}$$

to be the language *accepted* (or *recognized*) by the machine M. In keeping with previous terminology, we will call a language *PDA-acceptable* if and only if there exists some PDA that accepts it.

6.4.1 State Diagrams

As with NFAs and DFAs, PDAs (both deterministic and nondeterministic) have an obvious representation in terms of a state diagram. We retain all of the same conventions as with DFAs regarding the representation of states, transitions, and designating the start state and accept states. The only difference is how we label the arrows of the diagram. In the state diagram of a PDA, the transition $((p, a, s), (q, w))$ is denoted by an arrow from state p to state q bearing the label $a, s; w$. This transition means that when the machine is in state p, the read head is over the symbol a, and s is on top of the stack, then the machine is put into state q, the read head moves to the right by one cell, the s is popped off the stack, and the string w is pushed onto the stack so that its first symbol appears on top of the stack (i.e. the last symbol of the string is pushed first). There is no explicit representation of the stack; the operation of the stack is implicit in the definition of a PDA, and the sequence of transitions followed completely determines the stack contents at all times.

We also allow transitions of the form $((p, \varepsilon, s), (q, w))$, in which no symbol is read from the tape, and the read head does not move. Note that exactly one symbol is *always* popped from the stack.

Figure 6.2 depicts a state diagram for a PDA that accepts the language $\{a^n b^n \mid n \geq 0\}$. Here the tape and stack alphabets both contain the letters a and b. The machine pushes a's onto the stack to "remember" how many it encountered. If it then reads a string of b's, it can compare the number of b's to the number of a's by popping an a from the stack for each b that is read. When exactly as many b's have been read as a's, the stack marker will be on the top of the stack and the machine can enter state q_3. If there are still more symbols on the tape, the machine will reject, since there are no transitions leaving state q_3. Observe that the stack alphabet could be replaced with any two distinct symbols (plus the stack marker) without changing the language that is accepted.

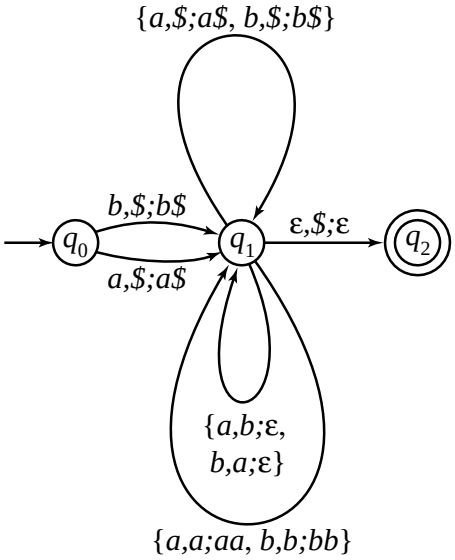

Figure 6.3: *This is a nondeterministic PDA that accepts the language* $\{w \in \{a,b\}^* \mid |w|_a = |w|_b\}$. *The machine is nondeterministic because of the* $\varepsilon, \$; \varepsilon$ *transition out of state* q_1*, which could always be chosen instead of* $b, \$; b\$$ *or* $a, \$; a\$$. *Several of the arrows are marked with sets of labels as a notational convenience. This language can also be recognized by a deterministic PDA.*

6.4.2 Configurations and the Yields Relation

6.4.3 Deterministic Pushdown Automata

A PDA is deterministic if two conditions hold. First, there cannot be multiple transitions that begin with the same 3-tuple. That is, there cannot be two distinct elements $((p,a,s),(q_1,w_1))$ and $((p,a,s),(q_2,w_2))$ in the transition relation Δ. This clearly leads to a nondeterministic choice when the machine is in state p, reading a, with s on top of the stack. However, there is also another condition that leads to nondeterminism: if there is a transition of the form $((p,a,s),(q_1,w_1))$ and also of the form $((p,\varepsilon,s),(q_2,w_2))$ in the relation Δ. In this case, the leading 3-tuples are in fact different, but still lead to a nondeterministic choice, as the symbol "a" may or may not be read from the tape. Thus, in a deterministic PDA, where there is an epsilon transition leading out of state q, there can be no other transitions out of q that pop the same symbol. The machine shown in Figure 6.2 is deterministic, and the machine shown in Figure 6.3 is nondeterministic since any transition of the form $a, \$; a\$$ or $b, \$; b\$$ while in state q_1 could also trigger the transition $\varepsilon, \$; \varepsilon$; thus, it is a nondeterministic choice.

Unless otherwise stated, when we will assume that a PDA is nondeterministic. If the distinction is important, we will refer to a deterministic PDA as a DPDA, and a nondeterministic PDA as an NPDA. Note that all PDAs, deterministic or not, have their transitions encoded as a Δ relation rather than a function; this is simply a matter of convention, since most PDAs are nondeterministic anyway.

6.5 Context-Free Languages and PDA's

Theorem 26 *A language is context-free if and only if it is accepted by some (nondeterministic) pushdown automaton.*

Proof: (To be supplied) □

6.6 An Intersection Theorem for Context-Free Languages

Although we have shown that context-free languages are not closed under intersection, it is possible to impose a constraint on one of the languages so that the intersection does remain context-free. The following theorem can be useful in proving that a given language is context-free.

Theorem 27 *The intersection of a context-free language and a regular language is context-free.*

Proof: Let M_1 and M_2 be a PDA and a DFA, respectively, over the same alphabet. In terms of tuples, we shall denote them as

$$\begin{aligned} M_1 &= (\Sigma, \Gamma, Q_1, q_1, \gamma, A_1, \Delta_1) \\ M_2 &= (\Sigma, Q_2, q_2, A_2, \delta_2) \end{aligned}$$

We will show how to construct another machine M_3, which is also a PDA, that accepts $\mathcal{L}(M_1) \cap \mathcal{L}(M_2)$. First, denote the components of M_3 by

$$M_3 = (\Sigma, \Gamma, Q_3, q_3, \gamma, A_3, \Delta_3).$$

The strategy is to create a PDA that behaves exactly the same way as M_1 with respect to reading the tape and managing the stack, but simultaneously keeps track of how the DFA M_2 would behave given the same input. Thus, each state of the new machine will record two things: the state of the original PDA, M_1, and the state of the NFA, M_2, which we imagine to be running in lock-step with M_1 (i.e. reading the same symbols from the tape at the same time). To do this, we introduce many more states into M_3; in particular, the new machine will have states corresponding to the Cartesian product of Q_1 and Q_2; that is, $Q_3 = Q_1 \times Q_2$.

Given this strategy, it is relatively straightforward to define the transition relation Δ_3 of M_3. We need only two rules:

1. For each $((p, a, s), (q, w)) \in \Delta_1$ and each $r \in Q_2$,
   add $((p', a, s), (q', w))$ to Δ_3, where $p' = (p, r)$, and $q' = (q, \delta_2(r, a))$.
2. For each $((p, \varepsilon, s), (q, w)) \in \Delta_1$ and each $r \in Q_2$,
   add $((p', \varepsilon, s), (q', w))$ to Δ_3, where $p' = (p, r)$, and $q' = (q, r)$.

To complete the definition of M_3, we let $q_3 = (q_1, q_2)$, and $A_3 = A_1 \times A_2$. Now, if we look at the first coordinate of the states visited by M_3 while processing a string, they exactly match what M_1 would do. Similarly, if we look at the second coordinate of the states visited while processing a string, they match what M_2 would do, except for possibly the addition of one or more null transitions. Finally, we observe that a string would be accepted by *both* M_1 and M_2 if and only if M_3 is left in a state that is in $A_1 \times A_2$, which consists of both an accept state of M_1 and an accept state of M_2.

□

6.7 Exercises

1. Given the set of terminals $\Sigma = \{a, b\}$, construct a context-free grammar for each of the following languages. You need only list the productions, provided that you adhere to the convention that upper-case letters are non-terminals, lower-case letters are terminals, and S is the start symbol.

   (a) $L_1 = b^*ab^*ab^*$. That is, all strings in Σ^* with exactly two a's.

   (b) $L_2 = \left\{a^n b^k u b^k a^n \mid n \geq 1, k \geq 1, u \in \Sigma^*\right\}$

   (c) $L_3 = \left\{a^m b^k \mid m \geq 2k\right\}$

2. For each of the following non-regular languages over the alphabet $\Sigma = \{a, b, c\}$, define a PDA that accepts them. You need only show the state diagram for each machine.

   (a) $\left\{a^n b^k c^{n+k} \mid n \geq 0, k \geq 0\right\}$

   (b) $\left\{a^n b^{n+k} c^{2k} \mid n \geq 0, k \geq 0\right\}$

   (c) $\left\{a^n b^k \mid n \neq k\right\}$

3. Let L be the language $\left\{a^n b^{2n} \mid n \geq 0\right\}$.

   (a) Give a context-free grammar that generates L.

   (b) Give the state diagram of a *deterministic* PDA that accepts the language L.

4. Show that for any regular language, there exists a *deterministic* PDA that accepts it.

5. Use the pumping lemma for context-free languages to show that the following languages are not context-free.

   (a) $\left\{a^n b^n a^{2n} \mid n \geq 0\right\}$.

   (b) $\{a^n \mid n$ is a power of two$\}$.

6. Demonstrate that a PDA with a *finite* stack is equivalent to an NFA. That is, for any given PDA and constant k, the collection of all strings that are accepted by the PDA without ever exceeding k symbols on the stack (including the stack marker) defines a new language that is a subset of the original language (that is, we now reject any string that causes the finite stack to "overflow" at any point). Show that this new language is regular, for any k.

7. Show that the class of context-free languages over a terminal alphabet Σ with exactly one symbol is regular.

8. Consider a modified PDA that has two stacks. At each transition it pops both stacks and pushes (possibly different) strings back onto both stacks. The new state is determined by the current state, the symbol under the read head, and the symbols at the top of both stacks. Diagrammatically, a transition would be labeled like this: $a, b, c; u, v$, where a is on the tape, b is on stack 1, c is on stack 2, and u and v get pushed onto stacks 1 and 2, respectively. In all other respects this modified PDA behaves like a 1-stack PDA. Discuss the languages accepted by such a machine. Can it accept languages that are not context-free? Justify your answer.

Bibliography

[1] Wilhelm Ackermann. On Hilbert's construction of the real numbers. In J. van Heijenoort, editor, *From Frege to Gödel. A Source Book in Mathematical Logic, 1879–1931*, pages 493–507. Harvard University Press, Cambridge, Massachusetts, 1967.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Massachusetts, 1974.

[3] Rudolf Carnap. *Introduction to Symbolic Logic and its Applications*. Dover Publications, New York, 1958.

[4] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Annual ACM Symposium on the Theory of Computing*, volume 3, pages 151–158, Ohio, May 1971.

[5] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw-Hill, New York, 1990.

[6] Martin D. Davis, Ron Sigal, and Elaine J. Weyuker. *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*. Academic Press, New York, second edition, 1994.

[7] Michael J. Fischer and Michael O. Rabin. Super-exponential complexity of Presburger arithmetic. In *Proceedings of the SIAM-AMS Symposium in Applied Mathematics*, volume 7, pages 27–41, 1974.

[8] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeeman and Company, San Francisco, 1979.

[9] Paul R. Halmos. *Naive Set Theory*. Springer-Verlag, New York, 1998.

[10] Fred Hennie. *Introduction to Computability*. Addison-Wesley, Reading, Massachusetts, 1977.

[11] John M. Howie. *Automata and Languages*. Oxford University Press, New York, 1991.

[12] Richard M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972.

[13] Stephen C. Kleene. Origins of recursive function theory. In *20th Annual Symposium on the Foundations of Computer Science*, pages 371–382, San Juan, Puerto Rico, October 1979.

[14] Dexter C. Kozen. *The Design and Analysis of Algorithms*. Springer-Verlag, New York, 1992.

[15] Dexter C. Kozen. *Automata and Computability*. Springer-Verlag, New York, 1997.

[16] E. V. Krishnamurthy. *Introductory Theory of Computer Science*. Springer-Verlag, New York, 1983.

[17] Harry R. Lewis and Christos H. Papadimitiou. *Elements of the Theory of Computation*. Prentice-Hall, Englewood Cliffs, New Jersey, second edition, 1998.

[18] Peter Linz. *An Introduction to Formal Languages and Automata*. Jones and Bartlett Publishers, Boston, second edition, 1997.

[19] Michael Machtey and Paul Young. *An Introduction to the General Theory of Algorithms*. Theory of Computation Series. North Holland, New York, 1978.

[20] Anil Nerode and Richard A. Shore. *Logic for Applications*. Graduate Texts in Computer Science. Springer-Verlag, New York, second edition, 1997.

[21] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, Boston, 1997.

[22] L. J. Stockmeyer. Planar 3-colorability is NP-complete. *SIGACT News*, 5(3):19–25, 1973.

[23] Thomas A. Sudkamp. *Languages and Machines: An Introduction to the Theory of Computer Science*. Addison-Wesley, Reading, Massachusetts, second edition, 1997.

[24] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, November 1936. (Also see correction, 43:544–546, 1937).

[25] Andrew Wiles. Modular elliptic curves and Fermat's last theorem. *Annals of Mathematics*, 141(3):443–551, May 1995.