1

Review of formal languages and automata theory

In this chapter we review material from a first course in the theory of computing. Much of this material should be familiar to you, but if not, you may want to read a more leisurely treatment contained in one of the texts suggested in the notes (Section 1.12).

1.1 Sets

A set is a collection of elements chosen from some domain. If S is a finite set, we use the notation $|S|$ to denote the number of elements or *cardinality* of the set. The empty set is denoted by $\emptyset$. By $A \cup B$ (respectively $A \cap B$, $A - B$) we mean the union of the two sets A and B (respectively intersection and set difference). The notation $\overline{A}$ means the complement of the set A with respect to some assumed universal set U; that is, $\overline{A} = \{x \in U \ : \ x \notin A\}$. Finally, 2^A denotes the *power set*, or set of all subsets, of A.

Some special sets that we talk about include $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$, the natural numbers, and $\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$, the integers.

1.2 Symbols, strings, and languages

One of the fundamental mathematical objects we study in this book is the *string*. In the literature, a string is sometimes called a *word* or *sentence*. A string is made up of *symbols* (or *letters*). (We treat the notion of symbol as primitive and do not define it further.) A nonempty set of symbols is called an *alphabet* and is often denoted by Σ; in this book, Σ will almost always be finite. An alphabet is called *unary* if it consists of a single symbol. We typically denote elements of Σ by using the lowercase italic letters a, b, c, d.

A *string* is a finite or infinite list of symbols chosen from Σ. The symbols themselves are usually written using the `typewriter` font. If unspecified, a string is assumed to be finite. We typically use the lowercase italic letters s, t, u, v, w, x, y, z to represent finite strings. We denote the *empty string* by ϵ. The set of all finite strings made up of letters chosen from Σ is denoted by Σ^*. For example, if $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, then $\Sigma^* = \{\epsilon, \mathtt{a}, \mathtt{b}, \mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}, \mathtt{aaa}, \ldots\}$. Note that Σ^* does not contain infinite strings. By Σ^+ for an alphabet Σ, we understand $\Sigma^* - \{\epsilon\}$, the set of all nonempty strings over Σ.

If w is a finite string, then its *length* (the number of symbols it contains) is denoted by $|w|$. (There should be no confusion with the same notation used for set cardinality.) For example, if $w = \mathtt{five}$, then $|w| = 4$. Note that $|\epsilon| = 0$. We can also count the number of occurrences of a particular letter in a string. If $a \in \Sigma$ and $w \in \Sigma^*$, then $|w|_a$ denotes the number of occurrences of a in w. Thus, for example, if $w = \mathtt{abbab}$, then $|w|_{\mathtt{a}} = 2$ and $|w|_{\mathtt{b}} = 3$.

We say a string y is a *subword* of a string w if there exist strings x, z such that $w = xyz$. We say x is a *prefix* of w if there exists y such that $w = xy$. The prefix is *proper* if $y \neq \epsilon$ and *nontrivial* if $x \neq \epsilon$. For example, if $w = \mathtt{antsy}$, then the set of prefixes of w is $\{\epsilon, \mathtt{a}, \mathtt{an}, \mathtt{ant}, \mathtt{ants}, \mathtt{antsy}\}$ (see Exercise 4). The set of proper prefixes of w is $\{\epsilon, \mathtt{a}, \mathtt{an}, \mathtt{ant}, \mathtt{ants}\}$, and the set of nontrivial prefixes of w is $\{\mathtt{a}, \mathtt{an}, \mathtt{ant}, \mathtt{ants}, \mathtt{antsy}\}$.

Similarly, we say that z is a *suffix* of w if there exists y such that $w = yz$. The suffix is *proper* if $y \neq \epsilon$ and *nontrivial* if $z \neq \epsilon$.

We say that x is a *subsequence* of y if we can obtain x by striking out 0 or more letters from y. For example, `gem` is a subsequence of `enlightenment`.

If $w = a_1 a_2 \cdots a_n$, then for $1 \leq i \leq n$, we define $w[i] = a_i$. If $1 \leq i \leq n$ and $i - 1 \leq j \leq n$, we define $w[i..j] = a_i a_{i+1} \cdots a_j$. Note that $w[i..i] = a_i$ and $w[i..i-1] = \epsilon$.

If $w = ux$, we sometimes write $x = u^{-1}w$ and $u = wx^{-1}$.

Now we turn to sets of strings. A *language* over Σ is a (finite or infinite) set of strings—in other words, a subset of Σ^*.

Example 1.2.1. The following are examples of languages:

$\mathtt{PRIMES2} = \{\mathtt{10}, \mathtt{11}, \mathtt{101}, \mathtt{111}, \mathtt{1011}, \mathtt{1101}, \mathtt{10001}, \ldots\}$ (the primes represented in base 2)

$\mathtt{EQ} = \{x \in \{\mathtt{0}, \mathtt{1}\}^* : |x|_{\mathtt{0}} = |x|_{\mathtt{1}}\}$ (strings containing an equal number of each symbol)

$= \{\epsilon, \mathtt{01}, \mathtt{10}, \mathtt{0011}, \mathtt{0101}, \mathtt{0110}, \mathtt{1001}, \mathtt{1010}, \mathtt{1100}, \ldots\}$

$\mathtt{EVEN} = \{x \in \{\mathtt{0}, \mathtt{1}\}^* : |x|_{\mathtt{0}} \equiv 0 \pmod 2\}$ (strings with an even number of 0s)

$\mathtt{SQ} = \{xx : x \in \{\mathtt{0}, \mathtt{1}\}^*\}$ (the language of squares)

Given a language $L \subseteq \Sigma^*$, we may consider its prefix and suffix languages. We define

$$\text{Pref}(L) = \{x \in \Sigma^* : \text{there exists } y \in L \text{ such that } x \text{ is a prefix of } y\};$$
$$\text{Suff}(L) = \{x \in \Sigma^* : \text{there exists } y \in L \text{ such that } x \text{ is a suffix of } y\}.$$

One of the fundamental operations on strings is *concatenation*. We concatenate two finite strings w and x by juxtaposing their symbols, and we denote this by wx. For example, if $w = \texttt{book}$ and $x = \texttt{case}$, then $wx = \texttt{bookcase}$. Concatenation of strings is, in general, not commutative; for example, we have $xw = \texttt{casebook}$. However, concatenation is associative: we have $w(xy) = (wx)y$ for all strings w, x, y.

In general, concatenation is treated notationally like multiplication, so that, for example, w^n denotes the string $www\cdots w$ (n times).

If $w = a_1a_2\cdots a_n$ and $x = b_1b_2\cdots b_n$ are finite words of the same length, then by $w \amalg x$ we mean the word $a_1b_1a_2b_2\cdots a_nb_n$, the *perfect shuffle* of w and x. For example, $\texttt{shoe} \amalg \texttt{cold} = \texttt{schooled}$, and $\texttt{clip} \amalg \texttt{aloe} = \texttt{calliope}$, and (appropriately for this book) $\texttt{term} \amalg \texttt{hoes} = \texttt{theorems}$.

If $w = a_1a_2\cdots a_n$ is a finite word, then by w^R we mean the *reversal* of the word w; that is, $w^R = a_na_{n-1}\cdots a_2a_1$. For example, $(\texttt{drawer})^R = \texttt{reward}$. Note that $(wx)^R = x^Rw^R$. A word w is a *palindrome* if $w = w^R$. Examples of palindromes in English include `radar`, `deified`, `rotator`, `repaper`, and `redivider`.

We now turn to orders on strings. Given a finite alphabet Σ, we can impose an order on the elements. For example, if $\Sigma = \Sigma_k = \{0, 1, 2, \ldots, k-1\}$, for some integer $k \geq 2$, then $0 < 1 < 2 < \cdots < k-1$. Suppose w, x are equal-length strings over Σ. We say that w is *lexicographically smaller* than x, and write $w < x$, if there exist strings z, w', x' and letters a, b such that $w = zaw'$, $x = zbx'$, and $a < b$. Thus, for example, $\texttt{trust} < \texttt{truth}$. We can extend this order to the *radix order* defined as follows: $w < x$ if $|w| < |x|$, or $|w| = |x|$ and w precedes x in lexicographic order. Thus, for example, $\texttt{rat} < \texttt{moose}$ in radix order.

1.3 Regular expressions and regular languages

As we have seen earlier, a language over Σ is a subset of Σ^*. Languages may be of finite or infinite cardinality. We start by defining some common operations on languages.

Let $L, L_1, L_2 \subseteq \Sigma^*$ be languages. We define the *product* or *concatenation* of languages by

$$L_1L_2 = \{wx : w \in L_1, x \in L_2\}.$$

Common Error 1.3.1. Note that the definition of language concatenation implies that $L\emptyset = \emptyset L = \emptyset$. Many students mistakenly believe that $L\emptyset = L$.

We define $L^0 = \{\epsilon\}$ and define L^i as LL^{i-1} for $i \geq 1$. We define

$$L^{\leq i} = L^0 \cup L^1 \cup \cdots \cup L^i.$$

We define L^* as $\bigcup_{i \geq 0} L^i$; the operation L^* is sometimes called *Kleene closure*. We define $L^+ = L\,L^*$; the operation $+$ in the superscript is sometimes called *positive closure*. If L is a language, then the reversed language is defined as follows: $L^R = \{x^R : x \in L\}$.

We now turn to a common notation for representing some kinds of languages. A *regular expression* over the base alphabet Σ is a well-formed string over the larger alphabet $\Sigma \cup A$, where $A = \{\epsilon,\ \emptyset,\ (,\ ),\ +,\ *\}$; we assume $\Sigma \cap A = \emptyset$. In evaluating such an expression, $*$ represents Kleene closure and has highest precedence. Concatenation is represented by juxtaposition, and has next highest precedence. Finally, $+$ represents union and has lowest precedence. Parentheses are used for grouping. A formal definition of regular expressions is given in Exercise 33.

If the word u is a regular expression, then $L(u)$ represents the language that u is shorthand for. For example, consider the regular expression $u = (0+10)^*(1+\epsilon)$. Then $L(u)$ represents all finite words of 0s and 1s that do not contain two consecutive 1s. Frequently we will abuse the notation by referring to the language as the naked regular expression without the surrounding $L(\ )$. A language L is said to be *regular* if $L = L(u)$ for some regular expression u.

1.4 Finite automata

A *deterministic finite automaton*, or DFA for short, is the simplest model of a computer. We imagine a finite control equipped with a read head and a tape, divided into cells, which holds a finite input. At each step, depending on the machine's internal state and the current symbol being scanned, the machine can change its internal state and move right to the next square on the tape. If, after scanning all the cells of the input the machine is in any one of a number of *final states*, we say the input is *accepted*; otherwise it is *rejected* (see Figure 1.1).

Formally, a DFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

- Q is a finite nonempty set of states;
- Σ is a finite nonempty input alphabet;
- $\delta : Q \times \Sigma \to Q$ is a transition function;

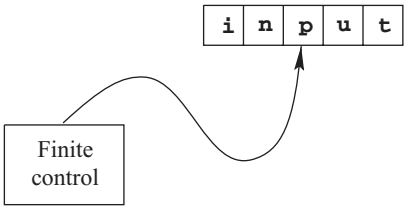

Figure 1.1: A deterministic finite automaton

- $q_0 \in Q$ is the start or initial state;
- $F \subseteq Q$ is the set of final states.

The transition function δ can be extended to a transition function $\delta^* : Q \times \Sigma^* \to Q$ as follows:

- $\delta^*(q, \epsilon) = q$ for all $q \in Q$;
- $\delta^*(q, xa) = \delta(\delta^*(q, x), a)$ for all $q \in Q$, $x \in \Sigma^*$, and $a \in \Sigma$.

Since δ^* agrees with δ on the domain of δ, we often just write δ for δ^*.

Now we give the formal definition of $L(M)$, the language accepted by a DFA M. We have

$$L(M) = \{x \in \Sigma^* \ : \ \delta(q_0, x) \in F\}.$$

We often describe deterministic finite automata by providing a *transition diagram*. This is a directed graph where states are represented by circles, final states represented by double circles, the initial state is labeled by a headless arrow entering a state, and transitions represented by directed arrows, labeled with a letter. For example, the transition diagram in Figure 1.2 represents the DFA that accepts the language $\texttt{EVEN} = \{x \in \{0, 1\}^* \ : \ |x|_0 \equiv 0 \pmod 2\}$.

Representation as a transition diagram suggests the following natural generalization of a DFA: we allow the automaton to have multiple choices (or none at all) on what state to enter on reading a given symbol. We accept an input if and only if *some* sequence of choices leads to a final state. For example, the

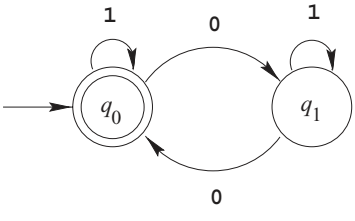

Figure 1.2: Transition diagram for a DFA

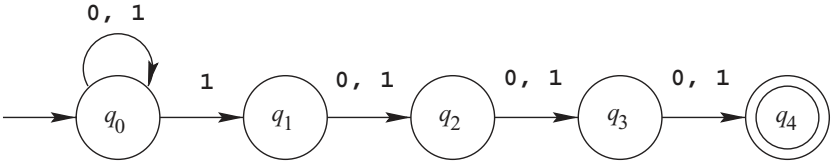

Figure 1.3: Transition diagram for an NFA

transition diagram in Figure 1.3 represents a *nondeterministic* finite automaton (NFA) that accepts the language L_4, where

$$L_n := \{x \in \{0,1\}^* \ : \ \text{the } n\text{th symbol from the right is } 1\}.$$

It is possible to show that the smallest DFA accepting L_n has at least 2^n states (see Exercise 3.14), so NFAs, while accepting the same class of languages as DFAs, can be exponentially more concise.

Formally, an NFA is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where $\delta : Q \times \Sigma \to 2^Q$. We define the extended transition function δ^* by

- $\delta^*(q, \epsilon) = \{q\}$;
- $\delta^*(q, xa) = \bigcup_{r \in \delta^*(q,x)} \delta(r, a)$.

The language accepted by an NFA, $L(M)$, is then given by

$$L(M) = \{x \in \Sigma^* \ : \ \delta^*(q_0, x) \ \cap \ F \neq \emptyset\}.$$

The following theorem shows that NFAs accept exactly the regular languages.

Theorem 1.4.1. *If M is an NFA, then there exists a DFA M' such that $L(M) = L(M')$.*

Proof Idea. We let the states of M' be all subsets of the state set of M. The final states of M' are those subsets containing at least one final state of M. ■

Exercise 31 asks you to show that the subset construction for NFA-to-DFA conversion can be carried out in $O(kn2^n)$ time, where $k = |\Sigma|$ and $n = |Q|$.

Yet another generalization of DFA is to allow the DFA to change state spontaneously without consuming a symbol of the input. This can be represented in a transition diagram by allowing arrows labeled ϵ, which are called ϵ-transitions. An NFA-ϵ is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$.

The most important theorem on regular languages is Kleene's theorem:

Theorem 1.4.2. *The following language classes are identical:*

(a) the class of languages specified by regular expressions;
(b) the class of languages accepted by DFAs;

(c) *the class of languages accepted by NFAs;*
(d) *the class of languages accepted by NFA-ϵ's.*

As a corollary, we can deduce some important *closure properties* of regular languages. We say a class of languages is *closed* under an operation if whenever the arguments to the operation are in the class, the result is also. If there are any counterexamples at all, we say the class is *not closed*.

Corollary 1.4.3. *The class of regular languages is closed under the operations of union, concatenation, Kleene $*$, and complement.*

The pumping lemma is an important tool for proving that certain languages are not regular.

Lemma 1.4.4. *Suppose L is a regular language. Then there exists a constant $n \geq 1$, depending on L, such that for all $z \in L$ with $|z| \geq n$, there exists a decomposition $z = uvw$ with $|uv| \leq n$ and $|v| \geq 1$ such that $uv^i w \in L$ for all $i \geq 0$. In fact, we may take n to be the number of states in any DFA accepting L.*

Proof Idea. The basic idea of the proof is that the path through the transition diagram for any sufficiently long accepted word must contain a loop. We may then go around this loop any number of times to obtain an infinite number of accepted words of the form $uv^i w$. ■

Example 1.4.5. Let us show that the language

$$\texttt{PRIMES1} = \{\mathtt{a}^2, \mathtt{a}^3, \mathtt{a}^5, \ldots\},$$

the prime numbers represented in unary, is not regular. Let n be the pumping lemma constant, and choose a prime $p > n$; we know such a prime exists by Euclid's theorem that there are infinitely many primes. Let $z = \mathtt{a}^p$. Then there exists a decomposition $z = uvw$ with $|uv| \leq n$ and $|v| \geq 1$ such that $uv^i w \in \texttt{PRIMES1}$ for all $i \geq 0$. Suppose $|v| = r$. Then choose $i = p + 1$. We have $|uv^i w| = p + (i-1)r = p(r+1)$. Since $r \geq 1$, this number is not a prime, a contradiction.

Example 1.4.6. Here is a deeper application of the pumping lemma. Let us show that the language

$$\texttt{PRIMES2} = \{10, 11, 101, 111, 1011, 1101, 10001, \ldots\},$$

the prime numbers represented in binary, is not regular. Let n be the pumping lemma constant and p be a prime $p > 2^n$. Let z be the base-2 representation

of p. If t is a string of 0s and 1s, let $[t]_2$ denote the integer whose base-2 representation is given by t. Write $z = uvw$. Now

$$[z]_2 = [u]_2 2^{|vw|} + [v]_2 2^{|w|} + [w]_2,$$

while

$$[uv^i w]_2 = [u]_2 2^{i|v|+|w|} + [v]_2(2^{|w|} + 2^{|vw|} + \cdots + 2^{|v^{i-1}w|}) + [w]_2.$$

Now $2^{|w|} + 2^{|vw|} + \cdots + 2^{|v^{i-1}w|}$ is, by the sum of a geometric series, equal to $2^{|w|}\frac{2^{i|v|}-1}{2^{|v|}-1}$. Now by Fermat's theorem, $2^p \equiv 2 \pmod{p}$ if p is a prime. Hence, setting $i = p$, we get $[uv^p w]_2 - [uvw]_2 \equiv 0 \pmod{p}$. But since z has no leading zeroes, $[uv^p w]_2 > [uvw]_2 = p$. (Note that $2^{|v|} - 1 \not\equiv 0 \pmod{p}$ since $|v| \geq 1$ and $|uv| \leq n \Rightarrow 2^{|v|} \leq 2^n < p$.) It follows that $[uv^p w]_2$ is an integer larger than p that is divisible by p, and so cannot represent a prime number. Hence, $uv^p w \notin$ PRIMES2. This contradiction proves that PRIMES2 is not regular.

1.5 Context-free grammars and languages

In the previous section, we saw two of the three important ways to specify languages: namely, as the language accepted by a machine or the language specified by a regular expression. In this section, we explore a third important way, the *grammar*. A machine receives a string as input and processes it, but a grammar actually constructs a string iteratively through a number of rewriting rules. We focus here on a particular kind of grammar, the *context-free grammar* (CFG).

Example 1.5.1. Consider the CFG given by the following production rules:

$$S \to \mathtt{a}$$
$$S \to \mathtt{b}$$
$$S \to \mathtt{a}S\mathtt{a}$$
$$S \to \mathtt{b}S\mathtt{b}.$$

The intention is to interpret each of these four rules as rewriting rules. We start with the symbol S and can choose to replace it by any of a, b, aSa, bSb. Suppose we replace S by aSa. Now the resulting string still has an S in it, and so we can choose any one of four strings to replace it. If we choose the rule $S \to \mathtt{b}S\mathtt{b}$, we get ab$S$ba. Now if we choose the rule $S \to \mathtt{b}$, we get the string abbba, and no more rules can be performed.

It is not hard to see that the language generated by this process is the set of palindromes over {a, b} of odd length, which we call ODDPAL.

Example 1.5.2. Here is a somewhat harder example. Let us create a CFG to generate the *nonpalindromes* over {a, b}.

$$S \to \mathtt{a}S\mathtt{a} \mid \mathtt{b}S\mathtt{b} \mid \mathtt{a}T\mathtt{b} \mid \mathtt{b}T\mathtt{a}$$
$$T \to \mathtt{a}T\mathtt{a} \mid \mathtt{a}T\mathtt{b} \mid \mathtt{b}T\mathtt{a} \mid \mathtt{b}T\mathtt{b} \mid \epsilon \mid \mathtt{a} \mid \mathtt{b}.$$

The basic idea is that if a string is a nonpalindrome, then there must be at least one position such that the character in that position does not match the character in the corresponding position from the end. The productions $S \to \mathtt{a}S\mathtt{a}$ and $S \to \mathtt{b}S\mathtt{b}$ are used to generate a prefix and suffix that match properly, but eventually one of the two productions involving T on the right-hand side must be used, at which point a mismatch is introduced. Now the remaining symbols can either match or not match, which accounts for the remaining productions involving T.

Example 1.5.3. Finally, we conclude with a genuinely challenging example. Consider the language

$$\begin{aligned} L &= \{x \in \{\mathtt{0},\mathtt{1}\}^* \ : \ x \text{ is not of the form } ww\} = \overline{\mathtt{SQ}} \\ &= \{\mathtt{0}, \mathtt{1}, \mathtt{01}, \mathtt{10}, \mathtt{000}, \mathtt{001}, \mathtt{010}, \mathtt{011}, \mathtt{100}, \mathtt{101}, \mathtt{110}, \mathtt{111}, \mathtt{0001}, \mathtt{0010}, \\ &\quad \mathtt{0011}, \mathtt{1000}, \ldots\}. \end{aligned}$$

Exercise 25 asks you to prove that this language can be generated by the following grammar:

$$S \to AB \mid BA \mid A \mid B$$
$$A \to \mathtt{0}A\mathtt{0} \mid \mathtt{0}A\mathtt{1} \mid \mathtt{1}A\mathtt{0} \mid \mathtt{1}A\mathtt{1} \mid \mathtt{0}$$
$$B \to \mathtt{0}B\mathtt{0} \mid \mathtt{0}B\mathtt{1} \mid \mathtt{1}B\mathtt{0} \mid \mathtt{1}B\mathtt{1} \mid \mathtt{1}.$$

Formally, we define a CFG G to be a 4-tuple $G = (V, \Sigma, P, S)$, where V is a nonempty finite set of variables, Σ is a nonempty finite set of terminal symbols, P is a finite set of productions of the form $A \to \alpha$, where $A \in V$ and $\alpha \in (V \cup \Sigma)^*$ (i.e., a finite subset of $V \times (V \cup \Sigma)^*$), and S is a distinguished element of V called the *start symbol*. We require that $V \cap \Sigma = \emptyset$. The term *context-free* comes from the fact that A may be replaced by α, independent of the context in which A appears.

A *sentential form* is any string of variables and terminals. We can go from one sentential form to another by applying a rule of the grammar. Formally, we write $\alpha B \gamma \Longrightarrow \alpha\beta\gamma$ if $B \to \beta$ is a production of P. We write $\stackrel{*}{\Longrightarrow}$ for the

reflexive, transitive closure of $\Longrightarrow$. In other words, we write $\alpha \stackrel{*}{\Longrightarrow} \beta$ if there exist sentential forms $\alpha = \alpha_0, \alpha_1, \ldots, \alpha_n = \beta$ such that

$$\alpha_0 \Longrightarrow \alpha_1 \Longrightarrow \alpha_2 \Longrightarrow \cdots \Longrightarrow \alpha_n.$$

A *derivation* consists of 0 or more applications of $\Longrightarrow$ to some sentential form.

If G is a CFG, then we define

$$L(G) = \{x \in \Sigma^* \ : \ S \stackrel{*}{\Longrightarrow} x\}.$$

A *leftmost derivation* is a derivation in which the variable replaced at each step is the leftmost one. A *rightmost derivation* is defined analogously. A grammar G is said to be *unambiguous* if every word $w \in L(G)$ has exactly one leftmost derivation and *ambiguous* otherwise.

A *parse tree* or *derivation tree* for $w \in L(G)$ is an ordered tree T where each vertex is labeled with an element of $V \cup \Sigma \cup \{\epsilon\}$. The root is labeled with a variable A and the leaves are labeled with elements of Σ or ϵ. If a node is labeled with $A \in V$ and its children are (from left to right) $X_1, X_2, \ldots, X_r$, then $A \to X_1 X_2 \cdots X_r$ is a production of G. The *yield* of the tree is w and consists of the concatenation of the leaf labels from left to right.

Theorem 1.5.4. *A grammar is unambiguous if and only if every word generated has exactly one parse tree.*

The class of languages generated by CFGs is called the *context-free languages* (CFLs).

We now recall some basic facts about CFGs. First, productions of the form $A \to \epsilon$ are called ϵ-productions and productions of the form $A \to B$ *unit* productions. There is an algorithm to transform a CFG G into a new grammar G' without ϵ-productions or unit productions, such that $L(G') = L(G) - \{\epsilon\}$ (see Exercise 27). Furthermore, it is possible to carry out this transformation in such a way that if G is unambiguous, G' is also.

We say a grammar is in *Chomsky normal form* if every production is of the form $A \to BC$ or $A \to a$, where A, B, C are variables and a is a single terminal. There is an algorithm to transform a grammar G into a new grammar G' in Chomsky normal form, such that $L(G') = L(G) - \{\epsilon\}$; (see Exercise 28).

We now recall a basic result about CFLs, known as the pumping lemma.

Theorem 1.5.5. *If L is context-free, then there exists a constant n such that for all $z \in L$ with $|z| \geq n$, there exists a decomposition $z = uvwxy$ with $|vwx| \leq n$ and $|vx| \geq 1$ such that for all $i \geq 0$, we have $uv^i wx^i y \in L$.*

Proof Idea. If L is context-free, then we can find a Chomsky normal form grammar G generating $L - \{\epsilon\}$. Let $n = 2^k$, where k is the number of variables