Introduction to the Theory of Computation
Languages, Automata and Grammars
Some Notes for CIS262

Jean Gallier
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104, USA
e-mail: `jean@cis.upenn.edu`

March 5, 2020

Contents

Chapter 1

Introduction

The theory of computation is concerned with algorithms and algorithmic systems: their design and representation, their completeness, and their complexity.

The purpose of these notes is to introduce some of the basic notions of the theory of computation, including concepts from formal languages and automata theory, the theory of computability, some basics of recursive function theory, and an introduction to complexity theory. Other topics such as correctness of programs will not be treated here (there just isn't enough time!).

The notes are divided into three parts. The first part is devoted to formal languages and automata. The second part deals with models of computation, recursive functions, and undecidability. The third part deals with computational complexity, in particular the classes $\mathcal{P}$ and $\mathcal{NP}$.

Chapter 2

Basics of Formal Language Theory

2.1 Review of Some Basic Math Notation and Definitions

$\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}, \mathbb{C}$.

The *natural numbers*,

$$\mathbb{N} = \{0, 1, 2, \ldots\}.$$

The *integers*,

$$\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}.$$

The *rationals*,

$$\mathbb{Q} = \left\{ \frac{p}{q} \mid p, q \in \mathbb{Z},\ q \neq 0 \right\}.$$

The *reals*, $\mathbb{R}$.

The *complex numbers*,

$$\mathbb{C} = \{a + ib \mid a, b \in \mathbb{R}\}.$$

Given any set X, the *power set* of X is the set of all subsets of X and is denoted 2^X.

The notation

$$f \colon X \to Y$$

denotes a *function* with *domain X* and *range*
(or *codomain*) Y.

$$\mathrm{graph}(f) = \{(x, f(x)) \mid x \in X\} \subseteq X \times Y$$

is the *graph* of f.

$$\mathrm{Im}(f) = f(X) = \{y \in Y \mid \exists x \in X,\, y = f(x)\} \subseteq Y$$

is the *image* of f.

More generally, if $A \subseteq X$, then

$$f(A) = \{y \in Y \mid \exists x \in A,\, y = f(x)\} \subseteq Y$$

is the *(direct) image* of A.

If $B \subseteq Y$, then

$$f^{-1}(B) = \{x \in X \mid f(x) \in B\} \subseteq X$$

is the *inverse image* (or *pullback*) of B.

$f^{-1}(B)$ is a set; it might be empty even if $B \neq \emptyset$.

Given two functions $f\colon X \to Y$ and $g\colon Y \to Z$, the function $g \circ f\colon X \to Z$ given by

$$(g \circ f)(x) = g(f(x)) \quad \text{for all } x \in X$$

is the *composition* of f and g.

The function $\mathrm{id}_X\colon X \to X$ given by

$$\mathrm{id}_X(x) = x \quad \text{for all } x \in X$$

is the *identity function* (of X).

A function $f\colon X \to Y$ is *injective* (old terminology *one-to-one)* if for all $x_1, x_2 \in X$, if $f(x_1) = f(x_2)$, then $x_1 = x_2$;

equivalently if $x_1 \neq x_2$, then $f(x_1) \neq f(x_2)$.

Fact: If $X \neq \emptyset$ (and so $Y \neq \emptyset$), a function $f\colon X \to Y$ is injective iff there is a function $r\colon Y \to X$ (a *left inverse*) such that

$$r \circ f = \mathrm{id}_X.$$

Note: r is surjective.

A function $f\colon X \to Y$ is *surjective* (old terminology *onto)* if for all $y \in Y$, there is some $x \in X$ such that $y = f(x)$, iff

$$f(X) = Y.$$

Fact: If $X \neq \emptyset$ (and so $Y \neq \emptyset$), a function $f\colon X \to Y$ is surjective iff there is a function $s\colon Y \to X$ (a *right inverse* or *section*) such that

$$f \circ s = \mathrm{id}_Y.$$

Note: s is injective.

A function $f\colon X \to Y$ is *bijective* if it is injective and surjective.

Fact: If $X \neq \emptyset$ (and so $Y \neq \emptyset$), a function $f\colon X \to Y$ is bijective if there is a function $f^{-1}\colon Y \to X$ which is a left and a right inverse, that is

$$f^{-1} \circ f = \mathrm{id}_X, \qquad f \circ f^{-1} = \mathrm{id}_Y.$$

The function f^{-1} is unique and called the *inverse* of f. The function f is said to be *invertible*.

A *binary relation* R between two sets X and Y is a subset

$$R \subseteq X \times Y = \{(x, y) \mid x \in X,\ y \in Y\}.$$

$$\mathrm{dom}(R) = \{x \in X \mid \exists y \in Y,\ (x, y) \in R\} \subseteq X$$

is the *domain* of R.

$$\mathrm{range}(R) = \{y \in Y \mid \exists x \in X,\ (x, y) \in R\} \subseteq Y$$

is the *range* of R.

We also write xRy instead of $(x, y) \in R$.

Given two relations $R \subseteq X \times Y$ and $S \subseteq Y \times Z$, their *composition* $R \circ S \subseteq X \times Z$ is given by

$$R \circ S = \{(x, z) \mid \exists y \in Y,\ (x, y) \in R \quad \text{and} \quad (y, z) \in S\}.$$

Note that if R and S are the graphs of two functions f and g, then $R \circ S$ is the graph of $g \circ f$.

$$I_X = \{(x, x) \mid x \in X\}$$

is the *identity relation on X*.

Given $R \subseteq X \times Y$, the *converse* $R^{-1} \subseteq Y \times X$ of R is given by

$$R^{-1} = \{(x, y) \in Y \times X \mid (y, x) \in R\}.$$

A relation $R \subseteq X \times X$ is *transitive* if for all $x, y, z \in X$, if $(x, y) \in R$ and $(y, z) \in R$, then $(x, z) \in R$.

A relation $R \subseteq X \times X$ is transitive iff $R \circ R \subseteq R$.

A relation $R \subseteq X \times X$ is *reflexive* if $(x, x) \in R$ for all $x \in X$

A relation $R \subseteq X \times X$ is reflexive iff $I_X \subseteq R$.

A relation $R \subseteq X \times X$ is *symmetric* if for all $x, y \in X$, if $(x, y) \in R$, then $(y, x) \in R$

A relation $R \subseteq X \times X$ is symmetric iff $R^{-1} \subseteq R$.

Given $R \subseteq X \times X$ (a relation on X), define R^n by

$$\begin{aligned} R^0 &= I_X \\ R^{n+1} &= R \circ R^n. \end{aligned}$$

The *transtive closure R^+ of R* is given by

$$R^+ = \bigcup_{n \geq 1} R^n.$$

Fact. R^+ is the smallest transitive relation containing R.

The *reflexive and transitive closure R^* of R* is given by

$$R^* = \bigcup_{n \geq 0} R^n = R^+ \cup I_X.$$

Fact. R^* is the smallest transitive and reflexive relation containing R.

A relation $R \subseteq X \times X$ is an *equivalence relation* if it is reflexive, symmetric, and transitive.

Fact. The smallest equivalence relation containing a relation $R \subseteq X \times X$ is given by

$$(R \cup R^{-1})^*.$$

A relation $R \subseteq X \times X$ is *antisymmetric* if for all $x, y \in X$, if $(x, y) \in R$ and $(y, x) \in R$, then $x = y$.

A relation $R \subseteq X \times X$ is a *partial order* if it is reflexive, transitive, and antisymmetic.

A partial order $R \subseteq X \times X$ is a *total order* if for all $x, y \in X$, either $(x, y) \in R$ or $(y, x) \in R$.

2.2 Alphabets, Strings, Languages

Our view of languages is that *a language is a set of strings*. In turn, a string is a finite sequence of letters from some alphabet. These concepts are defined rigorously as follows.

Definition 2.1. An *alphabet* Σ is any **finite** set.

We often write $\Sigma = \{a_1, \ldots, a_k\}$. The a_i are called the *symbols* of the alphabet.

Examples:

$\Sigma = \{a\}$

$\Sigma = \{a, b, c\}$

$\Sigma = \{0, 1\}$

$\Sigma = \{\alpha, \beta, \gamma, \delta, \epsilon, \lambda, \varphi, \psi, \omega, \mu, \nu, \rho, \sigma, \eta, \xi, \zeta\}$

A string is a finite sequence of symbols. Technically, it is convenient to define strings as functions. For any integer $n \geq 1$, let

$$[n] = \{1, 2, \ldots, n\},$$

and for $n = 0$, let

$$[0] = \emptyset.$$

Definition 2.2. Given an alphabet Σ, a *string over Σ (or simply a string) of length n* is any function

$$u \colon [n] \to \Sigma.$$

The integer n is the *length* of the string u, and it is denoted as $|u|$. When $n = 0$, the special string $u \colon [0] \to \Sigma$ of length 0 is called the *empty string, or null string*, and is denoted as ϵ.

Given a string $u \colon [n] \to \Sigma$ of length $n \geq 1$, $u(i)$ is the i-th letter in the string u. For simplicity of notation, *we write u_i instead of $u(i)$*, and we denote the string $u = u(1)u(2) \cdots u(n)$ as

$$u = u_1 u_2 \cdots u_n,$$

with each $u_i \in \Sigma$.

For example, if $\Sigma = \{a, b\}$ and $u \colon [3] \to \Sigma$ is defined such that $u(1) = a$, $u(2) = b$, and $u(3) = a$, we write

$$u = aba.$$

Other examples of strings are

$$work, \;\; fun, \;\; gabuzomeuh$$

Strings of length 1 are functions $u \colon [1] \to \Sigma$ simply picking some element $u(1) = a_i$ in Σ. Thus, we will identify every symbol $a_i \in \Sigma$ with the corresponding string of length 1.

The set of all strings over an alphabet Σ, including the empty string, is denoted as Σ^*.

Observe that when $\Sigma = \emptyset$, then
$$\emptyset^* = \{\epsilon\}.$$
When $\Sigma \neq \emptyset$, the set Σ^* is countably infinite. Later on, we will see ways of ordering and enumerating strings.

Strings can be juxtaposed, or concatenated.

Definition 2.3. Given an alphabet Σ, given any two strings $u\colon [m] \to \Sigma$ and $v\colon [n] \to \Sigma$, the *concatenation $u \cdot v$ (also written uv) of u and v* is the string
$uv\colon [m+n] \to \Sigma$, defined such that

$$uv(i) = \begin{cases} u(i) & \text{if } 1 \leq i \leq m, \\ v(i-m) & \text{if } m+1 \leq i \leq m+n. \end{cases}$$

In particular, $u\epsilon = \epsilon u = u$. Observe that

$$|uv| = |u| + |v|.$$

For example, if $u = ga$, and $v = buzo$, then

$$uv = gabuzo$$

It is immediately verified that

$$u(vw) = (uv)w.$$

Thus, concatenation is a binary operation on Σ^* which is associative and has ϵ as an identity.

Note that generally, $uv \neq vu$, for example for $u = a$ and $v = b$.

Given a string $u \in \Sigma^*$ and $n \geq 0$, we define u^n recursively as follows:

$$\begin{aligned} u^0 &= \epsilon \\ u^{n+1} &= u^n u \quad (n \geq 0). \end{aligned}$$

By setting $n = 0$ in
$$u^{n+1} = u^n u$$
and using the fact that $u^0 = \epsilon$ we get

$$u^1 = u^{0+1} = u^0 u = \epsilon u = u,$$

so $u^1 = u$. It is an easy exercise to show that

$$u^n u = u u^n, \quad \text{for all } n \geq 0.$$

For the base case $n = 0$, since $u^0 = \epsilon$, we have

$$u^0 u = \epsilon u = u = u\epsilon = uu^0.$$

For the induction step, we have

$$\begin{aligned}
u^{n+1}u &= (u^n u)u && \text{by definition of } u^{n+1}\\
&= (uu^n)u && \text{by the induction hypothesis}\\
&= u(u^n u) && \text{by associativity}\\
&= uu^{n+1} && \text{by definition of } u^{n+1}.
\end{aligned}$$

Definition 2.4. Given an alphabet Σ, given any two strings $u, v \in \Sigma^*$ we define the following notions as follows:

u is a prefix of v iff there is some $y \in \Sigma^*$ such that

$$v = uy.$$

u is a suffix of v iff there is some $x \in \Sigma^*$ such that

$$v = xu.$$

u is a substring of v iff there are some $x, y \in \Sigma^*$ such that

$$v = xuy.$$

We say that *u is a proper prefix (suffix, substring) of v* iff u is a prefix (suffix, substring) of v and $u \neq v$.

For example, *ga* is a prefix of *gabuzo*,

zo is a suffix of *gabuzo* and

buz is a substring of *gabuzo*.

Recall that a partial ordering $\leq$ on a set S is a binary relation $\leq \subseteq S \times S$ which is reflexive, transitive, and antisymmetric.

The concepts of prefix, suffix, and substring, define binary relations on Σ^* in the obvious way. It can be shown that these relations are partial orderings.

Another important ordering on strings is the lexicographic (or dictionary) ordering.

Definition 2.5. Given an alphabet $\Sigma = \{a_1, \ldots, a_k\}$ assumed totally ordered such that $a_1 < a_2 < \cdots < a_k$, given any two strings $u, v \in \Sigma^*$, we define the *lexicographic ordering $\preceq$* as follows:

$$u \preceq v \quad \begin{cases} (1) \text{ if } v = uy, \text{ for some } y \in \Sigma^*, \text{ or} \\ (2) \text{ if } u = xa_i y,\ v = xa_j z,\ a_i < a_j, \\ \text{with } a_i, a_j \in \Sigma, \text{ and for some } x, y, z \in \Sigma^*. \end{cases}$$

The idea is that we scan u and v simultaneously from left to right, comparing the mth symbol u_m in u to the mth symbol v_m in v, starting with $m = 1$. If no discrepancy arises, that is, if the m-th symbol u_m in u *agrees* with the m-th symbol v_m in v for $m = 1, \ldots, |u|$, then u is a prefix of v and we declare that u precedes v in the lexicographic ordering. Otherwise, for a while u and v agree along a common prefix x (possibly the empty string), and then there is a *leftmost discrepancy*, which means that u is of the form $u = xa_iy$ and v is of the form $v = xa_jz$, with $a_i \neq a_j$ (and $x, y, z \in \Sigma^*$ arbitrary). Then we need to break the tie, and to do this we use the fact that the symbols $a_1 < a_2 < \cdots < a_k$ are assumed to be (totally) ordered, so we see which of a_i and a_j comes first, say $a_i < a_j$, and we declare that $u = xa_iy$ precedes $v = xa_jz$ in the lexicographic ordering.

Note that cases (1) and (2) are mutually exclusive. In case (1), u is a prefix of v. In case (2) $v \not\preceq u$ and $u \neq v$.

For example

$$ab \preceq b, \quad gallhager \preceq gallier.$$

It is fairly tedious to prove that the lexicographic ordering is in fact a partial ordering. In fact, it is a *total ordering*, which means that for any two strings $u, v \in \Sigma^*$, either $u \preceq v$, or $v \preceq u$.

The *reversal w^R* of a string w is defined inductively as follows:

$$\begin{aligned} \epsilon^R &= \epsilon, \\ (ua)^R &= au^R, \end{aligned}$$

where $a \in \Sigma$ and $u \in \Sigma^*$.

For example

$$reillag = gallier^R.$$

By setting $u = \epsilon$ in

$$(ua)^R = au^R,$$

since $\epsilon^R = \epsilon$ and $a = \epsilon a$, we get

$$a^R = (\epsilon a)^R = a\epsilon^R = a\epsilon = a,$$

namely $a^R = a$ for all $a \in \Sigma$.

It can be shown by induction on $|v|$ that

$$(uv)^R = v^R u^R.$$

A useful trick that cuts down on cumbersome notation when doing induction on strings is the observation that a *nonempty string* $w \in \Sigma^*$ of length $n + 1$ $(n \geq 0)$ can be written as

$$w = ua, \quad \text{for some } u \in \Sigma^* \text{ and some symbol } a \in \Sigma, \text{ with } |u| = n.$$

Since $|w| = n + 1$ (as $w = ua$), we can do induction on u. This trick saves us from using many indices (you **do not want to write** $w = w_1 \cdots w_{n+1}$, *etc.*). Sometimes, it is more convenient to write $w = au$, with $a \in \Sigma$, $u \in \Sigma^*$, and $|u| = n$.

It follows (by induction on n) that

$$(u_1 \ldots u_n)^R = u_n^R \ldots u_1^R,$$

and when $u_i \in \Sigma$, we have

$$(u_1 \ldots u_n)^R = u_n \ldots u_1.$$

We can now define languages.

Definition 2.6. Given an alphabet Σ, a *language over Σ (or simply a language)* is any subset L of Σ^*.

If $\Sigma \neq \emptyset$, there are uncountably many languages.

A Quick Review of Finite, Infinite, Countable, and Uncountable Sets

For details and proofs, see *Discrete Mathematics*, by Gallier.

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ be the set of *natural numbers*.

Recall that a set X is *finite* if there is some natural number $n \in \mathbb{N}$ and a bijection between X and the set $[n] = \{1, 2, \ldots, n\}$. (When $n = 0$, $X = \emptyset$, the empty set.)

The number n is uniquely determined. It is called the *cardinality (or size)* of X and is denoted by $|X|$.

A set is *infinite* iff it is not finite.

Fact. Recall that any injection or surjection of a finite set to itself is in fact a *bijection*.

The above fails for infinite sets.

The *pigeonhole principle* asserts that *there is no bijection between a finite set X and any proper subset Y of X*.

Consequence: If we think of X as a set of n pigeons and if there are only $m < n$ boxes (corresponding to the elements of Y), then at least two of the pigeons must share the same box.

As a consequence of the pigeonhole principle, a set X is infinite iff it is in bijection with a proper subset of itself.

For example, we have a bijection $n \mapsto 2n$ between $\mathbb{N}$ and the set $2\mathbb{N}$ of even natural numbers, a proper subset of $\mathbb{N}$, so $\mathbb{N}$ is infinite.

Definition 2.7. A set X is *countable (or denumerable)* if there is an *injection* from X into $\mathbb{N}$.

If X is not the empty set, since $f\colon X \to \mathbb{N}$ is an injection iff there is a surjection $r\colon \mathbb{N} \to X$ such that $r \circ f = \mathrm{id}_X$, the set X is countable iff there is a *surjection* from $\mathbb{N}$ onto X.

Fact. It can be shown that a set X is countable if either it is finite or if it is in bijection with $\mathbb{N}$ (in which case it is infinite).

We will see later that $\mathbb{N}\times\mathbb{N}$ is countable. As a consequence, the set $\mathbb{Q}$ of rational numbers is countable.

A set is *uncountable* if it is not countable.

For example, $\mathbb{R}$ (the set of real numbers) is uncountable.

Similarly

$$(0, 1) = \{x \in \mathbb{R} \mid 0 < x < 1\}$$

is uncountable. However, there is a bijection between $(0, 1)$ and $\mathbb{R}$ (find one!)

The set $2^{\mathbb{N}}$ of all subsets of $\mathbb{N}$ is uncountable. This is a special case of Cantor's theorem discussed below.

Suppose $|\Sigma| = k$ with $\Sigma = \{a_1, \ldots, a_k\}$. First, observe that there are k^n strings of length n and $(k^{n+1} - 1)/(k - 1)$ strings of length at most n over Σ; when $k = 1$, the second formula should be replaced by $n + 1$. Indeed, since a string is a function $u\colon \{1, \ldots, n\} \to \Sigma$, the number of strings of length n is the number of functions from $\{1, \ldots, n\}$ to Σ, and since the cardinality of Σ is k, there are k^n such functions (this is immediately shown by induction on n). Then the number of strings of length at most n is

$$1 + k + k^2 + \cdots + k^n.$$

If $k = 1$, this number is $n + 1$, and if $k \geq 2$, as the sum of a geometric series, it is $(k^{n+1} - 1)/(k - 1)$.

If $\Sigma \neq \emptyset$, then the set Σ^* of all strings over Σ is infinite and countable, as we now show by constructing an explicit bijection from Σ^* onto $\mathbb{N}$.

If $k = 1$ write $a = a_1$, and then

$$\{a\}^* = \{\epsilon, a, aa, aaa, \ldots, a^n, \ldots\}.$$

We have the bijection $n \mapsto a^n$ from $\mathbb{N}$ to $\{a\}^*$.

If $k \geq 2$, then we can think of the string

$$u = a_{i_1} \cdots a_{i_n}$$

as a representation of the integer $\nu(u)$ in base k shifted by $(k^n - 1)/(k - 1)$, with

$$\begin{aligned}\nu(u) &= i_1k^{n-1} + i_2k^{n-2} + \cdots + i_{n-1}k + i_n \\ &= \frac{k^n - 1}{k-1} + (i_1 - 1)k^{n-1} + \cdots + (i_{n-1} - 1)k + i_n - 1.\end{aligned}$$

(and with $\nu(\epsilon) = 0$), where $1 \leq i_j \leq k$ for $j = 1, \ldots, n$.

We leave it as an exercise to show that $\nu \colon \Sigma^* \to \mathbb{N}$ is a bijection. Finding *explicitly* (that is, a formula) for the inverse of ν is surprisingly difficult.

In fact, ν corresponds to the enumeration of Σ^* where u precedes v if $|u| < |v|$, and u precedes v in the lexicographic ordering if $|u| = |v|$. It is easy to check that the above relation (u precedes v) is a total order.

For example, if $k = 2$ and if we write $\Sigma = \{a, b\}$, then the enumeration begins with

$$\begin{array}{llllllll} \epsilon, \\ 0 \\ a, & b, \\ 1, & 2, \\ aa, & ab, & ba, & bb, \\ 3, & 4, & 5, & 6, \\ aaa, & aab, & aba, & abb, & baa, & bab, & bba, & bbb \\ 7, & 8, & 9, & 10, & 11, & 12, & 13, & 14 \end{array}$$

To get the next row, concatenate a on the left, and then concatenate b on the left. We have

$$\nu(bab) = 2 \cdot 2^2 + 1 \cdot 2^1 + 2 = 8 + 2 + 2 = 12.$$

It works!

On the other hand, if $\Sigma \neq \emptyset$, the set 2^{Σ^*} of all subsets of Σ^* (all languages) is *uncountable*.

Indeed, we can show that there is no surjection from $\mathbb{N}$ onto 2^{Σ^*} First, we will show that there is no surjection from Σ^* onto 2^{Σ^*}. This is a special case of Cantor's theorem.

We claim that if there is no surjection from Σ^* onto 2^{Σ^*}, then there is no surjection from $\mathbb{N}$ onto 2^{Σ^*} either.

Proof. Assume by contradiction that there is a surjection $g \colon \mathbb{N} \to 2^{\Sigma^*}$. But, if $\Sigma \neq \emptyset$, then Σ^* is infinite and countable, thus we have the bijection $\nu \colon \Sigma^* \to \mathbb{N}$. Then the composition

$$\begin{tikzcd} \Sigma^* \arrow[r, "\nu"] & \mathbb{N} \arrow[r, "g"] & 2^{\Sigma^*} \end{tikzcd}$$

is a surjection, because the bijection ν is a surjection, g is a surjection, and the composition of surjections is a surjection, contradicting the hypothesis that there is no surjection from Σ^* onto 2^{Σ^*}. $\square$

To prove that that there is no surjection from Σ^* onto 2^{Σ^*}. We use a *diagonalization* argument. This is an instance of *Cantor's Theorem*.

Theorem 2.1. *(Cantor, 1873) For every set X, there is no surjection from X onto 2^X.*

Proof. Assume there is a surjection $h\colon X \to 2^X$, and consider the set

$$D = \{x \in X \mid x \notin h(x)\} \in 2^X.$$

By definition, for any $x \in X$ we have $x \in D$ iff $x \notin h(x)$. Since h is surjective, there is some $y \in X$ such that $h(y) = D$. Then, by definition of D and since $D = h(y)$, we have

$y \in D$ iff $y \notin h(y) = D$,

a contradiction. Therefore, h is not surjective. $\square$

This is a beautiful proof but it is very abstract. The reader should experiment with concrete examples. For example, if $X = \{a, b, c\}$ and $h_1\colon X \to 2^X$ is given by

$$h_1(a) = \{a\}, \quad h_1(b) = \{a, c\}, \quad h_1(c) = \{a, b\},$$

we have $D = \{b, c\}$. Indeed, $\{b, c\}$ is not in the image of h_1.

For the function $h_2\colon X \to 2^X$ given by

$$h_2(a) = \{a\}, \quad h_2(b) = \{a, c\}, \quad h_2(c) = \{a, c\},$$

we have $D = \{b\}$. Indeed, $\{b\}$ is not in the image of h_2.

The proof of Theorem 2.1 actually shows a stronger fact: *for every set X and* **every** *function $h\colon X \to 2^X$, the subset $D = \{x \in X \mid x \notin h(x)\}$ is not in the image of h; that is, there is no $y \in X$ such that $D = h(y)$.*

Applying Theorem 2.1 to the case where $X = \Sigma^*$, we deduce that there is no surjection from Σ^* onto 2^{Σ^*}. Therefore, if $\Sigma \neq \emptyset$, then 2^{Σ^*} is uncountable.

Applying Theorem 2.1 to the case where $X = \mathbb{N}$, we see that there is no surjection from $\mathbb{N}$ onto $2^{\mathbb{N}}$. This shows that $2^{\mathbb{N}}$ is uncountable, as we claimed earlier.

For any set X, there an injection of X into 2^X obtained by mapping $x \in X$ to $\{x\} \in 2^X$. Since $2^{\emptyset} = \{\emptyset\}$ is not the empty set(!), there is no injection from $2^{\emptyset}$ into $\emptyset$ (a function with a nonempty domain must have a nonempty range). If $X \neq \emptyset$, since by Cantor's theorem, there is no surjection from X onto 2^X, *there is no injection $f\colon 2^X \to X$ of 2^X into X*. Otherwise, by a fact stated earlier, there would be a surjection $r\colon X \to 2^X$ such that $r \circ f = \mathrm{id}_{2^X}$, a contradiction. Intuitively, 2^X is strictly larger than X.

Since 2^{Σ^*} is uncountable, we will try to single out countable "tractable" families of languages.

We will begin with the family of *regular languages*, and then proceed to the *context-free languages*.

We now turn to operations on languages.

2.3 Operations on Languages

A way of building more complex languages from simpler ones is to combine them using various operations. First, we review the set-theoretic operations of union, intersection, and complementation.

Given some alphabet Σ, for any two languages L_1, L_2 over Σ, the *union* $L_1 \cup L_2$ of L_1 and L_2 is the language

$$L_1 \cup L_2 = \{w \in \Sigma^* \mid w \in L_1 \quad \text{or} \quad w \in L_2\}.$$

The *intersection* $L_1 \cap L_2$ of L_1 and L_2 is the language

$$L_1 \cap L_2 = \{w \in \Sigma^* \mid w \in L_1 \quad \text{and} \quad w \in L_2\}.$$

The *difference* $L_1 - L_2$ of L_1 and L_2 is the language

$$L_1 - L_2 = \{w \in \Sigma^* \mid w \in L_1 \quad \text{and} \quad w \notin L_2\}.$$

The difference is also called the *relative complement*.

A special case of the difference is obtained when $L_1 = \Sigma^*$, in which case we define the *complement* $\overline{L}$ of a language L as

$$\overline{L} = \{w \in \Sigma^* \mid w \notin L\}.$$

The above operations do not use the structure of strings. The following operations use concatenation.

Definition 2.8. Given an alphabet Σ, for any two languages L_1, L_2 over Σ, the *concatenation L_1L_2 of L_1 and L_2* is the language

$$L_1L_2 = \{w \in \Sigma^* \mid \exists u \in L_1,\ \exists v \in L_2,\ w = uv\}.$$

For any language L, we define L^n as follows:

$$\begin{aligned} L^0 &= \{\epsilon\}, \\ L^{n+1} &= L^nL \quad (n \geq 0). \end{aligned}$$

By setting $n = 0$ in $L^{n+1} = L^nL$, since $L^0 = \{\epsilon\}$, we get

$$L^1 = L^{0+1} = L^0L = \{\epsilon\}L = L,$$

so $L^1 = L$.

The following properties are easily verified:

$$\begin{aligned} L\emptyset &= \emptyset, \\ \emptyset L &= \emptyset, \\ L\{\epsilon\} &= L, \\ \{\epsilon\}L &= L, \\ (L_1 \cup \{\epsilon\})L_2 &= L_1L_2 \cup L_2, \\ L_1(L_2 \cup \{\epsilon\}) &= L_1L_2 \cup L_1, \\ L^nL &= LL^n. \end{aligned}$$

In general, $L_1L_2 \neq L_2L_1$.

So far, the operations that we have introduced, except complementation (since $\overline{L} = \Sigma^* - L$ is infinite if L is finite and Σ is nonempty), preserve the finiteness of languages. This is not the case for the next two operations.

Definition 2.9. Given an alphabet Σ, for any language L over Σ, the *Kleene $*$-closure L^* of L* is the language

$$L^* = \bigcup_{n\geq 0} L^n.$$

The *Kleene $+$-closure L^+ of L* is the language

$$L^+ = \bigcup_{n\geq 1} L^n.$$

Thus, L^* is the infinite union

$$L^* = L^0 \cup L^1 \cup L^2 \cup \ldots \cup L^n \cup \ldots,$$

and L^+ is the infinite union

$$L^+ = L^1 \cup L^2 \cup \ldots \cup L^n \cup \ldots.$$

Since $L^1 = L$, both L^* and L^+ contain L. In fact,

$$\begin{aligned} L^+ = \{&w \in \Sigma^*, \exists n \geq 1, \\ &\exists u_1 \in L \cdots \exists u_n \in L,\ w = u_1 \cdots u_n\}, \end{aligned}$$

and since $L^0 = \{\epsilon\}$,

$$\begin{aligned} L^* = \{\epsilon\} \cup \{&w \in \Sigma^*, \exists n \geq 1, \\ &\exists u_1 \in L \cdots \exists u_n \in L,\ w = u_1 \cdots u_n\}. \end{aligned}$$

Thus, the language L^* always contains ϵ, and we have

$$L^* = L^+ \cup \{\epsilon\}.$$

However, if $\epsilon \notin L$, then $\epsilon \notin L^+$. The following is easily shown:

$$\begin{aligned}\emptyset^* &= \{\epsilon\},\\ L^+ &= L^*L,\\ L^{**} &= L^*,\\ L^*L^* &= L^*.\end{aligned}$$

The Kleene closures have many other interesting properties.

Homomorphisms are also very useful.

Given two alphabets Σ, Δ, a *homomorphism* $h\colon \Sigma^* \to \Delta^*$ *between* Σ^* *and* Δ^* is a function $h\colon \Sigma^* \to \Delta^*$ such that

$$h(uv) = h(u)h(v) \quad \text{for all } u, v \in \Sigma^*.$$

Letting $u = v = \epsilon$, we get

$$h(\epsilon) = h(\epsilon)h(\epsilon),$$

which implies that (why?)

$$h(\epsilon) = \epsilon.$$

If $\Sigma = \{a_1, \ldots, a_k\}$, it is easily seen that h is completely determined by $h(a_1), \ldots, h(a_k)$ (why?)

Example 2.1. Letg $\Sigma = \{a, b, c\}$, $\Delta = \{0, 1\}$, and

$$h(a) = 01, \quad h(b) = 011, \quad h(c) = 0111.$$

For example,

$$h(abbc) = 010110110111.$$

Given any language $L_1 \subseteq \Sigma^*$, we define the *image* $h(L_1)$ *of* L_1 as

$$h(L_1) = \{h(u) \in \Delta^* \mid u \in L_1\}.$$

Given any language $L_2 \subseteq \Delta^*$, we define the *inverse image* $h^{-1}(L_2)$ *of* L_2 as

$$h^{-1}(L_2) = \{u \in \Sigma^* \mid h(u) \in L_2\}.$$

We now turn to the first formalism for defining languages, Deterministic Finite Automata (DFA's)

Chapter 3

DFA's, NFA's, Regular Languages

The family of regular languages is the simplest, yet interesting family of languages.

We give six definitions of the regular languages.

1. Using *deterministic finite automata (DFAs)*.
2. Using *nondeterministic finite automata (NFAs)*.
3. Using a *closure definition* involving, union, concatenation, and Kleene $*$.
4. Using *regular expressions*.
5. Using *right-invariant equivalence relations of finite index* (the Myhill-Nerode characterization).
6. Using *right-linear context-free grammars*.

We prove the equivalence of these definitions, often by providing an *algorithm* for converting one formulation into another.

We find that the introduction of NFA's is motivated by the conversion of regular expressions into DFA's.

To finish this conversion, we also show that every NFA can be converted into a DFA (using the *subset construction*).

So, although NFA's often allow for more concise descriptions, they do not have more expressive power than DFA's.

NFA's operate according to the paradigm: *guess a successful path, and check it in polynomial time*.

This is the essence of an important class of hard problems known as $\mathcal{NP}$, which will be investigated later.

We will also discuss methods for proving that certain languages are not regular (Myhill-Nerode, pumping lemma).

We present algorithms to convert a DFA to an equivalent one with a minimal number of states.

3.1 Deterministic Finite Automata (DFA's)

First we define what DFA's are, and then we explain how they are used to accept or reject strings. Roughly speaking, a DFA is a finite transition graph whose edges are labeled with letters from an alphabet Σ.

The graph also satisfies certain properties that makes it deterministic. Basically, this means that given any string w, starting from any node, *there is a unique path in the graph "parsing" the string w*.

Example 3.1. A DFA for the language

$$L_1 = \{ab\}^+ = \{ab\}^*\{ab\},$$

i.e.,

$$L_1 = \{ab, abab, ababab, \ldots, (ab)^n, \ldots\}.$$

Input alphabet: $\Sigma = \{a, b\}$.

State set $Q_1 = \{0, 1, 2, 3\}$.

Start state: 0.

Set of accepting states: $F_1 = \{2\}$.

Transition table (function) δ_1:

	a	b
0	1	3
1	3	2
2	1	3
3	3	3

Note that state 3 is a *trap state* or *dead state*.

Here is a graph representation of the DFA specified by the transition function shown above:

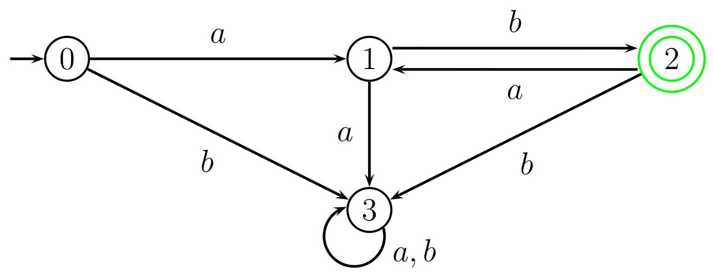


Figure 3.1: DFA for $\{ab\}^+$.

Example 3.2. A DFA for the language

$$L_2 = \{ab\}^* = L_1 \cup \{\epsilon\}$$

i.e.,

$$L_2 = \{\epsilon, ab, abab, ababab, \ldots, (ab)^n, \ldots\}.$$

Input alphabet: $\Sigma = \{a, b\}$.

State set $Q_2 = \{0, 1, 2\}$.

Start state: 0.

Set of accepting states: $F_2 = \{0\}$. The convention for the empty string to be accepted is that the start state is a final state.

Transition table (function) δ_2:

	a	b
0	1	2
1	2	0
2	2	2

State 2 is a *trap state* or *dead state*.

Here is a graph representation of the DFA specified by the transition function shown above:

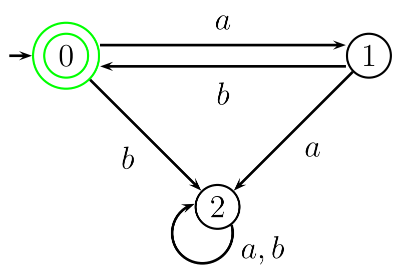


Figure 3.2: DFA for $\{ab\}^*$.

Example 3.3. A DFA for the language

$$L_3 = \{a, b\}^*\{abb\}.$$

Note that L_3 consists of all strings of a's and b's ending in abb.

Input alphabet: $\Sigma = \{a, b\}$.

State set $Q_3 = \{0, 1, 2, 3\}$.

Start state: 0.

Set of accepting states: $F_3 = \{3\}$.

Transition table (function) δ_3:

	a	b
0	1	0
1	1	2
2	1	3
3	1	0

Here is a graph representation of the DFA specified by the transition function shown above:

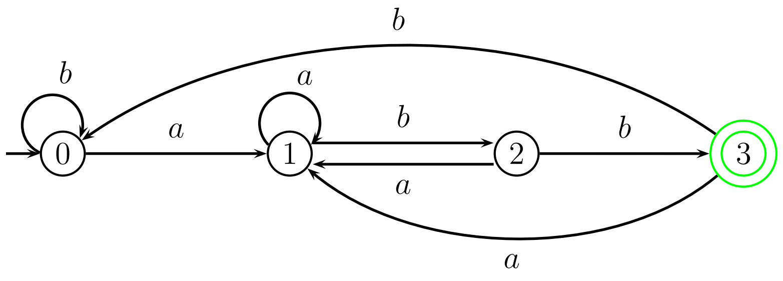


Figure 3.3: DFA for $\{a, b\}^*\{abb\}$.

Is this a minimal DFA?

Definition 3.1. A *deterministic finite automaton (or DFA)* is a quintuple $D = (Q, \Sigma, \delta, q_0, F)$, where

- Σ is a finite *input alphabet*;
- Q is a finite set of *states*;

- F is a subset of Q of *final (or accepting) states*;
- $q_0 \in Q$ is the *start state (or initial state)*;
- δ is the *transition function*, a function

$$\delta \colon Q \times \Sigma \to Q.$$

For any state $p \in Q$ and any input $a \in \Sigma$, the state $q = \delta(p, a)$ is uniquely determined.

Thus, it is possible to define the state reached from a given state $p \in Q$ on input $w \in \Sigma^*$, following the path specified by w.

Technically, this is done by defining the extended transition function $\delta^* \colon Q \times \Sigma^* \to Q$.

Definition 3.2. Given a DFA $D = (Q, \Sigma, \delta, q_0, F)$, the *extended transition function* $\delta^* \colon Q \times \Sigma^* \to Q$ is defined as follows:

$$\begin{aligned}\delta^*(p, \epsilon) &= p,\\ \delta^*(p, ua) &= \delta(\delta^*(p, u), a),\end{aligned}$$

where $a \in \Sigma$ and $u \in \Sigma^*$.

If we let $u = \epsilon$ in

$$\delta^*(p, ua) = \delta(\delta^*(p, u), a),$$

since $\delta^*(p, \epsilon) = p$, we get

$$\delta^*(p, a) = \delta^*(p, \epsilon a) = \delta(\delta^*(p, \epsilon), a) = \delta(p, a),$$

that is, $\delta^*(p, a) = \delta(p, a)$ for $a \in \Sigma$.

The meaning of $\delta^*(p, w)$ is that it is the state reached from state p following the path from p specified by w.

We can show (by induction on the length of v) that

$$\delta^*(p, uv) = \delta^*(\delta^*(p, u), v) \quad \text{for all } p \in Q \text{ and all } u, v \in \Sigma^*$$

For the base case $v = \epsilon$, since $\delta^*(q, \epsilon) = q$ for all $q \in Q$, we have

$$\delta^*(p, u\epsilon) = \delta^*(p, u) = \delta^*(\delta^*(p, u), \epsilon).$$

For the induction step, for $u \in \Sigma^*$, and all $v = ya$ with $y \in \Sigma^*$ and $a \in \Sigma$,

$$\begin{aligned}\delta^*(p, uya) &= \delta(\delta^*(p, uy), a) && \text{by definition of } \delta^*\\ &= \delta(\delta^*(\delta^*(p, u), y), a) && \text{by induction}\\ &= \delta^*(\delta^*(p, u), ya) && \text{by definition of } \delta^*.\end{aligned}$$

We can now define how a DFA accepts or rejects a string.

Definition 3.3. Given a DFA $D = (Q, \Sigma, \delta, q_0, F)$, the *language $L(D)$ accepted (or recognized) by D* is the language

$$L(D) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}.$$

Thus, a string $w \in \Sigma^*$ is accepted iff the path from q_0 on input w ends in a final state. Since $\delta^*(q_0, \epsilon) = q_0$, the empty string is accepted iff the start state is a final state, as we said before.

The definition of a DFA does not prevent the possibility that a DFA may have states that are not reachable from the start state q_0, which means that there is no path from q_0 to such states.

For example, in the DFA D_1 defined by the transition table below and the set of final states $F = \{1, 2, 3\}$, the states in the set $\{0, 1\}$ are reachable from the start state 0, but the states in the set $\{2, 3, 4\}$ are not (even though there are transitions from $2, 3, 4$ to 0, but they go in the wrong direction).

	a	b
0	1	0
1	0	1
2	3	0
3	4	0
4	2	0

Since there is no path from the start state 0 to any of the states in $\{2, 3, 4\}$, the states $2, 3, 4$ are useless as far as acceptance of strings, so they should be deleted as well as the transitions from them.

Given a DFA $D = (Q, \Sigma, \delta, q_0, F)$, the above suggests defining the following set.

Definition 3.4. Given any DFA $D = (Q, \Sigma, \delta, q_0, F)$, the set Q_r of *reachable* (or *accessible*) states is defined by

$$Q_r = \{p \in Q \mid (\exists u \in \Sigma^*)(p = \delta^*(q_0, u))\}.$$

The set Q_r consists of those states $p \in Q$ such that there is some path from q_0 to p (along some string u).

Computing the set Q_r is a reachability problem in a directed graph. There are various algorithms to solve this problem, including breadth-first search or depth-first search. They all run in in polynomial time (in the size of the graph).

Once the set Q_r has been computed, we can clean up the DFA D by deleting all redundant states in $Q - Q_r$ and all transitions from these states.

More precisely, we form the DFA defined as follows.

Definition 3.5. Given any DFA $D = (Q, \Sigma, \delta, q_0, F)$, the DFA D_r is defined as $D_r = (Q_r, \Sigma, \delta_r, q_0, Q_r \cap F)$, where $\delta_r \colon Q_r \times \Sigma \to Q_r$ is the restriction of $\delta \colon Q \times \Sigma \to Q$ to Q_r. A DFA D such that $Q = Q_r$ is said to be *trim* (or *reduced*).

It can be shown that $L(D_r) = L(D)$ (see the homework problems). Observe that the DFA D_r is trim. A minimal DFA must be trim.

If D_1 is the DFA of the previous example, then the DFA $(D_1)_r$ is obtained by deleting the states $2, 3, 4$:

	a	b
0	1	0
1	0	1

Computing Q_r gives us a method to test whether a DFA D accepts a nonempty language. Indeed

$$L(D) \neq \emptyset \quad \text{iff} \quad Q_r \cap F \neq \emptyset. \qquad (*_{\text{emptyness}})$$

We now come to the first of several equivalent definitions of the regular languages.

Regular Languages, Version 1

Definition 3.6. A language L is a *regular language* if it is accepted by some DFA.

Note that a regular language may be accepted by many different DFAs. Later on, we will investigate how to find minimal DFA's.

For a given regular language L, a minimal DFA for L is a DFA with the smallest number of states among all DFA's accepting L. A minimal DFA for L must exist since every nonempty subset of natural numbers has a smallest element.

In order to understand how complex the regular languages are, we will investigate the closure properties of the regular languages under union, intersection, complementation, concatenation, and Kleene $*$. It turns out that the family of regular languages is closed under all these operations. For union, intersection, and complementation, we can use the cross-product construction which preserves determinism.

However, for concatenation and Kleene $*$, there does not appear to be any method involving DFA's only. The way to do it is to introduce nondeterministic finite automata (NFA's), which we do a little later.

3.2 The "Cross-product" Construction

Let $\Sigma = \{a_1, \ldots, a_m\}$ be an alphabet.

Given any two DFA's $D_1 = (Q_1, \Sigma, \delta_1, q_{0,1}, F_1)$ and $D_2 = (Q_2, \Sigma, \delta_2, q_{0,2}, F_2)$, there is a very useful construction for showing that the union, the intersection, or the relative complement of regular languages, is a regular language.

Given any two languages L_1, L_2 over Σ, recall that

$$\begin{aligned} L_1 \cup L_2 &= \{w \in \Sigma^* \mid w \in L_1 \quad \text{or} \quad w \in L_2\}, \\ L_1 \cap L_2 &= \{w \in \Sigma^* \mid w \in L_1 \quad \text{and} \quad w \in L_2\}, \\ L_1 - L_2 &= \{w \in \Sigma^* \mid w \in L_1 \quad \text{and} \quad w \notin L_2\}. \end{aligned}$$

Let us first explain how to constuct a DFA accepting the intersection $L_1 \cap L_2$. Let D_1 and D_2 be DFA's such that $L_1 = L(D_1)$ and $L_2 = L(D_2)$. The idea is to construct a DFA *simulating D_1 and D_2 in parallel*. This can be done by using states which are pairs $(p_1, p_2) \in Q_1 \times Q_2$.

Thus, we define the DFA D as follows:

$$D = (Q_1 \times Q_2, \Sigma, \delta, (q_{0,1}, q_{0,2}), F_1 \times F_2),$$

where the transition function $\delta\colon (Q_1 \times Q_2) \times \Sigma \to Q_1 \times Q_2$ is defined as follows:

$$\delta((p_1, p_2),\, a) = (\delta_1(p_1, a),\, \delta_2(p_2, a)),$$

for all $p_1 \in Q_1$, $p_2 \in Q_2$, and $a \in \Sigma$.

Clearly, D is a DFA, since D_1 and D_2 are. Also, by the definition of δ, it is immediately shown by induction on $|w|$ that we have

$$\delta^*((p_1, p_2),\, w) = (\delta_1^*(p_1, w),\, \delta_2^*(p_2, w)),$$

for all $p_1 \in Q_1$, $p_2 \in Q_2$, and $w \in \Sigma^*$.

The choice of $F_1 \times F_2$ for the final states is motivated by the fact that a string w belongs to the intersection language $L(D_1) \cap L(D_2)$ iff w is accepted by D_1 *and* w is accepted by D_2 iff the path in D_1 from $q_{0,1}$ on input w ends with a state in F_1 and if the path in D_2 from $q_{0,2}$ on input w ends with a state in F_2. To prove rigorously that D accepts $L(D_1) \cap L(D_2)$ we proceed as follows.

Now for every $w \in \Sigma^*$, we have $w \in L(D_1) \cap L(D_2)$

$$\begin{aligned} &\text{iff} \quad w \in L(D_1) \text{ and } w \in L(D_2), \\ &\text{iff} \quad \delta_1^*(q_{0,1}, w) \in F_1 \text{ and } \delta_2^*(q_{0,2}, w) \in F_2, \\ &\text{iff} \quad (\delta_1^*(q_{0,1}, w), \delta_2^*(q_{0,2}, w)) \in F_1 \times F_2, \\ &\text{iff} \quad \delta^*((q_{0,1}, q_{0,2}),\, w) \in F_1 \times F_2, \\ &\text{iff} \quad w \in L(D). \end{aligned}$$

Thus $L(D) = L(D_1) \cap L(D_2)$, and our construction is correct.

We can now modify D very easily to accept $L(D_1) \cup L(D_2)$. We change the set of final states so that it becomes $(F_1 \times Q_2) \cup (Q_1 \times F_2)$. The choice of $(F_1 \times Q_2) \cup (Q_1 \times F_2)$ for the final states is motivated by the fact that a string w belongs to the union language $L(D_1) \cup L(D_2)$ iff w is accepted by D_1 *or* w is accepted by D_2 iff the path in D_1 from $q_{0,1}$ on input w ends with a state in F_1 *or* if the path in D_2 from $q_{0,2}$ on input w ends with a state in F_2. But if the path in D_1 from $q_{0,1}$ on input w ends with a state in F_1, then we don't care where we end in D_2, so we let the set of ending states in D_2 be the entire set Q_2, so acceptance in D_1 corresponds to ending in $F_1 \times Q_2$. Similarly, if the path in D_2 from $q_{0,2}$ on input w ends with a state in F_2, then we don't care where we end in D_1, so we let the set of ending states in D_1 be the entire set Q_1, so acceptance in D_2 corresponds to ending in $Q_1 \times F_2$. To prove rigorously that D accepts $L(D_1) \cup L(D_2)$ we proceed as follows.

For all $w \in \Sigma^*$, we have $w \in L(D_1) \cup L(D_2)$

$$\begin{aligned}
&\text{iff} \quad w \in L(D_1) \text{ or } w \in L(D_2),\\
&\text{iff} \quad \delta_1^*(q_{0,1}, w) \in F_1 \text{ or } \delta_2^*(q_{0,2}, w) \in F_2,\\
&\text{iff} \quad (\delta_1^*(q_{0,1}, w), \delta_2^*(q_{0,2}, w)) \in (F_1 \times Q_2) \cup (Q_1 \times F_2),\\
&\text{iff} \quad \delta^*((q_{0,1}, q_{0,2}),\, w) \in (F_1 \times Q_2) \cup (Q_1 \times F_2),\\
&\text{iff} \quad w \in L(D).
\end{aligned}$$

Thus $L(D) = L(D_1) \cup L(D_2)$, and our construction is correct.

We can also modify D very easily to accept $L(D_1) - L(D_2)$. We change the set of final states so that it becomes $F_1 \times (Q_2 - F_2)$.

The choice of $F_1 \times (Q_2 - F_2)$ for the final states is motivated by the fact that a string w belongs to the relative complement language $L(D_1) - L(D_2)$ iff w is *accepted* by D_1 *and* w is *rejected* by D_2 iff the path in D_1 from $q_{0,1}$ on input w ends with a state in F_1 *and* if the path in D_2 from $q_{0,2}$ on input w *does not end* with a state in F_2. Equivalently, the path in D_1 from $q_{0,1}$ on input w ends with a state in F_1 *and* the path in D_2 from $q_{0,2}$ on input w ends with a state in $Q_2 - F_2$. To prove rigorously that D accepts $L(D_1) - L(D_2)$ we proceed as follows.

For all $w \in \Sigma^*$, we have $w \in L(D_1) - L(D_2)$

$$\begin{aligned}
&\text{iff} \quad w \in L(D_1) \text{ and } w \notin L(D_2),\\
&\text{iff} \quad \delta_1^*(q_{0,1}, w) \in F_1 \text{ and } \delta_2^*(q_{0,2}, w) \notin F_2,\\
&\text{iff} \quad (\delta_1^*(q_{0,1}, w), \delta_2^*(q_{0,2}, w)) \in F_1 \times (Q_2 - F_2),\\
&\text{iff} \quad \delta^*((q_{0,1}, q_{0,2}),\, w) \in F_1 \times (Q_2 - F_2),\\
&\text{iff} \quad w \in L(D).
\end{aligned}$$

Thus $L(D) = L(D_1) - L(D_2)$, and our construction is correct.

In all cases, if D_1 has n_1 states and D_2 has n_2 states, the DFA D has $n_1 n_2$ states.

Example 3.4. Let $\Sigma = \{a, b\}$. Consider the languages

$$L_1 = \{w \in \Sigma^* \mid w \text{ contains an odd number of } b\text{'s}\}$$

and

$$L_2 = \{w \in \Sigma^* \mid w \text{ contains a number of } a\text{'s divisible by } 3\}.$$

The language L_1 is accepted by the DFA shown in Figure 3.4 and the language L_2 is accepted by the DFA shown in Figure 3.5

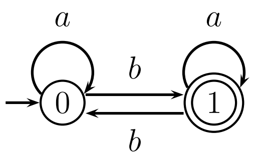


Figure 3.4: DFA for L_1.

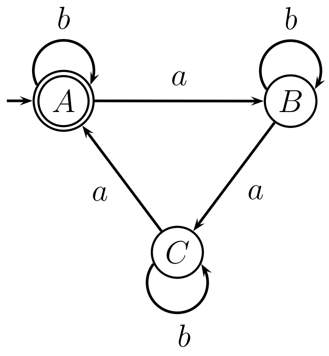


Figure 3.5: DFA for L_2.

The DFA accepting $L_3 = L_1 \cup L_2$ obtained by appying cross-product construction to D_1 and D_2 has the following transition table

	a	b
$(0, A)$	$(0, B)$	$(1, A)$
$(0, B)$	$(0, C)$	$(1, B)$
$(0, C)$	$(0, A)$	$(1, C)$
$(1, A)$	$(1, B)$	$(0, A)$
$(1, B)$	$(1, C)$	$(0, B)$
$(1, C)$	$(1, A)$	$(0, C)$

The final states are: $(0, A), (1, A), (1, B), (1, C)$ and the start state is $(0, A)$. The cross-product DFA is shown in Figure 3.6.

As an application of the cross-product construction we show how to solve the following important problem.

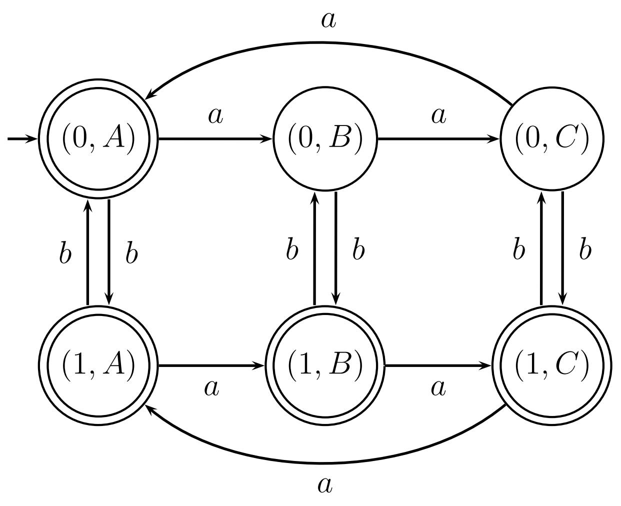


Figure 3.6: DFA for $L_1 \cup L_2$.

Definition 3.7. The *equivalence problem for DFA's* is the following problem: given some alphabet Σ, is there an algorithm which takes as input any two DFA's D_1 and D_2 and decides whether $L(D_1) = L(D_2)$.

Now $L(D_1) \neq L(D_2)$ if either some string $u \in \Sigma^*$ is accepted by D_1 and rejected by D_2, or some string $v \in \Sigma^*$ is accepted by D_2 and rejected by D_1. So if we enumerate all strings in Σ^* using the method of the section on countable and uncountable sets, eventually some u or some v as above will show up and we will know that $L(D_1) \neq L(D_2)$, but the problem is that we know of no upper bound on the length of u or v.

To solve our problem we make use of the following fact: given any two sets X and Y,

$$X = Y \quad \text{iff} \quad X - Y = \emptyset \text{ and } Y - X = \emptyset.$$

Applying the above fact to $X = L(D_1)$ and $Y = L(D_2)$, we get $L(D_1) = L(D_2)$ iff $L(D_1) - L(D_2) = \emptyset$ and $L(D_2) - L(D_1) = \emptyset$. But we just saw that the cross-product construction (for relative complement) yields two DFA's D_{12} and D_{21} such that $L(D_{12}) = L(D_1) - L(D_2)$ and $L(D_{21}) = L(D_2) - L(D_1)$, so we get

$$L(D_1) = L(D_2) \quad \text{iff} \quad L(D_{12}) = \emptyset \text{ and } L(D_{21}) = \emptyset.$$

The problem is reduced to testing whether a DFA does not accept any string, that is, $L(D) = \emptyset$. But we solved this problem before. Indeed, we know from $(*_{\text{emptyness}})$ that if Q_r is the set of reachable states of D, then $L(D) = \emptyset$ iff $Q_r \cap F = \emptyset$. Therefore, $L(D_{12}) = \emptyset$ iff $(Q_{12})_r \cap (F_1 \times \overline{F_2}) = \emptyset$, and $L(D_{21}) = \emptyset$ iff $(Q_{21})_r \cap (F_2 \times \overline{F_1}) = \emptyset$, where $(Q_{12})_r$ is the set of states reachable from $(q_{0,1}, q_{0,2})$ in the DFA's D_{12}, and $(Q_{21})_r$ is the set of states reachable from $(q_{0,2}, q_{0,1})$ in the DFA's D_{21}. But by definition of the cross-product, testing whether $(Q_{21})_r \cap (F_2 \times \overline{F_1}) = \emptyset$ is equivalent to testing whether $(Q_{12})_r \cap (\overline{F_1} \times F_2) = \emptyset$, so

$$L(D_1) = L(D_2) \quad \text{iff} \quad (Q_{12})_r \cap (F_1 \times \overline{F_2}) = \emptyset \quad \text{and} \quad (Q_{12})_r \cap (\overline{F_1} \times F_2) = \emptyset.$$

Therefore, we obtained an algorithm for deciding whether $L(D_1) = L(D_2)$ using the cross-product construction and reducing the problem to two reachability problems in the graph associated with D_{12}. This algorithm runs in time polynomial in $n_1 n_2$, where $n_1 = |Q_1|$ and $n_2 = |Q_2|$. This is not a pretty good algorithm, but there are faster algorithms based on methods for testing state equivalence, as we will see later.

Given a DFA $D = (Q, \Sigma, \delta, q_0, F)$, informally, two states $p, q \in Q$ are *equivalent*, written $p \equiv q$, if they have the same acceptance/rejection behavior. This means that if we make two copies D_p and D_q of D and if we view p as the start state of D_p and q as the start state of D_q, then any string $w \in \Sigma^*$ is accepted by D_p iff it is accepted by D_q. We can make this precise by setting

$$D_p = (Q, \Sigma, \delta, p, F), \qquad D_q = (Q, \Sigma, \delta, q, F)$$

(note how in D_p, the old start state q_0 is replaced by the new start state p, and in D_q, the old start state q_0 is replaced by the new start state q), and then

$$p, q \in Q \text{ are equivalent iff } \quad L(D_p) = L(D_q).$$

Our method for deciding whether $L(D_p) = L(D_q)$ yields an algorithm for testing state equivalence, but this is a rather inefficient method and there are much better methods discussed in Section 6.3. Nevertheless, it can be shown that if $p \equiv q$, then we can construct a smaller DFA by *merging p and q* and also merging the transitions in and out of p and q. By repeating this process, we will ultimately obtain a minimal DFA. Actually, it is better to find the equivalence classes of states under state equivalence, and then merge *all* states in each equivalence class. It is by no means obvious that this process is correct and that we get a minimal DFA, but it is, as we will see in Section 6.3.

3.3 Nondeteterministic Finite Automata (NFA's)

NFA's are obtained from DFA's by allowing multiple transitions from a given state on a given input. This can be done by defining $\delta(p, a)$ as a **subset** of Q rather than a single state. It will also be convenient to allow transitions on input ϵ.

We let 2^Q denote the set of all subsets of Q, including the empty set. The set 2^Q is the *power set* of Q.

Example 3.5. A NFA for the language

$$L_3 = \{a, b\}^*\{abb\}.$$

Input alphabet: $\Sigma = \{a, b\}$.

State set $Q_4 = \{0, 1, 2, 3\}$.

Start state: 0.

Set of accepting states: $F_4 = \{3\}$.

Transition table δ_4:

	a	b
0	$\{0,1\}$	$\{0\}$
1	$\emptyset$	$\{2\}$
2	$\emptyset$	$\{3\}$
3	$\emptyset$	$\emptyset$

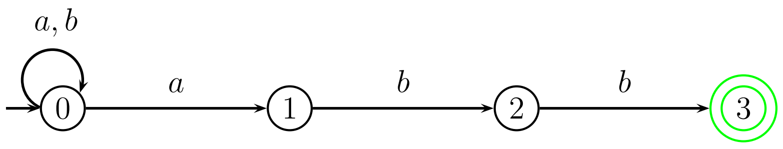


Figure 3.7: NFA for $\{a,b\}^*\{abb\}$.

Example 3.6. Let $\Sigma = \{a_1, \ldots, a_n\}$, with $n \geq 2$, let

$$L_n^i = \{w \in \Sigma^* \mid w \text{ contains an odd number of } a_i\text{'s}\},$$

and let

$$L_n = L_n^1 \cup L_n^2 \cup \cdots \cup L_n^n.$$

The language L_n consists of those strings in Σ^* that contain an odd number of some letter $a_i \in \Sigma$. Equivalently $\Sigma^* - L_n$ consists of those strings in Σ^* with an even number of *every* letter $a_i \in \Sigma$.

It can be shown that every DFA accepting L_n has at least 2^n states. However, there is an NFA with $2n + 1$ states accepting L_n.

We define NFA's as follows.

Definition 3.8. A *nondeterministic finite automaton (or NFA)* is a quintuple $N = (Q, \Sigma, \delta, q_0, F)$, where

- Σ is a finite *input alphabet*;
- Q is a finite set of *states*;
- F is a subset of Q of *final (or accepting) states*;
- $q_0 \in Q$ is the *start state (or initial state)*;
- δ is the *transition function*, a function

$$\delta \colon Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q.$$

For any state $p \in Q$ and any input $a \in \Sigma \cup \{\epsilon\}$, the set of states $\delta(p, a)$ is uniquely determined. We write $q \in \delta(p, a)$.

Given an NFA $N = (Q, \Sigma, \delta, q_0, F)$, we would like to define the language accepted by N. However, given an NFA N, unlike the situation for DFA's, given a state $p \in Q$ and some input $w \in \Sigma^*$, in general *there is no unique path from p on input w, but instead a tree of computation paths*.

Example 3.7. Given the NFA shown below,

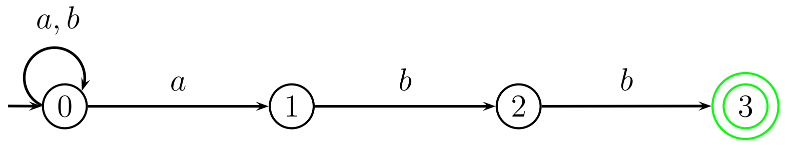


Figure 3.8: NFA for $\{a, b\}^*\{abb\}$.

from state 0 on input $w = ababb$ we obtain the following tree of computation paths:

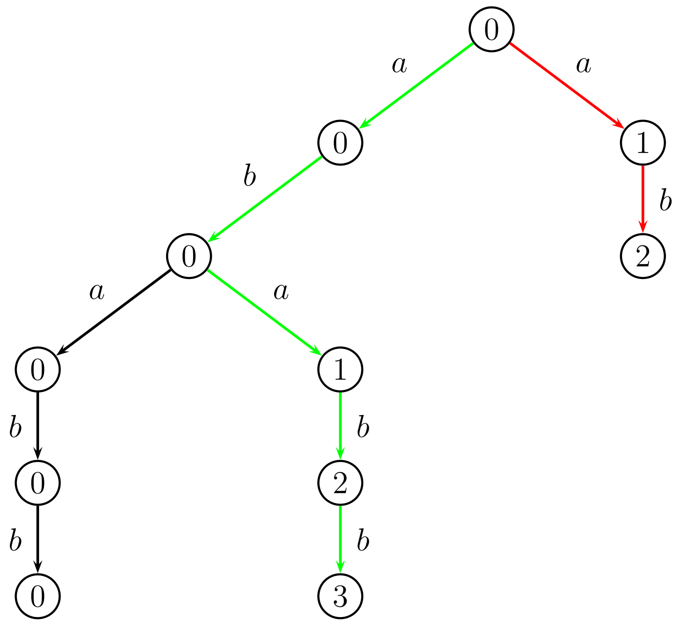


Figure 3.9: A tree of computation paths on input *ababb*.

Observe that there are three kinds of computation paths:

1. A path on input w ending in a rejecting state (for example, the lefmost path).

2. A path on some proper prefix of w, along which the computation gets stuck (for example, the rightmost path).

3. A path on input w ending in an accepting state (such as the path ending in state 3).

The acceptance criterion for NFA is *very lenient*: a string w is accepted iff the tree of computation paths contains *some accepting path* (of type (3)). Thus, all failed paths of type (1) and (2) are ignored. Furthermore, there is *no charge* for failed paths.

A string w is rejected iff all computation paths are failed paths of type (1) or (2). The "philosophy" of nondeterminism is that an NFA "guesses" an accepting path and then checks it in polynomial time by following this path. We are only charged for one accepting path (even if there are several accepting paths).

A way to capture this acceptance policy if to extend the transition function $\delta\colon Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$ to a function

$$\delta^*\colon Q \times \Sigma^* \to 2^Q.$$

The presence of ϵ-transitions (i.e., when $q \in \delta(p, \epsilon)$) causes technical problems, and to overcome these problems, we introduce the notion of ϵ-closure.

3.4 ϵ-Closure

Definition 3.9. Given an NFA $N = (Q, \Sigma, \delta, q_0, F)$ (with ϵ-transitions) for every state $p \in Q$, the *ϵ-closure of p* is set ϵ-closure(p) consisting of all states q such that there is a path from p to q whose spelling is ϵ (an *ϵ-path*). This means that either $q = p$, or that all the edges on the path from p to q have the label ϵ.

We can compute ϵ-closure(p) using a sequence of approximations as follows. Define the sequence of sets of states $(\epsilon\text{-clo}_i(p))_{i \geq 0}$ as follows:

$$\begin{aligned}\epsilon\text{-clo}_0(p) &= \{p\},\\ \epsilon\text{-clo}_{i+1}(p) &= \epsilon\text{-clo}_i(p) \cup \{q \in Q \mid \exists s \in \epsilon\text{-clo}_i(p),\ q \in \delta(s, \epsilon)\}.\end{aligned}$$

Since $\epsilon\text{-clo}_i(p) \subseteq \epsilon\text{-clo}_{i+1}(p)$, $\epsilon\text{-clo}_i(p) \subseteq Q$, for all $i \geq 0$, and Q is finite, it can be shown that

Fact 1. There is a smallest i, say i_0, such that

$$\epsilon\text{-clo}_{i_0}(p) = \epsilon\text{-clo}_{i_0+1}(p).$$

It suffices to show that there is some $i \geq 0$ such that $\epsilon\text{-clo}_i(p) = \epsilon\text{-clo}_{i+1}(p)$, because then there is a smallest such i (since every nonempty subset of $\mathbb{N}$ has a smallest element).

Proof. Assume by contradiction that

$$\epsilon\text{-clo}_i(p) \subset \epsilon\text{-clo}_{i+1}(p) \quad \text{for all } i \geq 0.$$

The symbol $\subset$ means strict inclusion, so $\epsilon\text{-clo}_i(p) \subseteq \epsilon\text{-clo}_{i+1}(p)$ and $\epsilon\text{-clo}_i(p) \neq \epsilon\text{-clo}_{i+1}(p)$.

I claim that $|\epsilon\text{-clo}_i(p)| \geq i+1$ for all $i \geq 0$. We prove this by induction on i.

This is true for $i = 0$ since $\epsilon\text{-clo}_0(p) = \{p\}$.

For the induction step, since $\epsilon\text{-clo}_i(p) \subset \epsilon\text{-clo}_{i+1}(p)$, there is some $q \in \epsilon\text{-clo}_{i+1}(p)$ that does not belong to $\epsilon\text{-clo}_i(p)$, and since by induction $|\epsilon\text{-clo}_i(p)| \geq i+1$, we get

$$|\epsilon\text{-clo}_{i+1}(p)| \geq |\epsilon\text{-clo}_i(p)| + 1 \geq i+1+1 = i+2,$$

establishing the induction step.

If $n = |Q|$, then $|\epsilon\text{-clo}_n(p)| \geq n+1$, a contradiction.

Therefore, there is indeed some $i \geq 0$ such that $\epsilon\text{-clo}_i(p) = \epsilon\text{-clo}_{i+1}(p)$, and for the least such $i = i_0$, we have $i_0 \leq n-1$. ☐

It can also be shown that

Fact 2.

$$\epsilon\text{-closure}(p) = \epsilon\text{-clo}_{i_0}(p).$$

For this, we prove (by induction on the length of paths) that

1. $\epsilon\text{-clo}_i(p) \subseteq \epsilon\text{-closure}(p)$, for all $i \geq 0$.

2. $\epsilon\text{-closure}(p)_i \subseteq \epsilon\text{-clo}_{i_0}(p)$, for all $i \geq 0$,

where $\epsilon\text{-closure}(p)_i$ is the set of states reachable from p by an ϵ-path of length $\leq i$.

Fact 1 proves that the method terminates and Fact 2 prove that it computes correctly $\epsilon\text{-closure}(p)$ as $\epsilon\text{-clo}_{i_0}(p)$.

When N has no ϵ-transitions, i.e., when $\delta(p, \epsilon) = \emptyset$ for all $p \in Q$ (which means that δ can be viewed as a function $\delta \colon Q \times \Sigma \to 2^Q$), we have

$$\epsilon\text{-closure}(p) = \{p\}.$$

It should be noted that there are more efficient ways of computing $\epsilon\text{-closure}(p)$, for example, using a stack (basically, a kind of depth-first search).

We present such an algorithm below. It is assumed that the types *NFA* and *stack* are defined. If n is the number of states of an NFA N, we let

eclotype = **array**[1..n] **of boolean**

```
function eclosure[$N$: NFA, $p$: integer]: eclotype;
    begin
      var eclo: eclotype, $q, s$: integer, st: stack;
      for each $q \in setstates(N)$ do
        $eclo[q] := false$;
      endfor
      $eclo[p] := true$; $st := empty$;
      $trans := deltatable(N)$;
      $st := push(st, p)$;
      while $st \neq emptystack$ do
        $q = pop(st)$;
        for each $s \in trans(q, \epsilon)$ do
          if $eclo[s] = false$ then
            $eclo[s] := true$; $st := push(st, s)$
          endif
        endfor
      endwhile;
      $eclosure := eclo$
    end
```

This algorithm can be easily adapted to compute the set of states reachable from a given state p (in a DFA or an NFA).

Definition 3.10. Given a subset S of Q, we define ϵ-closure(S) as

$$\epsilon\text{-closure}(S) = \bigcup_{s \in S} \epsilon\text{-closure}(s),$$

with

$$\epsilon\text{-closure}(\emptyset) = \emptyset.$$

When N has no ϵ-transitions, we have

$$\epsilon\text{-closure}(S) = S.$$

We are now ready to define the extension $\delta^* \colon Q \times \Sigma^* \to 2^Q$ of the transition function $\delta \colon Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$ in order to convert an NFA into a DFA.

3.5 Converting an NFA into a DFA

The intuition behind the definition of the extended transition function is that $\delta^*(p, w)$ is the set of all states reachable from p by a path whose spelling is w.

Definition 3.11. Given an NFA $N = (Q, \Sigma, \delta, q_0, F)$ (with ϵ-transitions), the *extended transition function* $\delta^* \colon Q \times \Sigma^* \to 2^Q$ is defined as follows: for every $p \in Q$, every $u \in \Sigma^*$, and every $a \in \Sigma$,

$$\delta^*(p, \epsilon) = \epsilon\text{-closure}(\{p\}),$$
$$\delta^*(p, ua) = \epsilon\text{-closure}\left(\bigcup_{s \in \delta^*(p,u)} \delta(s, a)\right).$$

In the second equation, if $\delta^*(p, u) = \emptyset$, then

$$\delta^*(p, ua) = \emptyset.$$

The *language $L(N)$ accepted by an NFA N* is the set

$$L(N) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \cap F \neq \emptyset\}.$$

Observe that the definition of $L(N)$ conforms to the lenient acceptance policy: a string w is accepted iff $\delta^*(q_0, w)$ contains *some final state*. Also, since $\delta^*(q_0, \epsilon) = \epsilon\text{-closure}(\{q_0\})$, the empty string is accepted iff some state in $\epsilon\text{-closure}(\{q_0\})$ is a final state.

The function δ^* satisfies the following property which generalizes a familiar property of δ^* when N is a DFA.

Proposition 3.1. *Given any NFA $N = (Q, \Sigma, \delta, q_0, F)$, for any state $p \in Q$ and for any two strings $u, v \in \Sigma^*$, we have*

$$\delta^*(p, uv) = \bigcup_{s \in \delta^*(p,u)} \delta^*(s, v).$$

Proof. We proceed by induction on the length of v. First, it is shown immediately by the definition of ϵ-closure that for any subset $S \subseteq Q$, we have

$$\epsilon\text{-closure}(\epsilon\text{-closure}(S)) = \epsilon\text{-closure}(S).$$

As subset $S \subseteq Q$ such that $\epsilon\text{-closure}(S) = S$ is said to be *ϵ-closed*. Observe that by definition, $\delta^*(p, w)$ is ϵ-closed for all $p \in Q$ and all $w \in \Sigma^*$.

Consider the base case $v = \epsilon$. We have

$$\begin{aligned}\bigcup_{s \in \delta^*(p,u)} \delta^*(s, \epsilon) &= \bigcup_{s \in \delta^*(p,u)} (\epsilon\text{-closure}(\{s\}) \\ &= \epsilon\text{-closure}\left(\delta^*(p, u)\right) \\ &= \delta^*(p, u),\end{aligned}$$

as desired.

For the induction step, assume $v = wa$, for some $w \in \Sigma^*$ and some $a \in \Sigma$. By the induction hypothesis,

$$\delta^*(p, uw) = \bigcup_{s \in \delta^*(p,u)} \delta^*(s, w).$$

Then we have

$$\begin{aligned}
\delta^*(p, uwa) &= \epsilon\text{-closure}\Bigg(\bigcup_{q \in \delta^*(p,uw)} \delta(q, a) \Bigg) \\
&= \epsilon\text{-closure}\Bigg(\bigcup_{s \in \delta^*(p,u)} \bigcup_{q \in \delta^*(s,w)} \delta(q, a) \Bigg) \\
&= \bigcup_{s \in \delta^*(p,u)} \epsilon\text{-closure}\Bigg(\bigcup_{q \in \delta^*(s,w)} \delta(q, a) \Bigg) \\
&= \bigcup_{s \in \delta^*(p,u)} \delta^*(s, wa),
\end{aligned}$$

proving the induction step. $\square$

In order to show how to convert an NFA to a DFA we also extend $\delta^* \colon Q \times \Sigma^* \to 2^Q$ to a function

$$\widehat{\delta} \colon 2^Q \times \Sigma^* \to 2^Q$$

defined as follows:

Definition 3.12. For every subset S of Q, for every $w \in \Sigma^*$,

$$\widehat{\delta}(S, w) = \bigcup_{s \in S} \delta^*(s, w),$$

with

$$\widehat{\delta}(\emptyset, w) = \emptyset.$$

Let $\mathcal{Q}$ be the subset of 2^Q consisting of those subsets S of Q that are *ϵ-closed*, i.e., such that

$$S = \epsilon\text{-closure}(S).$$

We have the following version of Proposition 3.1 for $\widehat{\delta}$.

Proposition 3.2. *Given any NFA $N = (Q, \Sigma, \delta, q_0, F)$, for any subset $S \subseteq Q$ and for any two strings $u, v \in \Sigma^*$, we have*

$$\widehat{\delta}(S, uv) = \widehat{\delta}(\widehat{\delta}(S, u), v).$$

Proof. Using Proposition 3.1 and the definition of $\widehat{\delta}$, we have

$$\begin{aligned}
\widehat{\delta}(\widehat{\delta}(S, u), v) &= \bigcup_{p \in \widehat{\delta}(S,u)} \delta^*(p, v) \\
&= \bigcup_{s \in S} \bigcup_{p \in \delta^*(s,u)} \delta^*(p, v) \\
&= \bigcup_{s \in S} \delta^*(s, uv) \\
&= \widehat{\delta}(S, uv),
\end{aligned}$$

as claimed. ☐

If we consider the restriction

$$\Delta \colon \mathcal{Q} \times \Sigma \to \mathcal{Q}$$

of $\widehat{\delta} \colon 2^Q \times \Sigma^* \to 2^Q$ to $\mathcal{Q}$ and Σ, we observe that *Δ is the transition function of a DFA*.

Indeed, this is the transition function of a DFA accepting $L(N)$. It is easy to show that Δ is defined directly as follows (on subsets S in $\mathcal{Q}$):

$$\Delta(S, a) = \epsilon\text{-closure}\left(\bigcup_{s \in S} \delta(s, a)\right),$$

with

$$\Delta(\emptyset, a) = \emptyset.$$

Definition 3.13. The DFA D corresponding to N is defined as follows:

$$D = (\mathcal{Q}, \Sigma, \Delta, \epsilon\text{-closure}(\{q_0\}), \mathcal{F}),$$

where $\mathcal{F} = \{S \in \mathcal{Q} \mid S \cap F \neq \emptyset\}$ and

$$\Delta(S, a) = \epsilon\text{-closure}\left(\bigcup_{s \in S} \delta(s, a)\right),$$

with

$$\Delta(\emptyset, a) = \emptyset.$$

Proposition 3.3. *The DFA D of Definition 3.13 has the property that $L(D) = L(N)$, that is, D is a DFA accepting $L(N)$.*

Proof. To prove the proposition, we show that

$$\Delta^*(S, w) = \widehat{\delta}(S, w) \quad \text{for all } S \in \mathcal{Q} \text{ and all } w \in \Sigma^* \qquad (\Delta)$$

by induction on $|w|$.

Proof of Equation (Δ). For the base case $w = \epsilon$, by definition of $\widehat{\delta}$ we have

$$\widehat{\delta}(S, \epsilon) = \bigcup_{s \in S} \delta^*(s, \epsilon) = \bigcup_{s \in S} \epsilon\text{-closure}(\{s\}) = \epsilon\text{-closure}(S) = S,$$

since S is ϵ-closed, and of course by definition $\Delta^*(S, \epsilon) = S$, so

$$\Delta^*(S, \epsilon) = \widehat{\delta}(S, \epsilon).$$

For the induction step, using the induction hypothesis $\Delta^*(S, u) = \widehat{\delta}(S, u)$ and the fact that Δ is the restriction of $\widehat{\delta}$ to Σ (and $\mathcal{Q}$), using Proposition 3.2, we have

$$\begin{aligned}\Delta^*(S, ua) &= \Delta(\Delta^*(S, u), a) \\ &= \widehat{\delta}(\widehat{\delta}(S, u), a) \\ &= \widehat{\delta}(S, ua),\end{aligned}$$

proving the induction step. □

Then, for any $w \in \Sigma^*$, we have

$$\begin{aligned}\Delta^*(\epsilon\text{-closure}(\{q_0\}), w) &= \widehat{\delta}(\epsilon\text{-closure}(\{q_0\}), w) \\ &= \bigcup_{p \in \epsilon\text{-closure}(\{q_0\})} \delta^*(p, w) \\ &= \bigcup_{p \in \delta^*(q_0, \epsilon)} \delta^*(p, w).\end{aligned}$$

By Proposition 3.1 applied to $u = \epsilon$, $v = w$, and $p = q_0$, we get

$$\bigcup_{p \in \delta^*(q_0, \epsilon)} \delta^*(p, w) = \delta^*(q_0, w),$$

so we obtain

$$\Delta^*(\epsilon\text{-closure}(\{q_0\}), w) = \delta^*(q_0, w). \tag{$*_\Delta$}$$

By the choice of final states of D ($\mathcal{F} = \{S \in \mathcal{Q} \mid S \cap F \neq \emptyset\}$), we have $w \in L(D)$ iff $\Delta^*(\epsilon\text{-closure}(\{q_0\}), w) \in \mathcal{F}$ iff $\delta^*(q_0, w) \in \mathcal{F}$ iff $\delta^*(q_0, w) \cap F \neq \emptyset$ (since $\delta^*(q_0, w) \in \mathcal{Q}$) iff $w \in L(N)$. Therefore $L(D) = L(N)$. □

Thus, we have converted the NFA N into a DFA D (and gotten rid of ϵ-transitions).

Since DFA's are special NFA's, the subset construction shows that DFA's and NFA's accept *the same* family of languages, the *regular languages, version 1* (although not with the same complexity).

The states of the DFA D equivalent to N are ϵ-closed subsets of Q. For this reason, the above construction is often called the *subset construction*.

This construction is due to Michael Rabin and Dana Scott. Michael Rabin and Dana Scott were awarded the prestigious *Turing Award* in 1976 for this important contribution and many others.

Note that among the Turing award winners, Dijsktra received the Turing Award in 1972, Donald Knuth in 1974, John Backus in 1977, Steve Cook in 1982, Richard Karp in 1985, John Hopcroft and Robert André Tarjan in 1986, and Leslie Lamport in 2013.

Although theoretically fine, the method may construct useless sets S that are not reachable from the start state ϵ-closure($\{q_0\}$). A more economical construction is given next.

An Algorithm to convert an NFA into a DFA: The "subset construction"

Given an input NFA $N = (Q, \Sigma, \delta, q_0, F)$, a DFA $D = (K, \Sigma, \Delta, S_0, \mathcal{F})$ is constructed. It is assumed that K is a linear array of sets of states $S \subseteq Q$, and Δ is a 2-dimensional array, where $\Delta[i, a]$ is the index of the target state of the transition from $K[i] = S$ on input a, with $S \in K$, and $a \in \Sigma$.

```
$S_0 := \epsilon$-closure($\{q_0\}$); $total := 1$; $K[1] := S_0$;
$marked := 0$;
**while** $marked < total$ **do**;
  $marked := marked + 1$; $S := K[marked]$;
  **for each** $a \in \Sigma$ **do**
    $U := \bigcup_{s \in S} \delta(s, a)$; $T := \epsilon$-closure($U$);
    **if** $T \notin K$ **then**
      $total := total + 1$; $K[total] := T$
    **endif**;
    $\Delta[marked, a] :=$ index($T$)
  **endfor**
**endwhile**;
$\mathcal{F} := \{S \in K \mid S \cap F \neq \emptyset\}$
```

Let us illustrate the subset construction on the NFA of Example 3.5.

Example 3.8. A NFA for the language

$$L_3 = \{a, b\}^* \{abb\}$$

is given by the transition table δ_4 below:

	a	b
0	$\{0,1\}$	$\{0\}$
1	$\emptyset$	$\{2\}$
2	$\emptyset$	$\{3\}$
3	$\emptyset$	$\emptyset$

The set of accepting states is $F_4 = \{3\}$.

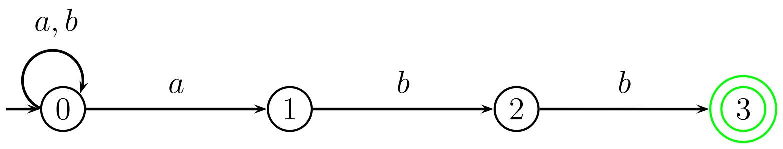


Figure 3.10: NFA for $\{a, b\}^*\{abb\}$

.

Here is the sequence of snapshots obtained by running the algorithm for converting an NFA into a DFA The pointer $\Rightarrow$ corresponds to *marked* and the pointer $\rightarrow$ to *total*.

Initial transition table Δ.
Start state $A = \epsilon$-closure$(\{0\}) = \{0\}$.

$\Rightarrow$	index	states	a	b
$\rightarrow$	A	$\{0\}$		

Just after entering the while loop

	index	states	a	b
$\Rightarrow\rightarrow$	A	$\{0\}$		

$S = \{0\}$.
$\bigcup_{s\in\{0\}} \delta(s, a) = \delta(0, a) = \{0, 1\}$; new state $B = \{0, 1\}$.
$\bigcup_{s\in\{0\}} \delta(s, b) = \delta(0, b) = \{0\} = A$.
After the first round through the while loop.

	index	states	a	b
$\Rightarrow$	A	$\{0\}$	B	A
$\rightarrow$	B	$\{0,1\}$		

After just reentering the while loop.

	index	states	a	b
	A	$\{0\}$	B	A
$\Rightarrow\rightarrow$	B	$\{0,1\}$		

$S = \{0, 1\}$.
$\bigcup_{s\in\{0,1\}} \delta(s, a) = \delta(0, a) \cup \delta(1, a) = \{0, 1\} \cup \emptyset = \{0, 1\} = B$.
$\bigcup_{s\in\{0,1\}} \delta(s, b) = \delta(0, b) \cup \delta(1, b) = \{0\} \cup \{2\} = \{0, 2\}$; new state $C = \{0, 2\}$.
After the second round through the while loop.

	index	states	a	b
	A	$\{0\}$	B	A
$\Rightarrow$	B	$\{0,1\}$	B	C
$\rightarrow$	C	$\{0,2\}$		

$S = \{0, 2\}$.
$\bigcup_{s\in\{0,2\}} \delta(s, a) = \delta(0, a) \cup \delta(2, a) = \{0, 1\} \cup \emptyset = \{0, 1\} = B$.
$\bigcup_{s\in\{0,2\}} \delta(s, b) = \delta(0, b) \cup \delta(2, b) = \{0\} \cup \{3\} = \{0, 3\}$; new state $D = \{0, 3\}$.
After the third round through the while loop.

	index	states	a	b
	A	$\{0\}$	B	A
	B	$\{0,1\}$	B	C
$\Rightarrow$	C	$\{0,2\}$	B	D
$\rightarrow$	D	$\{0,3\}$		

$S = \{0, 3\}$.
$\bigcup_{s\in\{0,3\}} \delta(s, a) = \delta(0, a) \cup \delta(3, a) = \{0, 1\} \cup \emptyset = \{0, 1\} = B$.
$\bigcup_{s\in\{0,3\}} \delta(s, b) = \delta(0, b) \cup \delta(3, b) = \{0\} \cup \emptyset = \{0\} = A$.
After the fourth round through the while loop.

	index	states	a	b
	A	$\{0\}$	B	A
	B	$\{0,1\}$	B	C
	C	$\{0,2\}$	B	D
$\Rightarrow\rightarrow$	D	$\{0,3\}$	B	A

This is the DFA of Figure 3.3, except that in that example A, B, C, D are renamed $0, 1, 2, 3$.

Here is another example invoving an ϵ-transition.

Example 3.9. Consider the language $L = \{aa, bb\}^*$. The transition table and the transition graph of an NFA with a single ϵ-transition accepting $L = \{aa, bb\}^*$ are shown below.

	ϵ	a	b
0	$\emptyset$	1	2
1	$\emptyset$	3	$\emptyset$
2	$\emptyset$	$\emptyset$	3
3	0	$\emptyset$	$\emptyset$

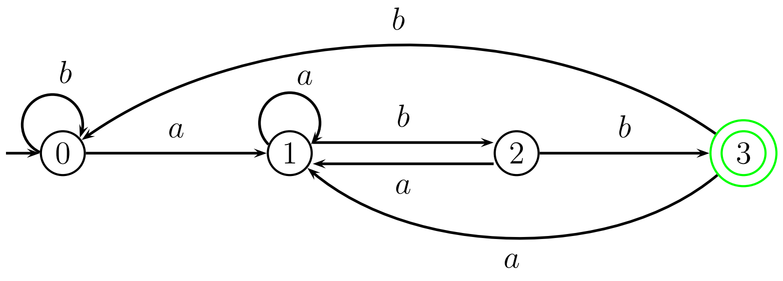


Figure 3.11: DFA for $\{a, b\}^*\{abb\}$.

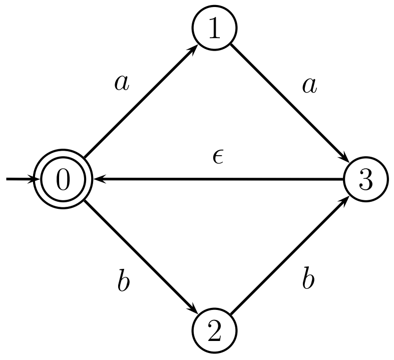


Figure 3.12: NFA for $L = \{aa, bb\}^*$.

The result of applying the subset construction to the above NFA we obtain the five state DFA with the transition table and graph shown in Figure 3.13.

	subsets	a	b
A	$\{0\}$	B	C
B	$\{1\}$	D	E
C	$\{2\}$	E	D
D	$\{0, 3\}$	B	C
E	$\emptyset$	E	E

The final states are A and D and the start state is A.

The next example requires computing bigger ϵ-closures.

Example 3.10. Consider the NFA with ϵ-transitions accepting $L = \{a, b\}^*\{abb\}$ shown in Figure 3.14.

The result of applying the subset constructions to the NFA shown in Figure 3.14 is the DFA whose transition table is shown below:

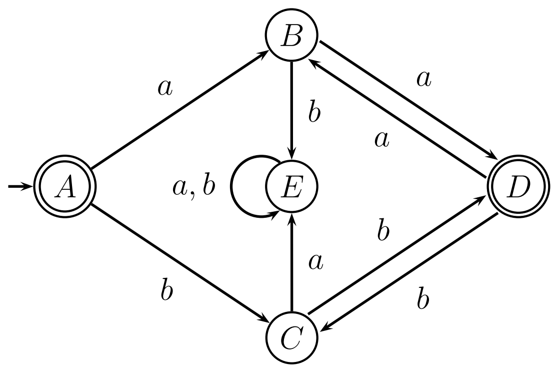


Figure 3.13: DFA for $L = \{aa, bb\}^*$.

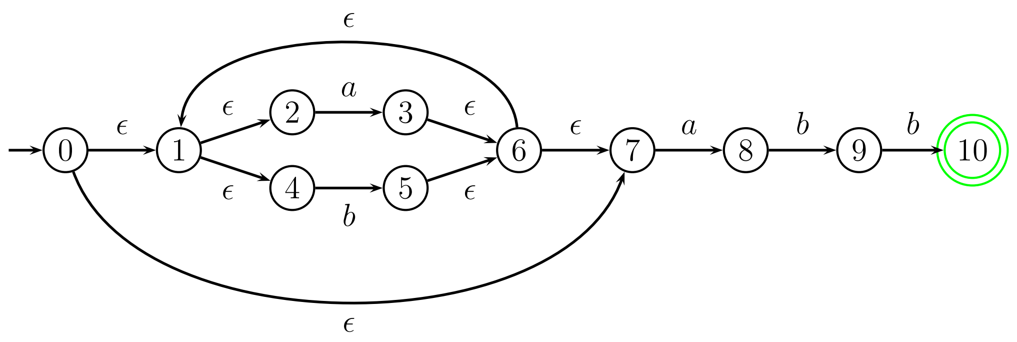


Figure 3.14: An NFA for $L = \{a, b\}^*\{abb\}$.

	subsets	a	b
A	$\{0, 1, 2, 4, 7\}$	B	C
B	$\{1, 2, 3, 4, 6, 7, 8\}$	B	D
C	$\{1, 2, 4, 5, 6, 7\}$	B	C
D	$\{1, 2, 4, 5, 6, 7, 9\}$	B	E
E	$\{1, 2, 4, 5, 6, 7, 10\}$	B	C

We have the following steps. The start state A is ϵ-closure$(\{0\}) = \{0, 1, 2, 4, 7\}$.
We have $U = \bigcup_{s \in A} \delta(s, a) = \emptyset \cup \emptyset \cup \delta(2, a) \cup \emptyset \cup \delta(7, a) = \{3\} \cup \{8\} = \{3, 8\}$. Then

$$\begin{aligned} T &= \epsilon\text{-closure}(U) = \epsilon\text{-closure}(\{3, 8\}) = \epsilon\text{-closure}(\{3\}) \cup \epsilon\text{-closure}(\{8\}) \\ &= \{3, 6, 7, 1, 2, 4\} \cup \{8\} = \{1, 2, 3, 4, 6, 7, 8\} = B. \end{aligned}$$

We have $U = \bigcup_{s \in A} \delta(s, b) = \emptyset \cup \emptyset \cup \emptyset \cup \delta(4, b) \cup \emptyset = \{5\}$. Then

$$\epsilon\text{-closure}(U) = \epsilon\text{-closure}(\{5\}) = \{1, 2, 4, 5, 6, 7\} = C.$$

We have $U = \bigcup_{s \in B} \delta(s, a) = \emptyset \cup \delta(2, a) \cup \emptyset \cup \emptyset \cup \emptyset \cup \delta(7, a) \cup \emptyset = \{3, 8\}$. Then

$$\epsilon\text{-closure}(U) = \epsilon\text{-closure}(\{3, 8\}) = \{1, 2, 4, 5, 6, 7, 8\} = B.$$

We have $U = \bigcup_{s \in B} \delta(s, b) = \emptyset \cup \emptyset \cup \emptyset \cup \delta(4, b) \cup \emptyset \cup \emptyset \cup \delta(8, b) = \{5, 9\}$. Then

$$\begin{aligned}\epsilon\text{-closure}(U) &= \epsilon\text{-closure}(\{5, 9\}) = \epsilon\text{-closure}(\{5\}) \cup \epsilon\text{-closure}(\{9\}) \\ &= \{1, 2, 4, 5, 6, 7\} \cup \{9\} = \{1, 2, 4, 5, 6, 7, 9\} = D.\end{aligned}$$

We have $U = \bigcup_{s \in C} \delta(s, a) = \emptyset \cup \delta(2, a) \cup \emptyset \cup \emptyset \cup \emptyset \cup \delta(7, a) = \{3, 8\}$. Then

$$\epsilon\text{-closure}(U) = \epsilon\text{-closure}(\{3, 8\}) = \{1, 2, 4, 5, 6, 7, 8\} = B.$$

We have $U = \bigcup_{s \in C} \delta(s, b) = \emptyset \cup \emptyset \cup \delta(4, b) \cup \emptyset \cup \emptyset \cup \emptyset = \{5\}$. Then

$$\epsilon\text{-closure}(U) = \epsilon\text{-closure}(\{5\}) = \{1, 2, 4, 5, 6, 7\} = C.$$

We have $U = \bigcup_{s \in D} \delta(s, a) = \emptyset \cup \delta(2, a) \cup \emptyset \cup \emptyset \cup \emptyset \cup \delta(7, a) \cup \emptyset = \{3, 8\}$. Then

$$\epsilon\text{-closure}(U) = \epsilon\text{-closure}(\{3, 8\}) = \{1, 2, 4, 5, 6, 7, 8\} = B.$$

We have $U = \bigcup_{s \in D} \delta(s, b) = \emptyset \cup \emptyset \cup \delta(4, b) \cup \emptyset \cup \emptyset \cup \emptyset \cup \delta(9, a) = \{5\} \cup \{10\} = \{5, 10\}$. Then

$$\begin{aligned}\epsilon\text{-closure}(U) &= \epsilon\text{-closure}(\{5, 10\}) = \epsilon\text{-closure}(\{5\}) \cup \epsilon\text{-closure}(\{10\}) \\ &= \{1, 2, 4, 5, 6, 7\} \cup \{10\} = \{1, 2, 4, 5, 6, 7, 10\} = E.\end{aligned}$$

We have $U = \bigcup_{s \in E} \delta(s, a) = \emptyset \cup \delta(2, a) \cup \emptyset \cup \emptyset \cup \emptyset \cup \delta(7, a) \cup \emptyset = \{3, 8\}$. Then

$$\epsilon\text{-closure}(U) = \epsilon\text{-closure}(\{3, 8\}) = \{1, 2, 4, 5, 6, 7, 8\} = B.$$

We have $U = \bigcup_{s \in E} \delta(s, b) = \emptyset \cup \emptyset \cup \delta(4, b) \cup \emptyset \cup \emptyset \cup \emptyset \cup \emptyset = \{5\}$. Then

$$\epsilon\text{-closure}(U) = \epsilon\text{-closure}(\{5\}) = \{1, 2, 4, 5, 6, 7\} = C.$$

The only final state is E. The graph of this DFA with 5 states is shown in Figure 3.15. It is *not* a minimal DFA for $L = \{a, b\}^*\{abb\}$.

3.6 Finite State Automata With Output: Transducers

So far, we have only considered automata that recognize languages, i.e., automata that do not produce any output on any input (except "accept" or "reject").

It is interesting and useful to consider input/output finite state machines. Such automata are called *transducers*. They compute functions or relations. First, we define a deterministic kind of transducer.

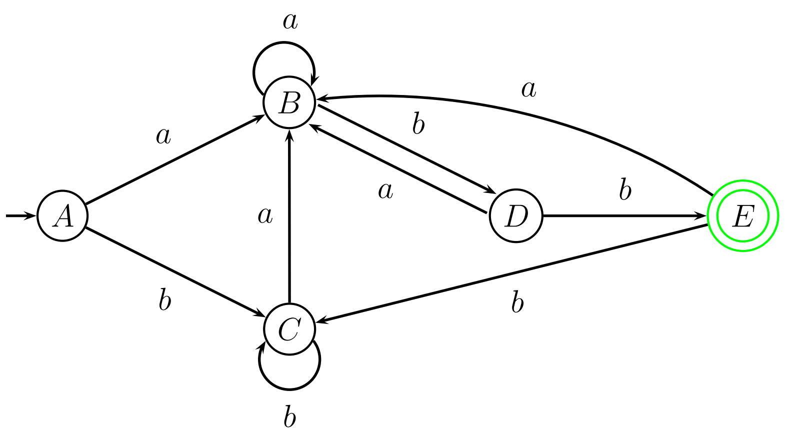


Figure 3.15: A non-minimal DFA for $\{a,b\}^*\{abb\}$.

Definition 3.14. A *general sequential machine (gsm)* is a sextuple $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$, where

(1) Q is a finite set of *states*,

(2) Σ is a finite *input alphabet*,

(3) Δ is a finite *output alphabet*,

(4) $\delta\colon Q \times \Sigma \to Q$ is the *transition function*,

(5) $\lambda\colon Q \times \Sigma \to \Delta^*$ is the *output function* and

(6) q_0 is the *initial* (or *start*) *state*.

If $\lambda(p, a) \neq \epsilon$, for all $p \in Q$ and all $a \in \Sigma$, then M is *nonerasing*. If $\lambda(p, a) \in \Delta$ for all $p \in Q$ and all $a \in \Sigma$, we say that M is a *complete sequential machine (csm)*.

An example of a gsm for which $\Sigma = \{a, b\}$ and $\Delta = \{0, 1, 2\}$ is shown in Figure 3.16. For example aab is converted to 102001.

In order to define how a gsm works, we extend the transition and the output functions. We define $\delta^*\colon Q \times \Sigma^* \to Q$ and $\lambda^*\colon Q \times \Sigma^* \to \Delta^*$ recursively as follows: For all $p \in Q$, all $u \in \Sigma^*$ and all $a \in \Sigma$

$$\begin{aligned}
\delta^*(p, \epsilon) &= p \\
\delta^*(p, ua) &= \delta(\delta^*(p, u), a) \\
\lambda^*(p, \epsilon) &= \epsilon \\
\lambda^*(p, ua) &= \lambda^*(p, u)\lambda(\delta^*(p, u), a).
\end{aligned}$$

For any $w \in \Sigma^*$, we let

$$M(w) = \lambda^*(q_0, w)$$

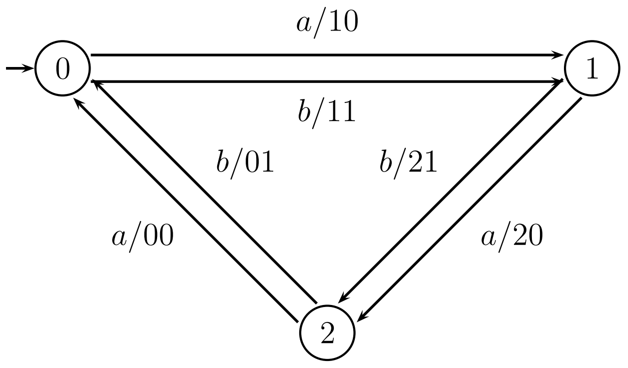


Figure 3.16: Example of a gsm.

and for any $L \subseteq \Sigma^*$ and $L' \subseteq \Delta^*$, let

$$M(L) = \{\lambda^*(q_0, w) \mid w \in L\}$$

and

$$M^{-1}(L') = \{w \in \Sigma^* \mid \lambda^*(q_0, w) \in L'\}.$$

Note that if M is a csm, then $|M(w)| = |w|$ for all $w \in \Sigma^*$. Also, a homomorphism is a special kind of gsm—it can be realized by a gsm with one state.

We can use gsm's and csm's to compute certain kinds of functions.

Definition 3.15. A function $f\colon \Sigma^* \to \Delta^*$ is a *gsm* (resp. *csm*) *mapping* iff there is a gsm (resp. csm) M so that $M(w) = f(w)$, for all $w \in \Sigma^*$.

Remark: Ginsburg and Rose (1966) characterized gsm mappings as follows:

A function $f\colon \Sigma^* \to \Delta^*$ is a gsm mapping iff

(a) f preserves prefixes, i.e., $f(x)$ is a prefix of $f(xy)$;

(b) There is an integer, m, such that for all $w \in \Sigma^*$ and all $a \in \Sigma$, we have $|f(wa)| - |f(w)| \leq m$;

(c) $f(\epsilon) = \epsilon$;

(d) For every regular language, $R \subseteq \Delta^*$, the language $f^{-1}(R) = \{w \in \Sigma^* \mid f(w) \in R\}$ is regular.

A function $f\colon \Sigma^* \to \Delta^*$ is a csm mapping iff f satisfies (a) and (d), and for all $w \in \Sigma^*$, $|f(w)| = |w|$. The following proposition is left as a homework problem.

Proposition 3.4. *The family of regular languages (over an alphabet Σ) is closed under both gsm and inverse gsm mappings.*

We can generalize the gsm model so that

(1) the device is nondeterministic,

(2) the device has a set of accepting states,

(3) transitions are allowed to occur without new input being processed,

(4) transitions are defined for input strings instead of individual letters.

Here is the definition of such a model, the *a-transducer*. A much more powerful model of transducer will be investigated later: the *Turing machine*.

Definition 3.16. An *a-transducer* (or *nondeterministic sequential transducer with accepting states*) is a sextuple $M = (K, \Sigma, \Delta, \lambda, q_0, F)$, where

(1) K is a finite set of *states*,

(2) Σ is a finite *input alphabet*,

(3) Δ is a finite *output alphabet*,

(4) $q_0 \in K$ is the *start* (or *initial*) *state*,

(5) $F \subseteq K$ is the set of *accepting* (of *final*) *states* and

(6) $\lambda \subseteq K \times \Sigma^* \times \Delta^* \times K$ is a finite set of quadruples called the *transition function* of M.

If $\lambda \subseteq K \times \Sigma^* \times \Delta^+ \times K$, then M is *ϵ-free*

Clearly, a gsm is a special kind of a-transducer.

An a-transducer defines a binary relation between Σ^* and Δ^*, or equivalently, a function $M\colon \Sigma^* \to 2^{\Delta^*}$.

We can explain what this function is by describing how an a-transducer makes a sequence of moves from configurations to configurations.

The current *configuration* of an a-transducer is described by a triple

$$(p, u, v) \in K \times \Sigma^* \times \Delta^*,$$

where p is the current state, u is the remaining input, and v is some ouput produced so far.

We define the binary relation $\vdash_M$ on $K \times \Sigma^* \times \Delta^*$ as follows: For all $p, q \in K$, $u, \alpha \in \Sigma^*$, $\beta, v \in \Delta^*$, if $(p, u, v, q) \in \lambda$, then

$$(p,\, u\alpha,\, \beta) \vdash_M (q,\, \alpha,\, \beta v).$$

Let $\vdash_M^*$ be the transitive and reflexive closure of $\vdash_M$. The function $M\colon \Sigma^* \to 2^{\Delta^*}$ is defined such that for every $w \in \Sigma^*$,

$$M(w) = \{y \in \Delta^* \mid (q_0,\, w,\, \epsilon) \vdash_M^* (f,\, \epsilon,\, y),\ f \in F\}.$$

For any language $L \subseteq \Sigma^*$ let

$$M(L) = \bigcup_{w \in L} M(w).$$

For any $y \in \Delta^*$, let

$$M^{-1}(y) = \{w \in \Sigma^* \mid y \in M(w)\}$$

and for any language $L' \subseteq \Delta^*$, let

$$M^{-1}(L') = \bigcup_{y \in L'} M^{-1}(y).$$

Remark: Notice that if $w \in M^{-1}(L')$, then there exists some $y \in L'$ such that $w \in M^{-1}(y)$, i.e.,
$y \in M(w)$. This **does not** imply that $M(w) \subseteq L'$, only that $M(w) \cap L' \neq \emptyset$.

One should realize that for any $L' \subseteq \Delta^*$ and any a-transducer, M, there is some a-transducer, M', (from Δ^* to 2^{Σ^*}) so that $M'(L') = M^{-1}(L')$.

The following proposition is left as a homework problem:

Proposition 3.5. *The family of regular languages (over an alphabet Σ) is closed under both a-transductions and inverse a-transductions.*

3.7 An Application of NFA's: Text Search

A common problem in the age of the Web (and on-line text repositories) is the following:

Given a set of words, called the *keywords*, find all the documents that contain one (or all) of those words. Search engines are a popular example of this process. Search engines use *inverted indexes* (for each word appearing on the Web, a list of all the places where that word occurs is stored).

However, there are applications that are unsuited for inverted indexes, but are good for automaton-based techniques.

Some text-processing programs, such as advanced forms of the UNIX `grep` command (such as `egrep` or `fgrep`) are based on automaton-based techniques.

The characteristics that make an application suitable for searches that use automata are:

(1) The repository on which the search is conducted is rapidly changing.

(2) The documents to be searched cannot be catalogued. For example, Amazon.com creates pages "on the fly" in response to queries.

We can use an NFA to find occurrences of a set of keywords in a text. This NFA signals by entering a final state that it has seen one of the keywords. The form of such an NFA is special.

(1) There is a start state, q_0, with a transition to itself on every input symbol from the alphabet, Σ.

(2) For each keyword, $w = w_1 \cdots w_k$ (with $w_i \in \Sigma$), there are k states, $q_1^{(w)}, \ldots, q_k^{(w)}$, and there is a transition from q_0 to $q_1^{(w)}$ on input w_1, a transition from $q_1^{(w)}$ to $q_2^{(w)}$ on input w_2, and so on, until a transition from $q_{k-1}^{(w)}$ to $q_k^{(w)}$ on input w_k. The state $q_k^{(w)}$ is an accepting state and indicates that the keyword $w = w_1 \cdots w_k$ has been found.

The NFA constructed above can then be converted to a DFA using the subset construction.

Example 3.11. Here is an example where $\Sigma = \{a, b\}$ and the set of keywords is

$$\{aba,\ ab,\ ba\}.$$

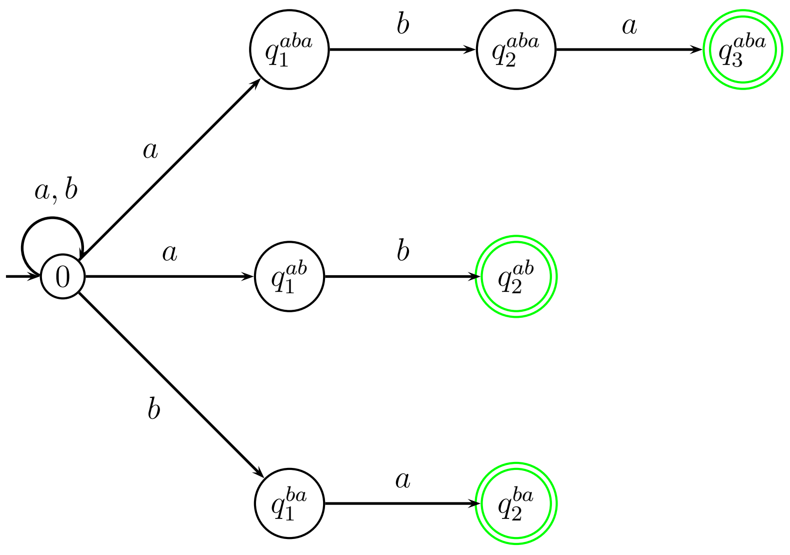


Figure 3.17: NFA for the keywords aba, ab, ba.

Applying the subset construction to the NFA, we obtain the DFA whose transition table is:

		a	b
0	0	1	2
1	$0, q_1^{aba}, q_1^{ab}$	1	3
2	$0, q_1^{ba}$	4	2
3	$0, q_1^{ba}, q_2^{aba}, q_2^{ab}$	5	2
4	$0, q_1^{aba}, q_1^{ab}, q_2^{ba}$	1	3
5	$0, q_1^{aba}, q_1^{ab}, q_2^{ba}, q_3^{aba}$	1	3

The final states are: 3, 4, 5.

The good news news is that, due to the very special structure of the NFA, the number of states of the corresponding DFA is *at most* the number of states of the original NFA!

We find that the states of the DFA are (check it yourself!):

(1) The set $\{q_0\}$, associated with the start state q_0 of the NFA.

(2) For any state $p \neq q_0$ of the NFA reached from q_0 along a path corresponding to a string $u = u_1 \cdots u_m$, the set consisting of:

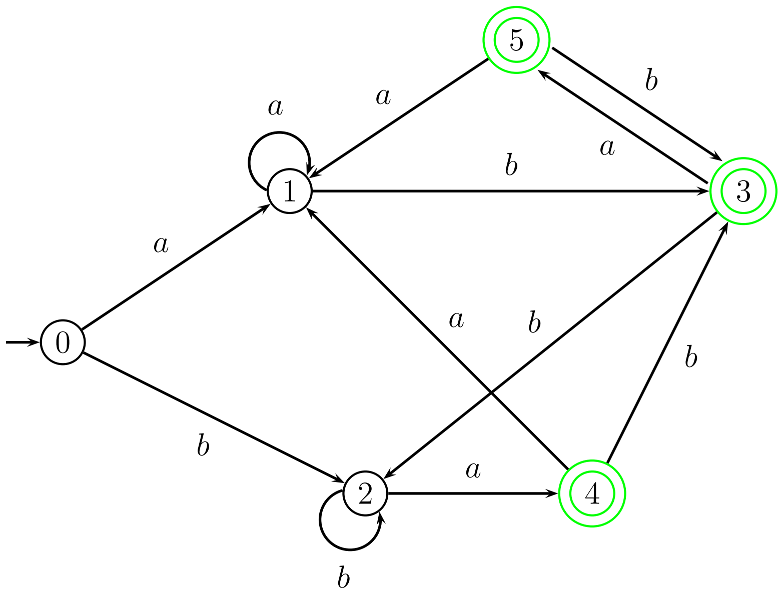


Figure 3.18: DFA for the keywords aba, ab, ba.

(a) q_0

(b) p

(c) The set of all states q of the NFA reachable from q_0 by following a path whose symbols form a nonempty suffix of u, i.e., a string of the form $u_j u_{j+1} \cdots u_m$.

As a consequence, we get an efficient (w.r.t. time and space) method to recognize a set of keywords. In fact, this DFA recognizes leftmost occurrences of keywords in a text (we can stop as soon as we enter a final state).

Chapter 4

Hidden Markov Models (HMMs)

4.1 Definition of a Hidden Markov Model (HMM)

There is a variant of the notion of DFA with output, for example a transducer such as a gsm (generalized sequential machine), which is widely used in machine learning. This machine model is known as *hidden Markov model*, for short *HMM*. These notes are only an *introduction* to HMMs and are by no means complete. For more comprehensive presentations of HMMs, see the references at the end of this chapter.

There are three new twists compared to traditional gsm models:

(1) There is a finite set of states Q with n elements, a bijection $\sigma\colon Q \to \{1, \ldots, n\}$, and the transitions between states are labeled with probabilities rather that symbols from an alphabet. For any two states p and q in Q, the edge from p to q is labeled with a probability $A(i, j)$, with $i = \sigma(p)$ and $j = \sigma(q)$. The probabilities $A(i, j)$ form an $n \times n$ matrix $A = (A(i, j))$.

(2) There is a finite set $\mathbb{O}$ of size m (called the *observation space*) of possible outputs that can be emitted, a bijection $\omega\colon \mathbb{O} \to \{1, \ldots, m\}$, and for every state $q \in Q$, there is a probability $B(i, j)$ that output $O \in \mathbb{O}$ is emitted (produced), with $i = \sigma(q)$ and $j = \omega(O)$. The probabilities $B(i, j)$ form an $n \times m$ matrix $B = (B(i, j))$.

(3) *Sequences of outputs* $\mathcal{O} = (O_1, \ldots, O_T)$ (with $O_t \in \mathbb{O}$ for $t = 1, \ldots, T$) emitted by the model are *directly observable*, but the sequences of states $\mathcal{S} = (q_1, \ldots, q_T)$ (with $q_t \in Q$ for $t = 1, \ldots, T$) that caused some sequence of output to be emitted are *not observable*. In this sense the states are hidden, and this is the reason for calling this model a *hidden Markov model*.

Remark: We could define a state transition probability function $\mathbb{A}\colon Q \times Q \to [0, 1]$ by $\mathbb{A}(p, q) = A(\sigma(p), \sigma(q))$, and a state observation probability function $\mathbb{B}\colon Q \times \mathbb{O} \to [0, 1]$ by $\mathbb{B}(p, O) = B(\sigma(p), \omega(O))$. The function $\mathbb{A}$ conveys exactly the same amount of information as the matrix A, and the function $\mathbb{B}$ conveys exactly the same amount of information as the matrix B. The only difference is that the arguments of $\mathbb{A}$ are states rather than integers, so in that sense it is perhaps more natural. We can think of A as an implementation of $\mathbb{A}$. Similarly, the arguments of $\mathbb{B}$ are states and outputs rather than integers. Again, we can think of B as an implementation of $\mathbb{B}$. Most of the literature is rather sloppy about this. We will use matrices.

Here is an example illustrating the notion of HMM.

Example 4.1. Say we consider the following behavior of some professor at some university. On a hot day (denoted by Hot), the professor comes to class with a drink (denoted D) with probability 0.7, and with no drink (denoted N) with probability 0.3. On the other hand, on a cold day (denoted Cold), the professor comes to class with a drink with probability 0.2, and with no drink with probability 0.8.

Suppose a student intrigued by this behavior recorded a sequence showing whether the professor came to class with a drink or not, say NNND. Several months later, the student would like to know whether the weather was hot or cold the days he recorded the drinking behavior of the professor.

Now the student heard about machine learning, so he constructs a probabilistic (hidden Markov) model of the weather. Based on some experiments, he determines the probability of going from a hot day to another hot day to be 0.75, the probability of going from a hot to a cold day to be 0.25, the probability of going from a cold day to another cold day to be 0.7, and the probability of going from a cold day to a hot day to be 0.3. He also knows that when he started his observations, it was a cold day with probability 0.45, and a hot day with probability 0.55.

In this example, the set of states is $Q = \{\mathrm{Cold}, \mathrm{Hot}\}$, and the set of outputs is $\mathbb{O} = \{\mathrm{N}, \mathrm{D}\}$. We have the bijection $\sigma\colon \{\mathrm{Cold}, \mathrm{Hot}\} \to \{1, 2\}$ given by $\sigma(\mathrm{Cold}) = 1$ and $\sigma(\mathrm{Hot}) = 2$, and the bijection $\omega\colon \{\mathrm{N}, \mathrm{D}\} \to \{1, 2\}$ given by $\omega(\mathrm{N}) = 1$ and $\omega(\mathrm{D}) = 2$.

The above data determine an HMM depicted in Figure 4.1.

The portion of the state diagram involving the states Cold, Hot, is analogous to an NFA in which the transition labels are probabilities; it is the underlying Markov model of the HMM. For any given state, the probabilities on the outgoing edges sum to 1. The start state is a convenient way to express the probabilities of starting either in state Cold or in state Hot. Also, from each of the states Cold and Hot, we have emission probabilities of producing the ouput N or D, and these probabilities also sum to 1.

We can also express these data using matrices. The matrix

$$A = \begin{pmatrix} 0.7 & 0.3 \\ 0.25 & 0.75 \end{pmatrix}$$

describes the transitions of the Markov model, the vector

$$\pi = \begin{pmatrix} 0.45 \\ 0.55 \end{pmatrix}$$

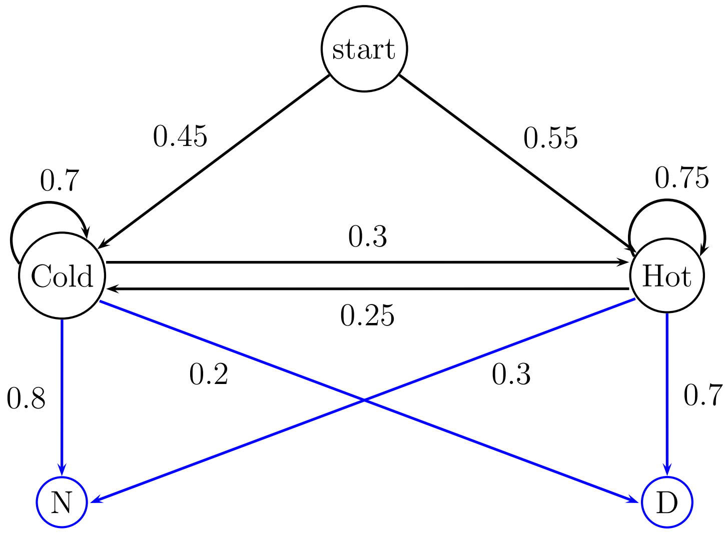


Figure 4.1: Example of an HMM modeling the "drinking behavior" of a professor at the University of Pennsylvania.

describes the probabilities of starting either in state Cold or in state Hot, and the matrix

$$B = \begin{pmatrix} 0.8 & 0.2 \\ 0.3 & 0.7 \end{pmatrix}$$

describes the emission probabilities. Observe that the rows of the matrices A and B sum to 1. Such matrices are called *row-stochastic matrices*. The entries in the vector π also sum to 1.

The student would like to solve what is known as the *decoding problem*. Namely, given the output sequence NNND, find the *most likely state sequence* of the Markov model that produces the output sequence NNND. Is it $(\text{Cold}, \text{Cold}, \text{Cold}, \text{Cold})$, or $(\text{Hot}, \text{Hot}, \text{Hot}, \text{Hot})$, or $(\text{Hot}, \text{Cold}, \text{Cold}, \text{Hot})$, or $(\text{Cold}, \text{Cold}, \text{Cold}, \text{Hot})$? Given the probabilities of the HMM, it seems unlikely that it is $(\text{Hot}, \text{Hot}, \text{Hot}, \text{Hot})$, but how can we find the most likely one?

Before going any further, we wish to address a notational issue that everyone who writes about state-processes faces. This issue is a bit of a headache which needs to be resolved to avoid a lot of confusion.

The issue is how to denote the states, the ouputs, as well as (ordered) sequences of states and sequences of output. In most problems, states and outputs have "meaningful" names. For example, if we wish to describe the evolution of the temperature from day to day, it makes sense to use two states "Cold" and "Hot," and to describe whether a given individual has a drink by "D," and no drink by "N." Thus our set of states is $Q = \{\text{Cold}, \text{Hot}\}$, and our set of outputs is $\mathbb{O} = \{\text{N}, \text{D}\}$.

However, when computing probabilities, we need to use matrices whose rows and columns are indexed by positive integers, so we need a mechanism to associate a *numerical index* to every state and to every output, and this is the purpose of the bijections $\sigma\colon Q \to \{1, \ldots, n\}$ and $\omega\colon \mathbb{O} \to \{1, \ldots, m\}$. In our example, we define σ by $\sigma(\text{Cold}) = 1$ and $\sigma(\text{Hot}) = 2$, and ω by $\omega(\text{N}) = 1$ and $\omega(\text{D}) = 2$.

Some author circumvent (or do they?) this notational issue by assuming that the set of outputs is $\mathbb{O} = \{1, 2, \ldots, m\}$, and that the set of states is $Q = \{1, 2, \ldots, n\}$. The disadvantage of doing this is that in "real" situations, it is often more convenient to name the outputs and the states with more meaningful names than $1, 2, 3$ *etc.* With respect to this, Mitch Marcus pointed out to me that the task of naming the elements of the output alphabet can be challenging, for example in speech recognition.

Let us now turn to sequences. For example, consider the sequence of six states (from the set $Q = \{\text{Cold}, \text{Hot}\}$),

$$\mathcal{S} = (\text{Cold}, \text{Cold}, \text{Hot}, \text{Cold}, \text{Hot}, \text{Hot}).$$

Using the bijection $\sigma\colon \{\text{Cold}, \text{Hot}\} \to \{1, 2\}$ defined above, the sequence $\mathcal{S}$ is completely determined by the sequence of indices

$$\sigma(\mathcal{S}) = (\sigma(\text{Cold}), \sigma(\text{Cold}), \sigma(\text{Hot}), \sigma(\text{Cold}), \sigma(\text{Hot}), \sigma(\text{Hot})) = (1, 1, 2, 1, 2, 2).$$

More generally, we will denote a sequence of length $T \geq 1$ of states from a set Q of size n by

$$\mathcal{S} = (q_1, q_2, \ldots, q_T),$$

with $q_t \in Q$ for $t = 1, \ldots, T$. Note that sequences start at time $t = 1$, and not at time $t = 0$. This is not the convention used in the theory of stochastic discrete-parameter processes where the starting time is $t = 0$, but it has the advantage that a sequence of T elements is written as $(q_1, q_2, \ldots, q_T)$ instead of $(q_0, q_1, \ldots, q_{T-1})$.

Using the bijection $\sigma\colon Q \to \{1, \ldots, n\}$, the sequence $\mathcal{S}$ is completely determined by the sequence of indices

$$\sigma(\mathcal{S}) = (\sigma(q_1), \sigma(q_2), \ldots, \sigma(q_T)),$$

where $\sigma(q_t)$ is some index from the set $\{1, \ldots, n\}$, for $t = 1, \ldots, T$. The problem now is, *what is a better notation for the index denoted by $\sigma(q_t)$?*

Of course, we could use $\sigma(q_t)$, but this is a heavy notation, so *we adopt the notational convention to denote the index $\sigma(q_t)$ by i_t.*[1]

Going back to our example

$$\mathcal{S} = (q_1, q_2, q_3, q_4, q_4, q_6) = (\text{Cold}, \text{Cold}, \text{Hot}, \text{Cold}, \text{Hot}, \text{Hot}),$$

[1]We contemplated using the notation σ_t for $\sigma(q_t)$ instead of i_t. However, we feel that this would deviate too much from the common practice found in the literature, which uses the notation i_t. This is not to say that the literature is free of horribly confusing notation!

we have

$$\sigma(\mathcal{S}) = (\sigma(q_1), \sigma(q_2), \sigma(q_3), \sigma(q_4), \sigma(q_5), \sigma(q_6)) = (1, 1, 2, 1, 2, 2),$$

so the sequence of indices $(i_1, i_2, i_3, i_4, i_5, i_6) = (\sigma(q_1), \sigma(q_2), \sigma(q_3), \sigma(q_4), \sigma(q_5), \sigma(q_6))$ is given by

$$\sigma(\mathcal{S}) = (i_1, i_2, i_3, i_4, i_5, i_6) = (1, 1, 2, 1, 2, 2).$$

So, the fourth index i_4 is has the value 1.

We apply a similar convention to sequences of outputs. For example, consider the sequence of six outputs (from the set $\mathbb{O} = \{\mathrm{N}, \mathrm{D}\}$),

$$\mathcal{O} = (\mathrm{N}, \mathrm{D}, \mathrm{N}, \mathrm{N}, \mathrm{N}, \mathrm{D}).$$

Using the bijection $\omega\colon \{\mathrm{N}, \mathrm{D}\} \to \{1, 2\}$ defined above, the sequence $\mathcal{O}$ is completely determined by the sequence of indices

$$\omega(\mathcal{O}) = (\omega(\mathrm{N}), \omega(\mathrm{D}), \omega(\mathrm{N}), \omega(\mathrm{N}), \omega(\mathrm{N}), \omega(\mathrm{D})) = (1, 2, 1, 1, 1, 2).$$

More generally, we will denote a sequence of length $T \geq 1$ of outputs from a set $\mathbb{O}$ of size m by

$$\mathcal{O} = (O_1, O_2, \ldots, O_T),$$

with $O_t \in \mathbb{O}$ for $t = 1, \ldots, T$. Using the bijection $\omega\colon \mathbb{O} \to \{1, \ldots, m\}$, the sequence $\mathcal{O}$ is completely determined by the sequence of indices

$$\omega(\mathcal{O}) = (\omega(O_1), \omega(O_2), \ldots, \omega(O_T)),$$

where $\omega(O_t)$ is some index from the set $\{1, \ldots, m\}$, for $t = 1, \ldots, T$. This time, *we adopt the notational convention to denote the index $\omega(O_t)$ by ω_t.*

Going back to our example

$$\mathcal{O} = (O_1, O_2, O_3, O_4, O_5, O_6) = (\mathrm{N}, \mathrm{D}, \mathrm{N}, \mathrm{N}, \mathrm{N}, \mathrm{D}),$$

we have

$$\omega(\mathcal{O}) = (\omega(O_1), \omega(O_2), \omega(O_3), \omega(O_4), \omega(O_5), \omega(O_6)) = (1, 2, 1, 1, 1, 2),$$

so the sequence of indices $(\omega_1, \omega_2, \omega_3, \omega_4, \omega_5, \omega_6) = (\omega(O_1), \omega(O_2), \omega(O_3), \omega(O_4), \omega(O_5), \omega(O_6))$ is given by

$$\omega(\mathcal{O}) = (\omega_1, \omega_2, \omega_3, \omega_4, \omega_5, \omega_6) = (1, 2, 1, 1, 1, 2).$$

Remark: What is very confusing is this: to assume that the state set is $Q = \{q_1, \ldots, q_n\}$, and to denote a sequence of states of length T as $\mathcal{S} = (q_1, q_2, \ldots, q_T)$. The symbol q_1 in the sequence $\mathcal{S}$ may actually refer to q_3 in Q, *etc.* At least, the states in Q or the states in the sequences should be denoted using a different letter, say $\mathcal{S} = (s_1, \ldots, s_T)$.

We feel that the explicit introduction of the bijections $\sigma\colon Q \to \{1, \ldots, n\}$ and $\omega\colon \mathbb{O} \to \{1, \ldots, m\}$, although not standard in the literature, yields a mathematically clean way to deal with sequences which is not too cumbersome, although this latter point is a matter of taste.

HMM's are among the most effective tools to solve the following types of problems:

(1) **DNA and protein sequence alignment** in the face of mutations and other kinds of evolutionary change.

(2) **Speech understanding**, also called **Automatic speech recognition**. When we talk, our mouths produce sequences of sounds from the sentences that we want to say. This process is complex. Multiple words may map to the same sound, words are pronounced differently as a function of the word before and after them, we all form sounds slightly differently, and so on. All a listener can hear (perhaps a computer system) is the sequence of sounds, and the listener would like to reconstruct the mapping (backward) in order to determine what words we were attempting to say. For example, when you "talk to your TV" to pick a program, say *game of thrones*, you don't want to get *Jessica Jones*.

(3) **Optical character recognition (OCR)**. When we write, our hands map from an idealized symbol to some set of marks on a page (or screen). The marks are observable, but the process that generates them isn't. A system performing OCR, such as a system used by the post office to read addresses, must discover which word is most likely to correspond to the mark it reads.

The reader should review Example 4.1 illustrating the notion of HMM. Let us consider another example taken from Stamp [8].

Example 4.2. Suppose we want to determine the average annual temperature at a particular location over a series of years in a distant past where thermometers did not exist. Since we can't go back in time, we look for indirect evidence of the temperature, say in terms of the size of tree growth rings. For simplicity, assume that we consider the two temperatures Cold and Hot, and three different sizes of tree rings: small, medium and large, which we denote by S, M, L.

In this example, the set of states is $Q = \{\text{Cold}, \text{Hot}\}$, and the set of outputs is $\mathbb{O} = \{\text{S}, \text{M}, \text{L}\}$. We have the bijection $\sigma\colon \{\text{Cold}, \text{Hot}\} \to \{1, 2\}$ given by $\sigma(\text{Cold}) = 1$ and $\sigma(\text{Hot}) = 2$, and the bijection $\omega\colon \{\text{S}, \text{M}, \text{L}\} \to \{1, 2, 3\}$ given by $\omega(\text{S}) = 1$, $\omega(\text{M}) = 2$, and $\omega(\text{L}) = 3$. The HMM shown in Figure 4.2 is a model of the situation.

Suppose we observe the sequence of tree growth rings (S, M, S, L). What is the most likely sequence of temperatures over a four-year period which yields the observations (S, M, S, L)?

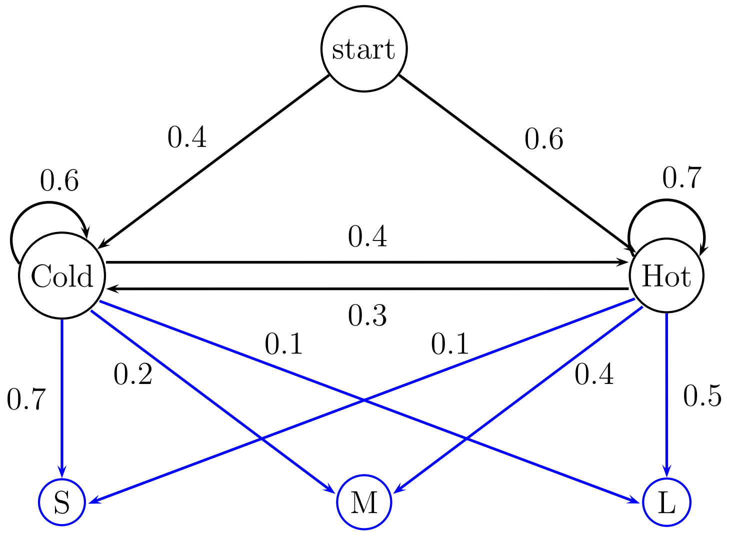


Figure 4.2: Example of an HMM modeling the temperature in terms of tree growth rings.

Going back to Example 4.1, which corresponds to the HMM graph shown in Figure 4.3, we need to figure out the probability that a sequence of states $\mathcal{S} = (q_1, q_2, \ldots, q_T)$ produces the output sequence $\mathcal{O} = (O_1, O_2, \ldots, O_T)$.

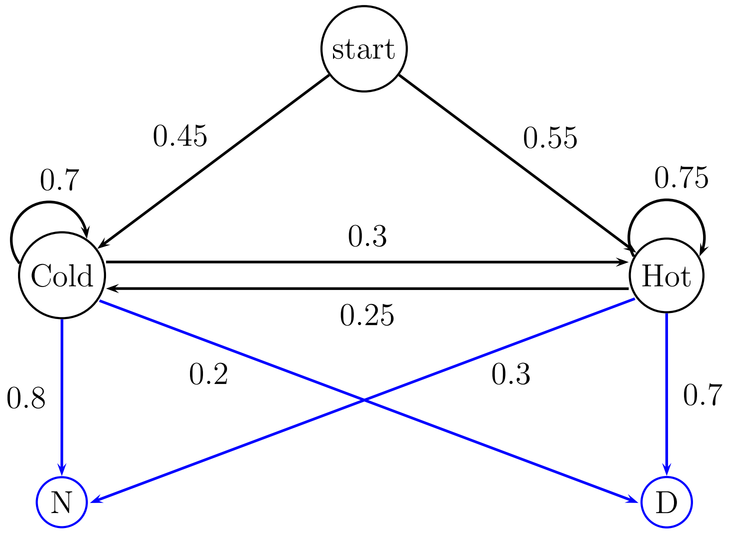


Figure 4.3: Example of an HMM modeling the "drinking behavior" of a professor at the University of Pennsylvania.

Then the probability that we want is just the product of the probability that we begin with state q_1, times the product of the probabilities of each of the transitions, times the product of the emission probabilities. With our notational conventions, $\sigma(q_t) = i_t$ and $\omega(O_t) = \omega_t$, so we have

$$\mathsf{Pr}(\mathcal{S}, \mathcal{O}) = \pi(i_1)B(i_1, \omega_1)\prod_{t=2}^{T} A(i_{t-1}, i_t)B(i_t, \omega_t).$$

In our example, $\omega(\mathcal{O}) = (\omega_1, \omega_2, \omega_3, \omega_4) = (1, 1, 1, 2)$, which corresponds to NNND. The brute-force method is to compute these probabilities for all $2^4 = 16$ sequences of states of length 4 (in general, there are n^T sequences of length T). For example, for the sequence $\mathcal{S} = (\text{Cold}, \text{Cold}, \text{Cold}, \text{Hot})$, associated with the sequence of indices $\sigma(\mathcal{S}) = (i_1, i_2, i_3, i_4) = (1, 1, 1, 2)$, we find that

$$\begin{aligned}\mathsf{Pr}(\mathcal{S}, \text{NNND}) &= \pi(1)B(1,1)A(1,1)B(1,1)A(1,1)B(1,1)A(1,2)B(2,2)\\ &= 0.45 \times 0.8 \times 0.7 \times 0.8 \times 0.7 \times 0.8 \times 0.3 \times 0.7 = 0.0237.\end{aligned}$$

A much more efficient way to proceed is to use a method based on *dynamic programming*. Recall the bijection $\sigma\colon \{\text{Cold}, \text{Hot}\} \to \{1, 2\}$, so that we will refer to the state Cold as 1, and to the state Hot as 2. For $t = 1, 2, 3, 4$, for every state $i = 1, 2$, *we compute $score(i, t)$ to be the highest probability that a sequence of length t ending in state i produces the output sequence $(O_1, \ldots, O_t)$*, and for $t \geq 2$, we let $pred(i, t)$ be the state that precedes state i in a best sequence of length t ending in i.

Recall that in our example, $\omega(\mathcal{O}) = (\omega_1, \omega_2, \omega_3, \omega_4) = (1, 1, 1, 2)$, which corresponds to NNND. Initially, we set

$$score(j, 1) = \pi(j)B(j, \omega_1), \quad j = 1, 2,$$

and since $\omega_1 = 1$ we get $score(1, 1) = 0.45 \times 0.8 = 0.36$, which is the probability of starting in state Cold and emitting N, and $score(2, 1) = 0.55 \times 0.3 = 0.165$, which is the probability of starting in state Hot and emitting N.

Next we compute $score(1, 2)$ and $score(2, 2)$ as follows. For $j = 1, 2$, for $i = 1, 2$, compute temporary scores

$$tscore(i, j) = score(i, 1)A(i, j)B(j, \omega_2);$$

then pick the best of the temporary scores,

$$score(j, 2) = \max_i tscore(i, j).$$

Since $\omega_2 = 1$, we get $tscore(1, 1) = 0.36 \times 0.7 \times 0.8 = 0.2016$, $tscore(2, 1) = 0.165 \times 0.25 \times 0.8 = 0.0330$, and $tscore(1, 2) = 0.36 \times 0.3 \times 0.3 = 0.0324$, $tscore(2, 2) = 0.165 \times 0.75 \times 0.3 = 0.0371$. Then

$$score(1, 2) = \max\{tscore(1, 1), tscore(2, 1)\} = \max\{0.2016, 0.0330\} = 0.2016,$$

which is the largest probability that a sequence of two states emitting the output (N, N) ends in state Cold, and

$$\mathit{score}(2,2) = \max\{\mathit{tscore}(1,2), \mathit{tscore}(2,2)\} = \max\{0.0324, 0.0371\} = 0.0371.$$

which is the largest probability that a sequence of two states emitting the output (N, N) ends in state Hot. Since the state that leads to the optimal score score$(1,2)$ is 1, we let $\mathit{pred}(1,2) = 1$, and since the state that leads to the optimal score score$(2,2)$ is 2, we let $\mathit{pred}(2,2) = 2$.

We compute $\mathit{score}(1,3)$ and $\mathit{score}(2,3)$ in a similar way. For $j = 1,2$, for $i = 1,2$, compute

$$\mathit{tscore}(i,j) = \mathit{score}(i,2)A(i,j)B(j,\omega_3);$$

then pick the best of the temporary scores,

$$\mathit{score}(j,3) = \max_i \mathit{tscore}(i,j).$$

Since $\omega_3 = 1$, we get $\mathit{tscore}(1,1) = 0.2016 \times 0.7 \times 0.8 = 0.1129$, $\mathit{tscore}(2,1) = 0.0371 \times 0.25 \times 0.8 = 0.0074$, and $\mathit{tscore}(1,2) = 0.2016 \times 0.3 \times 0.3 = 0.0181$, $\mathit{tscore}(2,2) = 0.0371 \times 0.75 \times 0.3 = 0.0083$. Then

$$\mathit{score}(1,3) = \max\{\mathit{tscore}(1,1), \mathit{tscore}(2,1)\} = \max\{0.1129, 0.0074\} = 0.1129,$$

which is the largest probability that a sequence of three states emitting the output (N, N, N) ends in state Cold, and

$$\mathit{score}(2,3) = \max\{\mathit{tscore}(1,2), \mathit{tscore}(2,2)\} = \max\{0.0181, 0.0083\} = 0.0181,$$

which is the largest probability that a sequence of three states emitting the output (N, N, N) ends in state Hot. We also get $\mathit{pred}(1,3) = 1$ and $\mathit{pred}(2,3) = 1$. Finally, we compute $\mathit{score}(1,4)$ and $\mathit{score}(2,4)$ in a similar way. For $j = 1,2$, for $i = 1,2$, compute

$$\mathit{tscore}(i,j) = \mathit{score}(i,3)A(i,j)B(j,\omega_4);$$

then pick the best of the temporary scores,

$$\mathit{score}(j,4) = \max_i \mathit{tscore}(i,j).$$

Since $\omega_4 = 2$, we get $\mathit{tscore}(1,1) = 0.1129 \times 0.7 \times 0.2 = 0.0158$, $\mathit{tscore}(2,1) = 0.0181 \times 0.25 \times 0.2 = 0.0009$, and $\mathit{tscore}(1,2) = 0.1129 \times 0.3 \times 0.7 = 0.0237$, $\mathit{tscore}(2,2) = 0.0181 \times 0.75 \times 0.7 = 0.0095$. Then

$$\mathit{score}(1,4) = \max\{\mathit{tscore}(1,1), \mathit{tscore}(2,1)\} = \max\{0.0158, 0.0009\} = 0.0158,$$

which is the largest probability that a sequence of four states emitting the output (N, N, N, D) ends in state Cold, and

$$score(2,4) = \max\{tscore(1,2), tscore(2,2)\} = \max\{0.0237, 0.0095\} = 0.0237,$$

which is the largest probability that a sequence of four states emitting the output (N, N, N, D) ends in state Hot, and $pred(1,4) = 1$ and $pred(2,4) = 1$

Since $\max\{score(1,4), score(2,4)\} = \max\{0.0158, 0.0237\} = 0.0237$, the state with the maximum score is Hot, and by following the predecessor list (also called backpointer list), we find that the most likely state sequence to produce the output sequence NNND is (Cold, Cold, Cold, Hot).

The stages of the computations of $score(j,t)$ for $i = 1, 2$ and $t = 1, 2, 3, 4$ can be recorded in the following diagram called a *lattice*, or a *trellis* (which means lattice in French!):

$$\begin{tikzcd}
\text{Cold} & 0.36 \arrow[r, Rightarrow, "0.2016"] \arrow[dr, "0.0324"] & 0.2016 \arrow[r, Rightarrow, "0.1129"] \arrow[dr, Rightarrow, "0.0181"] & 0.1129 \arrow[r, Rightarrow, "0.0158"] \arrow[dr, Rightarrow, "0.0237"] & 0.0158 \\
\text{Hot} & 0.1650 \arrow[r, Rightarrow, "0.0371"'] \arrow[ur, "0.033"] & 0.0371 \arrow[r, "0.0083"'] \arrow[ur, "0.0074"] & 0.0181 \arrow[r, "0.0095"'] \arrow[ur, "0.0009"] & \mathbf{0.0237}
\end{tikzcd}$$

Note that the trellis contains 16 paths corresponding to the 16 sequences of states of length 4. Double arrows represent the predecessor edges. For example, the predecessor $pred(2,3)$ of the third node on the bottom row labeled with the score 0.0181 (which corresponds to Hot), is the second node on the first row labeled with the score 0.2016 (which corresponds to Cold). The two incoming arrows to the third node on the bottom row are labeled with the temporary scores 0.0181 and 0.0083. The node with the highest score at time $t = 4$ is Hot, with score 0.0237 (showed in bold), and by following the double arrows backward from this node, we obtain the most likely state sequence (Cold, Cold, Cold, Hot).

The method we just described is known as the *Viterbi algorithm*. We now define HHM's in general, and then present the Viterbi algorithm.

Definition 4.1. A *hidden Markov model*, for short *HMM*, is a quintuple $M = (Q, \mathbb{O}, \pi, A, B)$ where

- Q is a finite set of *states* with n elements, and there is a bijection $\sigma \colon Q \to \{1, \ldots, n\}$.
- $\mathbb{O}$ is a finite *output alphabet* (also called *set of possible observations*) with m observations, and there is a bijection $\omega \colon \mathbb{O} \to \{1, \ldots, m\}$.
- $A = (A(i,j))$ is an $n \times n$ matrix called the *state transition probability matrix*, with

$$A(i,j) \geq 0, \quad 1 \leq i, j \leq n, \quad \text{and} \quad \sum_{j=1}^{n} A(i,j) = 1, \quad i = 1, \ldots, n.$$

- $B = (B(i,j))$ is an $n \times m$ matrix called the *state observation probability matrix* (also called *confusion matrix*), with

$$B(i,j) \geq 0, \quad 1 \leq i,j \leq n, \quad \text{and} \quad \sum_{j=1}^{m} B(i,j) = 1, \quad i = 1, \ldots, n.$$

  A matrix satisfying the above conditions is said to be *row stochastic*. Both A and B are row-stochastic.

We also need to state the conditions that make M a Markov model. To do this rigorously requires the notion of random variable and is a bit tricky (see the remark below), so we will cheat as follows:

(a) Given any sequence of states $(q_1, \ldots, q_{t-2}, p, q)$, the conditional probability that q is the tth state given that the previous states were $q_1, \ldots, q_{t-2}, p$ is equal to the conditional probability that q is the tth state given that the previous state at time $t-1$ is p:

$$\mathsf{Pr}(q \mid q_1, \ldots, q_{t-2}, p) = \mathsf{Pr}(q \mid p).$$

This is the *Markov property*. Informally, the "next" state q of the process at time t is independent of the "past" states $q_1, \ldots, q_{t-2}$, provided that the "present" state p at time $t-1$ is known.

(b) Given any sequence of states $(q_1, \ldots, q_i, \ldots, q_t)$, and given any sequence of outputs $(O_1, \ldots, O_i, \ldots, O_t)$, the conditional probability that the output O_i is emitted depends only on the state q_i, and not any other states or any other observations:

$$\mathsf{Pr}(O_i \mid q_1, \ldots, q_i, \ldots, q_t, O_1, \ldots, O_i, \ldots, O_t) = \mathsf{Pr}(O_i \mid q_i).$$

This is the *output independence* condition. Informally, the output function is near-sighted.

Examples of HMMs are shown in Figure 4.1 and Figure 4.2 (see also Figure 4.4 below).

Note that an output is emitted when visiting a state, not when making a transition, as in the case of a gsm. So the analogy with the gsm model is only partial; it is meant as a motivation for HMMs.

The hidden Markov model was developed by L. E. Baum and colleagues at the Institue for Defence Analysis at Princeton (including Petrie, Eagon, Sell, Soules, and Weiss) starting in 1966.

If we ignore the output components $\mathbb{O}$ and B, then we have what is called a *Markov chain*. A good interpretation of a Markov chain is the evolution over (discrete) time of the populations of n species that may change from one species to another. The probability $A(i,j)$ is the fraction of the population of the ith species that changes to the jth species. If we denote the populations at time t by the row vector $x = (x_1, \ldots, x_n)$, and the populations at time $t+1$ by $y = (y_1, \ldots, y_n)$, then

$$y_j = A(1,j)x_1 + \cdots + A(i,j)x_i + \cdots + A(n,j)x_n, \quad 1 \leq j \leq n,$$

in matrix form, $y = xA$. The condition $\sum_{j=1}^{n} A(i,j) = 1$ expresses that the total population is preserved, namely $y_1 + \cdots + y_n = x_1 + \cdots + x_n$.

Remark: This remark is intended for the reader who knows some probability theory, and *it can be skipped without any negative effect on understanding the rest of this chapter*. Given a probability space $(\Omega, \mathcal{F}, \mu)$ and any countable set Q (for simplicity we may assume Q is finite), a *stochastic discrete-parameter process with state space* Q is a countable family $(X_t)_{t\in\mathbb{N}}$ of random variables $X_t \colon \Omega \to Q$. We can think of t as time, and for any $q \in Q$, of $\mathsf{Pr}(X_t = q)$ as the probability that the process X is in state q at time t. Note that for such a process, the stating time is $t = 0$. If

$$\mathsf{Pr}(X_t = q \mid X_0 = q_0, \ldots, X_{t-2} = q_{t-2}, X_{t-1} = p) = \mathsf{Pr}(X_t = q \mid X_{t-1} = p)$$

for all $q_0, , \ldots, q_{t-2}, p, q \in Q$ and for all $t \geq 1$, and if the probability on the right-hand side is independent of t, then we say that $X = (X_t)_{t\in\mathbb{N}}$ is a *time-homogeneous Markov chain*, for short, *Markov chain*. Informally, the "next" state X_t of the process is independent of the "past" states $X_0, \ldots, X_{t-2}$, provided that the "present" state X_{t-1} is known.

Since for simplicity Q is assumed to be finite, there is a bijection $\sigma \colon Q \to \{1, \ldots, n\}$, and then, the process X is completely determined by the probabilities

$$a_{ij} = \mathsf{Pr}(X_t = q \mid X_{t-1} = p), \quad i = \sigma(p),\ j = \sigma(q),\ p, q \in Q,$$

and if Q is a finite state space of size n, these form an $n \times n$ matrix $A = (a_{ij})$ called the *Markov matrix* of the process X. It is a row-stochastic matrix.

The beauty of Markov chains is that if we write

$$\pi(i) = \mathsf{Pr}(X_0 = i)$$

for the initial probability distribution, then the joint probability distribution of $X_0, X_1, \ldots,$ X_t is given by

$$\mathsf{Pr}(X_0 = i_0, X_1 = i_1, \ldots, X_t = i_t) = \pi(i_0)A(i_0, i_1) \cdots A(i_{t-1}, i_t).$$

The above expression only involves π and the matrix A, and makes no mention of the original measure space. Therefore, it *doesn't matter what the probability space is!*

Conversely, given an $n \times n$ row-stochastic matrix A, let Ω be the set of all countable sequences $\omega = (\omega_0, \omega_1, \ldots, \omega_t, \ldots)$ with $\omega_t \in Q = \{1, \ldots, n\}$ for all $t \in \mathbb{N}$, and let $X_t \colon \Omega \to Q$

be the projection on the tth component, namely $X_t(\omega) = \omega_t$.[2] Then it is possible to define a σ-algebra (also called a σ-field) $\mathcal{B}$ and a measure μ on $\mathcal{B}$ such that $(\Omega, \mathcal{B}, \mu)$ is a probability space, and $X = (X_t)_{t\in\mathbb{N}}$ is a Markov chain with corresponding Markov matrix A.

To define $\mathcal{B}$, proceed as follows. For every $t \in \mathbb{N}$, let $\mathcal{F}_t$ be the family of all unions of subsets of Ω of the form

$$\{\omega \in \Omega \mid (X_0(\omega) \in S_0) \wedge (X_1(\omega) \in S_1) \wedge \cdots \wedge (X_t(\omega) \in S_t)\},$$

where $S_0, S_1, \ldots, S_t$ are subsets of the state space $Q = \{1, \ldots, n\}$. It is not hard to show that each $\mathcal{F}_t$ is a σ-algebra. Then let

$$\mathcal{F} = \bigcup_{t\geq 0} \mathcal{F}_t.$$

Each set in $\mathcal{F}$ is a set of paths for which a finite number of outcomes are restricted to lie in certain subsets of $Q = \{1, \ldots, n\}$. All other outcomes are unrestricted. In fact, every subset C in $\mathcal{F}$ is a countable union

$$C = \bigcup_{i\in\mathbb{N}} B_i^{(t)}$$

of sets of the form

$$\begin{aligned} B_i^{(t)} &= \{\omega \in \Omega \mid \omega = (q_0, q_1, \ldots, q_t, s_{t+1}, \ldots . s_j, \ldots,) \mid q_0, q_1, \ldots, q_t \in Q\} \\ &= \{\omega \in \Omega \mid X_0(\omega) = q_0, X_1(\omega) = q_1, \ldots, X_t(\omega) = q_t\}. \end{aligned}$$

The sequences in $B_i^{(t)}$ are those beginning with the fixed sequence $(q_0, q_1, \ldots, q_t)$. One can show that $\mathcal{F}$ is a field of sets (a boolean algebra), but not necessarily a σ-algebra, so we form the smallest σ-algebra $\mathcal{G}$ containing $\mathcal{F}$.

Using the matrix A we can define the measure $\nu(B_i^{(t)})$ as the product of the probabilities along the sequence $(q_0, q_1, \ldots, q_t)$. Then it can be shown that ν can be extended to a measure μ on $\mathcal{G}$, and we let $\mathcal{B}$ be the σ-algebra obtained by adding to $\mathcal{G}$ all subsets of sets of measure zero. The resulting probability space $(\Omega, \mathcal{B}, \mu)$ is usually called the *sequence space*, and the measure μ is called the *tree measure*. Then it is easy to show that the family of random variables $X_t \colon \Omega \to Q$ on the probability space$(\Omega, \mathcal{B}, \mu)$ is a time-homogeneous Markov chain whose Markov matrix is the original matrix A. The above construction is presented in full detail in Kemeny, Snell, and Knapp [4] (Chapter 2, Sections 1 and 2).

Most presentations of Markov chains do not even mention the probability space over which the random variables X_t are defined. This makes the whole thing quite mysterious, since the probabilities $\mathsf{Pr}(X_t = q)$ are by definition given by

$$\mathsf{Pr}(X_t = q) = \mu(\{\omega \in \Omega \mid X_t(\omega) = q\}),$$

[2]It is customary in probability theory to denote events by the letter ω. In the present case, ω denotes a countable sequence of elements from Q. This notation has nothing do with the bijection $\omega \colon \mathbb{O} \to \{1, \ldots, m\}$ occurring in Definition 4.1.

which requires knowing the measure μ. This is more problematic if we start with a stochastic matrix. What are the random variables X_t, what are they defined on? The above construction puts things on firm grounds.

After this long digression we now return to HMM's. There are three types of problems that can be solved using HMMs:

(1) **The decoding problem**: Given an HMM $M = (Q, \mathbb{O}, \pi, A, B)$, for any observed output sequence $\mathcal{O} = (O_1, O_2, \ldots, O_T)$ of length $T \geq 1$, find a most likely sequence of states $\mathcal{S} = (q_1, q_2, \ldots, q_T)$ that produces the output sequence $\mathcal{O}$. More precisely, with our notational convention that $\sigma(q_t) = i_t$ and $\omega(O_t) = \omega_t$, this means finding a sequence $\mathcal{S}$ such that the probability

$$\mathsf{Pr}(\mathcal{S}, \mathcal{O}) = \pi(i_1)B(i_1, \omega_1)\prod_{t=2}^{T} A(i_{t-1}, i_t)B(i_t, \omega_t)$$

is maximal. This problem is solved effectively by the *Viterbi algorithm* that we outlined before.

(2) **The evaluation problem**, also called **the likelyhood problem**: Given a finite collection $\{M_1, \ldots, M_L\}$ of HMM's with the same output alphabet O, for any output sequence $\mathcal{O} = (O_1, O_2, \ldots, O_T)$ of length $T \geq 1$, find which model M_ℓ is most likely to have generated $\mathcal{O}$. More precisely, given any model M_k, we compute the probability $tprob_k$ that M_k could have produced $\mathcal{O}$ along any path. Then we pick an HMM M_ℓ for which $tprob_\ell$ is maximal. We will return to this point after having described the Viterbi algoritm. A variation of the Viterbi algorithm called the *forward algorithm* effectively solves the evaluation problem.

(3) **The training problem**, also called **the learning problem**: Given a set $\{\mathcal{O}_1, \ldots, \mathcal{O}_r\}$ of output sequences on the same output alpabet O, usually called a set of *training data*, given Q, find the "best" π, A, and B for an HMM M that produces all the sequences in the training set, in the sense that the HMM $M = (Q, \mathbb{O}, \pi, A, B)$ is the most likely to have produced the sequences in the training set. The technique used here is called *expectation maximization*, or *EM*. It is an iterative method that starts with an initial triple π, A, B, and tries to impove it. There is such an algorithm known as the *Baum-Welch* or *forward-backward algorithm*, but it is beyond the scope of this introduction.

Let us now describe the Viterbi algorithm in more details.

4.2 The Viterbi Algorithm and the Forward Algorithm

Given an HMM $M = (Q, \mathbb{O}, \pi, A, B)$, for any observed output sequence $\mathcal{O} = (O_1, O_2, \ldots, O_T)$ of length $T \geq 1$, we want to find a most likely sequence of states $\mathcal{S} = (q_1, q_2, \ldots, q_T)$ that produces the output sequence $\mathcal{O}$.

Using the bijections $\sigma\colon Q \to \{1, \ldots, n\}$ and $\omega\colon \mathbb{O} \to \{1, \ldots, m\}$, we can work with sequences of indices, and recall that we denote the index $\sigma(q_t)$ associated with the tth state q_t in the sequence $\mathcal{S}$ by i_t, and the index $\omega(O_t)$ associated with the tth output O_t in the sequence $\mathcal{O}$ by ω_t. Then we need to find a sequence $\mathcal{S}$ such that the probability

$$\mathsf{Pr}(\mathcal{S}, \mathcal{O}) = \pi(i_1)B(i_1, \omega_1)\prod_{t=2}^{T} A(i_{t-1}, i_t)B(i_t, \omega_t)$$

is maximal.

In general, there are n^T sequences of length T. We can draw a trellis consisting of T verticals layers of n nodes (the states), and draw n oriented edges from the ith state in the jth vertical layer to all n states in the $(j+1)$th vertical layer $(1 \leq i \leq n, 1 \leq j \leq T-1)$. There are exactly n^T paths in this trellis.

The problem can be solved efficiently by a method based on *dynamic programming*. For any t, $1 \leq t \leq T$, for any state $q \in Q$, if $\sigma(q) = j$, then we compute $score(j, t)$, which is the largest probability that a sequence $(q_1, \ldots, q_{t-1}, q)$ of length t ending with q has produced the output sequence $(O_1, \ldots, O_{t-1}, O_t)$.

The point is that if we know $score(k, t-1)$ for $k = 1, \ldots, n$ (with $t \geq 2$), then we can find $score(j, t)$ for $j = 1, \ldots, n$, because if we write $k = \sigma(q_{t-1})$ and $j = \sigma(q)$ (recall that $\omega_t = \omega(O_t)$), then the probability associated with the path $(q_1, \ldots, q_{t-1}, q)$ is

$$tscore(k, j) = score(k, t-1)A(k, j)B(j, \omega_t).$$

See the illustration below:

$$\begin{tikzcd}
\text{state indices} & i_1 & \cdots & k & j \\
\text{states} & q_1 \arrow[u, "\sigma"] \arrow[d, Rightarrow] & \cdots \arrow[r, dotted, "score(k,t-1)"] & q_{t-1} \arrow[u, "\sigma"] \arrow[d, Rightarrow] \arrow[r, dotted, "A(k,j)"] & q \arrow[u, "\sigma"] \arrow[d, Rightarrow, "B(j,\omega_t)"] \\
\text{outputs} & O_1 \arrow[d, "\omega"] & \cdots & O_{t-1} \arrow[d, "\omega"] & O_t \arrow[d, "\omega"] \\
\text{output indices} & \omega_1 & \cdots & \omega_{t-1} & \omega_t
\end{tikzcd}$$

So to maximize this probability, we just have to find the maximum of the probabilities $tscore(k, j)$ over all k, that is, we must have

$$score(j, t) = \max_k tscore(k, j).$$

See the illustration below:

$$\begin{tikzcd}
\sigma^{-1}(1) \arrow[dr, "tscore(1{,}j)"] & \\
\sigma^{-1}(k) \arrow[r, "tscore(k{,}j)"] & q = \sigma^{-1}(j) \\
\sigma^{-1}(n) \arrow[ur, "tscore(n{,}j)"'] &
\end{tikzcd}$$

To get started, we set $score(j, 1) = \pi(j)B(j, \omega_1)$ for $j = 1, \ldots, n$.

The algorithm goes through a forward phase for $t = 1, \ldots, T$, during which it computes the probabilities $score(j, t)$ for $j = 1, \ldots, n$. When $t = T$, we pick an index j such that $score(j, T)$ is maximal. The machine learning community is fond of the notation

$$j = \arg\max_k score(k, T)$$

to express the above fact. Typically, the *smallest index* j corresponding the maximum element in the list of probabilities

$$(score(1, T), score(2, T), \ldots, score(n, T))$$

is returned. This gives us the last state $q_T = \sigma^{-1}(j)$ in an optimal sequence that yields the output sequence $\mathcal{O}$.

The algorithm then goes through a path retrieval phase. To do this, when we compute

$$score(j, t) = \max_k tscore(k, j),$$

we also record the index $k = \sigma(q_{t-1})$ of the state q_{t-1} in the best sequence $(q_1, \ldots, q_{t-1}, q_t)$ for which $tscore(k, j)$ is maximal (with $j = \sigma(q_t)$), as $pred(j, t) = k$. The index k is often called the *backpointer* of j at time t. This index may not be unique, we just pick one of them. Again, this can be expressed by

$$pred(j, t) = \arg\max_k tscore(k, j).$$

Typically, the *smallest index* k corresponding the maximum element in the list of probabilities

$$(tscore(1, j), tscore(2, j), \ldots, tscore(n, j))$$

is returned.

The predecessors $pred(j, t)$ are only defined for $t = 2, \ldots, T$, but we can let $pred(j, 1) = 0$.

Observe that the path retrieval phase of the Viterbi algorithm is very similar to the phase of Dijkstra's algorithm for finding a shortest path that follows the *prev* array. One should not confuse this phase with what is called the *backward algorithm*, which is used in solving the learning problem. The forward phase of the Viterbi algorithm is quite different from the Dijkstra's algorithm, and the Viterbi algorithm is actually simpler (it computes $\mathit{score}(j,t)$ for all states and for $t = 1,\dots,T$), whereas Dijkstra's algorithm maintains a list of unvisited vertices, and needs to pick the next vertex). The major difference is that the Viterbi algorithm *maximizes a product* of weights along a path, but Dijkstra's algorithm *minimizes a sum* of weights along a path. Also, the Viterbi algorithm knows the length of the path (T) ahead of time, but Dijkstra's algorithm does not.

The Viterbi algorithm, invented by Andrew Viterbi in 1967, is shown below.

The input to the algorithm is $M = (Q, \mathbb{O}, \pi, A, B)$ and the sequence of indices $\omega(\mathcal{O}) = (\omega_1,\dots,\omega_T)$ associated with the observed sequence $\mathcal{O} = (O_1, O_2, \dots, O_T)$ of length $T \geq 1$, with $\omega_t = \omega(O_t)$ for $t = 1,\dots,T$.

The output is a sequence of states $(q_1,\dots,q_T)$. This sequence is determined by the sequence of indices $(I_1,\dots,I_T)$; namely, $q_t = \sigma^{-1}(I_t)$.

The Viterbi Algorithm

```
begin
  for $j = 1$ to $n$ do
    $\mathit{score}(j,1) = \pi(j)B(j,\omega_1)$
  endfor;
  (∗ forward phase to find the best (highest) scores ∗)
  for $t = 2$ to $T$ do
    for $j = 1$ to $n$ do
      for $k = 1$ to $n$ do
        $\mathit{tscore}(k) = \mathit{score}(k,t-1)A(k,j)B(j,\omega_t)$
      endfor;
      $\mathit{score}(j,t) = \max_k \mathit{tscore}(k)$;
      $\mathit{pred}(j,t) = \arg\max_k \mathit{tscore}(k)$
    endfor
  endfor;
  (∗ second phase to retrieve the optimal path ∗)
  $I_T = \arg\max_j \mathit{score}(j,T)$;
  $q_T = \sigma^{-1}(I_T)$;
  for $t = T$ to 2 by $-1$ do
    $I_{t-1} = \mathit{pred}(I_t,t)$;
```

```
        $q_{t-1} = \sigma^{-1}(I_{t-1})$
    **endfor**
  **end**
```

An illustration of the Viterbi algorithm applied to Example 4.1 was presented after Example 4.2. If we run the Viterbi algorithm on the output sequence (S, M, S, L) of Example 4.2, we find that the sequence (Cold, Cold, Cold, Hot) has the highest probability, 0.00282, among all sequences of length four.

One may have noticed that the numbers involved, being products of probabilities, become quite small. Indeed, underflow may arise in dynamic programming. Fortunately, there is a simple way to avoid underflow by taking logarithms. We initialize the algorithm by computing

$$score(j, 1) = \log[\pi(j)] + \log[B(j, \omega_1)],$$

and in the step where *tscore* is computed we use the formula

$$tscore(k) = score(k, t-1) + \log[A(k, j)] + \log[B(j, \omega_t)].$$

It immediately verified that the time complexity of the Viterbi algorithm is $O(n^2T)$.

Let us now to turn to the second problem, the evaluation problem (or likelyhood problem).

This time, given a finite collection $\{M_1, \ldots, M_L\}$ of HMM's with the same output alphabet O, for any observed output sequence $\mathcal{O} = (O_1, O_2, \ldots, O_T)$ of length $T \geq 1$, find which model M_ℓ is most likely to have generated $\mathcal{O}$. More precisely, given any model M_k, we compute the probability $tprob_k$ that M_k could have produced $\mathcal{O}$ along any sequence of states $\mathcal{S} = (q_1, \ldots, q_T)$. Then we pick an HMM M_ℓ for which $tprob_\ell$ is maximal.

The probability $tprob_k$ that we are seeking is given by

$$\begin{aligned} tprob_k &= \mathsf{Pr}(\mathcal{O}) \\ &= \sum_{(i_1,\ldots,i_T)\in\{1,\ldots,n\}^T} \mathsf{Pr}((q_{i_1}, \ldots, q_{i_T}), \mathcal{O}) \\ &= \sum_{(i_1,\ldots,i_T)\in\{1,\ldots,n\}^T} \pi(i_1)B(i_1, \omega_1)\prod_{t=2}^{T} A(i_{t-1}, i_t)B(i_t, \omega_t), \end{aligned}$$

where $\{1, \ldots, n\}^T$ denotes the set of all sequences of length T consisting of elements from the set $\{1, \ldots, n\}$.

It is not hard to see that a brute-force computation requires $2Tn^T$ multiplications. Fortunately, it is easy to adapt the Viterbi algorithm to compute $tprob_k$ efficiently. Since we are not looking for an explicity path, there is no need for the second phase, and during the forward phase, going from $t-1$ to t, rather than finding the maximum of the scores $tscore(k)$ for $k = 1, \ldots, n$, we just set $score(j, t)$ to the sum over k of the temporary scores $tscore(k)$. At the end, $tprob_k$ is the sum over j of the probabilities $score(j, T)$.

The algorithm solving the evaluation problem known as the *forward algorithm* is shown below.

The input to the algorithm is $M = (Q, \mathbb{O}, \pi, A, B)$ and the sequence of indices $\omega(\mathcal{O}) = (\omega_1, \ldots, \omega_T)$ associated with the observed sequence $\mathcal{O} = (O_1, O_2, \ldots, O_T)$ of length $T \geq 1$, with $\omega_t = \omega(O_t)$ for $t = 1, \ldots, T$. The output is the probability *tprob*.

The Foward Algorithm

```
begin
  for $j = 1$ to $n$ do
    $score(j, 1) = \pi(j)B(j, \omega_1)$
  endfor;
  for $t = 2$ to $T$ do
    for $j = 1$ to $n$ do
      for $k = 1$ to $n$ do
        $tscore(k) = score(k, t-1)A(k, j)B(j, \omega_t)$
      endfor;
      $score(j, t) = \sum_k tscore(k)$
    endfor
  endfor;
  $tprob = \sum_j score(j, T)$
end
```

We can now run the above algorithm on $M_1, \ldots, M_L$ to compute $tprob_1, \ldots, tprob_L$, and we pick the model M_ℓ for which $tprob_\ell$ is maximum.

As for the Viterbi algorithm, the time complexity of the forward algorithm is $O(n^2T)$.

Underflow is also a problem with the forward algorithm. At first glance it looks like taking logarithms does not help because there is no simple expression for $\log(x_1 + \cdots + x_n)$ in terms of the $\log x_i$. Fortunately, we can use the *log-sum exp trick* (which I learned from Mitch Marcus), namely the identity

$$\log\left(\sum_{i=1}^n e^{x_i}\right) = a + \log\left(\sum_{i=1}^n e^{x_i - a}\right)$$

for all $x_1, \ldots, x_n \in \mathbb{R}$ and $a \in \mathbb{R}$ (take exponentials on both sides). Then, if we pick $a = \max_{1 \leq i \leq n} x_i$, we get

$$1 \leq \sum_{i=1}^n e^{x_i - a} \leq n,$$

so

$$\max_{1\leq i\leq n} x_i \leq \log\left(\sum_{i=1}^{n} e^{x_i}\right) \leq \max_{1\leq i\leq n} x_i + \log n,$$

which shows that $\max_{1\leq i\leq n} x_i$ is a good approximation for $\log\left(\sum_{i=1}^{n} e^{x_i}\right)$. For any positive reals $y_1, \ldots, y_n$, if we let $x_i = \log y_i$, then we get

$$\log\left(\sum_{i=1}^{n} y_i\right) = \max_{1\leq i\leq n} \log y_i + \log\left(\sum_{i=1}^{n} e^{\log(y_i)-a}\right), \quad \text{with} \quad a = \max_{1\leq i\leq n} \log y_i.$$

We will use this trick to compute

$$\log(score(j,k)) = \log\left(\sum_{k=1}^{n} e^{\log(tscore(k))}\right) = a + \log\left(\sum_{k=1}^{n} e^{\log(tscore(k))-a}\right)$$

with $a = \max_{1\leq k\leq n} \log(tscore(k))$, where $tscore((k)$ could be very small, but $\log(tscore(k))$ is not, so computing $\log(tscore(k)) - a$ does not cause underflow, and

$$1 \leq \sum_{k=1}^{n} e^{\log(tscore(k))-a} \leq n,$$

since $\log(tscore(k)) - a \leq 0$ and one of these terms is equal to zero, so even if some of the terms $e^{\log(tscore(k))-a}$ are very small, this does not cause any trouble. We will also use this trick to compute $\log(tprob) = \log\left(\sum_{j=1}^{n} score(j,T)\right)$ in terms of the $\log(score(j,T))$.

We leave it as an exercise to the reader to modify the forward algorithm so that it computes $\log(score(j,t))$ and $\log(tprob)$ using the log-sum exp trick. If you use `Matlab`, then this is quite easy because `Matlab` does a lot of the work for you since it can apply operators such as exp or $\sum$ (sum) to vectors.

Example 4.3. To illustrate the forward algorithm, assume that our observant student also recorded the drinking behavior of a professor at Harvard, and that he came up with the HHM shown in Figure 4.4.

However, the student can't remember whether he observed the sequence NNND at Penn or at Harvard. So he runs the forward algorithm on both HMM's to find the most likely model. Do it!

Following Jurafsky, the following chronology shows how of the Viterbi algorithm has had applications in many separate fields.

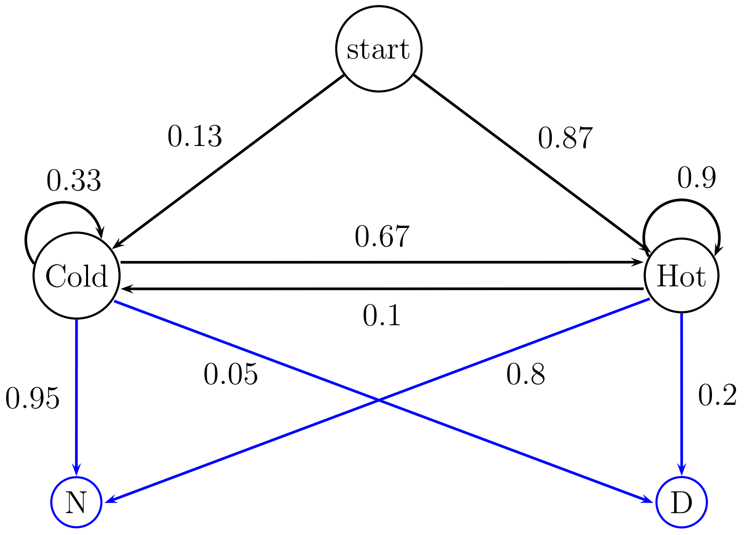


Figure 4.4: Example of an HMM modeling the "drinking behavior" of a professor at Harvard.

Citation	**Field**
Viterbi (1967)	information theory
Vintsyuk (1968)	speech processing
Needleman and Wunsch (1970)	molecular biology
Sakoe and Chiba (1971)	speech processing
Sankoff (1972)	molecular biology
Reichert et al. (1973)	molecular biology
Wagner and Fischer (1974)	computer science

Readers who wish to learn more about HMMs should begin with Stamp [8], a great tutorial which contains a very clear and easy to read presentation. Another nice introduction is given in Rich [7] (Chapter 5, Section 5.11). A much more complete, yet accessible, coverage of HMMs is found in Rabiner's tutorial [6]. Jurafsky and Martin's online Chapter 9 (Hidden Markov Models) is also a very good and informal tutorial (see https://web.stanford.edu/˜jurafsky/slp3/9.pdf).

A very clear and quite accessible presentation of Markov chains is given in Cinlar [2]. Another thorough but a bit more advanced presentation is given in Brémaud [1]. Other presentations of Markov chains can be found in Mitzenmacher and Upfal [5], and in Grimmett and Stirzaker [3].

Acknowledgments: I would like to thank Mitch Marcus, Jocelyn Qaintance, and Joao Sedoc, for scrutinizing my work and for many insightful comments.

Chapter 5

Regular Languages and Regular Expressions

5.1 Directed Graphs and Paths

It is often useful to view DFA's and NFA's as labeled directed graphs.

Definition 5.1. A *directed graph* is a quadruple $G = (V, E, s, t)$, where V is a set of *vertices, or nodes*, E is a set of *edges, or arcs*, and $s, t \colon E \to V$ are two functions, s being called the *source* function, and t the *target* function. Given an edge $e \in E$, we also call $s(e)$ the *origin* (or *source*) of e, and $t(e)$ the *endpoint* (or *target*) of e.

Remark: The functions s, t need not be injective or surjective. Thus, we allow "isolated vertices."

Example 5.1. Let G be the directed graph defined such that

$E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}$,

$V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$, and

$$\begin{array}{llll}
s(e_1) = v_1, & s(e_2) = v_2, & s(e_3) = v_3, & s(e_4) = v_4, \\
s(e_5) = v_2, & s(e_6) = v_5, & s(e_7) = v_5, & s(e_8) = v_5, \\
t(e_1) = v_2, & t(e_2) = v_3, & t(e_3) = v_4, & t(e_4) = v_2, \\
t(e_5) = v_5, & t(e_6) = v_5, & t(e_7) = v_6, & t(e_8) = v_6.
\end{array}$$

Such a graph can be represented by the diagram showen in Figure 5.1.

In drawing directed graphs, we will usually omit edge names (the e_i), and sometimes even the node names (the v_j).

We now define paths in a directed graph.

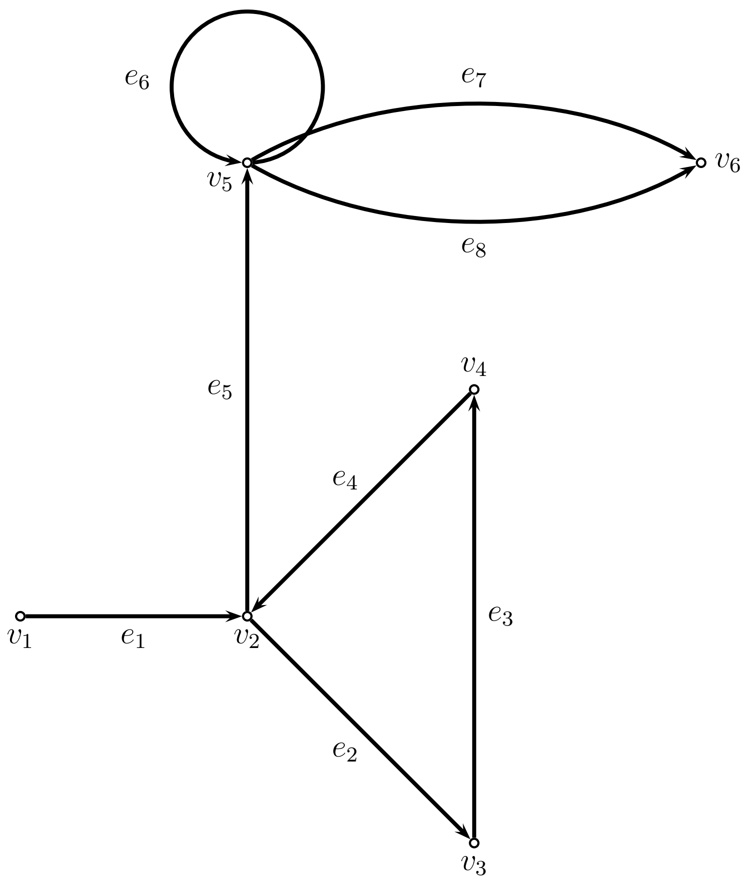


Figure 5.1: A directed graph.

Definition 5.2. Given a directed graph $G = (V, E, s, t)$, for any two nodes $u, v \in V$, a *path from u to v* is a triple $\pi = (u, e_1 \dots e_n, v)$, where $e_1 \dots e_n$ is a string (sequence) of edges in E such that, $s(e_1) = u$, $t(e_n) = v$, and $t(e_i) = s(e_{i+1})$, for all i such that $1 \leq i \leq n - 1$. When $n = 0$, we must have $u = v$, and the path (u, ϵ, u) is called the *null path from u to u*. The number n is the *length* of the path. We also call u the *source* (or *origin*) of the path, and v the *target* (or *endpoint*) of the path. When there is a nonnull path π from u to v, we say that *u and v are connected*.

Remark: In a path $\pi = (u, e_1 \dots e_n, v)$, the expression $e_1 \dots e_n$ is a **sequence**, and thus, the e_i are **not** necessarily distinct.

Example 5.2. The following are paths:

$$\pi_1 = (v_1, e_1e_5e_7, v_6),$$

$$\pi_2 = (v_2, e_2e_3e_4e_2e_3e_4e_2e_3e_4, v_2),$$

and

$$\pi_3 = (v_1, e_1e_2e_3e_4e_2e_3e_4e_5e_6e_6e_8, v_6).$$

Clearly, π_2 and π_3 are of a different nature from π_1. Indeed, they contain cycles. This is formalized as follows.

Definition 5.3. Given a directed graph $G = (V, E, s, t)$, for any node $u \in V$ a *cycle (or loop) through u* is a nonnull path of the form $\pi = (u, e_1 \ldots e_n, u)$ (equivalently, $t(e_n) = s(e_1)$). More generally, a nonnull path $\pi = (u, e_1 \ldots e_n, v)$ *contains a cycle* iff for some i, j, with $1 \leq i \leq j \leq n$, $t(e_j) = s(e_i)$. In this case, letting $w = t(e_j) = s(e_i)$, the path $(w, e_i \ldots e_j, w)$ is a cycle through w. A path π is *acyclic* iff it does not contain any cycle. Note that each null path (u, ϵ, u) is acyclic.

Obviously, a cycle $\pi = (u, e_1 \ldots e_n, u)$ through u is also a cycle through every node $t(e_i)$. Also, a path π may contain several different cycles.

Paths can be concatenated as follows.

Definition 5.4. Given a directed graph $G = (V, E, s, t)$, two paths $\pi_1 = (u, e_1 \ldots e_m, v)$ and $\pi_2 = (u', e'_1 \ldots e'_n, v')$ can be *concatenated* provided that $v = u'$, in which case their *concatenation* is the path

$$\pi_1\pi_2 = (u, e_1 \ldots e_m e'_1 \ldots e'_n, v').$$

It is immediately verified that the concatenation of paths is associative, and that the concatenation of the path $\pi = (u, e_1 \ldots e_m, v)$ with the null path (u, ϵ, u) or with the null path (v, ϵ, v) is the path π itself.

The following fact, although almost trivial, is used all the time, and is worth stating in detail. The proof uses the pigeonhole principle.

Proposition 5.1. *Given a directed graph $G = (V, E, s, t)$, if the set of nodes V contains $m \geq 1$ nodes, then every path π of length at least m contains some cycle.*

A consequence of Proposition 5.1 is that in a finite graph with m nodes, given any two nodes $u, v \in V$, in order to find out whether there is a path from u to v, it is enough to consider paths of length $\leq m - 1$. Indeed, if there is path between u and v, then there is some path π of minimal length (not necessarily unique, but this doesn't matter).

If this minimal path has length at least m, then by the Proposition, it contains a cycle. However, by deleting this cycle from the path π, we get an even shorter path from u to v, contradicting the minimality of π.

We now turn to labeled graphs.

5.2 Labeled Graphs and Automata

In fact, we only need edge-labeled graphs.

Definition 5.5. A *labeled directed graph* is a tuple $G = (V, E, L, s, t, \lambda)$, where V is a set of *vertices, or nodes*, E is a set of *edges, or arcs*, L is a set of *labels*, $s, t \colon E \to V$ are two functions, s being called the *source* function, and t the *target* function, and $\lambda \colon E \to L$ is the *labeling function*. Given an edge $e \in E$, we also call $s(e)$ the *origin* (or *source*) of e, $t(e)$ the *endpoint* (or *target*) of e, and $\lambda(e)$ the *label* of e.

Note that the function λ need not be injective or surjective. Thus, distinct edges may have the same label.

Example 5.3. Let G be the directed graph defined such that

$E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}$,

$V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$, $L = \{a, b\}$,

and

$$\begin{array}{llll}
s(e_1) = v_1, & s(e_2) = v_2, & s(e_3) = v_3, & s(e_4) = v_4, \\
s(e_5) = v_2, & s(e_6) = v_5, & s(e_7) = v_5, & s(e_8) = v_5, \\
t(e_1) = v_2, & t(e_2) = v_3, & t(e_3) = v_4, & t(e_4) = v_2, \\
t(e_5) = v_5, & t(e_6) = v_5, & t(e_7) = v_6, & t(e_8) = v_6 \\
\lambda(e_1) = a, & \lambda(e_2) = b, & \lambda(e_3) = a, & \lambda(e_4) = a, \\
\lambda(e_5) = b, & \lambda(e_6) = a, & \lambda(e_7) = a, & \lambda(e_8) = b.
\end{array}$$

Such a labeled graph can be represented by the diagram shown in Figure 5.2.

In drawing labeled graphs, we will usually omit edge names (the e_i), and sometimes even the node names (the v_j).

Paths, cycles, and concatenation of paths are defined just as before (that is, we ignore the labels). However, we can now define the *spelling* of a path.

Definition 5.6. Given a labeled directed graph $G = (V, E, L, s, t, \lambda)$ for any two nodes $u, v \in V$, for any path $\pi = (u, e_1 \ldots e_n, v)$, the *spelling of the path* π is the string of labels

$$\lambda(e_1) \cdots \lambda(e_n).$$

When $n = 0$, the spelling of the null path (u, ϵ, u) is the null string ϵ.

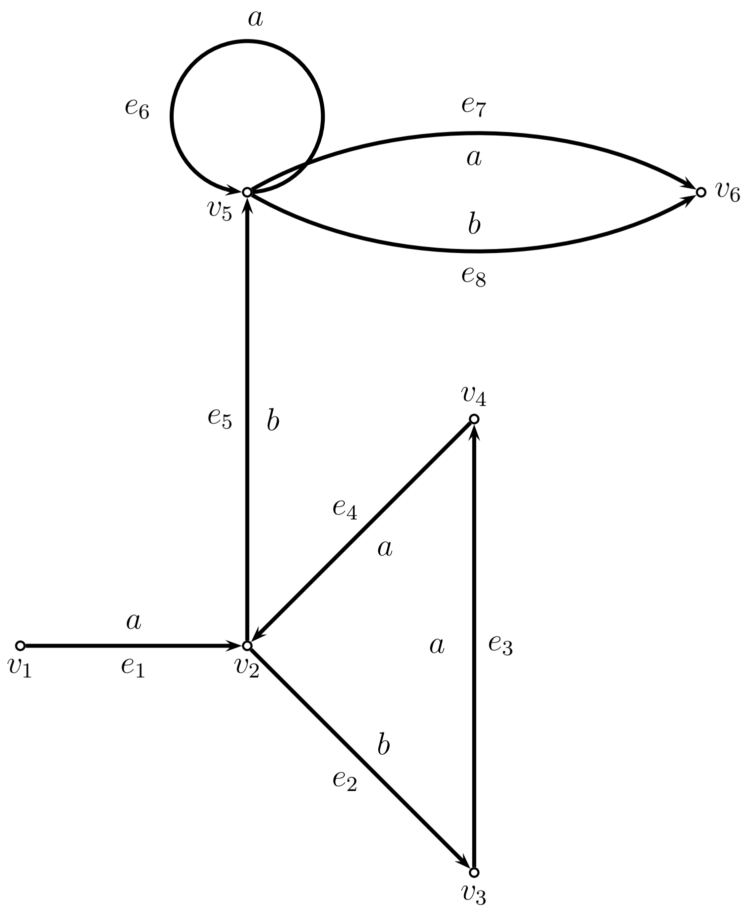


Figure 5.2: A labeled directed graph.

For example, the spelling of the path

$$\pi_3 = (v_1, e_1e_2e_3e_4e_2e_3e_4e_5e_6e_6e_8, v_6)$$

is

$$abaabaabaab.$$

Every DFA and every NFA can be viewed as a labeled graph, in such a way that the set of spellings of paths from the start state to some final state is the language accepted by the automaton in question.

Given a DFA $D = (Q, \Sigma, \delta, q_0, F)$, where $\delta\colon Q \times \Sigma \to Q$, we associate the labeled directed graph $G_D = (V, E, L, s, t, \lambda)$ defined as follows:

$$V = Q, \quad E = \{(p, a, q) \mid q = \delta(p, a),\ p, q \in Q,\ a \in \Sigma\},$$

$L = \Sigma$, $s((p, a, q)) = p$, $t((p, a, q)) = q$, and $\lambda((p, a, q)) = a$.

Such labeled graphs have a special structure that can easily be characterized.

It is easily shown that a string $w \in \Sigma^*$ is in the language $L(D) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}$ iff w is the spelling of some path in G_D from q_0 to some final state.

Similarly, given an NFA $N = (Q, \Sigma, \delta, q_0, F)$, where $\delta \colon Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$, we associate the labeled directed graph $G_N = (V, E, L, s, t, \lambda)$ defined as follows: $V = Q$

$$E = \{(p, a, q) \mid q \in \delta(p, a),\ p, q \in Q,\ a \in \Sigma \cup \{\epsilon\}\},$$

$$L = \Sigma \cup \{\epsilon\},\ \ s((p, a, q)) = p,\ \ t((p, a, q)) = q,$$
$$\lambda((p, a, q)) = a.$$

Remark: : When N has no ϵ-transitions, we can let $L = \Sigma$.

Such labeled graphs have also a special structure that can easily be characterized.

Again, a string $w \in \Sigma^*$ is in the language $L(N) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \cap F \neq \emptyset\}$ iff w is the spelling of some path in G_N from q_0 to some final state.

5.3 The Closure Definition of the Regular Languages

Let $\Sigma = \{a_1, \ldots, a_m\}$ be some alphabet. We would like to define a family of languages, $R(\Sigma)$, by singling out some very basic (atomic) languages, namely the languages $\{a_1\}, \ldots, \{a_m\}$, the empty language, and the trivial language, $\{\epsilon\}$, and then forming more complicated languages by repeatedly forming union, concatenation and Kleene $*$ of previously constructed languages. By doing so, we hope to get a family of languages ($R(\Sigma)$) that is closed under union, concatenation, and Kleene $*$. This means that for any two languages, $L_1, L_2 \in R(\Sigma)$, we also have $L_1 \cup L_2 \in R(\Sigma)$ and $L_1 L_2 \in R(\Sigma)$, and for any language $L \in R(\Sigma)$, we have $L^* \in R(\Sigma)$. Furthermore, we would like $R(\Sigma)$ to be the smallest family with these properties. How do we achieve this rigorously?

Informally, we define the family of languages $R(\Sigma)$ using the following rules:

(1) The languages $\{a_1\}, \ldots, \{a_m\}$, the empty language, and the trivial language $\{\epsilon\}$, called *base languages*, belong to $R(\Sigma)$.

(2a) If L_1 and L_2 belong to $R(\Sigma)$, then $L_1 \cup L_2$ also belongs to $R(\Sigma)$.

(2b) If L_1 and L_2 belong to $R(\Sigma)$, then $L_1 L_2$ also belongs to $R(\Sigma)$.

(2c) If L belongs to $R(\Sigma)$, then L^* also belongs to $R(\Sigma)$.

The issue is to show that the above rules define a family of languages which is the smallest family containing the base languages and closed under union, concaternation, and Kleene $*$.

First, let us look more closely at what we mean by a family of languages. Recall that a language (over Σ) is *any* subset, L, of Σ^*. Thus, the set of all languages is 2^{Σ^*}, the power set of Σ^*. If Σ is nonempty, this is an uncountable set. Next, we define a *family*, $\mathcal{L}$, of languages to be any subset of 2^{Σ^*}. This time, the set of families of languages is $2^{2^{\Sigma^*}}$. This is a huge set. We can use the inclusion relation on $2^{2^{\Sigma^*}}$ to define a partial order on families of languages. So, $\mathcal{L}_1 \subseteq \mathcal{L}_2$ iff for every language, L, if $L \in \mathcal{L}_1$ then $L \in \mathcal{L}_2$.

We can now state more precisely what we are trying to do. Consider the following properties for a family of languages, $\mathcal{L}$:

(1) We have $\{a_1\}, \ldots, \{a_m\}, \emptyset, \{\epsilon\} \in \mathcal{L}$, i.e., $\mathcal{L}$ contains the "atomic" languages.

(2a) For all $L_1, L_2 \in \mathcal{L}$, we also have $L_1 \cup L_2 \in \mathcal{L}$.

(2b) For all $L_1, L_2 \in \mathcal{L}$, we also have $L_1 L_2 \in \mathcal{L}$.

(2c) For all $L \in \mathcal{L}$, we also have $L^* \in \mathcal{L}$.

In other words, $\mathcal{L}$ is closed under union, concatenation and Kleene $*$.

Now, what we want is the smallest (w.r.t. inclusion) family of languages that satisfies properties (1) and (2)(a)(b)(c). We can construct such a family using an *inductive definition*. This inductive definition constructs a sequence of families of languages, $(R(\Sigma)_n)_{n\geq 0}$, called the *stages of the inductive definition*, as follows:

$$\begin{aligned} R(\Sigma)_0 &= \{\{a_1\}, \ldots, \{a_m\}, \emptyset, \{\epsilon\}\}, \\ R(\Sigma)_{n+1} &= R(\Sigma)_n \cup \{L_1 \cup L_2,\ L_1L_2,\ L^* \mid L_1, L_2, L \in R(\Sigma)_n\}. \end{aligned}$$

Then, we define $R(\Sigma)$ by

$$R(\Sigma) = \bigcup_{n\geq 0} R(\Sigma)_n.$$

Thus, a language L belongs to $R(\Sigma)$ iff it belongs L_n, for some $n \geq 0$.

For example, if $\Sigma = \{a, b\}$, we have

$$\begin{aligned} R(\Sigma)_1 = \{&\{a\}, \{b\}, \emptyset, \{\epsilon\}, \\ &\{a, b\}, \{a, \epsilon\}, \{b, \epsilon\}, \\ &\{ab\}, \{ba\}, \{aa\}, \{bb\}, \{a\}^*, \{b\}^*\}. \end{aligned}$$

Some of the languages that will appear in $R(\Sigma)_2$ are:

$$\{a, bb\},\ \{ab, ba\},\ \{abb\},\ \{aabb\},\ \{a\}\{a\}^*,\ \{aa\}\{b\}^*,\ \{bb\}^*.$$

Observe that

$$R(\Sigma)_0 \subseteq R(\Sigma)_1 \subseteq R(\Sigma)_2 \subseteq \cdots R(\Sigma)_n \subseteq R(\Sigma)_{n+1} \subseteq \cdots \subseteq R(\Sigma),$$

so that if $L \in R(\Sigma)_n$, then $L \in R(\Sigma)_p$, for all $p \geq n$. Also, there is some smallest n for which $L \in R(\Sigma)_n$ (the *birthdate* of L!). In fact, all these inclusions are strict. Note that each $R(\Sigma)_n$ only contains a finite number of languages (but some of the languages in $R(\Sigma)_n$ are infinite, because of Kleene $*$).

Then we define the *Regular languages, Version 2*, as the family $R(\Sigma)$.

Of course, it is far from obvious that $R(\Sigma)$ coincides with the family of languages accepted by DFA's (or NFA's), what we call the regular languages, version 1. However, this is the case, and this can be demonstrated by giving two algorithms. Actually, it will be slightly more convenient to define a notation system, the *regular expressions*, to denote the languages in $R(\Sigma)$. Then, we will give an algorithm that converts a regular expression, R, into an NFA, N_R, so that $L_R = L(N_R)$, where L_R is the language (in $R(\Sigma)$) denoted by R. We will also give an algorithm that converts an NFA, N, into a regular expression, R_N, so that $L(R_N) = L(N)$.

But before doing all this, we should make sure that $R(\Sigma)$ is indeed the family that we are seeking. This is the content of

Proposition 5.2. *The family, $R(\Sigma)$, is the smallest family of languages which contains the atomic languages $\{a_1\}, \ldots, \{a_m\}$, $\emptyset$, $\{\epsilon\}$, and is closed under union, concatenation, and Kleene $*$.*

Proof. There are two things to prove.

(i) We need to prove that $R(\Sigma)$ has properties (1) and (2)(a)(b)(c).

(ii) We need to prove that $R(\Sigma)$ is the smallest family having properties (1) and (2)(a)(b)(c).

(i) Since

$$R(\Sigma)_0 = \{\{a_1\}, \ldots, \{a_m\}, \emptyset, \{\epsilon\}\},$$

it is obvious that (1) holds. Next, assume that $L_1, L_2 \in R(\Sigma)$. This means that there are some integers $n_1, n_2 \geq 0$, so that $L_1 \in R(\Sigma)_{n_1}$ and $L_2 \in R(\Sigma)_{n_2}$. Now, it is possible that $n_1 \neq n_2$, but if we let $n = \max\{n_1, n_2\}$, as we observed that $R(\Sigma)_p \subseteq R(\Sigma)_q$ whenever $p \leq q$, we are guaranteed that both $L_1, L_2 \in R(\Sigma)_n$. However, by the definition of $R(\Sigma)_{n+1}$ (that's why we defined it this way!), we have $L_1 \cup L_2 \in R(\Sigma)_{n+1} \subseteq R(\Sigma)$. The same argument proves that $L_1L_2 \in R(\Sigma)_{n+1} \subseteq R(\Sigma)$. Also, if $L \in R(\Sigma)_n$, we immediately have $L^* \in R(\Sigma)_{n+1} \subseteq R(\Sigma)$. Therefore, $R(\Sigma)$ has properties (1) and (2)(a)(b)(c).

(ii) Let $\mathcal{L}$ be any family of languages having properties (1) and (2)(a)(b)(c). We need to prove that $R(\Sigma) \subseteq \mathcal{L}$. If we can prove that $R(\Sigma)_n \subseteq \mathcal{L}$, for all $n \geq 0$, we are done (since then, $R(\Sigma) = \bigcup_{n\geq 0} R(\Sigma)_n \subseteq \mathcal{L}$). We prove by induction on n that $R(\Sigma)_n \subseteq \mathcal{L}$, for all $n \geq 0$.

The base case $n = 0$ is trivial, since $\mathcal{L}$ has (1), which says that $R(\Sigma)_0 \subseteq \mathcal{L}$. Assume inductively that $R(\Sigma)_n \subseteq \mathcal{L}$. We need to prove that $R(\Sigma)_{n+1} \subseteq \mathcal{L}$. Pick any $L \in R(\Sigma)_{n+1}$.

Recall that

$$R(\Sigma)_{n+1} = R(\Sigma)_n \cup \{L_1 \cup L_2,\ L_1L_2,\ L^* \ \mid L_1, L_2, L \in R(\Sigma)_n\}.$$

If $L \in R(\Sigma)_n$, then $L \in \mathcal{L}$, since $R(\Sigma)_n \subseteq \mathcal{L}$, by the induction hypothesis. Otherwise, there are three cases:

(a) $L = L_1 \cup L_2$, where $L_1, L_2 \in R(\Sigma)_n$. By the induction hypothesis, $R(\Sigma)_n \subseteq \mathcal{L}$, so, we get $L_1, L_2 \in \mathcal{L}$; since $\mathcal{L}$ has 2(a), we have $L_1 \cup L_2 \in \mathcal{L}$.

(b) $L = L_1L_2$, where $L_1, L_2 \in R(\Sigma)_n$. By the induction hypothesis, $R(\Sigma)_n \subseteq \mathcal{L}$, so, we get $L_1, L_2 \in \mathcal{L}$; since $\mathcal{L}$ has 2(b), we have $L_1L_2 \in \mathcal{L}$.

(c) $L = L_1^*$, where $L_1 \in R(\Sigma)_n$. By the induction hypothesis, $R(\Sigma)_n \subseteq \mathcal{L}$, so, we get $L_1 \in \mathcal{L}$; since $\mathcal{L}$ has 2(c), we have $L_1^* \in \mathcal{L}$.

Thus, in all cases, we showed that $L \in \mathcal{L}$, and so, $R(\Sigma)_{n+1} \subseteq \mathcal{L}$, which proves the induction step. $\square$

Note: a given language L may be built up in different ways. For example,

$$\{a, b\}^* = (\{a\}^*\{b\}^*)^*.$$

Students should study carefully the above proof. Although simple, it is the prototype of many proofs appearing in the theory of computation.

5.4 Regular Expressions

The definition of the family of languages $R(\Sigma)$ given in the previous section in terms of an inductive definition is good to prove properties of these languages but is it not very convenient to manipulate them in a practical way. To do so, it is better to introduce a symbolic notation system, the *regular expressions*. Regular expressions are certain strings formed according to rules that mimic the inductive rules for constructing the families $R(\Sigma)_n$. The set of regular expressions $\mathcal{R}(\Sigma)$ over an alphabet Σ is a language defined on an alphabet Δ defined as follows.

Given an alphabet $\Sigma = \{a_1, \ldots, a_m\}$, consider the new alphabet

$$\Delta = \Sigma \cup \{+, \cdot, *, (,), \emptyset, \epsilon\}.$$

Informally, we define the family of regular expressions $\mathcal{R}(\Sigma)$ using the following rules:

(1) The strings $a_1, \ldots, a_m$, the empty string ϵ, and the empty set $\emptyset$, called *base regular expressions*, belong to $\mathcal{R}(\Sigma)$.

(2a) If R_1 and R_2 are regular expressions (*i.e.*, belong to $\mathcal{R}(\Sigma)$), then $(R_1 + R_2)$ is a regular expression (*i.e.*, belongs to $\mathcal{R}(\Sigma)$).

(2b) If R_1 and R_2 are regular expressions (*i.e.*, belong to $\mathcal{R}(\Sigma)$), then $(R_1 \cdot R_2)$ is a regular expression (*i.e.*, belongs to $\mathcal{R}(\Sigma)$).

(2c) If R is a regular expression (*i.e.*, belongs to $\mathcal{R}(\Sigma)$), then R^* is a regular expression (*i.e.*, belongs to $\mathcal{R}(\Sigma)$).

More precisely, we define the family $(\mathcal{R}(\Sigma)_n)$ of languages over Δ as follows:

$$\begin{aligned}\mathcal{R}(\Sigma)_0 &= \{a_1, \ldots, a_m, \emptyset, \epsilon\},\\ \mathcal{R}(\Sigma)_{n+1} &= \mathcal{R}(\Sigma)_n \cup \{(R_1 + R_2),\ (R_1 \cdot R_2),\ R^* \mid R_1, R_2, R \in \mathcal{R}(\Sigma)_n\}.\end{aligned}$$

Then, we define $\mathcal{R}(\Sigma)$ as

$$\mathcal{R}(\Sigma) = \bigcup_{n \geq 0} \mathcal{R}(\Sigma)_n.$$

Note that every language $\mathcal{R}(\Sigma)_n$ is finite.

For example, if $\Sigma = \{a, b\}$, we have

$$\begin{aligned}\mathcal{R}(\Sigma)_1 = \{&a, b, \emptyset, \epsilon,\\ &(a+b), (b+a), (a+a), (b+b), (a+\epsilon), (\epsilon+a),\\ &(b+\epsilon), (\epsilon+b), (a+\emptyset), (\emptyset+a), (b+\emptyset), (\emptyset+b),\\ &(\epsilon+\epsilon), (\epsilon+\emptyset), (\emptyset+\epsilon), (\emptyset+\emptyset),\\ &(a\cdot b), (b\cdot a), (a\cdot a), (b\cdot b), (a\cdot\epsilon), (\epsilon\cdot a),\\ &(b\cdot\epsilon), (\epsilon\cdot b), (\epsilon\cdot\epsilon), (a\cdot\emptyset), (\emptyset\cdot a),\\ &(b\cdot\emptyset), (\emptyset\cdot b), (\epsilon\cdot\emptyset), (\emptyset\cdot\epsilon), (\emptyset\cdot\emptyset),\\ &a^*, b^*, \epsilon^*, \emptyset^*\}.\end{aligned}$$

Some of the regular expressions appearing in $\mathcal{R}(\Sigma)_2$ are:

$$\begin{aligned}&(a + (b\cdot b)),\ ((a\cdot b) + (b\cdot a)),\ ((a\cdot b)\cdot b),\\ &((a\cdot a)\cdot(b\cdot b)),\ (a\cdot a^*),\ ((a\cdot a)\cdot b^*),\ (b\cdot b)^*.\end{aligned}$$

Definition 5.7. The set $\mathcal{R}(\Sigma)$ is the set of *regular expressions* (over Σ).

Proposition 5.3. *The language $\mathcal{R}(\Sigma)$ is the smallest language which contains the symbols a_1, ..., a_m, $\emptyset$, ϵ, from Δ, and such that $(R_1 + R_2)$, $(R_1 \cdot R_2)$, and R^*, also belong to $\mathcal{R}(\Sigma)$, when $R_1, R_2, R \in \mathcal{R}(\Sigma)$.*

For simplicity of notation, we write

$$(R_1 R_2)$$

instead of

$$(R_1 \cdot R_2).$$

Example 5.4. The following are regular expressions. $R = (a + b)^*$, $S = (a^* b^*)^*$,

$$T = ((a + b)^* a)(\underbrace{(a + b) \cdots (a + b)}_{n}).$$

5.5 Regular Expressions and Regular Languages

Every regular expression $R \in \mathcal{R}(\Sigma)$ can be viewed as the *name*, or *denotation*, of some language $L \in R(\Sigma)$. Similarly, every language $L \in R(\Sigma)$ is the *interpretation* (or *meaning*) of some regular expression $R \in \mathcal{R}(\Sigma)$.

Think of a regular expression R as a *program*, and of $\mathcal{L}(R)$ as the result of the *execution* or *evaluation*, of R by $\mathcal{L}$. This can be made rigorous by defining a function

$$\mathcal{L} \colon \mathcal{R}(\Sigma) \to R(\Sigma).$$

This function is defined recursively as follows:

$$\begin{aligned} \mathcal{L}[a_i] &= \{a_i\}, \\ \mathcal{L}[\emptyset] &= \emptyset, \\ \mathcal{L}[\epsilon] &= \{\epsilon\}, \\ \mathcal{L}[(R_1 + R_2)] &= \mathcal{L}[R_1] \cup \mathcal{L}[R_2], \\ \mathcal{L}[(R_1 R_2)] &= \mathcal{L}[R_1]\mathcal{L}[R_2], \\ \mathcal{L}[R^*] &= \mathcal{L}[R]^*. \end{aligned}$$

Proposition 5.4. *For every regular expression $R \in \mathcal{R}(\Sigma)$, the language $\mathcal{L}[R]$ is regular (version 2), i.e. $\mathcal{L}[R] \in R(\Sigma)$. Conversely, for every regular (version 2) language $L \in R(\Sigma)$, there is some regular expression $R \in \mathcal{R}(\Sigma)$ such that $L = \mathcal{L}[R]$.*

Proof. To prove that $\mathcal{L}[R] \in R(\Sigma)$ for all $R \in \mathcal{R}(\Sigma)$, we prove by induction on $n \geq 0$ that if $R \in \mathcal{R}(\Sigma)_n$, then $\mathcal{L}[R] \in R(\Sigma)_n$. To prove that $\mathcal{L}$ is surjective, we prove by induction on $n \geq 0$ that if $L \in R(\Sigma)_n$, then there is some $R \in \mathcal{R}(\Sigma)_n$ such that $L = \mathcal{L}[R]$. ☐

Note: the function $\mathcal{L}$ is **not** injective.

Example 5.5. If $R = (a + b)^*$, $S = (a^* b^*)^*$, then

$$\mathcal{L}[R] = \mathcal{L}[S] = \{a, b\}^*.$$

For simplicity, we often denote $\mathcal{L}[R]$ as L_R.

Example 5.6. As examples, we have

$$\begin{aligned}
\mathcal{L}[(((ab)b)+a)] &= \{a, abb\}\\
\mathcal{L}[((((a^*b)a^*)b)a^*)] &= \{w \in \{a,b\}^* \mid w \text{ has two } b\text{'s}\}\\
\mathcal{L}[(((((a^*b)a^*)b)a^*)^*a^*)] &= \{w \in \{a,b\}^* \mid w \text{ has an even \# of } b\text{'s}\}\\
\mathcal{L}[(((((((a^*b)a^*)b)a^*)^*a^*)b)a^*)] &= \{w \in \{a,b\}^* \mid w \text{ has an odd \# of } b\text{'s}\}
\end{aligned}$$

Remark: If

$$R = ((a+b)^*a)(\underbrace{(a+b)\cdots(a+b)}_{n}),$$

it can be shown that any minimal DFA accepting L_R has 2^{n+1} states. Yet, both $((a+b)^*a)$ and $(\underbrace{(a+b)\cdots(a+b)}_{n})$ denote languages that can be accepted by "small" DFA's (of size 2 and $n+2$).

Definition 5.8. Two regular expressions $R, S \in \mathcal{R}(\Sigma)$ are *equivalent*, denoted as $R \cong S$, iff $\mathcal{L}[R] = \mathcal{L}[S]$.

It is immediate that $\cong$ is an equivalence relation. The relation $\cong$ satisfies some (nice) identities. For example:

$$\begin{aligned}
(((aa)+b)+c) &\cong ((aa)+(b+c))\\
((aa)(b(cc))) &\cong (((aa)b)(cc))\\
(a^*a^*) &\cong a^*,
\end{aligned}$$

and more generally

$$\begin{aligned}
((R_1+R_2)+R_3) &\cong (R_1+(R_2+R_3)),\\
((R_1R_2)R_3) &\cong (R_1(R_2R_3)),\\
(R_1+R_2) &\cong (R_2+R_1),\\
(R^*R^*) &\cong R^*,\\
R^{**} &\cong R^*.
\end{aligned}$$

There is an algorithm to test the equivalence of regular expressions, but its complexity is exponential. Such an algorithm uses the conversion of a regular expression to an NFA, and the subset construction for converting an NFA to a DFA. Then the problem of deciding whether two regular expressions R and S are equivalent is reduced to testing whether two DFA D_1 and D_2 accept the same languages (the *equivalence problem for DFA's*; see Definition 3.7). As shown in Section 3.2, this last problem is equivalent to testing whether

$$L(D_1) - L(D_2) = \emptyset \quad \text{and} \quad L(D_2) - L(D_1) = \emptyset.$$

But $L(D_1) - L(D_2)$ (and similarly $L(D_2) - L(D_1)$) is accepted by a DFA obtained by the cross-product construction for the relative complement (with final states $F_1 \times \overline{F_2}$ and $F_2 \times \overline{F_1}$). Thus in the end, the equivalence problem for regular expressions reduces to the problem of testing whether a DFA $D = (Q, \Sigma, \delta, q_0, F)$ accepts the empty language, which is equivalent to $Q_r \cap F = \emptyset$. This last problem is a reachability problem in a directed graph which is easily solved in polynomial time.

It is an *open problem* to prove that the problem of testing the equivalence of regular expressions cannot be decided in polynomial time.

In the next two sections we show the equivalence of NFA's and regular expressions, by providing an algorithm to construct an NFA from a regular expression, and an algorithm for constructing a regular expression from an NFA. This will show that the regular languages Version 1 coincide with the regular languages Version 2.

5.6 Regular Expressions and NFA's

Proposition 5.5. *There is an algorithm, which, given any regular expression $R \in \mathcal{R}(\Sigma)$, constructs an NFA N_R accepting L_R, i.e., such that $L_R = L(N_R)$.*

In order to ensure the correctness of the construction as well as to simplify the description of the algorithm it is convenient to assume that our NFA's satisfy the following conditions:

1. Each NFA has a *single* final state, t, distinct from the start state, s.

2. There are *no incoming transitions* into the the start state, s, and *no outgoing transitions* from the final state, t.

3. Every state has at most two incoming and two outgoing transitions.

Here is the algorithm.

For the base case, either

(a) $R = a_i$, in which case, N_R is the following NFA:

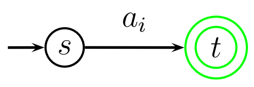


Figure 5.3: NFA for a_i.

(b) $R = \epsilon$, in which case, N_R is the following NFA:

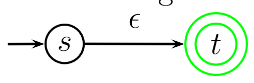


Figure 5.4: NFA for ϵ.

(c) $R = \emptyset$, in which case, N_R is the following NFA:

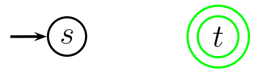


Figure 5.5: NFA for $\emptyset$.

The recursive clauses are as follows:

(i) If our expression is $(R+S)$, the algorithm is applied recursively to R and S, generating NFA's N_R and N_S, and then these two NFA's are combined in parallel as shown in Figure 5.6:

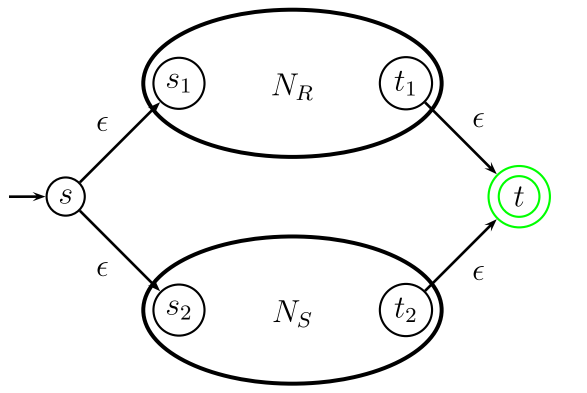


Figure 5.6: NFA for $(R+S)$.

(ii) If our expression is $(R \cdot S)$, the algorithm is applied recursively to R and S, generating NFA's N_R and N_S, and then these NFA's are combined sequentially as shown in Figure 5.7 by merging the "old" final state, t_1, of N_R, with the "old" start state, s_2, of N_S:

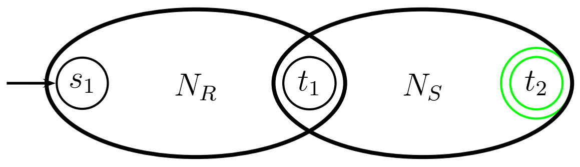


Figure 5.7: NFA for $(R \cdot S)$.

Note that since there are no incoming transitions into s_2 in N_S, once we enter N_S, there is no way of reentering N_R, and so the construction is correct (it yields the concatenation $L_R L_S$).

(iii) If our expression is R^*, the algorithm is applied recursively to R, generating the NFA N_R. Then we construct the NFA shown in Figure 5.8 by adding an ϵ-transition from the "old" final state, t_1, of N_R to the "old" start state, s_1, of N_R and, as ϵ is not necessarily accepted by N_R, we add an ϵ-transition from s to t:

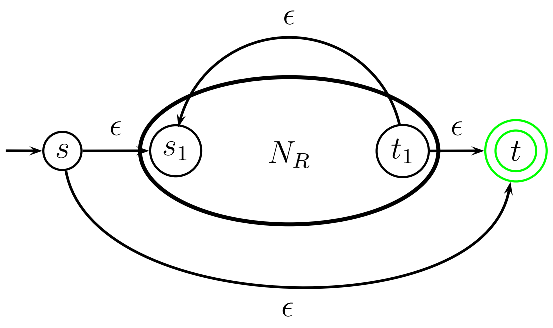


Figure 5.8: NFA for R^*.

Since there are no outgoing transitions from t_1 in N_R, we can only loop back to s_1 from t_1 using the new ϵ-transition from t_1 to s_1 and so the NFA of Figure 5.8 does accept N_R^*.

The algorithm that we just described is sometimes called the "sombrero construction." As a corollary of this construction, we get

Reg. languages version 2 $\subseteq$ Reg. languages, version 1.

Example 5.7. The reader should check that if one constructs the NFA corresponding to the regular expression $(a+b)^*abb$, we obtain the NFA shown in Figure 5.9.

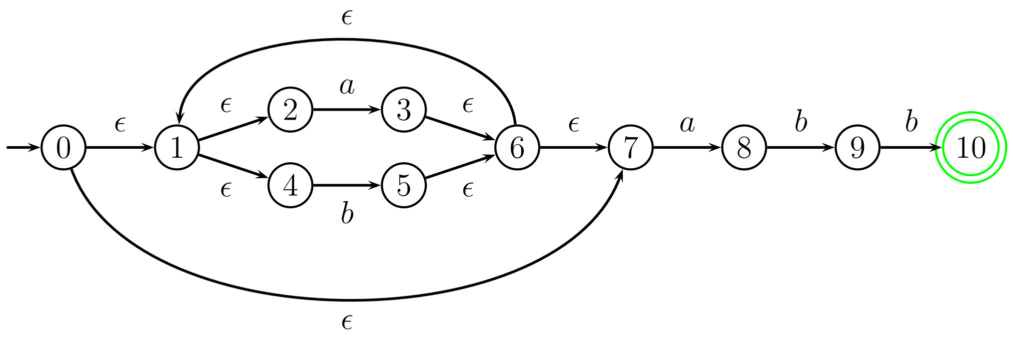


Figure 5.9: An NFA for $R = (a+b)^*abb$.

If we apply the subset construction, one gets the following DFA:

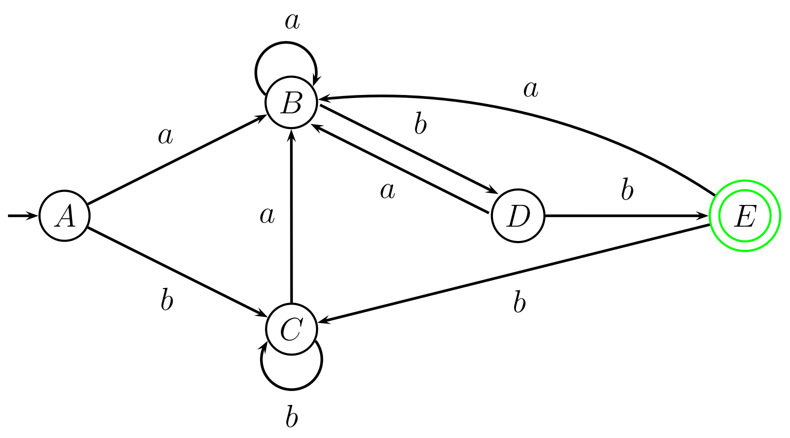


Figure 5.10: A non-minimal DFA for $\{a, b\}^*\{abb\}$.

We now consider the construction of a regular expression from an NFA.

Proposition 5.6. *There is an algorithm, which, given any NFA N, constructs a regular expression $R \in \mathcal{R}(\Sigma)$, denoting $L(N)$, i.e., such that $L_R = L(N)$.*

As a corollary,

Reg. languages version 1 $\subseteq$ Reg. languages, version 2.

This is the *node elimination algorithm*.

The general idea is to allow more general labels on the edges of an NFA, namely, regular expressions. Then, such generalized NFA's are simplified by eliminating nodes one at a time, and readjusting labels.

Preprocessing, phase 1:

If necessary, we need to add a new start state with an ϵ-transition to the old start state, if there are incoming edges into the old start state.

If necessary, we need to add a new (unique) final state with ϵ-transitions from each of the old final states to the new final state, if there is more than one final state or some outgoing edge from any of the old final states.

At the end of this phase, the start state, say s, is a source (no incoming edges), and the final state, say t, is a sink (no outgoing edges).

Preprocessing, phase 2:

We need to "flatten" parallel edges. For any pair of states (p, q) ($p = q$ is possible), if there are k edges from p to q labeled $u_1, \ldots, u_k$, then create a single edge labeled with the regular expression

$$u_1 + \cdots + u_k.$$

For any pair of states (p, q) ($p = q$ is possible) such that there is **no** edge from p to q, we put an edge labeled $\emptyset$.

At the end of this phase, the resulting "*generalized NFA*" is such that for any pair of states (p, q) (where $p = q$ is possible), there is a unique edge labeled with some regular expression denoted as $R_{p,q}$. When $R_{p,q} = \emptyset$, this really means that there is no edge from p to q in the original NFA N.

By interpreting each $R_{p,q}$ as a function call (really, a macro) to the NFA $N_{p,q}$ accepting $\mathcal{L}[R_{p,q}]$ (constructed using the previous algorithm), we can verify that the original language $L(N)$ is accepted by this new generalized NFA.

Node elimination only applies if the generalized NFA has at least one node distinct from s and t.

Pick any node r distinct from s and t. For every pair (p, q) where $p \neq r$ and $q \neq r$, replace the label of the edge from p to q as indicated below:

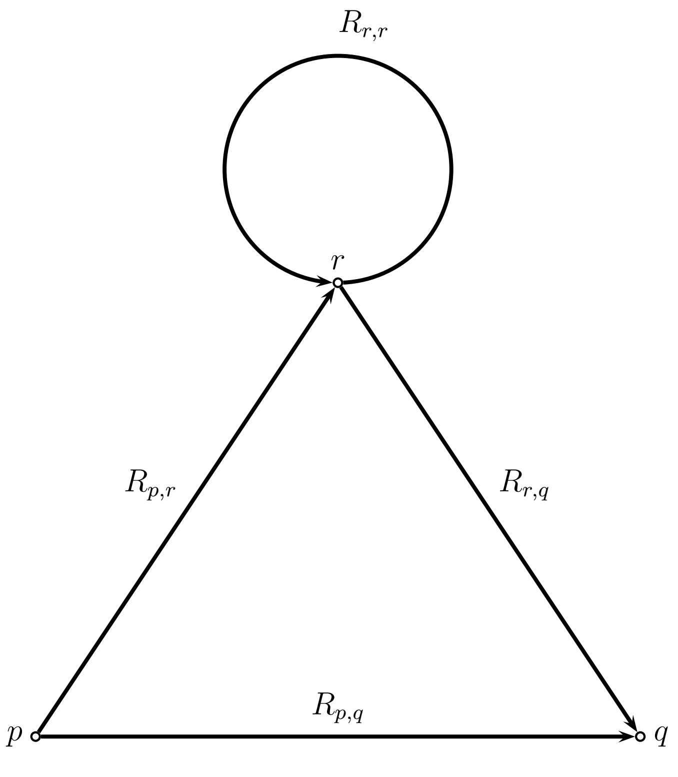


Figure 5.11: Before Eliminating node r.

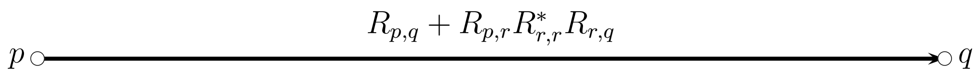


Figure 5.12: After Eliminating node r.

At the end of this step, delete the node r and all edges adjacent to r.

Note that $p = q$ is possible, in which case the triangle is "flat". It is also possible that $p = s$ or $q = t$. Also, this step is performed for all **pairs** (p, q), which means that both (p, q) and (q, p) are considered (when $p \neq q$)).

Note that this step only has an effect if there are edges from p to r and from r to q in the original NFA N. Otherwise, r can simply be deleted, as well as the edges adjacent to r.

Other simplifications can be made. For example, when $R_{r,r} = \emptyset$, we can simplify $R_{p,r}R_{r,r}^*R_{r,q}$ to $R_{p,r}R_{r,q}$. When $R_{p,q} = \emptyset$, we have $R_{p,r}R_{r,r}^*R_{r,q}$.

The order in which the nodes are eliminated is irrelevant, although it affects the size of the final expression.

The algorithm stops when the only remaining nodes are s and t. Then, the label R of the edge from s to t is a regular expression denoting $L(N)$.

Example 5.8. Let

$$\begin{aligned} L = \{w \in \Sigma^* \mid\ & w \text{ contains an odd number of } a\text{'s} \\ & \text{or an odd number of } b\text{'s}\}. \end{aligned}$$

An NFA for L after the preprocessing phase is:

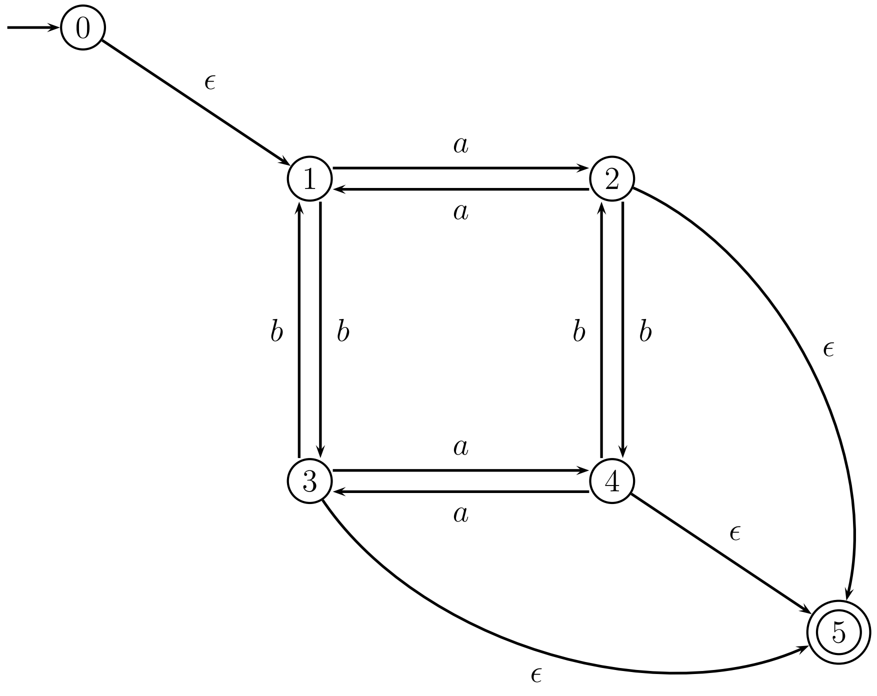


Figure 5.13: NFA for L (after preprocessing phase).

After eliminating node 2:

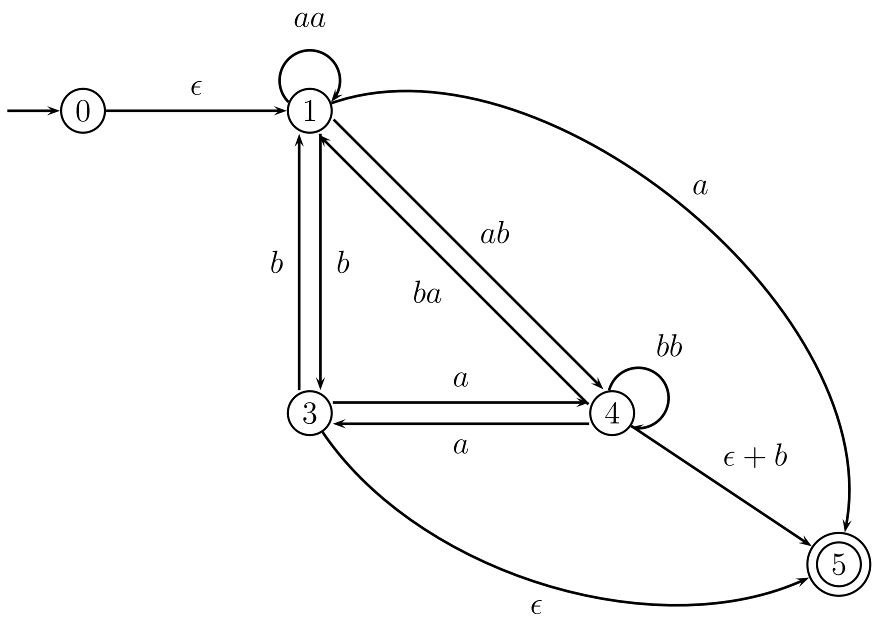


Figure 5.14: NFA for L (after eliminating node 2).

After eliminating node 3:

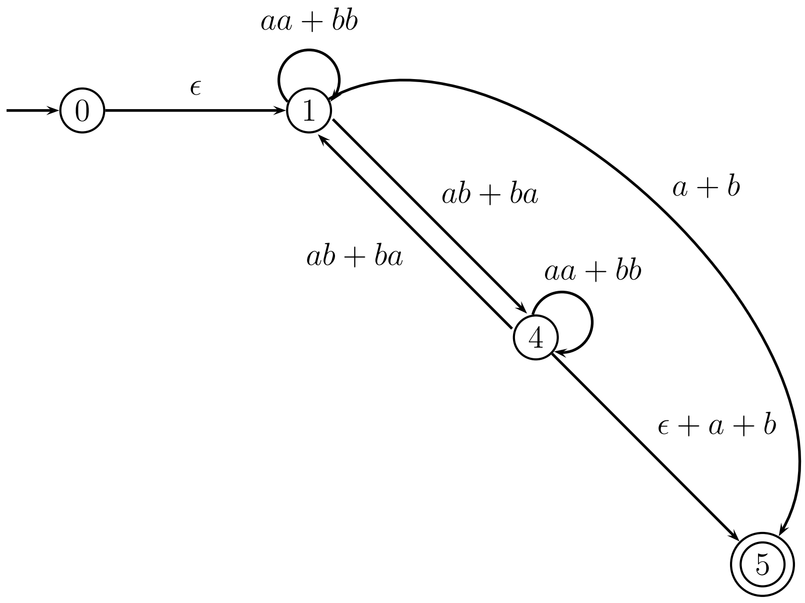


Figure 5.15: NFA for L (after eliminating node 3).

After eliminating node 4:

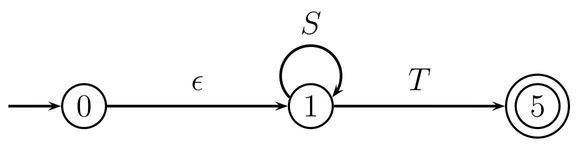


Figure 5.16: NFA for L (after eliminating node 4).

where

$$T = a + b + (ab + ba)(aa + bb)^*(\epsilon + a + b)$$

and

$$S = aa + bb + (ab + ba)(aa + bb)^*(ab + ba).$$

Finally, after eliminating node 1, we get:

$$R = (aa + bb + (ab + ba)(aa + bb)^*(ab + ba))^*(a + b + (ab + ba)(aa + bb)^*(\epsilon + a + b)).$$

5.7 Applications of Regular Expressions: Lexical analysis, Finding patterns in text

Regular expressions have several practical applications. The first important application is to *lexical analysis*.

A *lexical analyzer* is the first component of a *compiler*. The purpose of a lexical analyzer is to scan the source program and break it into atomic components, known as *tokens*, i.e., substrings of consecutive characters that belong together logically.

Examples of tokens are: identifiers, keywords, numbers (in fixed point notation or floating point notation, etc.), arithmetic operators ($+, \cdot, -,$ ^), comparison operators ($<, >, =, <>$), assignment operator (:=), etc.

Tokens can be described by regular expressions. For this purpose, it is useful to enrich the syntax of regular expressions, as in UNIX.

For example, the 26 upper case letters of the (roman) alphabet, $A, \ldots, Z$, can be specified by the expression

[A-Z]

Similarly, the ten digits, $0, 1, \ldots, 9$, can be specified by the expression

[0-9]

The regular expression

$$R_1 + R_2 + \cdots + R_k$$

is denoted

$$[R_1 R_2 \cdots R_k]$$

So, the expression

$$[\text{A-Za-z0-9}]$$

denotes any letter (upper case or lower case) or digit. This is called an *alphanumeric*.

If we define an identifier as a string beginning with a letter (upper case or lower case) followed by any number of alphanumerics (including none), then we can use the following expression to specify identifiers:

$$[\text{A-Za-z}][\text{A-Za-z0-9}]*$$

There are systems, such as `lex` or `flex` that accept as input a list of regular expressions describing the tokens of a programming language and construct a lexical analyzer for these tokens. Such systems are called *lexical analyzer generators*. Basically, they build a DFA from the set of regular expressions using the algorithms that have been described earlier.

Usually, it is possible associate with every expression some action to be taken when the corresponding token is recognized

Another application of regular expressions is finding patterns in text. Using a regular expression, we can specify a "vaguely defined" class of patterns.

Take the example of a street address. Most street addresses end with "Street", or "Avenue", or "Road" or "St.", or "Ave.", or "Rd.".

We can design a regular expression that captures the shape of most street addresses and then convert it to a DFA that can be used to search for street addresses in text.

For more on this, see Hopcroft-Motwani and Ullman.

5.8 Summary of Closure Properties of the Regular Languages

The family of regular languages is closed under many operations. In particular, it is closed under the following operations listed below. Some of the closure properties are left as a homework problem.

(1) Union, intersection, relative complement.

(2) Concatenation, Kleene $*$, Kleene $+$.

(3) Homomorphisms and inverse homomorphisms.

(4) gsm and inverse gsm mappings, a-transductions and inverse a-transductions.

Another useful operation is substitution. Given any two alphabets Σ, Δ, a *substitution* is a function, $\tau\colon \Sigma \to 2^{\Delta^*}$, assigning some language, $\tau(a) \subseteq \Delta^*$, to every symbol $a \in \Sigma$.

Definition 5.9. A substitution $\tau\colon \Sigma \to 2^{\Delta^*}$ is extended to a map $\tau\colon 2^{\Sigma^*} \to 2^{\Delta^*}$ by first extending τ to strings using the following definition

$$\begin{aligned} \tau(\epsilon) &= \{\epsilon\}, \\ \tau(ua) &= \tau(u)\tau(a), \end{aligned}$$

where $u \in \Sigma^*$ and $a \in \Sigma$, and then to languages by letting

$$\tau(L) = \bigcup_{w \in L} \tau(w),$$

for every language $L \subseteq \Sigma^*$.

Observe that a homomorphism is a special kind of substitution.

A substitution is a *regular* substitution iff $\tau(a)$ is a regular language for every $a \in \Sigma$. The proof of the next proposition is left as a homework problem.

Proposition 5.7. *If L is a regular language and τ is a regular substitution, then $\tau(L)$ is also regular. Thus, the family of regular languages is closed under regular substitutions.*

Chapter 6

Regular Language and Right-Invariant Equivalence Relations

6.1 Right-Invariant Equivalence Relations on Σ^*

The purpose of this chapter is to give one more characterization of the regular languages in terms of certain kinds of equivalence relations on strings. Pushing this characterization a bit further, we will be able to show how minimal DFA's can be found.

Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA. The DFA D may be redundant, for example, if there are states that are not accessible from the start state. Recall (see Section 3.1, especially Definition 3.4) that the set Q_r of *accessible or reachable states* is the subset of Q defined as

$$Q_r = \{p \in Q \mid \exists w \in \Sigma^*,\ \delta^*(q_0, w) = p\}.$$

If $Q \neq Q_r$, we can "clean up" D by deleting the states in $Q - Q_r$ and restricting the transition function δ to Q_r. This way, we get an equivalent DFA D_r such that $L(D) = L(D_r)$, where all the states of D_r are reachable. From now on, we assume that we are dealing with DFA's such that $D = D_r$, called *trim, or reachable*.

Recall that an *equivalence relation* $\simeq$ on a set A is a relation which is *reflexive*, *symmetric*, and *transitive*. Given any $a \in A$, the set

$$\{b \in A \mid a \simeq b\}$$

is called the *equivalence class of* a, and it is denoted as $[a]_\simeq$, or even as $[a]$. Recall that for any two elements $a, b \in A$, $[a] \cap [b] = \emptyset$ iff $a \not\simeq b$, and $[a] = [b]$ iff $a \simeq b$. The set of equivalence classes associated with the equivalence relation $\simeq$ is a *partition* Π of A (also denoted as $A/\simeq$). This means that it is a family of nonempty pairwise disjoint sets whose union is equal to A itself. The equivalence classes are also called the *blocks* of the partition Π. The number of blocks in the partition Π is called the *index* of $\simeq$ (and Π).

Given any two equivalence relations $\simeq_1$ and $\simeq_2$ with associated partitions Π_1 and Π_2,

$$\simeq_1 \subseteq \simeq_2$$

iff every block of the partition Π_1 is contained in some block of the partition Π_2. Then, every block of the partition Π_2 is the union of blocks of the partition Π_1, and we say that $\simeq_1$ is a *refinement* of $\simeq_2$ (and similarly, Π_1 is a refinement of Π_2). Note that Π_2 has at most as many blocks as Π_1 does.

We now define an equivalence relation on strings induced by a DFA. This equivalence is a kind of "observational" equivalence, in the sense that we decide that two strings u, v are equivalent iff, when feeding first u and then v to the DFA, u and v drive the DFA to the same state. From the point of view of the observer, u and v have the same effect (reaching the same state).

Definition 6.1. Given a DFA $D = (Q, \Sigma, \delta, q_0, F)$, we define the relation $\simeq_D$ on Σ^* as follows: for any two strings $u, v \in \Sigma^*$,

$$u \simeq_D v \quad \text{iff} \quad \delta^*(q_0, u) = \delta^*(q_0, v).$$

Example 6.1. We can figure out what the equivalence classes of $\simeq_D$ are for the following DFA:

	a	b
0	1	0
1	2	1
2	0	2

with 0 both start state and (unique) final state. For example

$$\begin{aligned} abbabbb &\simeq_D aa \\ ababab &\simeq_D \epsilon \\ bba &\simeq_D a. \end{aligned}$$

There are three equivalences classes:

$$[\epsilon]_{\simeq}, \quad [a]_{\simeq}, \quad [aa]_{\simeq}.$$

Observe that $L(D) = [\epsilon]_{\simeq}$. Also, the equivalence classes are in one–to–one correspondence with the states of D.

The relation $\simeq_D$ turns out to have some interesting properties. In particular, it is *right-invariant*, which means that for all $u, v, w \in \Sigma^*$, if $u \simeq v$, then $uw \simeq vw$.

Proposition 6.1. *Given any (trim) DFA $D = (Q, \Sigma, \delta, q_0, F)$, the relation $\simeq_D$ is an equivalence relation which is right-invariant and has finite index. Furthermore, if Q has n states, then the index of $\simeq_D$ is n, and every equivalence class of $\simeq_D$ is a regular language. Finally, $L(D)$ is the union of some of the equivalence classes of $\simeq_D$.*

Proof. The fact that $\simeq_D$ is an equivalence relation is a trivial verification. To prove that $\simeq_D$ is right-invariant, we first prove by induction on the length of v that for all $u, v \in \Sigma^*$, for all $p \in Q$,

$$\delta^*(p, uv) = \delta^*(\delta^*(p, u),\, v).$$

Then, if $u \simeq_D v$, which means that $\delta^*(q_0, u) = \delta^*(q_0, v)$, we have

$$\delta^*(q_0, uw) = \delta^*(\delta^*(q_0, u),\, w) = \delta^*(\delta^*(q_0, v),\, w) = \delta^*(q_0, vw),$$

which means that $uw \simeq_D vw$. Thus, $\simeq_D$ is right-invariant. We still have to prove that $\simeq_D$ has index n. Define the function $f\colon \Sigma^* \to Q$ such that

$$f(u) = \delta^*(q_0, u).$$

Note that if $u \simeq_D v$, which means that $\delta^*(q_0, u) = \delta^*(q_0, v)$, then $f(u) = f(v)$. Thus, the function $f\colon \Sigma^* \to Q$ has the *same value* on all the strings in some equivalence class $[u]$, so it induces a function $\widehat{f}\colon \Pi \to Q$ defined such that

$$\widehat{f}([u]) = f(u)$$

for every equivalence class $[u] \in \Pi$, where $\Pi = \Sigma^*/\simeq$ is the partition associated with $\simeq_D$. This function is well defined since $f(v)$ has the same value for all elements v in the equivalence class $[u]$.

However, the function $\widehat{f}\colon \Pi \to Q$ is injective (one-to-one), since $\widehat{f}([u]) = \widehat{f}([v])$ is equivalent to $f(u) = f(v)$ (since by definition of $\widehat{f}$ we have $\widehat{f}([u]) = f(u)$ and $\widehat{f}([v]) = f(v)$), which by definition of f means that $\delta^*(q_0, u) = \delta^*(q_0, v)$, which means precisely that $u \simeq_D v$, that is, $[u] = [v]$.

Since Q has n states, Π has at most n blocks. Moreover, since every state is accessible, for every $q \in Q$, there is some $w \in \Sigma^*$ so that $\delta^*(q_0, w) = q$, which shows that $\widehat{f}([w]) = f(w) = q$. Consequently, $\widehat{f}$ is also surjective. But then, being injective and surjective, $\widehat{f}$ is bijective and Π has exactly n blocks.

Every equivalence class of Π is a set of strings of the form

$$\{w \in \Sigma^* \mid \delta^*(q_0, w) = p\},$$

for some $p \in Q$, which is accepted by the DFA

$$D_p = (Q, \Sigma, \delta, q_0, \{p\})$$

obtained from D by changing F to $\{p\}$. Thus, every equivalence class is a regular language. Finally, since

$$\begin{aligned} L(D) &= \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\} \\ &= \bigcup_{f \in F} \{w \in \Sigma^* \mid \delta^*(q_0, w) = f\} \\ &= \bigcup_{f \in F} L(D_f), \end{aligned}$$

we see that $L(D)$ is the union of the equivalence classes corresponding to the final states in F. $\square$

One should not be too optimistic and hope that every equivalence relation on strings is right-invariant.

Example 6.2. For example, if $\Sigma = \{a\}$, the equivalence relation $\simeq$ given by the partition

$$\big\{\epsilon, a, a^4, a^9, a^{16}, \ldots, a^{n^2}, \ldots \mid n \geq 0\big\} \cup \big\{a^2, a^3, a^5, a^6, a^7, a^8, \ldots, a^m, \ldots \mid m \text{ is not a square}\big\}$$

we have $a \simeq a^4$, yet by concatenating on the right with a^5, since $aa^5 = a^6$ and $a^4a^5 = a^9$ we get

$$a^6 \not\simeq a^9,$$

that is, a^6 and a^9 are *not* equivalent. It turns out that the problem is that neither equivalence class is a regular language.

It is worth noting that a right-invariant equivalence relation is not necessarily *left-invariant*, which means that if $u \simeq v$ then $wu \simeq wv$.

Example 6.3. For example, if $\simeq$ is given by the four equivalence classes

$$C_1 = \{bb\}^*, \;\; C_2 = \{bb\}^*a, \;\; C_3 = b\{bb\}^*, \;\; C_4 = \{bb\}^*a\{a,b\}^+ \cup b\{bb\}^*a\{a,b\}^*,$$

then we can check that $\simeq$ is right-invariant by figuring out the inclusions $C_ia \subseteq C_j$ and $C_ib \subseteq C_j$, which are recorded in the following table:

	a	b
C_1	C_2	C_3
C_2	C_4	C_4
C_3	C_4	C_1
C_4	C_4	C_4

However, both $ab, ba \in C_4$, yet $bab \in C_4$ and $bba \in C_2$, so $\simeq$ is not left-invariant.

The remarkable fact due to Myhill and Nerode is that Proposition 6.1 has a converse. Indeed, given a right-invariant equivalence relation of finite index it is possible to reconstruct a DFA, and by a suitable choice of final state, every equivalence class is accepted by such a DFA. Let us show how this DFA is constructed using a simple example.

Example 6.4. Consider the equivalence relation $\simeq$ on $\{a, b\}^*$ given by the three equivalence classes

$$C_1 = \{\epsilon\}, \;\; C_2 = a\{a, b\}^*, \;\; C_3 = b\{a, b\}^*.$$

We leave it as an easy exercise to check that $\simeq$ is right-invariant. For example, if $u \simeq v$ and $u, v \in C_2$, then $u = ax$ and $v = ay$ for some $x, y \in \{a, b\}^*$, so for any $w \in \{a, b\}^*$ we have $uw = axw$ and $vw = ayw$, which means that we also have $uw, vw \in C_2$, thus $uw \simeq vw$.

For any subset $C \subseteq \{a, b\}^*$ and any string $w \in \{a, b\}^*$ define Cw as the set of strings

$$Cw = \{uw \mid u \in C\}.$$

There are two reasons why a DFA can be recovered from the right-invariant equivalence relation $\simeq$:

(1) For every equivalence class C_i and every string w, there is a unique equivalence class C_j such that
$$C_i w \subseteq C_j.$$
Actually, it is enough to check the above property for strings w of length 1 (*i.e.* symbols in the alphabet) because the property for arbitrary strings follows by induction.

(2) For every $w \in \Sigma^*$ and every class C_i,
$$C_1 w \subseteq C_i \quad \text{iff} \quad w \in C_i,$$
where C_1 is the equivalence class of the empty string.

We can make a table recording these inclusions.

Example 6.5. Continuing Example 6.4, we get:

	a	b
C_1	C_2	C_3
C_2	C_2	C_2
C_3	C_3	C_3

For example, from $C_1 = \{\epsilon\}$ we have $C_1 a = \{a\} \subseteq C_2$ and $C_1 b = \{b\} \subseteq C_3$, for $C_2 = a\{a, b\}^*$, we have $C_2 a = a\{a, b\}^* a \subseteq C_2$ and $C_2 a = a\{a, b\}^* b \subseteq C_2$, and for $C_3 = b\{a, b\}^*$, we have $C_3 a = b\{a, b\}^* a \subseteq C_3$ and $C_3 b = b\{a, b\}^* b \subseteq C_3$.

The key point is that the above table is the transition table of a DFA with start state $C_1 = [\epsilon]$. Furthermore, if C_i $(i = 1, 2, 3)$ is chosen as a single final state, the corresponding DFA D_i accepts C_i. This is the converse of Myhill-Nerode!

Observe that the inclusions $C_i w \subseteq C_j$ may be strict inclusions. For example, $C_1 a = \{a\}$ is a proper subset of $C_2 = a\{a, b\}^*$

Let us do another example.

Example 6.6. Consider the equivalence relation $\simeq$ given by the four equivalence classes

$$C_1 = \{\epsilon\}, \;\; C_2 = \{a\}, \;\; C_3 = \{b\}^+, \;\; C_4 = a\{a, b\}^+ \cup \{b\}^+ a\{a, b\}^*.$$

We leave it as an easy exercise to check that $\simeq$ is right-invariant.

We obtain the following table of inclusions $C_ia \subseteq C_j$ and $C_ib \subseteq C_j$:

	a	b
C_1	C_2	C_3
C_2	C_4	C_4
C_3	C_4	C_3
C_4	C_4	C_4

For example, from $C_3 = \{b\}^+$ we get $C_3a = \{b\}^+a \subseteq C_4$, and $C_3b = \{b\}^+b \subseteq C_3$.

The above table is the transition function of a DFA with four states and start state C_1. If C_i ($i = 1, 2, 3, 4$) is chosen as a single final state, the corresponding DFA D_i accepts C_i.

Here is the general result.

Proposition 6.2. *Given any equivalence relation $\simeq$ on Σ^*, if $\simeq$ is right-invariant and has finite index n, then every equivalence class (block) in the partition Π associated with $\simeq$ is a regular language.*

Proof. Let $C_1, \ldots, C_n$ be the blocks of Π, and assume that $C_1 = [\epsilon]$ is the equivalence class of the empty string.

First, we claim that for every block C_i and every $w \in \Sigma^*$, there is a unique block C_j such that $C_iw \subseteq C_j$, where $C_iw = \{uw \mid u \in C_i\}$.

For every $u \in C_i$, the string uw belongs to one and only one of the blocks of Π, say C_j. For any other string $v \in C_i$, since (by definition) $u \simeq v$, by right invariance, we get $uw \simeq vw$, but since $uw \in C_j$ and C_j is an equivalence class, we also have $vw \in C_j$. This proves the first claim.

We also claim that for every $w \in \Sigma^*$, for every block C_i,

$$C_1w \subseteq C_i \quad \text{iff} \quad w \in C_i.$$

If $C_1w \subseteq C_i$, since $C_1 = [\epsilon]$, we have $\epsilon w = w \in C_i$. Conversely, if $w \in C_i$, for any $v \in C_1 = [\epsilon]$, since $\epsilon \simeq v$, by right invariance we have $w \simeq vw$, and thus $vw \in C_i$, which shows that $C_1w \subseteq C_i$.

For every class C_k, let
$$D_k = (\{1, \ldots, n\}, \Sigma, \delta, 1, \{k\}),$$
where $\delta(i, a) = j$ iff $C_ia \subseteq C_j$. We will prove the following equivalence:

$$\delta^*(i, w) = j \quad \text{iff} \quad C_iw \subseteq C_j.$$

For this, we prove the following two implications by induction on $|w|$:

(a) If $\delta^*(i, w) = j$, then $C_iw \subseteq C_j$, and

(b) If $C_i w \subseteq C_j$, then $\delta^*(i, w) = j$.

The base case ($w = \epsilon$) is trivial for both (a) and (b). We leave the proof of the induction step for (a) as an exercise and give the proof of the induction step for (b) because it is more subtle. Let $w = ua$, with $a \in \Sigma$ and $u \in \Sigma^*$. If $C_i ua \subseteq C_j$, then by the first claim, we know that there is a unique block, C_k, such that $C_i u \subseteq C_k$. Furthermore, there is a unique block, C_h, such that $C_k a \subseteq C_h$, but $C_i u \subseteq C_k$ implies $C_i ua \subseteq C_k a$ so we get $C_i ua \subseteq C_h$. However, by the uniqueness of the block, C_j, such that $C_i ua \subseteq C_j$, we must have $C_h = C_j$. By the induction hypothesis, as $C_i u \subseteq C_k$, we have

$$\delta^*(i, u) = k$$

and, by definition of δ, as $C_k a \subseteq C_j$ $(= C_h)$, we have $\delta(k, a) = j$, so we deduce that

$$\delta^*(i, ua) = \delta(\delta^*(i, u), a) = \delta(k, a) = j,$$

as desired. Then, using the equivalence just proved and the second claim, we have

$$\begin{aligned} L(D_k) &= \{w \in \Sigma^* \mid \delta^*(1, w) \in \{k\}\} \\ &= \{w \in \Sigma^* \mid \delta^*(1, w) = k\} \\ &= \{w \in \Sigma^* \mid C_1 w \subseteq C_k\} \\ &= \{w \in \Sigma^* \mid w \in C_k\} = C_k, \end{aligned}$$

proving that every block, C_k, is a regular language. $\square$

In general it is false that $C_i a = C_j$ for some block C_j, and we can only claim that $C_i a \subseteq C_j$.

We can combine Proposition 6.1 and Proposition 6.2 to get the following characterization of a regular language due to Myhill and Nerode:

Theorem 6.3. *(Myhill-Nerode) A language L (over an alphabet Σ) is a regular language iff it is the union of some of the equivalence classes of an equivalence relation $\simeq$ on Σ^*, which is right-invariant and has finite index.*

Theorem 6.3 can also be used to prove that certain languages are not regular. A general scheme (not the only one) goes as follows: If L is not regular, then it must be infinite. Now, we argue by contradiction. If L was regular, then by Myhill-Nerode, there would be some equivalence relation $\simeq$, which is right-invariant and of finite index, and such that L is the union of some of the classes of $\simeq$. Because Σ^* is infinite and $\simeq$ has only finitely many equivalence classes, there are strings $x, y \in \Sigma^*$ with $x \neq y$ so that

$$x \simeq y.$$

If we can find a third string, $z \in \Sigma^*$, such that

$$xz \in L \quad \text{and} \quad yz \notin L,$$

then we reach a contradiction. Indeed, by right invariance, from $x \simeq y$, we get $xz \simeq yz$. But, L is the union of equivalence classes of $\simeq$, so if $xz \in L$, then we should also have $yz \in L$, contradicting $yz \notin L$. Therefore, L is not regular.

Then the scenario is this: to prove that L is not regular, first we check that L is infinite. If so, we try finding three strings x, y, z, where and x and $y \neq x$ are prefixes of strings in L such that

$$x \simeq y,$$

where $\simeq$ is a right-invariant relation of finite index such that L is the union of equivalence of L (which must exist by Myhill–Nerode since we are assuming by contradiction that L is regular), and where z is chosen so that

$$xz \in L \quad \text{and} \quad yz \notin L.$$

Example 6.7. For example, we prove that $L = \{a^n b^n \mid n \geq 1\}$ is not regular.

Assuming for the sake of contradiction that L is regular, there is some equivalence relation $\simeq$ which is right-invariant and of finite index and such that L is the union of some of the classes of $\simeq$. Since the sequence

$$a, aa, aaa, \ldots, a^i, \ldots$$

is infinite and $\simeq$ has a finite number of classes, two of these strings must belong to the same class, which means that $a^i \simeq a^j$ for some $i \neq j$. But since $\simeq$ is right invariant, by concatenating with b^i on the right, we see that $a^i b^i \simeq a^j b^i$ for some $i \neq j$. However $a^i b^i \in L$, and since L is the union of classes of $\simeq$, we also have $a^j b^i \in L$ for $i \neq j$, which is absurd, given the definition of L. Thus, in fact, L is not regular.

Here is another illustration of the use of the Myhill-Nerode Theorem to prove that a language is not regular.

Example 6.8. We claim that the language,

$$L' = \{a^{n!} \mid n \geq 1\},$$

is not regular, where $n!$ (n factorial) is given by $0! = 1$ and $(n+1)! = (n+1)n!$.

Assume L' is regular. Then, there is some equivalence relation $\simeq$ which is right-invariant and of finite index and such that L' is the union of some of the classes of $\simeq$. Since the sequence

$$a, a^2, \ldots, a^n, \ldots$$

is infinite, two of these strings must belong to the same class, which means that $a^p \simeq a^q$ for some p, q with $1 \leq p < q$. As $q! \geq q$ for all $q \geq 0$ and $q > p$, we can concatenate on the right with $a^{q!-p}$ and we get

$$a^p a^{q!-p} \simeq a^q a^{q!-p},$$

that is,

$$a^{q!} \simeq a^{q!+q-p}.$$

Since $p < q$ we have $q! < q! + q - p$. If we can show that

$$q! + q - p < (q+1)!$$

we will obtain a contradiction because then $a^{q!+q-p} \notin L'$, yet $a^{q!+q-p} \simeq a^{q!}$ and $a^{q!} \in L'$, contradicting Myhill-Nerode. Now, as $1 \leq p < q$, we have $q - p \leq q - 1$, so if we can prove that

$$q! + q - p \leq q! + q - 1 < (q+1)!$$

we will be done. However, $q! + q - 1 < (q+1)!$ is equivalent to

$$q - 1 < (q+1)! - q!,$$

and since $(q+1)! - q! = (q+1)q! - q! = qq!$, we simply need to prove that

$$q - 1 < q \leq qq!,$$

which holds for $q \geq 1$.

There is another version of the Myhill-Nerode Theorem involving congruences which is also quite useful. An equivalence relation, $\simeq$, on Σ^* is *left and right-invariant* iff for all $x, y, u, v \in \Sigma^*$,

$$\text{if} \quad x \simeq y \quad \text{then} \quad uxv \simeq uyv.$$

An equivalence relation, $\simeq$, on Σ^* is a *congruence* iff for all $u_1, u_2, v_1, v_2 \in \Sigma^*$,

$$\text{if} \quad u_1 \simeq v_1 \quad \text{and} \quad u_2 \simeq v_2 \quad \text{then} \quad u_1u_2 \simeq v_1v_2.$$

It is easy to prove that an equivalence relation is a congruence iff it is left and right-invariant.

For example, assume that $\simeq$ is a left and right-invariant equivalence relation, and assume that

$$u_1 \simeq v_1 \quad \text{and} \quad u_2 \simeq v_2.$$

By right-invariance applied to $u_1 \simeq v_1$, we get

$$u_1u_2 \simeq v_1u_2$$

and by left-invariance applied to $u_2 \simeq v_2$ we get

$$v_1u_2 \simeq v_1v_2.$$

By transitivity, we conclude that

$$u_1u_2 \simeq v_1v_2.$$

which shows that $\simeq$ is a congruence.

Proving that a congruence is left and right-invariant is even easier.

There is a version of Proposition 6.1 that applies to congruences and for this we define the relation $\sim_D$ as follows: For any (trim) DFA, $D = (Q, \Sigma, \delta, q_0, F)$, for all $x, y \in \Sigma^*$,

$$x \sim_D y \quad \text{iff} \quad (\forall q \in Q)(\delta^*(q, x) = \delta^*(q, y)).$$

Proposition 6.4. *Given any (trim) DFA, $D = (Q, \Sigma, \delta, q_0, F)$, the relation $\sim_D$ is an equivalence relation which is left and right-invariant and has finite index. Furthermore, if Q has n states, then the index of $\sim_D$ is at most n^n and every equivalence class of $\sim_D$ is a regular language. Finally, $L(D)$ is the union of some of the equivalence classes of $\sim_D$.*

Proof. We leave most of the proof of Proposition 6.4 as an exercise. The last two parts of the proposition are proved using the following facts:

(1) Since $\sim_D$ is left and right-invariant and has finite index, in particular, $\sim_D$ is right-invariant and has finite index, so by Proposition 6.2 every equivalence class of $\sim_D$ is regular.

(2) Observe that

$$\sim_D \subseteq \simeq_D,$$

since the condition $\delta^*(q, x) = \delta^*(q, y)$ holds for every $q \in Q$, so in particular for $q = q_0$. But then, every equivalence class of $\simeq_D$ is the union of equivalence classes of $\sim_D$ and since, by Proposition 6.1, L is the union of equivalence classes of $\simeq_D$, we conclude that L is also the union of equivalence classes of $\sim_D$.

This completes the proof. □

Using Proposition 6.4 and Proposition 6.2, we obtain another version of the Myhill-Nerode Theorem.

Theorem 6.5. *(Myhill-Nerode, Congruence Version) A language L (over an alphabet Σ) is a regular language iff it is the union of some of the equivalence classes of an equivalence relation $\simeq$ on Σ^*, which is a congruence and has finite index.*

We now consider an equivalence relation associated with a language L.

6.2 Finding minimal DFA's

Given any language L (not necessarily regular), we can define an equivalence relation ρ_L on Σ^* which is right-invariant, but not necessarily of finite index. The equivalence relation ρ_L is such that L is the union of equivalence classes of ρ_L. Furthermore, when L is regular, the relation ρ_L has finite index. In fact, this index is the size of a smallest DFA accepting L. As a consequence, if L is regular, a simple modification of the proof of Proposition 6.2 applied to $\simeq\; = \;\rho_L$ yields a minimal DFA D_{ρ_L} accepting L.

Then, given any trim DFA D accepting L, the equivalence relation ρ_L can be translated to an equivalence relation $\equiv$ on states, in such a way that for all $u, v \in \Sigma^*$,

$$u \rho_L v \quad \text{iff} \quad \varphi(u) \equiv \varphi(v),$$

where $\varphi\colon \Sigma^* \to Q$ is the function (run the DFA D on u from q_0) given by

$$\varphi(u) = \delta^*(q_0, u).$$

One can then construct a quotient DFA $D/\equiv$ whose states are obtained by merging all states in a given equivalence class of states into a single state, and the resulting DFA $D/\equiv$ is a mininal DFA. Even though $D/\equiv$ appears to depend on D, it is in fact unique, and isomorphic to the abstract DFA D_{ρ_L} induced by ρ_L.

The last step in obtaining the minimal DFA $D/\equiv$ is to give a constructive method to compute the state equivalence relation $\equiv$. This can be done by constructing a sequence of approximations $\equiv_i$, where each $\equiv_{i+1}$ refines $\equiv_i$. It turns out that if D has n states, then there is some index $i_0 \leq n-2$ such that

$$\equiv_j = \equiv_{i_0} \quad \text{for all } j \geq i_0 + 1,$$

and that

$$\equiv = \equiv_{i_0}.$$

Furthermore, $\equiv_{i+1}$ can be computed inductively from $\equiv_i$. In summary, we obtain a iterative algorithm for computing $\equiv$ that terminates in at most $n-2$ steps.

Definition 6.2. Given any language L (over Σ), we define the *right-invariant equivalence* ρ_L *associated with* L as the relation on Σ^* defined as follows: for any two strings $u, v \in \Sigma^*$,

$$u \rho_L v \quad \text{iff} \quad \forall w \in \Sigma^* (uw \in L \quad \text{iff} \quad vw \in L).$$

It is clear that the relation ρ_L is an equivalence relation, and it is right-invariant. To show right-invariance, argue as follows: if $u\rho_L v$, then for any $w \in \Sigma^*$, since $u\rho_L v$ means that

$$uz \in L \quad \text{iff} \quad vz \in L$$

for all $z \in \Sigma^*$, in particular the above equivalence holds for all z of the form $z = wy$ for any arbitary $y \in \Sigma^*$, so we have

$$uwy \in L \quad \text{iff} \quad vwy \in L$$

for all $y \in \Sigma^*$, which means that $uw\rho_L vw$.

It is also clear that L is the union of the equivalence classes of strings in L. This is because if $u \in L$ and $u\rho_L v$, by letting $w = \epsilon$ in the definition of ρ_L, we get

$$u \in L \quad \text{iff} \quad v \in L,$$

and since $u \in L$, we also have $v \in L$. This implies that if $u \in L$ then $[u]_{\rho_L} \subseteq L$ and so,

$$L = \bigcup_{u \in L} [u]_{\rho_L}.$$

Example 6.9. For example, consider the regular language

$$L = \{a\} \cup \{b^m \mid m \geq 1\}.$$

We leave it as an exercise to show that the equivalence relation ρ_L consists of the four equivalence classes

$$C_1 = \{\epsilon\},\ C_2 = \{a\},\ C_3 = \{b\}^+,\ C_4 = a\{a,b\}^+ \cup \{b\}^+ a\{a,b\}^*$$

encountered earlier in Example 6.6. Observe that

$$L = C_2 \cup C_3.$$

When L is regular, we have the following remarkable result:

Proposition 6.6. *Given any regular language L, for any (trim) DFA $D = (Q, \Sigma, \delta, q_0, F)$ such that $L = L(D)$, ρ_L is a right-invariant equivalence relation, and we have $\simeq_D \subseteq \rho_L$. Furthermore, if ρ_L has m classes and Q has n states, then $m \leq n$.*

Proof. By definition, $u \simeq_D v$ iff $\delta^*(q_0, u) = \delta^*(q_0, v)$. Since $w \in L(D)$ iff $\delta^*(q_0, w) \in F$, the fact that $u\rho_L v$ can be expressed as

$$\begin{aligned}
&\forall w \in \Sigma^*(uw \in L \quad \text{iff} \quad vw \in L)\\
&\text{iff}\\
&\forall w \in \Sigma^*(\delta^*(q_0, uw) \in F \quad \text{iff} \quad \delta^*(q_0, vw) \in F)\\
&\text{iff}\\
&\forall w \in \Sigma^*(\delta^*(\delta^*(q_0, u), w) \in F \quad \text{iff} \quad \delta^*(\delta^*(q_0, v), w) \in F),
\end{aligned}$$

and if $\delta^*(q_0, u) = \delta^*(q_0, v)$, this shows that $u\rho_L v$. Since the number of classes of $\simeq_D$ is n and $\simeq_D \subseteq \rho_L$, the equivalence relation ρ_L has fewer classes than $\simeq_D$, and $m \leq n$. $\square$

Proposition 6.6 shows that when L is regular, the index m of ρ_L is finite, and it is a lower bound on the size of all DFA's accepting L. It remains to show that a DFA with m states accepting L exists.

However, going back to the proof of Proposition 6.2 starting with the right-invariant equivalence relation ρ_L of finite index m, if L is the union of the classes $C_{i_1}, \ldots, C_{i_k}$, the DFA

$$D_{\rho_L} = (\{1, \ldots, m\}, \Sigma, \delta, 1, \{i_1, \ldots, i_k\}),$$

where $\delta(i, a) = j$ iff $C_i a \subseteq C_j$, is such that $L = L(D_{\rho_L})$.

In summary, if L is regular, then the index of ρ_L is equal to the number of states of a minimal DFA for L, and D_{ρ_L} is a minimal DFA accepting L.

Example 6.10. For example, if

$$L = \{a\} \cup \{b^m \mid m \geq 1\}.$$

then we saw in Example 6.9 that ρ_L consists of the four equivalence classes

$$C_1 = \{\epsilon\},\ C_2 = \{a\},\ C_3 = \{b\}^+,\ C_4 = a\{a, b\}^+ \cup \{b\}^+ a\{a, b\}^*,$$

and we showed in Example 6.6 that the transition table of D_{ρ_L} is given by

	a	b
C_1	C_2	C_3
C_2	C_4	C_4
C_3	C_4	C_3
C_4	C_4	C_4

By picking the final states to be C_2 and C_3, we obtain the minimal DFA D_{ρ_L} accepting $L = \{a\} \cup \{b^m \mid m \geq 1\}$.

In the next section, we give an algorithm which allows us to find D_{ρ_L}, given any DFA D accepting L. This algorithms finds which states of D are equivalent.

6.3 State Equivalence and Minimal DFA's

The proof of Proposition 6.6 suggests the following definition of an equivalence between states:

Definition 6.3. Given any DFA $D = (Q, \Sigma, \delta, q_0, F)$, the relation $\equiv$ on Q, called *state equivalence*, is defined as follows: for all $p, q \in Q$,

$$p \equiv q \quad \text{iff} \quad \forall w \in \Sigma^*(\delta^*(p, w) \in F \quad \text{iff} \quad \delta^*(q, w) \in F). \qquad (*)$$

When $p \equiv q$, we say that *p and q are indistinguishable*.

It is trivial to verify that $\equiv$ is an equivalence relation, and that it satisfies the following property:

$$\text{if } p \equiv q \text{ then } \delta(p,a) \equiv \delta(q,a), \quad \text{for all } a \in \Sigma.$$

To prove the above, since the condition defining $\equiv$ must hold for all strings $w \in \Sigma^*$, in particular it must hold for all strings of the form $w = au$ with $a \in \Sigma$ and $u \in \Sigma^*$, so if $p \equiv q$ then we have

$$\begin{aligned}
&(\forall a \in \Sigma)(\forall u \in \Sigma^*)(\delta^*(p,au) \in F \quad \text{iff} \quad \delta^*(q,au) \in F)\\
\text{iff}\quad &(\forall a \in \Sigma)(\forall u \in \Sigma^*)(\delta^*(\delta^*(p,a),u) \in F \quad \text{iff} \quad \delta^*(\delta^*(q,a),u) \in F)\\
\text{iff}\quad &(\forall a \in \Sigma)(\forall u \in \Sigma^*)(\delta^*(\delta(p,a),u) \in F \quad \text{iff} \quad \delta^*(\delta(q,a),u) \in F)\\
\text{iff}\quad &(\forall a \in \Sigma)(\delta(p,a) \equiv \delta(q,a)).
\end{aligned}$$

$$\delta^*(p,\epsilon) \in F \quad \text{iff} \quad \delta^*(q,\epsilon) \in F,$$

which, since $\delta^*(p,\epsilon) = p$ and $\delta^*(q,\epsilon) = q$, is equivalent to

$$p \in F \quad \text{iff} \quad q \in F.$$

Therefore, if two states p, q are equivalent, then either both $p, q \in F$ or both $p, q \in \overline{F}$. This implies that a final state and a rejecting states are *never* equivalent.

Example 6.11. The reader should check that states A and C in the DFA below are equivalent and that no other distinct states are equivalent.

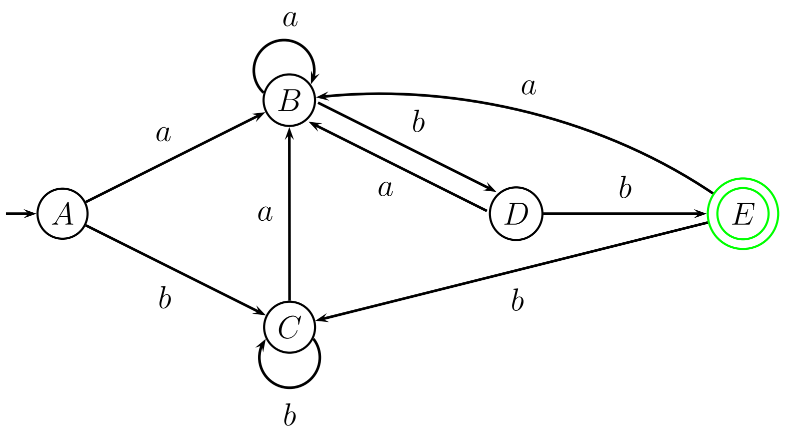


Figure 6.1: A non-minimal DFA for $\{a,b\}^*\{abb\}$.

It is illuminating to express state equivalence as the equality of two languages. Given the DFA $D = (Q, \Sigma, \delta, q_0, F)$, let $D_p = (Q, \Sigma, \delta, p, F)$ be the DFA obtained from D by redefining the start state to be p. Then, it is clear that

$$p \equiv q \quad \text{iff} \quad L(D_p) = L(D_q).$$

This simple observation implies that there is an algorithm to test state equivalence. Indeed, we simply have to test whether the DFA's D_p and D_q accept the same language and this can be done using the cross-product construction. Indeed, $L(D_p) = L(D_q)$ iff $L(D_p) - L(D_q) = \emptyset$ and $L(D_q) - L(D_p) = \emptyset$. Now, if $(D_p \times D_q)_{1-2}$ denotes the cross-product DFA with starting state (p, q) and with final states $F \times (Q - F)$ and $(D_p \times D_q)_{2-1}$ denotes the cross-product DFA also with starting state (p, q) and with final states $(Q - F) \times F$, we know that

$$L((D_p \times D_q)_{1-2}) = L(D_p) - L(D_q) \quad \text{and} \quad L((D_p \times D_q)_{2-1}) = L(D_q) - L(D_p),$$

so all we need to do if to test whether $(D_p \times D_q)_{1-2}$ and $(D_p \times D_q)_{2-1}$ accept the empty language. However, we know that this is the case iff the set of states reachable from (p, q) in $(D_p \times D_q)_{1-2}$ contains no state in $F \times (Q - F)$ and the set of states reachable from (p, q) in $(D_p \times D_q)_{2-1}$ contains no state in $(Q - F) \times F$.

Actually, the graphs of $(D_p \times D_q)_{1-2}$ and $(D_p \times D_q)_{2-1}$ are identical, so we only need to check that no state in $(F \times (Q - F)) \cup ((Q - F) \times F)$ is reachable from (p, q) in that graph. This algorithm to test state equivalence is not the most efficient but it is quite reasonable (it runs in polynomial time).

If $L = L(D)$, Theorem 6.7 below shows the relationship between ρ_L and $\equiv$ and, more generally, between the DFA, D_{ρ_L}, and the DFA, $D/\equiv$, obtained as the quotient of the DFA D modulo the equivalence relation $\equiv$ on Q.

The minimal DFA $D/\equiv$ is obtained by merging the states in each block C_i of the partition Π associated with $\equiv$, forming states corresponding to the blocks C_i, and drawing a transition on input a from a block C_i to a block C_j of Π iff there is a transition $q = \delta(p, a)$ from any state $p \in C_i$ to any state $q \in C_j$ on input a.

The start state is the block containing q_0, and the final states are the blocks consisting of final states.

Example 6.12. For example, consider the DFA D_1 accepting $L = \{ab, ba\}^*$ shown in Figure 6.2.

This is not a minimal DFA. In fact,

$$0 \equiv 2 \quad \text{and} \quad 3 \equiv 5.$$

Here is the minimal DFA for L:

The minimal DFA D_2 is obtained by merging the states in the equivalence class $\{0, 2\}$ into a single state, similarly merging the states in the equivalence class $\{3, 5\}$ into a single state, and drawing the transitions between equivalence classes. We obtain the DFA shown in Figure 6.3.

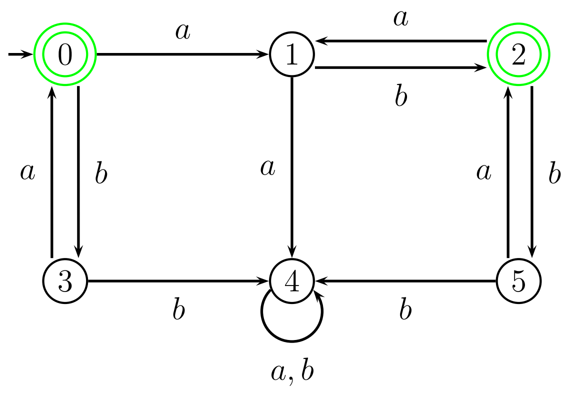


Figure 6.2: A nonminimal DFA D_1 for $L = \{ab, ba\}^*$.

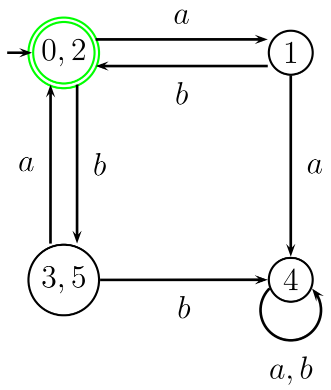


Figure 6.3: A minimal DFA D_2 for $L = \{ab, ba\}^*$.

Formally, the quotient DFA $D/\equiv$ is defined such that

$$D/\equiv = (Q/\equiv, \Sigma, \delta/\equiv, [q_0]_\equiv, F/\equiv),$$

where

$$\delta/\equiv([p]_\equiv, a) = [\delta(p, a)]_\equiv.$$

Theorem 6.7. *For any (trim) DFA $D = (Q, \Sigma, \delta, q_0, F)$ accepting the regular language $L = L(D)$, the function $\varphi\colon \Sigma^* \to Q$ defined such that*

$$\varphi(u) = \delta^*(q_0, u)$$

satisfies the property

$$u\rho_L v \quad \textit{iff} \quad \varphi(u) \equiv \varphi(v) \quad \textit{for all } u, v \in \Sigma^*,$$

and induces a bijection $\widehat{\varphi}\colon \Sigma^*/\rho_L \to Q/\equiv$, defined such that

$$\widehat{\varphi}([u]_{\rho_L}) = [\delta^*(q_0, u)]_\equiv.$$

Furthermore, we have

$$[u]_{\rho_L} a \subseteq [v]_{\rho_L} \quad \textit{iff} \quad \delta(\varphi(u), a) \equiv \varphi(v).$$

Consequently, $\widehat{\varphi}$ induces an isomorphism of DFA's, $\widehat{\varphi}\colon D_{\rho_L} \to D/\equiv$.

Proof. Since $\varphi(u) = \delta^*(q_0, u)$ and $\varphi(v) = \delta^*(q_0, v)$, the fact that $\varphi(u) \equiv \varphi(v)$ can be expressed as

$$\begin{aligned}&\forall w \in \Sigma^*(\delta^*(\delta^*(q_0, u), w) \in F \quad \text{iff} \quad \delta^*(\delta^*(q_0, v), w) \in F)\\ &\text{iff}\\ &\forall w \in \Sigma^*(\delta^*(q_0, uw) \in F \quad \text{iff} \quad \delta^*(q_0, vw) \in F),\end{aligned}$$

which is exactly $u \,\rho_L\, v$. Therefore,

$$u \,\rho_L\, v \quad \text{iff} \quad \varphi(u) \equiv \varphi(v).$$

From the above, we see that the equivalence class $[\varphi(u)]_\equiv$ of $\varphi(u)$ does not depend on the choice of the representative in the equivalence class $[u]_{\rho_L}$ of $u \in \Sigma^*$, since for any $v \in \Sigma^*$, if $u \,\rho_L\, v$ then $\varphi(u) \equiv \varphi(v)$, so $[\varphi(u)]_\equiv = [\varphi(v)]_\equiv$. Therefore, the function $\varphi\colon \Sigma^* \to Q$ maps each equivalence class $[u]_{\rho_L}$ modulo ρ_L to the equivalence class $[\varphi(u)]_\equiv$ modulo $\equiv$, and so the function $\widehat{\varphi}\colon \Sigma^*/\rho_L \to Q/\equiv$ given by

$$\widehat{\varphi}([u]_{\rho_L}) = [\varphi(u)]_\equiv = [\delta^*(q_0, u)]_\equiv$$

is well-defined. Moreover, $\widehat{\varphi}$ is injective, since $\widehat{\varphi}([u]) = \widehat{\varphi}([v])$ iff $\varphi(u) \equiv \varphi(v)$ iff (from above) $u\rho_v v$ iff $[u] = [v]$. Since every state in Q is accessible, for every $q \in Q$, there is some $u \in \Sigma^*$ so that $\varphi(u) = \delta^*(q_0, u) = q$, so $\widehat{\varphi}([u]) = [q]_\equiv$ and $\widehat{\varphi}$ is surjective. Therefore, we have a bijection $\widehat{\varphi}\colon \Sigma^*/\rho_L \to Q/\equiv$.

Since $\varphi(u) = \delta^*(q_0, u)$, we have

$$\delta(\varphi(u), a) = \delta(\delta^*(q_0, u), a) = \delta^*(q_0, ua) = \varphi(ua),$$

and thus, $\delta(\varphi(u), a) \equiv \varphi(v)$ can be expressed as $\varphi(ua) \equiv \varphi(v)$. By the previous part, this is equivalent to $ua\rho_L v$, and we claim that this is equivalent to

$$[u]_{\rho_L} a \subseteq [v]_{\rho_L}.$$

First, if $[u]_{\rho_L} a \subseteq [v]_{\rho_L}$, then $ua \in [v]_{\rho_L}$, that is, $ua\rho_L v$. Conversely, if $ua\rho_L v$, then for every $u' \in [u]_{\rho_L}$, we have $u'\rho_L u$, so by right-invariance we get $u'a\rho_L ua$, and since $ua\rho_L v$, we get $u'a\rho_L v$, that is, $u'a \in [v]_{\rho_L}$. Since $u' \in [u]_{\rho_L}$ is arbitrary, we conclude that $[u]_{\rho_L} a \subseteq [v]_{\rho_L}$. Therefore, we proved that

$$\delta(\varphi(u), a) \equiv \varphi(v) \quad \text{iff} \quad [u]_{\rho_L} a \subseteq [v]_{\rho_L}.$$

The above shows that the transitions of D_{ρ_L} correspond to the transitions of $D/\equiv$. ☐

Theorem 6.7 shows that the DFA D_{ρ_L} is isomorphic to the DFA $D/\equiv$ obtained as the quotient of the DFA D modulo the equivalence relation $\equiv$ on Q. Since D_{ρ_L} is a minimal DFA accepting L, so is $D/\equiv$.

Example 6.13. Consider the following DFA D,

	a	b
1	2	3
2	4	4
3	4	3
4	5	5
5	5	5

with start state 1 and final states 2 and 3. It is easy to see that

$$L(D) = \{a\} \cup \{b^m \mid m \geq 1\}.$$

It is not hard to check that states 4 and 5 are equivalent, and no other pairs of distinct states are equivalent. The quotient DFA $D/\equiv$ is obtained my merging states 4 and 5, and we obtain the following minimal DFA:

	a	b
1	2	3
2	4	4
3	4	3
4	4	4

with start state 1 and final states 2 and 3. This DFA is isomorphic to the DFA D_{ρ_L} of Example 6.10.

There are other characterizations of the regular languages. Among those, the characterization in terms of right derivatives is of particular interest because it yields an alternative construction of minimal DFA's.

Definition 6.4. Given any language, $L \subseteq \Sigma^*$, for any string, $u \in \Sigma^*$, the *right derivative of L by u*, denoted L/u, is the language

$$L/u = \{w \in \Sigma^* \mid uw \in L\}.$$

Theorem 6.8. *If $L \subseteq \Sigma^*$ is any language, then L is regular iff it has finitely many right derivatives. Furthermore, if L is regular, then all its right derivatives are regular and their number is equal to the number of states of the minimal DFA's for L.*

Proof. It is easy to check that

$$L/u = L/v \qquad \text{iff} \qquad u\rho_L v.$$

The above shows that ρ_L has a finite number of classes, say m, iff there is a finite number of right derivatives, say n, and if so, $m = n$. If L is regular, then we know that the number of equivalence classes of ρ_L is the number of states of the minimal DFA's for L, so the number of right derivatives of L is equal to the size of the minimal DFA's for L.

Conversely, if the number of derivatives is finite, say m, then ρ_L has m classes and by Myhill-Nerode, L is regular. It remains to show that if L is regular then every right derivative is regular.

Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA accepting L. If $p = \delta^*(q_0, u)$, then let

$$D_p = (Q, \Sigma, \delta, p, F),$$

that is, D with with p as start state. It is clear that

$$L/u = L(D_p),$$

so L/u is regular for every $u \in \Sigma^*$. Also observe that if $|Q| = n$, then there are at most n DFA's D_p, so there is at most n right derivatives, which is another proof of the fact that a regular language has a finite number of right derivatives. $\square$

If L is regular then the construction of a minimal DFA for L can be recast in terms of right derivatives. Let $L/u_1, L/u_2, \ldots, L/u_m$ be the set of all the right derivatives of L. Of course, we may assume that $u_1 = \epsilon$. We form a DFA whose states are the right derivatives, L/u_i. For every state, L/u_i, for every $a \in \Sigma$, there is a transition on input a from L/u_i to $L/u_j = L/(u_i a)$. The start state is $L = L/u_1$ and the final states are the right derivatives, L/u_i, for which $\epsilon \in L/u_i$.

We leave it as an exercise to check that the above DFA accepts L. One way to do this is to recall that $L/u = L/v$ iff $u \rho_L v$ and to observe that the above construction mimics the construction of D_{ρ_L} as in the Myhill-Nerode proposition (Proposition 6.2). This DFA is minimal since the number of right derivatives is equal to the size of the minimal DFA's for L.

We now return to state equivalence. Note that if $F = \emptyset$, then $\equiv$ has a single block (Q), and if $F = Q$, then $\equiv$ has a single block (F). In the first case, the minimal DFA is the one state DFA rejecting all strings. In the second case, the minimal DFA is the one state DFA accepting all strings. When $F \neq \emptyset$ and $F \neq Q$, there are at least two states in Q, and $\equiv$ also has at least two blocks, as we shall see shortly.

It remains to compute $\equiv$ explicitly. This is done using a sequence of approximations. In view of the previous discussion, we are assuming that $F \neq \emptyset$ and $F \neq Q$, which means that $n \geq 2$, where n is the number of states in Q.

Definition 6.5. Given any DFA $D = (Q, \Sigma, \delta, q_0, F)$, for every $i \geq 0$, the relation $\equiv_i$ on Q, called *i-state equivalence*, is defined as follows: for all $p, q \in Q$,

$$p \equiv_i q \quad \text{iff} \quad \forall w \in \Sigma^*,\ |w| \leq i\ (\delta^*(p, w) \in F \quad \text{iff} \quad \delta^*(q, w) \in F).$$

When $p \equiv_i q$, we say that *p and q are i-indistinguishable.*

Since state equivalence $\equiv$ is defined such that

$$p \equiv q \quad \text{iff} \quad \forall w \in \Sigma^* (\delta^*(p, w) \in F \quad \text{iff} \quad \delta^*(q, w) \in F),$$

we note that testing the condition

$$\delta^*(p, w) \in F \quad \text{iff} \quad \delta^*(q, w) \in F$$

for all strings in Σ^* is equivalent to testing the above condition for all strings of length at most i for all $i \geq 0$, i.e.

$$p \equiv q \quad \text{iff} \quad \forall i \geq 0\, \forall w \in \Sigma^*,\ |w| \leq i\, (\delta^*(p, w) \in F \quad \text{iff} \quad \delta^*(q, w) \in F).$$

Since $\equiv_i$ is defined such that

$$p \equiv_i q \quad \text{iff} \quad \forall w \in \Sigma^*,\ |w| \leq i\, (\delta^*(p, w) \in F \quad \text{iff} \quad \delta^*(q, w) \in F),$$

we conclude that

$$p \equiv q \quad \text{iff} \quad \forall i \geq 0\, (p \equiv_i q).$$

This identity can also be expressed as

$$\equiv\, = \bigcap_{i \geq 0} \equiv_i\ .$$

If we assume that $F \neq \emptyset$ and $F \neq Q$, observe that $\equiv_0$ has exactly two equivalence classes F and $Q - F$, since ϵ is the only string of length 0, and since the condition

$$\delta^*(p, \epsilon) \in F \quad \text{iff} \quad \delta^*(q, \epsilon) \in F$$

is equivalent to the condition

$$p \in F \quad \text{iff} \quad q \in F.$$

It is also obvious from the definition of $\equiv_i$ that

$$\equiv\, \subseteq \cdots \subseteq\, \equiv_{i+1}\, \subseteq\, \equiv_i\, \subseteq \cdots \subseteq\, \equiv_1\, \subseteq\, \equiv_0\ .$$

If this sequence was strictly decreasing for all $i \geq 0$, the partition associated with $\equiv_{i+1}$ would contain at least one more block than the partition associated with $\equiv_i$ and since we start with a partition with two blocks, the partition associated with $\equiv_i$ would have at least $i+2$ blocks. But then, for $i = n-1$, the partition associated with $\equiv_{n-1}$ would have at least $n+1$ blocks, which is absurd since Q has only n states. Therefore, there is a smallest integer, $i_0 \leq n-2$, such that

$$\equiv_{i_0+1}\, =\, \equiv_{i_0}\ .$$

Thus, it remains to compute $\equiv_{i+1}$ from $\equiv_i$, which can be done using the following proposition: The proposition also shows that

$$\equiv\, =\, \equiv_{i_0}\ .$$

Proposition 6.9. *For any (trim) DFA $D = (Q, \Sigma, \delta, q_0, F)$, for all $p, q \in Q$, $p \equiv_{i+1} q$ iff $p \equiv_i q$ and $\delta(p, a) \equiv_i \delta(q, a)$, for every $a \in \Sigma$. Furthermore, if $F \neq \emptyset$ and $F \neq Q$, there is a smallest integer $i_0 \leq n - 2$, such that*

$$\equiv_{i_0+1} = \equiv_{i_0} = \equiv .$$

Proof. By the definition of the relation $\equiv_i$,

$$p \equiv_{i+1} q \quad \text{iff} \quad \forall w \in \Sigma^*,\ |w| \leq i+1\, (\delta^*(p, w) \in F \quad \text{iff} \quad \delta^*(q, w) \in F).$$

The trick is to observe that the condition

$$\delta^*(p, w) \in F \quad \text{iff} \quad \delta^*(q, w) \in F$$

holds for all strings of length at most $i + 1$ iff it holds for all strings of length at most i and for all strings of length between 1 and $i + 1$. This is expressed as

$$\begin{aligned} p \equiv_{i+1} q \quad \text{iff} \quad & \forall w \in \Sigma^*,\ |w| \leq i\, (\delta^*(p, w) \in F \quad \text{iff} \quad \delta^*(q, w) \in F) \\ & \text{and} \\ & \forall w \in \Sigma^*,\ 1 \leq |w| \leq i+1\, (\delta^*(p, w) \in F \quad \text{iff} \quad \delta^*(q, w) \in F). \end{aligned}$$

Obviously, the first condition in the conjunction is $p \equiv_i q$, and since every string w such that $1 \leq |w| \leq i+1$ can be written as au where $a \in \Sigma$ and $0 \leq |u| \leq i$, the second condition in the conjunction can be written as

$$\forall a \in \Sigma\, \forall u \in \Sigma^*,\ |u| \leq i\, (\delta^*(p, au) \in F \quad \text{iff} \quad \delta^*(q, au) \in F).$$

However, $\delta^*(p, au) = \delta^*(\delta(p, a), u)$ and $\delta^*(q, au) = \delta^*(\delta(q, a), u)$, so that the above condition is really

$$\forall a \in \Sigma\, (\delta(p, a) \equiv_i \delta(q, a)).$$

Thus, we showed that

$$p \equiv_{i+1} q \quad \text{iff} \quad p \equiv_i q \quad \text{and} \quad \forall a \in \Sigma\, (\delta(p, a) \equiv_i \delta(q, a)).$$

We claim that if $\equiv_{i+1} = \equiv_i$ for some $i \geq 0$, then $\equiv_{i+j} = \equiv_i$ for all $j \geq 1$. This claim is proved by induction on j. For the base case j, the claim is that $\equiv_{i+1} = \equiv_i$, which is the hypothesis.

Assume inductively that $\equiv_{i+j} = \equiv_i$ for any $j \geq 1$. Since $p \equiv_{i+j+1} q$ iff $p \equiv_{i+j} q$ and $\delta(p, a) \equiv_{i+j} \delta(q, a)$, for every $a \in \Sigma$, and since by the induction hypothesis $\equiv_{i+j} = \equiv_i$, we obtain $p \equiv_{i+j+1} q$ iff $p \equiv_i q$ and $\delta(p, a) \equiv_i \delta(q, a)$, for every $a \in \Sigma$, which is equivalent to $p \equiv_{i+1} q$, and thus $\equiv_{i+j+1} = \equiv_{i+1}$. But $\equiv_{i+1} = \equiv_i$, so $\equiv_{i+j+1} = \equiv_i$, establishing the induction step.

Since

$$\equiv = \bigcap_{i \geq 0} \equiv_i, \quad \equiv_{i+1} \subseteq \equiv_i,$$

and since we know that there is a smallest index say i_0, such that $\equiv_j = \equiv_{i_0}$, for all $j \geq i_0 + 1$, we have $\equiv = \bigcap_{i=0}^{i_0} \equiv_i = \equiv_{i_0}$. ☐

Using Proposition 6.9, we can compute $\equiv$ inductively, starting from $\equiv_0 = (F, Q - F)$, and computing $\equiv_{i+1}$ from $\equiv_i$, until the sequence of partitions associated with the $\equiv_i$ stabilizes.

Note that if $F = Q$ or $F = \emptyset$, then $\equiv\, = \,\equiv_0$, and the inductive characterization of Proposition 6.9 holds trivially.

There are a number of algorithms for computing $\equiv$, or to determine whether $p \equiv q$ for some given $p, q \in Q$.

A simple method to compute $\equiv$ is described in Hopcroft and Ullman. The basic idea is to propagate inequivalence, rather than equivalence.

The method consists in forming a triangular array corresponding to all unordered pairs (p, q), with $p \neq q$ (the rows and the columns of this triangular array are indexed by the states in Q, where the entries are below the descending diagonal). Initially, the entry (p, q) is marked iff p and q are **not** **0-equivalent**, which means that p and q are not both in F or not both in $Q - F$.

Then, we process every unmarked entry on every row as follows: for any unmarked pair (p, q), we consider pairs $(\delta(p, a), \delta(q, a))$, for all $a \in \Sigma$. If any pair $(\delta(p, a), \delta(q, a))$ is already marked, this means that $\delta(p, a)$ and $\delta(q, a)$ are *inequivalent*, and thus p and q are *inequivalent*, and we mark the pair (p, q). We continue in this fashion, until at the end of a round during which all the rows are processed, nothing has changed. When the algorithm stops, all marked pairs are inequivalent, and all unmarked pairs correspond to equivalent states.

Let us illustrates the above method.

Example 6.14. Consider the following DFA accepting $\{a, b\}^*\{abb\}$:

	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

The start state is A, and the set of final states is $F = \{E\}$. (This is the DFA displayed in Figure 5.10.)

The initial (half) array is as follows, using $\times$ to indicate that the corresponding pair (say, (E, A)) consists of inequivalent states, and $\square$ to indicate that nothing is known yet.

B	$\square$			
C	$\square$	$\square$		
D	$\square$	$\square$	$\square$	
E	$\times$	$\times$	$\times$	$\times$
	A	B	C	D

After the first round, we have

B	□			
C	□	□		
D	$\times$	$\times$	$\times$	
E	$\times$	$\times$	$\times$	$\times$
	A	B	C	D

After the second round, we have

B	$\times$			
C	□	$\times$		
D	$\times$	$\times$	$\times$	
E	$\times$	$\times$	$\times$	$\times$
	A	B	C	D

Finally, nothing changes during the third round, and thus, only A and C are equivalent, and we get the four equivalence classes

$$(\{A, C\}, \{B\}, \{D\}, \{E\}).$$

We obtain the minimal DFA showed in Figure 6.4.

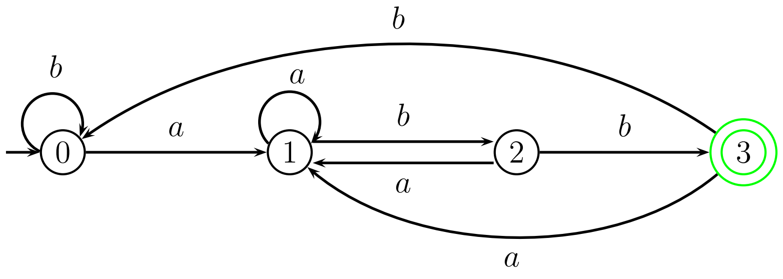


Figure 6.4: A minimal DFA accepting $\{a, b\}^*\{abb\}$.

There are ways of improving the efficiency of this algorithm, see Hopcroft and Ullman for such improvements. Fast algorithms for testing whether $p \equiv q$ for some given $p, q \in Q$ also exist. One of these algorithms is based on "forward closures," following an idea of Knuth. Such an algorithm is related to a fast unification algorithm; see Section 6.5.

6.4 The Pumping Lemma

Another useful tool for proving that languages are not regular is the so-called *pumping lemma*.

Proposition 6.10. *(Pumping lemma) Given any DFA $D = (Q, \Sigma, \delta, q_0, F)$, there is some $m \geq 1$ such that for every $w \in \Sigma^*$, if $w \in L(D)$ and $|w| \geq m$, then there exists a decomposition of w as $w = uxv$, where*

(1) $x \neq \epsilon$,

(2) $ux^iv \in L(D)$, for all $i \geq 0$, and

(3) $|ux| \leq m$.

Moreover, m can be chosen to be the number of states of the DFA D.

Proof. Let m be the number of states in Q, and let $w = w_1 \dots w_n$. Since Q contains the start state q_0, $m \geq 1$. Since $|w| \geq m$, we have $n \geq m$. Since $w \in L(D)$, let $(q_0, q_1, \dots, q_n)$, be the sequence of states in the accepting computation of w (where $q_n \in F$). Consider the subsequence

$$(q_0, q_1, \dots, q_m).$$

This sequence contains $m + 1$ states, but there are only m states in Q, and thus, we have $q_i = q_j$, for some i, j such that $0 \leq i < j \leq m$. Then, letting $u = w_1 \dots w_i$, $x = w_{i+1} \dots w_j$, and $v = w_{j+1} \dots w_n$, it is clear that the conditions of the proposition hold. □

An important consequence of the pumping lemma is that if a DFA D has m states and if there is some string $w \in L(D)$ such that $|w| \geq m$, then $L(D)$ is infinite.

Indeed, by the pumping lemma, $w \in L(D)$ can be written as $w = uxv$ with $x \neq \epsilon$, and

$$ux^iv \in L(D) \quad \text{for all } i \geq 0.$$

Since $x \neq \epsilon$, we have $|x| > 0$, so for all $i, j \geq 0$ with $i < j$ we have

$$|ux^iv| < |ux^iv| + (j - i)|x| = |ux^jv|,$$

which implies that $ux^iv \neq ux^jv$ for all $i < j$, and the set of strings

$$\{ux^iv \mid i \geq 0\} \subseteq L(D)$$

is an *infinite* subset of $L(D)$, which is itself infinite.

As a consequence, if $L(D)$ is finite, there are *no* strings w in $L(D)$ such that $|w| \geq m$. In this case, since the premise of the pumping lemma is false, the pumping lemma holds vacuously; that is, if $L(D)$ is finite, the pumping lemma yields no information.

Another corollary of the pumping lemma is that there is a test to decide whether a DFA D accepts an infinite language $L(D)$.

Proposition 6.11. *Let D be a DFA with m states, The language $L(D)$ accepted by D is infinite iff there is some string $w \in L(D)$ such that $m \leq |w| < 2m$.*

If $L(D)$ is infinite, there are strings of length $\geq m$ in $L(D)$, but a prirori there is no guarantee that there are "short" strings w in $L(D)$, that is, strings whose length is uniformly bounded by some function of m independent of D. The pumping lemma ensures that there are such strings, and the function is $m \mapsto 2m$.

Typically, the pumping lemma is used to prove that a language is not regular. The method is to proceed by contradiction, i.e., to assume (contrary to what we wish to prove) that a language L is indeed regular, and derive a contradiction of the pumping lemma. Thus, it would be helpful to see what the negation of the pumping lemma is, and for this, we first state the pumping lemma as a logical formula. We will use the following abbreviations:

$$\begin{aligned}
\mathit{nat} &= \{0, 1, 2, \ldots\}, \\
\mathit{pos} &= \{1, 2, \ldots\}, \\
A &\equiv w = uxv, \\
B &\equiv x \neq \epsilon, \\
C &\equiv |ux| \leq m, \\
P &\equiv \forall i \colon \mathit{nat}\ (ux^i v \in L(D)).
\end{aligned}$$

The pumping lemma can be stated as

$$\forall D\colon \text{DFA}\ \exists m\colon \mathit{pos}\ \forall w\colon \Sigma^* \Big((w \in L(D) \wedge |w| \geq m) \supset (\exists u, x, v\colon \Sigma^*\ A \wedge B \wedge C \wedge P)\Big).$$

Recalling that

$$\neg(A \wedge B \wedge C \wedge P) \equiv \neg(A \wedge B \wedge C) \vee \neg P \equiv (A \wedge B \wedge C) \supset \neg P$$

and

$$\neg(R \supset S) \equiv R \wedge \neg S,$$

the negation of the pumping lemma can be stated as

$$\exists D\colon \text{DFA}\ \forall m\colon \mathit{pos}\ \exists w\colon \Sigma^* \Big((w \in L(D) \wedge |w| \geq m) \wedge (\forall u, x, v\colon \Sigma^*\ (A \wedge B \wedge C) \supset \neg P)\Big).$$

Since

$$\neg P \equiv \exists i\colon \mathit{nat}\ (ux^i v \notin L(D)),$$

in order to show that the pumping lemma is contradicted, one needs to show that for some DFA D, for every $m \geq 1$, there is some string $w \in L(D)$ of length at least m, such that for every possible decomposition $w = uxv$ satisfying the constraints $x \neq \epsilon$ and $|ux| \leq m$, there is some $i \geq 0$ such that $ux^i v \notin L(D)$.

When proceeding by contradiction, we have a language L that we are (wrongly) assuming to be regular, and we can use any DFA D accepting L. The creative part of the argument is to pick the right $w \in L$ (not making any assumption on $m \leq |w|$).

As an illustration, let us use the pumping lemma to prove that $L_1 = \{a^n b^n \mid n \geq 1\}$ is not regular. The usefulness of the condition $|ux| \leq m$ lies in the fact that it reduces the number of legal decomposition uxv of w. We proceed by contradiction. Thus, let us assume that $L_1 = \{a^n b^n \mid n \geq 1\}$ is regular. If so, it is accepted by some DFA D. Now, we wish to contradict the pumping lemma. For every $m \geq 1$, let $w = a^m b^m$. Clearly, $w = a^m b^m \in L_1$ and $|w| \geq m$. Then, every legal decomposition u, x, v of w is such that

$$w = \underbrace{a \ldots a}_{u}\underbrace{a \ldots a}_{x}\underbrace{a \ldots a b \ldots b}_{v}$$

where $x \neq \epsilon$ and x ends within the a's, since $|ux| \leq m$. Since $x \neq \epsilon$, the string $uxxv$ is of the form $a^n b^m$ where $n > m$, and thus $uxxv \notin L_1$, contradicting the pumping lemma.

Let us consider two more examples. let $L_2 = \{a^m b^n \mid 1 \leq m < n\}$. We claim that L_2 is not regular. Our first proof uses the pumping lemma. For any $m \geq 1$, pick $w = a^m b^{m+1}$. We have $w \in L_2$ and $|w| \geq m$ so we need to contradict the pumping lemma. Every legal decomposition u, x, v of w is such that

$$w = \underbrace{a \ldots a}_{u}\underbrace{a \ldots a}_{x}\underbrace{a \ldots a b \ldots b}_{v}$$

where $x \neq \epsilon$ and x ends within the a's, since $|ux| \leq m$. Since $x \neq \epsilon$ and x consists of a's the string $ux^2v = uxxv$ contains at least $m+1$ a's and still $m+1$ b's, so $ux^2v \notin L_2$, contradicting the pumping lemma.

Our second proof uses Myhill-Nerode. Let $\simeq$ be a right-invariant equivalence relation of finite index such that L_2 is the union of classes of $\simeq$. If we consider the infinite sequence

$$a, a^2, \ldots, a^n, \ldots$$

since $\simeq$ has a finite number of classes there are two strings a^m and a^n with $m < n$ such that

$$a^m \simeq a^n.$$

By right-invariance by concatenating on the right with b^n we obtain

$$a^m b^n \simeq a^n b^n,$$

and since $m < n$ we have $a^m b^n \in L_2$ but $a^n b^n \notin L_2$, a contradiction.

Let us now consider the language $L_3 = \{a^m b^n \mid m \neq n\}$. This time let us begin by using Myhill-Nerode to prove that L_3 is not regular. The proof is the same as before, we obtain

$$a^m b^n \simeq a^n b^n,$$

and the contradiction is that $a^m b^n \in L_3$ and $a^n b^n \notin L_3$.

Let use now try to use the pumping lemma to prove that L_3 is not regular. For any $m \geq 1$ pick $w = a^m b^{m+1} \in L_3$. As in the previous case, every legal decomposition u, x, v of w is such that

$$w = \underbrace{a \ldots a}_{u}\underbrace{a \ldots a}_{x}\underbrace{a \ldots ab \ldots b}_{v}$$

where $x \neq \epsilon$ and x ends within the a's, since $|ux| \leq m$. However this time we have a problem, namely that we know that x is a nonempty string of a's but we don't know how many, so we can't guarantee that pumping up x will yield exactly the string $a^{m+1}b^{m+1}$. We made the wrong choice for w. There is a choice that will work but it is a bit tricky.

Fortunately, there is another simpler approach. Recall that the regular languages are closed under the boolean operations (union, intersection and complementation). Thus, L_3 is not regular iff its complement $\overline{L}_3$ is not regular. Observe that $\overline{L}_3$ contains $\{a^n b^n \mid n \geq 1\}$, which we showed to be nonregular. But there is another problem, which is that $\overline{L}_3$ contains other strings besides strings of the form $a^n b^n$, for example strings of the form $b^m a^n$ with $m, n > 0$.

Again, we can take care of this difficulty using the closure operations of the regular languages. If we can find a regular language R such that $\overline{L}_3 \cap R$ is not regular, then $\overline{L}_3$ itself is not regular, since otherwise as $\overline{L}_3$ and R are regular then $\overline{L}_3 \cap R$ is also regular. In our case, we can use $R = \{a\}^+\{b\}^+$ to obtain

$$\overline{L}_3 \cap \{a\}^+\{b\}^+ = \{a^n b^n \mid n \geq 1\}.$$

Since $\{a^n b^n \mid n \geq 1\}$ is not regular, we reached our final contradiction. Observe how we use the language R to "clean up" $\overline{L}_3$ by intersecting it with R.

To complete a direct proof using the pumping lemma, the reader should try $w = a^{m!}b^{(m+1)!}$.

The use of the closure operations of the regular languages is often a quick way of showing that a language L is not regular by reducing the problem of proving that L is not regular to the problem of proving that some well-known language is not regular.

6.5 A Fast Algorithm for Checking State Equivalence Using a "Forward-Closure"

Given two states $p, q \in Q$, if $p \equiv q$, then we know that $\delta(p, a) \equiv \delta(q, a)$, for all $a \in \Sigma$. This suggests a method for testing whether two distinct states p, q are equivalent. Starting with the relation $R = \{(p, q)\}$, construct the smallest equivalence relation $R^\dagger$ containing R with the property that whenever $(r, s) \in R^\dagger$, then $(\delta(r, a), \delta(s, a)) \in R^\dagger$, for all $a \in \Sigma$. If we ever encounter a pair (r, s) such that $r \in F$ and $s \in \overline{F}$, or $r \in \overline{F}$ and $s \in F$, then r and

s are inequivalent, and so are p and q. Otherwise, it can be shown that p and q are indeed equivalent. Thus, testing for the equivalence of two states reduces to finding an efficient method for computing the "forward closure" of a relation defined on the set of states of a DFA.

Such a method was worked out by John Hopcroft and Richard Karp and published in a 1971 Cornell technical report. This method is based on an idea of Donald Knuth for solving Exercise 11, in Section 2.3.5 of *The Art of Computer Programming*, Vol. 1, second edition, 1973. A sketch of the solution for this exercise is given on page 594. As far as I know, Hopcroft and Karp's method was never published in a journal, but a simple recursive algorithm does appear on page 144 of Aho, Hopcroft and Ullman's *The Design and Analysis of Computer Algorithms*, first edition, 1974. Essentially the same idea was used by Paterson and Wegman to design a fast unification algorithm (in 1978). We make a few definitions.

A relation $S \subseteq Q \times Q$ is a *forward closure* iff it is an equivalence relation and whenever $(r, s) \in S$, then $(\delta(r, a), \delta(s, a)) \in S$, for all $a \in \Sigma$. The *forward closure* of a relation $R \subseteq Q \times Q$ is the smallest equivalence relation $R^{\dagger}$ containing R which is forward closed.

We say that a forward closure S is *good* iff whenever $(r, s) \in S$, then $good(r, s)$, where $good(r, s)$ holds iff either both $r, s \in F$, or both $r, s \notin F$. Obviously, $bad(r, s)$ iff $\neg good(r, s)$.

Given any relation $R \subseteq Q \times Q$, recall that the smallest equivalence relation $R_{\approx}$ containing R is the relation $(R \cup R^{-1})^*$ (where $R^{-1} = \{(q, p) \mid (p, q) \in R\}$, and $(R \cup R^{-1})^*$ is the reflexive and transitive closure of $(R \cup R^{-1})$). The forward closure of R can be computed inductively by defining the sequence of relations $R_i \subseteq Q \times Q$ as follows:

$$\begin{aligned} R_0 &= R_{\approx} \\ R_{i+1} &= (R_i \cup \{(\delta(r, a), \delta(s, a)) \mid (r, s) \in R_i,\ a \in \Sigma\})_{\approx}. \end{aligned}$$

It is not hard to prove that $R_{i_0+1} = R_{i_0}$ for some least i_0, and that $R^{\dagger} = R_{i_0}$ is the smallest forward closure containing R. The following two facts can also been established.

(a) if $R^{\dagger}$ is good, then
$$R^{\dagger} \subseteq \equiv . \tag{6.1}$$

(b) if $p \equiv q$, then
$$R^{\dagger} \subseteq \equiv,$$
that is, equation (6.1) holds. This implies that $R^{\dagger}$ is good.

As a consequence, we obtain the correctness of our procedure: $p \equiv q$ iff the forward closure $R^{\dagger}$ of the relation $R = \{(p, q)\}$ is good.

In practice, we maintain a partition Π representing the equivalence relation that we are closing under forward closure. We add each new pair $(\delta(r, a), \delta(s, a))$ one at a time, and immediately form the smallest equivalence relation containing the new relation. If $\delta(r, a)$ and $\delta(s, a)$ already belong to the same block of Π, we consider another pair, else we merge the blocks corresponding to $\delta(r, a)$ and $\delta(s, a)$, and then consider another pair.

The algorithm is recursive, but it can easily be implemented using a stack. To manipulate partitions efficiently, we represent them as lists of trees (forests). Each equivalence class C in the partition Π is represented by a tree structure consisting of nodes and parent pointers, with the pointers from the sons of a node to the node itself. The root has a null pointer. Each node also maintains a counter keeping track of the number of nodes in the subtree rooted at that node.

Note that pointers can be avoided. We can represent a forest of n nodes as a list of n pairs of the form $(father, count)$. If $(father, count)$ is the ith pair in the list, then $father = 0$ iff node i is a root node, otherwise, $father$ is the index of the node in the list which is the parent of node i. The number $count$ is the total number of nodes in the tree rooted at the ith node.

For example, the following list of nine nodes

$$((0, 3), (0, 2), (1, 1), (0, 2), (0, 2), (1, 1), (2, 1), (4, 1), (5, 1))$$

represents a forest consisting of the following four trees:

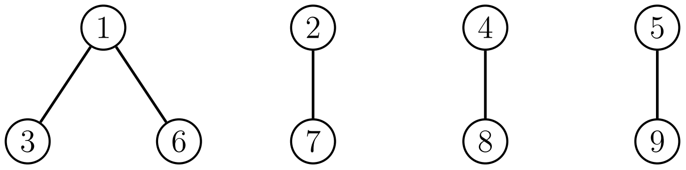


Figure 6.5: A forest of four trees.

Two functions *union* and *find* are defined as follows. Given a state p, $find(p, \Pi)$ finds the root of the tree containing p as a node (not necessarily a leaf). Given two root nodes p, q, $union(p, q, \Pi)$ forms a new partition by merging the two trees with roots p and q as follows: if the counter of p is smaller than that of q, then let the root of p point to q, else let the root of q point to p.

For example, given the two trees shown on the left in Figure 6.6, $find(6, \Pi)$ returns 3 and $find(8, \Pi)$ returns 4. Then $union(3, 4, \Pi)$ yields the tree shown on the right in Figure 6.6.

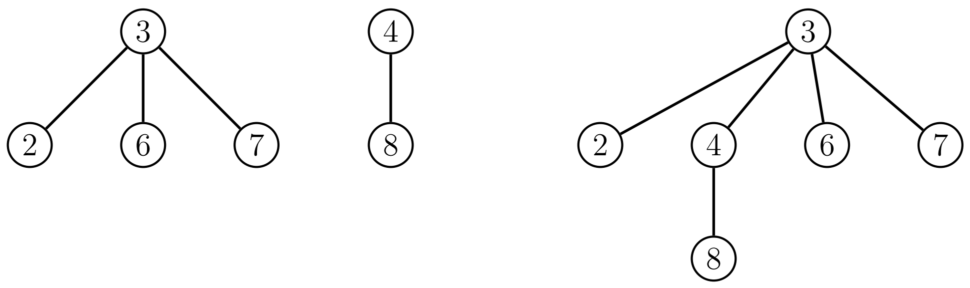


Figure 6.6: Applying the function *union* to the trees rooted at 3 and 4.

In order to speed up the algorithm, using an idea due to Tarjan, we can modify *find* as follows: during a call *find*(p, Π), as we follow the path from p to the root r of the tree containing p, we redirect the parent pointer of every node q on the path from p (including p itself) to r (we perform *path compression*). For example, applying *find*$(8, \Pi)$ to the tree shown on the right in Figure 6.6 yields the tree shown in Figure 6.7

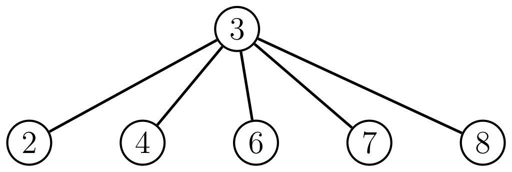


Figure 6.7: The result of applying *find* with path compression.

Then, the algorithm is as follows:

```
function $unif[p, q, \Pi, dd]$: $flag$;
  begin
    $trans := left(dd)$; $ff := right(dd)$; $pq := (p, q)$; $st := (pq)$; $flag := 1$;
    $k := Length(first(trans))$;
    while $st \neq () \land flag \neq 0$ do
      $uv := top(st)$; $uu := left(uv)$; $vv := right(uv)$;
      $pop(st)$;
      if $bad(ff, uv) = 1$ then $flag := 0$
      else
        $u := find(uu, \Pi)$; $v := find(vv, \Pi)$;
        if $u \neq v$ then
          $union(u, v, \Pi)$;
          for $i = 1$ to $k$ do
            $u1 := delta(trans, uu, k - i + 1)$; $v1 := delta(trans, vv, k - i + 1)$;
            $uv := (u1, v1)$; $push(st, uv)$
          endfor
        endif
      endif
    endwhile
  end
```

The initial partition Π is the identity relation on Q, i.e., it consists of blocks $\{q\}$ for all states $q \in Q$. The algorithm uses a stack *st*. We are assuming that the DFA *dd* is specified by a list of two sublists, the first list, denoted *left*(*dd*) in the pseudo-code above, being a representation of the transition function, and the second one, denoted *right*(*dd*), the set of final states. The transition function itself is a list of lists, where the i-th list represents the i-th row of the transition table for *dd*. The function *delta* is such that $delta(trans, i, j)$ returns the j-th state in the i-th row of the transition table of *dd*. For example, we have the DFA

$$dd = (((2, 3), (2, 4), (2, 3), (2, 5), (2, 3), (7, 6), (7, 8), (7, 9), (7, 6)), (5, 9))$$

consisting of 9 states labeled $1, \ldots, 9$, and two final states 5 and 9 shown in Figure 6.8. Also, the alphabet has two letters, since every row in the transition table consists of two entries. For example, the two transitions from state 3 are given by the pair $(2, 3)$, which indicates that $\delta(3, a) = 2$ and $\delta(3, b) = 3$.

The sequence of steps performed by the algorithm starting with $p = 1$ and $q = 6$ is shown below. At every step, we show the current pair of states, the partition, and the stack.

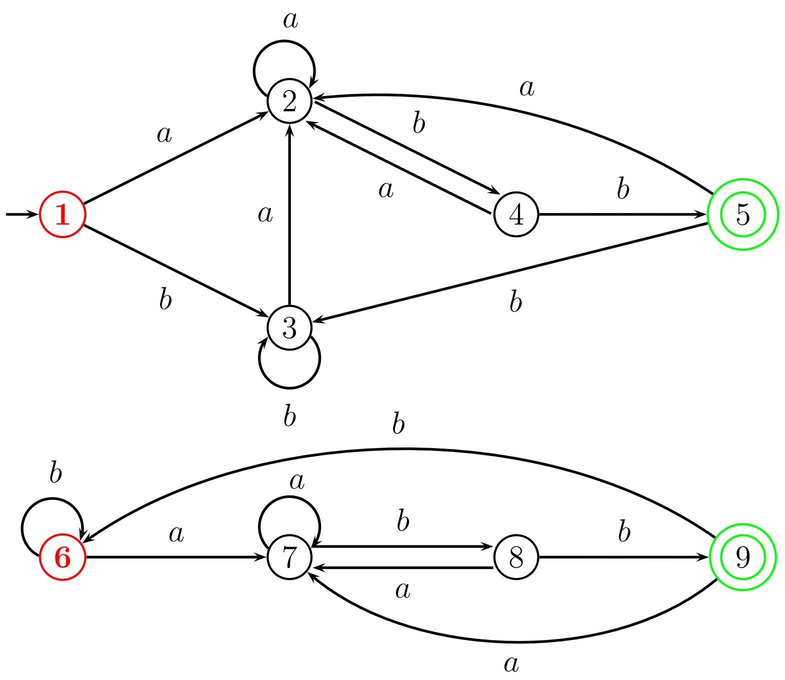


Figure 6.8: Testing state equivalence in a DFA.

$p = 1, q = 6,\ \Pi = \{\{1,6\},\{2\},\{3\},\{4\},\{5\},\{7\},\{8\},\{9\}\},\ st = \{\{1,6\}\}$

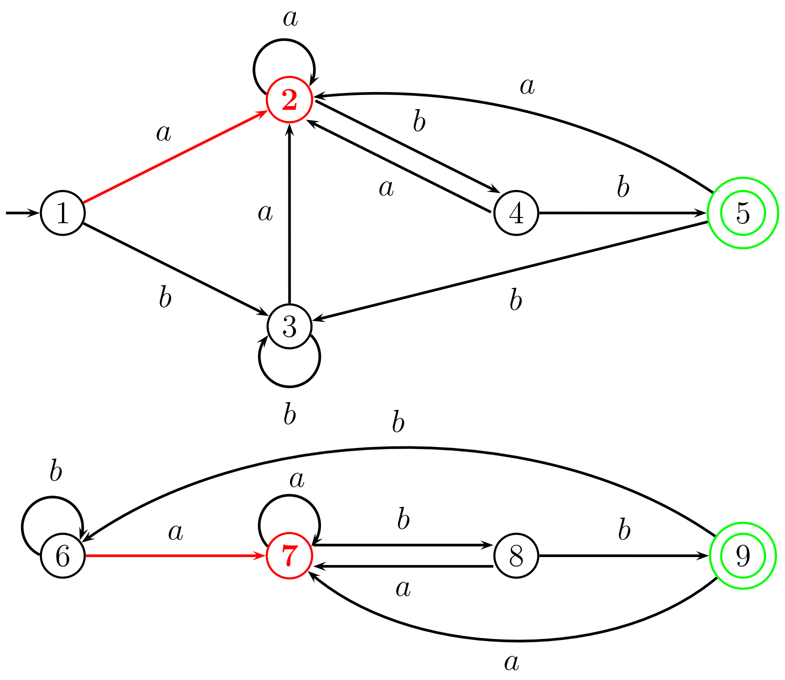


Figure 6.9: Testing state equivalence in a DFA.

$p = 2, q = 7,\ \Pi = \{\{1,6\},\{2,7\},\{3\},\{4\},\{5\},\{8\},\{9\}\},\ st = \{\{3,6\},\{2,7\}\}$

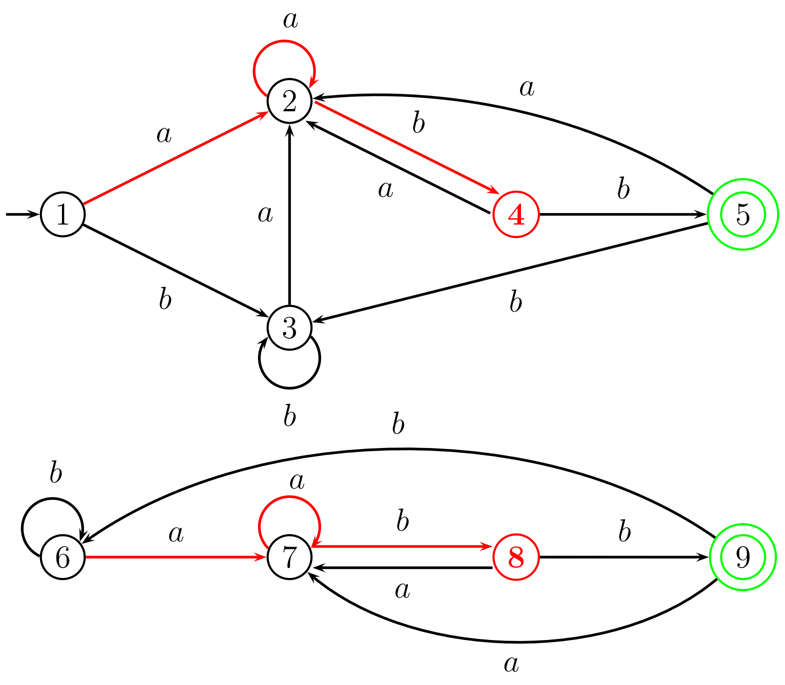


Figure 6.10: Testing state equivalence in a DFA.

$p = 4, q = 8,\ \Pi = \{\{1,6\},\{2,7\},\{3\},\{4,8\},\{5\},\{9\}\},\ st = \{\{3,6\},\{4,8\}\}$

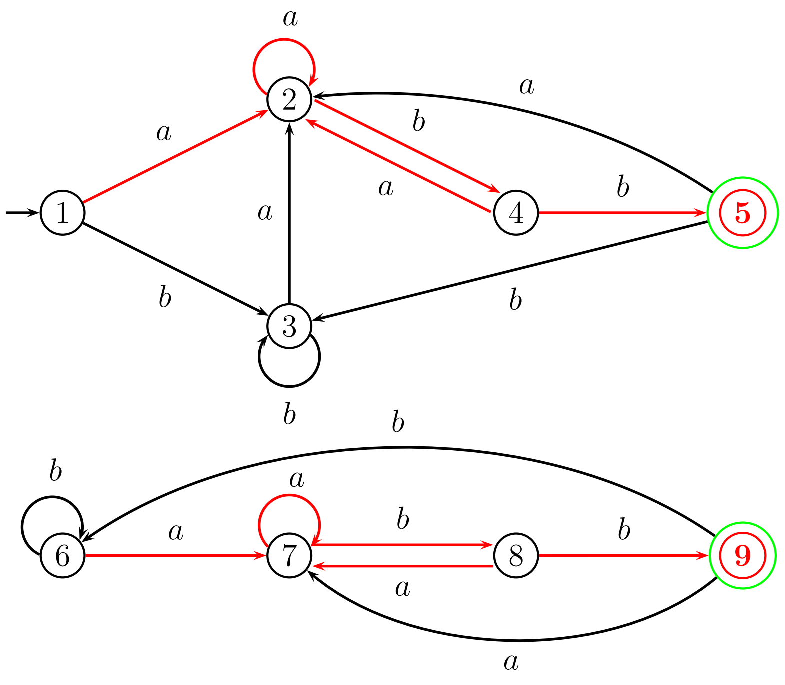


Figure 6.11: Testing state equivalence in a DFA.

$p = 5, q = 9,\ \Pi = \{\{1,6\},\{2,7\},\{3\},\{4,8\},\{5,9\}\},\ st = \{\{3,6\},\{5,9\}\}$

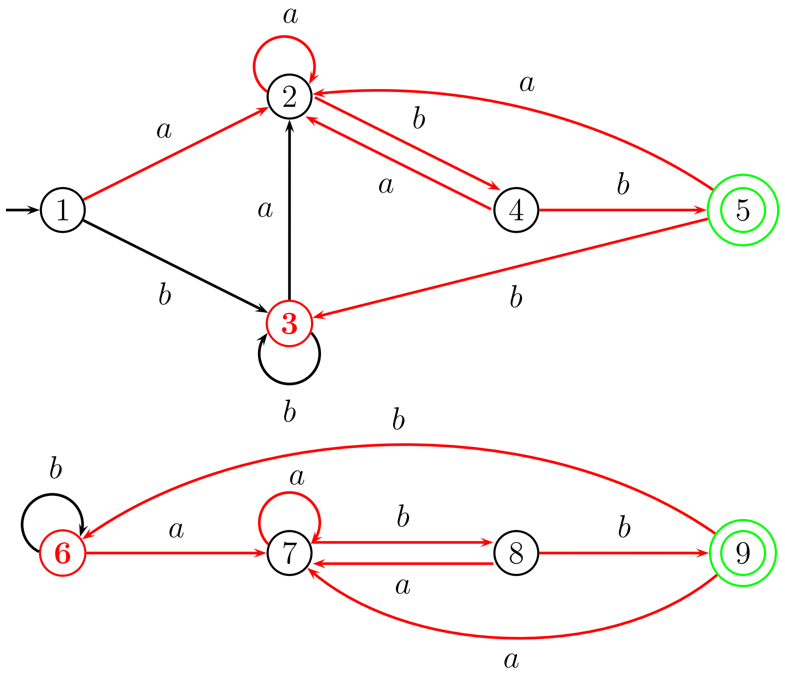


Figure 6.12: Testing state equivalence in a DFA.

$p = 3, q = 6, \ \Pi = \{\{1,3,6\},\{2,7\},\{4,8\},\{5,9\}\}, \ st = \{\{3,6\},\{3,6\}\}$

Since states 3 and 6 belong to the first block of the partition, the algorithm terminates. Since no block of the partition contains a bad pair, the states $p = 1$ and $q = 6$ are equivalent.

Let us now test whether the states $p = 3$ and $q = 7$ are equivalent.

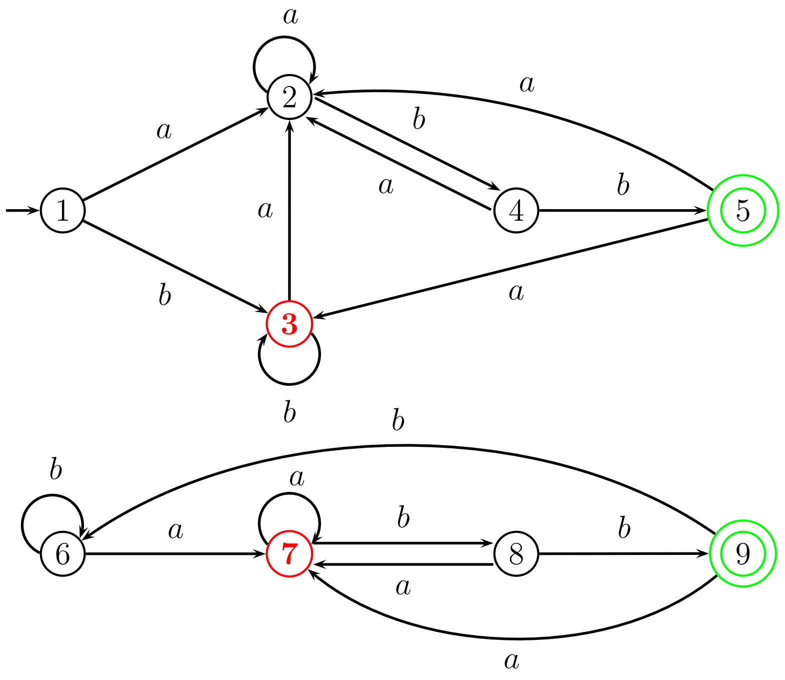


Figure 6.13: Testing state equivalence in a DFA.

$p = 3, q = 7,\ \Pi = \{\{1\}, \{2\}, \{3, 7\}, \{4\}, \{5\}, \{6\}, \{8\}, \{9\}\},\ st = \{\{3, 7\}\}$

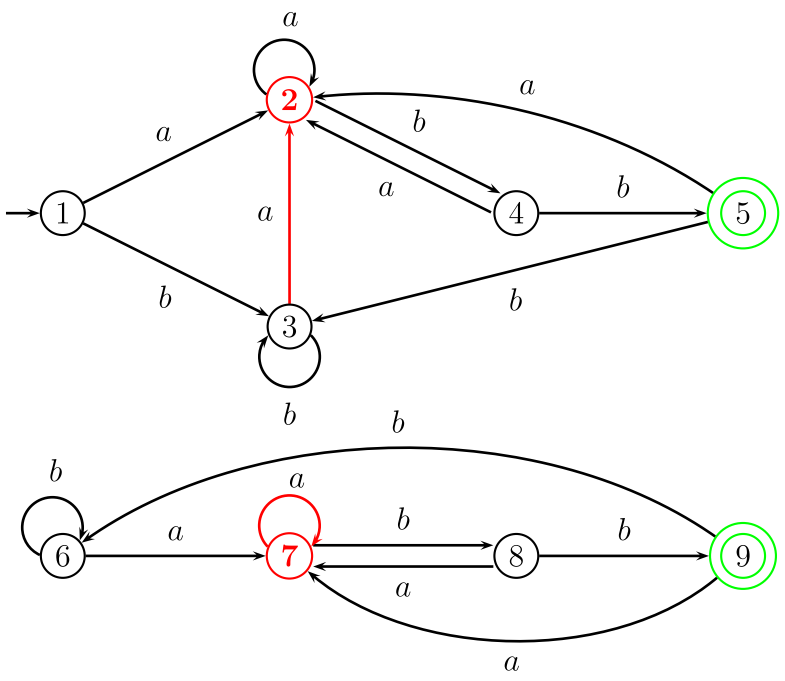


Figure 6.14: Testing state equivalence in a DFA.

$p = 2, q = 7,\ \Pi = \{\{1\}, \{2, 3, 7\}, \{4\}, \{5\}, \{6\}, \{8\}, \{9\}\},\ st = \{\{3, 8\}, \{2, 7\}\}$

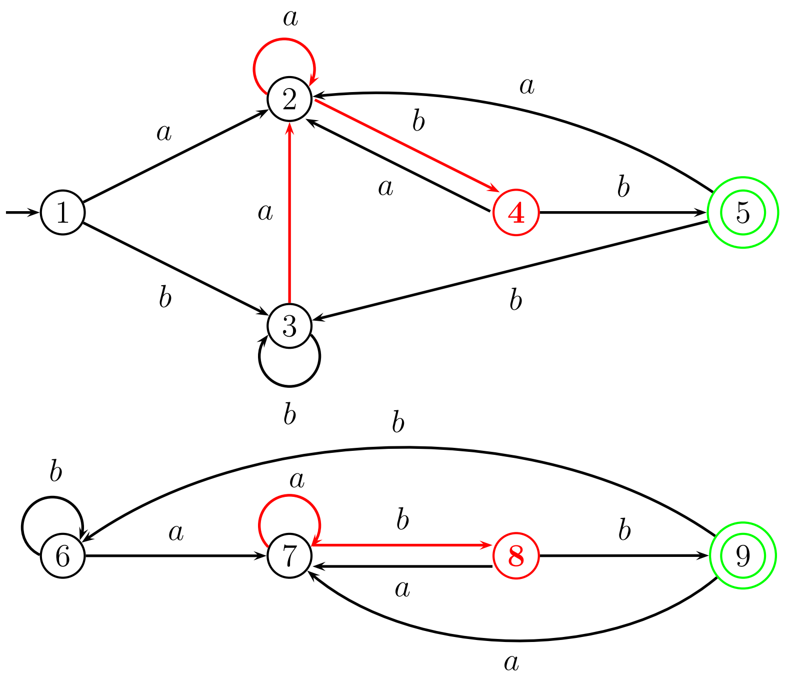


Figure 6.15: Testing state equivalence in a DFA.

$p = 4, q = 8,\ \Pi = \{\{1\}, \{2, 3, 7\}, \{4, 8\}, \{5\}, \{6\}, \{9\}\},\ st = \{\{3, 8\}, \{4, 8\}\}$

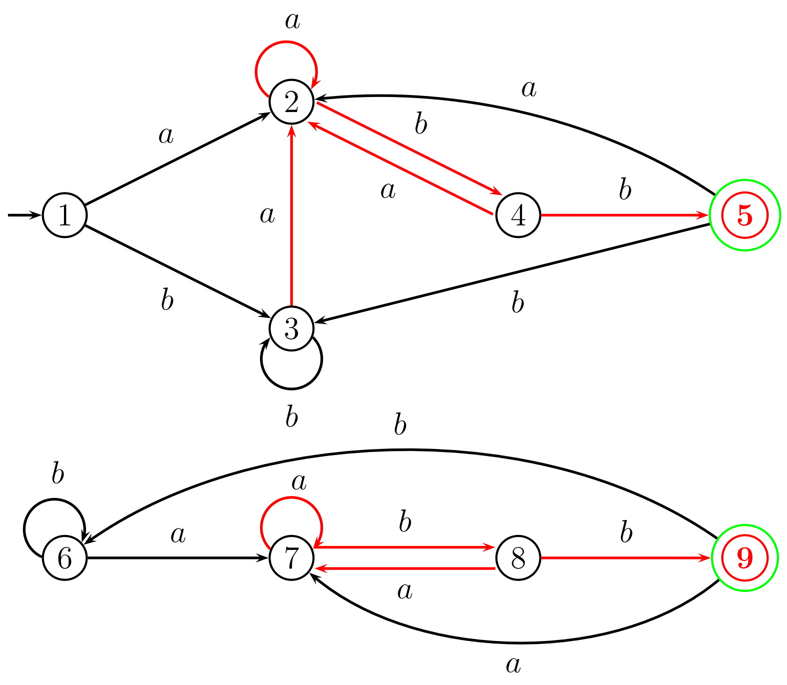


Figure 6.16: Testing state equivalence in a DFA.

$p = 5, q = 9,\ \Pi = \{\{1\}, \{2, 3, 7\}, \{4, 8\}, \{5, 9\}, \{6\}\},\ st = \{\{3, 8\}, \{5, 9\}\}$

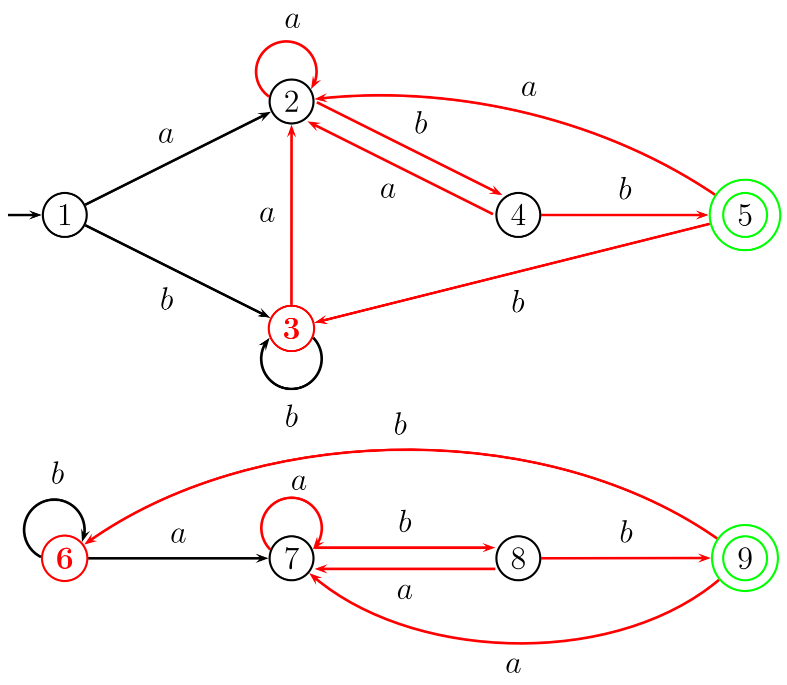


Figure 6.17: Testing state equivalence in a DFA.

$p = 3, q = 6,\ \Pi = \{\{1\}, \{2, 3, 6, 7\}, \{4, 8\}, \{5, 9\}\},\ st = \{\{3, 8\}, \{3, 6\}\}$

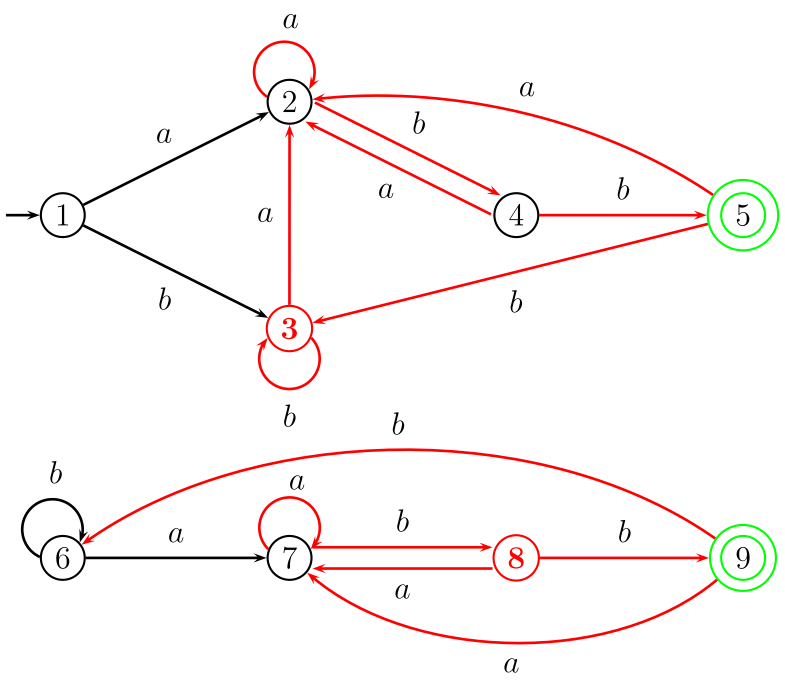


Figure 6.18: Testing state equivalence in a DFA.

$p = 3, q = 8,\ \Pi = \{\{1\}, \{2, 3, 4, 6, 7, 8\}, \{5, 9\}\},\ st = \{\{3, 8\}\}$

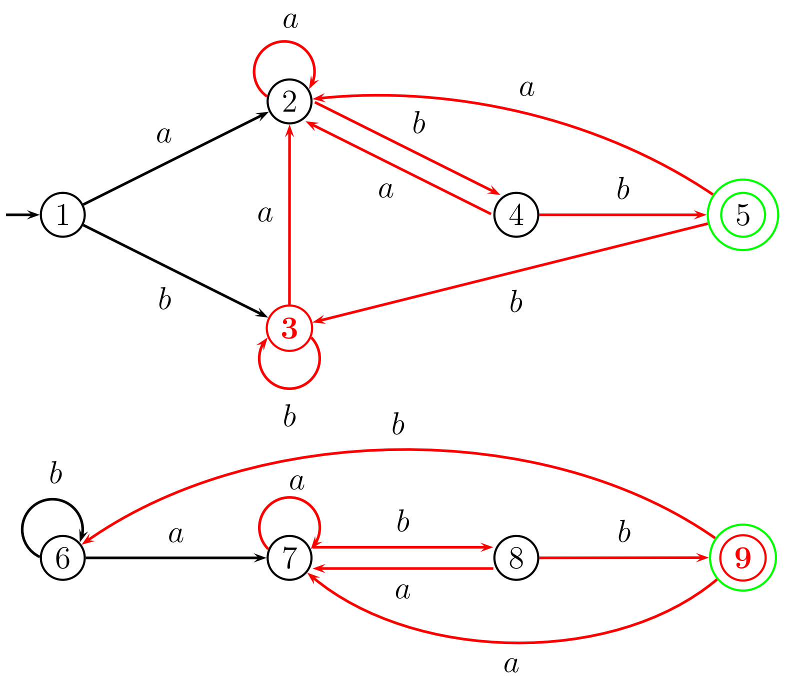


Figure 6.19: Testing state equivalence in a DFA.

$p = 3, q = 9,\ \Pi = \{\{1\}, \{2, 3, 4, 6, 7, 8\}, \{5, 9\}\},\ st = \{\{3, 9\}\}$

Since the pair $(3, 9)$ is a bad pair, the algorithm stops, and the states $p = 3$ and $q = 7$ are inequivalent.

Chapter 7

Context-Free Grammars, Context-Free Languages, Parse Trees and Ogden's Lemma

7.1 Context-Free Grammars

A context-free grammar basically consists of a finite set of grammar rules. In order to define grammar rules, we assume that we have two kinds of symbols: the terminals, which are the symbols of the alphabet underlying the languages under consideration, and the nonterminals, which behave like variables ranging over strings of terminals. A rule is of the form $A \to \alpha$, where A is a single nonterminal, and the right-hand side α is a string of terminal and/or nonterminal symbols. As usual, first we need to define what the object is (a context-free grammar), and then we need to explain how it is used. Unlike automata, grammars are used to *generate* strings, rather than recognize strings.

Definition 7.1. A *context-free grammar (for short, CFG)* is a quadruple $G = (V, \Sigma, P, S)$, where

- V is a finite set of symbols called the *vocabulary (or set of grammar symbols)*;
- $\Sigma \subseteq V$ is the set of *terminal symbols (for short, terminals)*;
- $S \in (V - \Sigma)$ is a designated symbol called the *start symbol*;
- $P \subseteq (V - \Sigma) \times V^*$ is a finite set of *productions (or rewrite rules, or rules)*.

The set $N = V - \Sigma$ is called the set of *nonterminal symbols (for short, nonterminals)*. Thus, $P \subseteq N \times V^*$, and every production $\langle A, \alpha \rangle$ is also denoted as $A \to \alpha$. A production of the form $A \to \epsilon$ is called an *epsilon rule, or null rule*.

Remark: Context-free grammars are sometimes defined as $G = (V_N, V_T, P, S)$. The correspondence with our definition is that $\Sigma = V_T$ and $N = V_N$, so that $V = V_N \cup V_T$. Thus, in this other definition, it is necessary to assume that $V_T \cap V_N = \emptyset$.

Example 1. $G_1 = (\{E, a, b\}, \{a, b\}, P, E)$, where P is the set of rules

$$\begin{aligned} E &\longrightarrow aEb, \\ E &\longrightarrow ab. \end{aligned}$$

As we will see shortly, this grammar generates the language $L_1 = \{a^n b^n \mid n \geq 1\}$, which is not regular.

Example 2. $G_2 = (\{E, +, *, (,), a\}, \{+, *, (,), a\}, P, E)$, where P is the set of rules

$$\begin{aligned} E &\longrightarrow E + E, \\ E &\longrightarrow E * E, \\ E &\longrightarrow (E), \\ E &\longrightarrow a. \end{aligned}$$

This grammar generates a set of arithmetic expressions.

7.2 Derivations and Context-Free Languages

The productions of a grammar are used to derive strings. In this process, the productions are used as rewrite rules. Formally, we define the derivation relation associated with a context-free grammar. First, let us review the concepts of transitive closure and reflexive and transitive closure of a binary relation.

Given a set A, a *binary relation R on A* is any set of ordered pairs, i.e. $R \subseteq A \times A$. For short, instead of binary relation, we often simply say relation. Given any two relations R, S on A, their *composition* $R \circ S$ is defined as

$$R \circ S = \{(x,\, y) \in A \times A \mid \exists z \in A,\, (x,\;\, z) \in R \text{ and } (z,\, y) \in S\}.$$

The *identity relation I_A on A* is the relation I_A defined such that

$$I_A = \{(x,\, x) \mid x \in A\}.$$

For short, we often denote I_A as I. Note that

$$R \circ I = I \circ R = R$$

for every relation R on A. Given a relation R on A, for any $n \geq 0$ we define R^n as follows:

$$\begin{aligned} R^0 &= I, \\ R^{n+1} &= R^n \circ R. \end{aligned}$$

It is obvious that $R^1 = R$. It is also easily verified by induction that $R^n \circ R = R \circ R^n$. The *transitive closure R^+ of the relation R* is defined as

$$R^+ = \bigcup_{n \geq 1} R^n.$$

It is easily verified that R^+ is the smallest transitive relation containing R, and that $(x, y) \in R^+$ iff there is some $n \geq 1$ and some $x_0, x_1, \ldots, x_n \in A$ such that $x_0 = x$, $x_n = y$, and $(x_i, x_{i+1}) \in R$ for all i, $0 \leq i \leq n - 1$. The *transitive and reflexive closure R^* of the relation R* is defined as

$$R^* = \bigcup_{n \geq 0} R^n.$$

Clearly, $R^* = R^+ \cup I$. It is easily verified that R^* is the smallest transitive and reflexive relation containing R.

Definition 7.2. Given a context-free grammar $G = (V, \Sigma, P, S)$, the (one-step) *derivation relation $\Longrightarrow_G$ associated with G* is the binary relation $\Longrightarrow_G \subseteq V^* \times V^*$ defined as follows: for all $\alpha, \beta \in V^*$, we have

$$\alpha \Longrightarrow_G \beta$$

iff there exist $\lambda, \rho \in V^*$, and some production $(A \to \gamma) \in P$, such that

$$\alpha = \lambda A \rho \quad \text{and} \quad \beta = \lambda \gamma \rho.$$

The transitive closure of $\Longrightarrow_G$ is denoted as $\overset{+}{\Longrightarrow}_G$ and the reflexive and transitive closure of $\Longrightarrow_G$ is denoted as $\overset{*}{\Longrightarrow}_G$.

When the grammar G is clear from the context, we usually omit the subscript G in $\Longrightarrow_G$, $\overset{+}{\Longrightarrow}_G$, and $\overset{*}{\Longrightarrow}_G$.

A string $\alpha \in V^*$ such that $S \overset{*}{\Longrightarrow} \alpha$ is called a *sentential form*, and a string $w \in \Sigma^*$ such that $S \overset{*}{\Longrightarrow} w$ is called a *sentence*. A derivation $\alpha \overset{*}{\Longrightarrow} \beta$ involving n steps is denoted as $\alpha \overset{n}{\Longrightarrow} \beta$.

Note that a derivation step

$$\alpha \Longrightarrow_G \beta$$

is rather nondeterministic. Indeed, one can choose among various occurrences of nonterminals A in α, and also among various productions $A \to \gamma$ with left-hand side A.

For example, using the grammar $G_1 = (\{E, a, b\}, \{a, b\}, P, E)$, where P is the set of rules

$$E \longrightarrow aEb,$$
$$E \longrightarrow ab,$$

every derivation from E is of the form

$$E \overset{*}{\Longrightarrow} a^n E b^n \Longrightarrow a^n abb^n = a^{n+1}b^{n+1},$$

or

$$E \overset{*}{\Longrightarrow} a^n E b^n \Longrightarrow a^n aEbb^n = a^{n+1}Eb^{n+1},$$

where $n \geq 0$.

Grammar G_1 is very simple: every string $a^n b^n$ has a unique derivation. This is usually not the case. For example, using the grammar $G_2 = (\{E, +, *, (,), a\}, \{+, *, (,), a\}, P, E)$, where P is the set of rules

$$\begin{aligned} E &\longrightarrow E + E, \\ E &\longrightarrow E * E, \\ E &\longrightarrow (E), \\ E &\longrightarrow a, \end{aligned}$$

the string $a + a * a$ has the following distinct derivations, where the boldface indicates which occurrence of E is rewritten:

$$\begin{aligned} &\mathbf{E} \Longrightarrow \mathbf{E} * E \Longrightarrow \mathbf{E} + E * E \\ &\Longrightarrow a + \mathbf{E} * E \Longrightarrow a + a * \mathbf{E} \Longrightarrow a + a * a, \end{aligned}$$

and

$$\begin{aligned} &\mathbf{E} \Longrightarrow \mathbf{E} + E \Longrightarrow a + \mathbf{E} \\ &\Longrightarrow a + \mathbf{E} * E \Longrightarrow a + a * \mathbf{E} \Longrightarrow a + a * a. \end{aligned}$$

In the above derivations, the leftmost occurrence of a nonterminal is chosen at each step. Such derivations are called *leftmost derivations*. We could systematically rewrite the rightmost occurrence of a nonterminal, getting *rightmost derivations*. The string $a + a * a$ also has the following two rightmost derivations, where the boldface indicates which occurrence of E is rewritten:

$$\begin{aligned} &\mathbf{E} \Longrightarrow E + \mathbf{E} \Longrightarrow E + E * \mathbf{E} \\ &\Longrightarrow E + \mathbf{E} * a \Longrightarrow \mathbf{E} + a * a \Longrightarrow a + a * a, \end{aligned}$$

and

$$\begin{aligned} &\mathbf{E} \Longrightarrow E * \mathbf{E} \Longrightarrow \mathbf{E} * a \\ &\Longrightarrow E + \mathbf{E} * a \Longrightarrow \mathbf{E} + a * a \Longrightarrow a + a * a. \end{aligned}$$

The language generated by a context-free grammar is defined as follows.

Definition 7.3. Given a context-free grammar $G = (V, \Sigma, P, S)$, the *language generated by* G is the set

$$L(G) = \{w \in \Sigma^* \mid S \overset{+}{\Longrightarrow} w\}.$$

A language $L \subseteq \Sigma^*$ is a *context-free language (for short, CFL)* iff $L = L(G)$ for some context-free grammar G.

It is technically very useful to consider derivations in which the leftmost nonterminal is always selected for rewriting, and dually, derivations in which the rightmost nonterminal is always selected for rewriting.

Definition 7.4. Given a context-free grammar $G = (V, \Sigma, P, S)$, the (one-step) *leftmost derivation relation* $\underset{lm}{\Longrightarrow}$ *associated with* G is the binary relation $\underset{lm}{\Longrightarrow} \subseteq V^* \times V^*$ defined as follows: for all $\alpha, \beta \in V^*$, we have

$$\alpha \underset{lm}{\Longrightarrow} \beta$$

iff there exist $u \in \Sigma^*$, $\rho \in V^*$, and some production $(A \to \gamma) \in P$, such that

$$\alpha = uA\rho \quad \text{and} \quad \beta = u\gamma\rho.$$

The transitive closure of $\underset{lm}{\Longrightarrow}$ is denoted as $\overset{+}{\underset{lm}{\Longrightarrow}}$ and the reflexive and transitive closure of $\underset{lm}{\Longrightarrow}$ is denoted as $\overset{*}{\underset{lm}{\Longrightarrow}}$. The (one-step) *rightmost derivation relation* $\underset{rm}{\Longrightarrow}$ *associated with* G is the binary relation $\underset{rm}{\Longrightarrow} \subseteq V^* \times V^*$ defined as follows: for all $\alpha, \beta \in V^*$, we have

$$\alpha \underset{rm}{\Longrightarrow} \beta$$

iff there exist $\lambda \in V^*$, $v \in \Sigma^*$, and some production $(A \to \gamma) \in P$, such that

$$\alpha = \lambda Av \quad \text{and} \quad \beta = \lambda\gamma v.$$

The transitive closure of $\underset{rm}{\Longrightarrow}$ is denoted as $\overset{+}{\underset{rm}{\Longrightarrow}}$ and the reflexive and transitive closure of $\underset{rm}{\Longrightarrow}$ is denoted as $\overset{*}{\underset{rm}{\Longrightarrow}}$.

Remarks: It is customary to use the symbols a, b, c, d, e for terminal symbols, and the symbols A, B, C, D, E for nonterminal symbols. The symbols u, v, w, x, y, z denote terminal strings, and the symbols $\alpha, \beta, \gamma, \lambda, \rho, \mu$ denote strings in V^*. The symbols X, Y, Z usually denote symbols in V.

Given a context-free grammar $G = (V, \Sigma, P, S)$, *parsing a string* w consists in finding out whether $w \in L(G)$, and if so, in producing a derivation for w. The following proposition is technically very important. It shows that leftmost and rightmost derivations are "universal". This has some important practical implications for the complexity of parsing algorithms.

Proposition 7.1. *Let $G = (V, \Sigma, P, S)$ be a context-free grammar. For every $w \in \Sigma^*$, for every derivation $S \overset{+}{\Longrightarrow} w$, there is a leftmost derivation $S \underset{lm}{\overset{+}{\Longrightarrow}} w$, and there is a rightmost derivation $S \underset{rm}{\overset{+}{\Longrightarrow}} w$.*

Proof. Of course, we have to somehow use induction on derivations, but this is a little tricky, and it is necessary to prove a stronger fact. We treat leftmost derivations, rightmost derivations being handled in a similar way.

Claim: For every $w \in \Sigma^*$, for every $\alpha \in V^+$, for every $n \geq 1$, if $\alpha \overset{n}{\Longrightarrow} w$, then there is a leftmost derivation $\alpha \underset{lm}{\overset{n}{\Longrightarrow}} w$.

The claim is proved by induction on n.

For $n = 1$, there exist some $\lambda, \rho \in V^*$ and some production $A \to \gamma$, such that $\alpha = \lambda A \rho$ and $w = \lambda\gamma\rho$. Since w is a terminal string, λ, ρ, and γ, are terminal strings. Thus, A is the only nonterminal in α, and the derivation step $\alpha \overset{1}{\Longrightarrow} w$ is a leftmost step (and a rightmost step!).

If $n > 1$, then the derivation $\alpha \overset{n}{\Longrightarrow} w$ is of the form

$$\alpha \Longrightarrow \alpha_1 \overset{n-1}{\Longrightarrow} w.$$

There are two subcases.

Case 1. If the derivation step $\alpha \Longrightarrow \alpha_1$ is a leftmost step $\alpha \underset{lm}{\Longrightarrow} \alpha_1$, by the induction hypothesis, there is a leftmost derivation $\alpha_1 \underset{lm}{\overset{n-1}{\Longrightarrow}} w$, and we get the leftmost derivation

$$\alpha \underset{lm}{\Longrightarrow} \alpha_1 \underset{lm}{\overset{n-1}{\Longrightarrow}} w.$$

Case 2. The derivation step $\alpha \Longrightarrow \alpha_1$ is a not a leftmost step. In this case, there must be some $u \in \Sigma^*$, $\mu, \rho \in V^*$, some nonterminals A and B, and some production $B \to \delta$, such that

$$\alpha = uA\mu B\rho \quad \text{and} \quad \alpha_1 = uA\mu\delta\rho,$$

where A is the leftmost nonterminal in α. Since we have a derivation $\alpha_1 \overset{n-1}{\Longrightarrow} w$ of length $n - 1$, by the induction hypothesis, there is a leftmost derivation

$$\alpha_1 \underset{lm}{\overset{n-1}{\Longrightarrow}} w.$$

Since $\alpha_1 = uA\mu\delta\rho$ where A is the leftmost terminal in α_1, the first step in the leftmost derivation $\alpha_1 \underset{lm}{\overset{n-1}{\Longrightarrow}} w$ is of the form

$$uA\mu\delta\rho \underset{lm}{\Longrightarrow} u\gamma\mu\delta\rho,$$

for some production $A \to \gamma$. Thus, we have a derivation of the form

$$\alpha = uA\mu B\rho \Longrightarrow uA\mu\delta\rho \underset{lm}{\Longrightarrow} u\gamma\mu\delta\rho \overset{n-2}{\underset{lm}{\Longrightarrow}} w.$$

We can commute the first two steps involving the productions $B \to \delta$ and $A \to \gamma$, and we get the derivation

$$\alpha = uA\mu B\rho \underset{lm}{\Longrightarrow} u\gamma\mu B\rho \Longrightarrow u\gamma\mu\delta\rho \overset{n-2}{\underset{lm}{\Longrightarrow}} w.$$

This may no longer be a leftmost derivation, but the first step is leftmost, and we are back in case 1. Thus, we conclude by applying the induction hypothesis to the derivation $u\gamma\mu B\rho \overset{n-1}{\Longrightarrow} w$, as in case 1. $\square$

Proposition 7.1 implies that

$$L(G) = \{w \in \Sigma^* \mid S \overset{+}{\underset{lm}{\Longrightarrow}} w\} = \{w \in \Sigma^* \mid S \overset{+}{\underset{rm}{\Longrightarrow}} w\}.$$

We observed that if we consider the grammar $G_2 = (\{E, +, *, (,), a\}, \{+, *, (,), a\}, P, E)$, where P is the set of rules

$$\begin{aligned} E &\longrightarrow E + E,\\ E &\longrightarrow E * E,\\ E &\longrightarrow (E),\\ E &\longrightarrow a, \end{aligned}$$

the string $a + a * a$ has the following two distinct leftmost derivations, where the boldface indicates which occurrence of E is rewritten:

$$\begin{aligned} &\mathbf{E} \Longrightarrow \mathbf{E} * E \Longrightarrow \mathbf{E} + E * E\\ &\Longrightarrow a + \mathbf{E} * E \Longrightarrow a + a * \mathbf{E} \Longrightarrow a + a * a, \end{aligned}$$

and

$$\begin{aligned} &\mathbf{E} \Longrightarrow \mathbf{E} + E \Longrightarrow a + \mathbf{E}\\ &\Longrightarrow a + \mathbf{E} * E \Longrightarrow a + a * \mathbf{E} \Longrightarrow a + a * a. \end{aligned}$$

When this happens, we say that we have an ambiguous grammars. In some cases, it is possible to modify a grammar to make it unambiguous. For example, the grammar G_2 can be modified as follows.

Let $G_3 = (\{E, T, F, +, *, (,), a\}, \{+, *, (,), a\}, P, E)$, where P is the set of rules

$$\begin{aligned} E &\longrightarrow E + T,\\ E &\longrightarrow T,\\ T &\longrightarrow T * F,\\ T &\longrightarrow F,\\ F &\longrightarrow (E),\\ F &\longrightarrow a. \end{aligned}$$

We leave as an exercise to show that $L(G_3) = L(G_2)$, and that every string in $L(G_3)$ has a unique leftmost derivation. Unfortunately, it is not always possible to modify a context-free grammar to make it unambiguous. There exist context-free languages that have no unambiguous context-free grammars. For example, the language

$$L_3 = \{a^m b^m c^n \mid m, n \geq 1\} \cup \{a^m b^n c^n \mid m, n \geq 1\}$$

is context-free, since it is generated by the following context-free grammar:

$$\begin{aligned}
S &\to S_1, \\
S &\to S_2, \\
S_1 &\to XC, \\
S_2 &\to AY, \\
X &\to aXb, \\
X &\to ab, \\
Y &\to bYc, \\
Y &\to bc, \\
A &\to aA, \\
A &\to a, \\
C &\to cC, \\
C &\to c.
\end{aligned}$$

However, it can be shown that L_3 has no unambiguous grammars. All this motivates the following definition.

Definition 7.5. A context-free grammar $G = (V, \Sigma, P, S)$ is *ambiguous* if there is some string $w \in L(G)$ that has two distinct leftmost derivations (or two distinct rightmost derivations). Thus, a grammar G is *unambiguous* if every string $w \in L(G)$ has a unique leftmost derivation (or a unique rightmost derivation). A context-free language L is *inherently ambiguous* if every CFG G for L is ambiguous.

Whether or not a grammar is ambiguous affects the complexity of parsing. Parsing algorithms for unambiguous grammars are more efficient than parsing algorithms for ambiguous grammars.

We now consider various normal forms for context-free grammars.

7.3 Normal Forms for Context-Free Grammars, Chomsky Normal Form

One of the main goals of this section is to show that every CFG G can be converted to an equivalent grammar in *Chomsky Normal Form (for short, CNF)*. A context-free grammar

$G = (V, \Sigma, P, S)$ is in Chomsky Normal Form iff its productions are of the form

$$\begin{aligned} &A \to BC, \\ &A \to a, \quad \text{or} \\ &S \to \epsilon, \end{aligned}$$

where $A, B, C \in N$, $a \in \Sigma$, $S \to \epsilon$ is in P iff $\epsilon \in L(G)$, and S does not occur on the right-hand side of any production.

Note that a grammar in Chomsky Normal Form does not have ϵ-rules, i.e., rules of the form $A \to \epsilon$, except when $\epsilon \in L(G)$, in which case $S \to \epsilon$ is the only ϵ-rule. It also does not have *chain rules*, i.e., rules of the form $A \to B$, where $A, B \in N$. Thus, in order to convert a grammar to Chomsky Normal Form, we need to show how to eliminate ϵ-rules and chain rules. This is not the end of the story, since we may still have rules of the form $A \to \alpha$ where either $|\alpha| \geq 3$ or $|\alpha| \geq 2$ and α contains terminals. However, dealing with such rules is a simple recoding matter, and we first focus on the elimination of ϵ-rules and chain rules. It turns out that ϵ-rules must be eliminated first.

The first step to eliminate ϵ-rules is to compute the set $E(G)$ of *erasable (or nullable) nonterminals*

$$E(G) = \{A \in N \mid A \overset{+}{\Longrightarrow} \epsilon\}.$$

The set $E(G)$ is computed using a sequence of approximations E_i defined as follows:

$$\begin{aligned} E_0 &= \{A \in N \mid (A \to \epsilon) \in P\}, \\ E_{i+1} &= E_i \cup \{A \mid \exists (A \to B_1 \ldots B_j \ldots B_k) \in P,\ B_j \in E_i,\ 1 \leq j \leq k\}. \end{aligned}$$

Clearly, the E_i form an ascending chain

$$E_0 \subseteq E_1 \subseteq \cdots \subseteq E_i \subseteq E_{i+1} \subseteq \cdots \subseteq N,$$

and since N is finite, there is a least i, say i_0, such that $E_{i_0} = E_{i_0+1}$. We claim that $E(G) = E_{i_0}$. Actually, we prove the following proposition.

Proposition 7.2. *Given a context-free grammar $G = (V, \Sigma, P, S)$, one can construct a context-free grammar $G' = (V', \Sigma, P', S')$ such that:*

(1) $L(G') = L(G)$;

(2) P' contains no ϵ-rules other than $S' \to \epsilon$, and $S' \to \epsilon \in P'$ iff $\epsilon \in L(G)$;

(3) S' does not occur on the right-hand side of any production in P'.

Proof. We begin by proving that $E(G) = E_{i_0}$. For this, we prove that $E(G) \subseteq E_{i_0}$ and $E_{i_0} \subseteq E(G)$.

To prove that $E_{i_0} \subseteq E(G)$, we proceed by induction on i. Since $E_0 = \{A \in N \mid (A \to \epsilon) \in P\}$, we have $A \overset{1}{\Longrightarrow} \epsilon$, and thus $A \in E(G)$. By the induction hypothesis, $E_i \subseteq$

$E(G)$. If $A \in E_{i+1}$, either $A \in E_i$ and then $A \in E(G)$, or there is some production $(A \rightarrow B_1 \ldots B_j \ldots B_k) \in P$, such that $B_j \in E_i$ for all j, $1 \leq j \leq k$. By the induction hypothesis, $B_j \overset{+}{\Longrightarrow} \epsilon$ for each j, $1 \leq j \leq k$, and thus

$$A \Longrightarrow B_1 \ldots B_j \ldots B_k \overset{+}{\Longrightarrow} B_2 \ldots B_j \ldots B_k \overset{+}{\Longrightarrow} B_j \ldots B_k \overset{+}{\Longrightarrow} \epsilon,$$

which shows that $A \in E(G)$.

To prove that $E(G) \subseteq E_{i_0}$, we also proceed by induction, but on the length of a derivation $A \overset{+}{\Longrightarrow} \epsilon$. If $A \overset{1}{\Longrightarrow} \epsilon$, then $A \rightarrow \epsilon \in P$, and thus $A \in E_0$ since $E_0 = \{A \in N \mid (A \rightarrow \epsilon) \in P\}$. If $A \overset{n+1}{\Longrightarrow} \epsilon$, then

$$A \Longrightarrow \alpha \overset{n}{\Longrightarrow} \epsilon,$$

for some production $A \rightarrow \alpha \in P$. If α contains terminals of nonterminals not in $E(G)$, it is impossible to derive ϵ from α, and thus, we must have $\alpha = B_1 \ldots B_j \ldots B_k$, with $B_j \in E(G)$, for all j, $1 \leq j \leq k$. However, $B_j \overset{n_j}{\Longrightarrow} \epsilon$ where $n_j \leq n$, and by the induction hypothesis, $B_j \in E_{i_0}$. But then, we get $A \in E_{i_0+1} = E_{i_0}$, as desired. $\square$

Having shown that $E(G) = E_{i_0}$, we construct the grammar G'. Its set of production P' is defined as follows. First, we create the production $S' \rightarrow S$ where $S' \notin V$, to make sure that S' does not occur on the right-hand side of any rule in P'. Let

$$P_1 = \{A \rightarrow \alpha \in P \mid \alpha \in V^+\} \cup \{S' \rightarrow S\},$$

and let P_2 be the set of productions

$$P_2 = \{A \rightarrow \alpha_1\alpha_2 \ldots \alpha_k\alpha_{k+1} \mid \exists\alpha_1 \in V^*, \ldots, \exists\alpha_{k+1} \in V^*,\ \exists B_1 \in E(G), \ldots, \exists B_k \in E(G)$$
$$A \rightarrow \alpha_1 B_1 \alpha_2 \ldots \alpha_k B_k \alpha_{k+1} \in P,\ k \geq 1,\ \alpha_1 \ldots \alpha_{k+1} \neq \epsilon\}.$$

Note that $\epsilon \in L(G)$ iff $S \in E(G)$. If $S \notin E(G)$, then let $P' = P_1 \cup P_2$, and if $S \in E(G)$, then let $P' = P_1 \cup P_2 \cup \{S' \rightarrow \epsilon\}$. We claim that $L(G') = L(G)$, which is proved by showing that every derivation using G can be simulated by a derivation using G', and vice-versa. All the conditions of the proposition are now met. $\square$

From a practical point of view, the construction or Proposition 7.2 is very costly. For example, given a grammar containing the productions

$$\begin{aligned}
&S \rightarrow ABCDEF,\\
&A \rightarrow \epsilon,\\
&B \rightarrow \epsilon,\\
&C \rightarrow \epsilon,\\
&D \rightarrow \epsilon,\\
&E \rightarrow \epsilon,\\
&F \rightarrow \epsilon,\\
&\ldots \rightarrow \ldots,
\end{aligned}$$

eliminating ϵ-rules will create $2^6 - 1 = 63$ new rules corresponding to the 63 nonempty subsets of the set $\{A, B, C, D, E, F\}$. We now turn to the elimination of chain rules.

It turns out that matters are greatly simplified if we first apply Proposition 7.2 to the input grammar G, and we explain the construction assuming that $G = (V, \Sigma, P, S)$ satisfies the conditions of Proposition 7.2. For every nonterminal $A \in N$, we define the set

$$I_A = \{B \in N \mid A \overset{+}{\Longrightarrow} B\}.$$

The sets I_A are computed using approximations $I_{A,i}$ defined as follows:

$$\begin{aligned} I_{A,0} &= \{B \in N \mid (A \to B) \in P\}, \\ I_{A,i+1} &= I_{A,i} \cup \{C \in N \mid \exists (B \to C) \in P, \text{ and } B \in I_{A,i}\}. \end{aligned}$$

Clearly, for every $A \in N$, the $I_{A,i}$ form an ascending chain

$$I_{A,0} \subseteq I_{A,1} \subseteq \cdots \subseteq I_{A,i} \subseteq I_{A,i+1} \subseteq \cdots \subseteq N,$$

and since N is finite, there is a least i, say i_0, such that $I_{A,i_0} = I_{A,i_0+1}$. We claim that $I_A = I_{A,i_0}$. Actually, we prove the following proposition.

Proposition 7.3. *Given a context-free grammar $G = (V, \Sigma, P, S)$, one can construct a context-free grammar $G' = (V', \Sigma, P', S')$ such that:*

(1) $L(G') = L(G)$;

(2) Every rule in P' is of the form $A \to \alpha$ where $|\alpha| \geq 2$, or $A \to a$ where $a \in \Sigma$, or $S' \to \epsilon$ iff $\epsilon \in L(G)$;

(3) S' does not occur on the right-hand side of any production in P'.

Proof. First, we apply Proposition 7.2 to the grammar G, obtaining a grammar $G_1 = (V_1, \Sigma, S_1, P_1)$. The proof that $I_A = I_{A,i_0}$ is similar to the proof that $E(G) = E_{i_0}$. First, we prove that $I_{A,i} \subseteq I_A$ by induction on i. This is staightforward. Next, we prove that $I_A \subseteq I_{A,i_0}$ by induction on derivations of the form $A \overset{+}{\Longrightarrow} B$. In this part of the proof, we use the fact that G_1 has no ϵ-rules except perhaps $S_1 \to \epsilon$, and that S_1 does not occur on the right-hand side of any rule. This implies that a derivation $A \overset{n+1}{\Longrightarrow} C$ is necessarily of the form $A \overset{n}{\Longrightarrow} B \Longrightarrow C$ for some $B \in N$. Then, in the induction step, we have $B \in I_{A,i_0}$, and thus $C \in I_{A,i_0+1} = I_{A,i_0}$.

We now define the following sets of rules. Let

$$P_2 = P_1 - \{A \to B \mid A \to B \in P_1\},$$

and let

$$P_3 = \{A \to \alpha \mid B \to \alpha \in P_1,\ \alpha \notin N_1,\ B \in I_A\}.$$

We claim that $G' = (V_1, \Sigma, P_2 \cup P_3, S_1)$ satisfies the conditions of the proposition. For example, S_1 does not appear on the right-hand side of any production, since the productions in P_3 have right-hand sides from P_1, and S_1 does not appear on the right-hand side in P_1. It is also easily shown that $L(G') = L(G_1) = L(G)$. ☐

Let us apply the method of Proposition 7.3 to the grammar

$$G_3 = (\{E, T, F, +, *, (,), a\}, \{+, *, (,), a\}, P, E),$$

where P is the set of rules

$$\begin{aligned}
E &\longrightarrow E + T,\\
E &\longrightarrow T,\\
T &\longrightarrow T * F,\\
T &\longrightarrow F,\\
F &\longrightarrow (E),\\
F &\longrightarrow a.
\end{aligned}$$

We get $I_E = \{T, F\}$, $I_T = \{F\}$, and $I_F = \emptyset$. The new grammar G'_3 has the set of rules

$$\begin{aligned}
E &\longrightarrow E + T,\\
E &\longrightarrow T * F,\\
E &\longrightarrow (E),\\
E &\longrightarrow a,\\
T &\longrightarrow T * F,\\
T &\longrightarrow (E),\\
T &\longrightarrow a,\\
F &\longrightarrow (E),\\
F &\longrightarrow a.
\end{aligned}$$

At this stage, the grammar obtained in Proposition 7.3 no longer has ϵ-rules (except perhaps $S' \to \epsilon$ iff $\epsilon \in L(G)$) or chain rules. However, it may contain rules $A \to \alpha$ with $|\alpha| \geq 3$, or with $|\alpha| \geq 2$ and where α contains terminals(s). To obtain the Chomsky Normal Form. we need to eliminate such rules. This is not difficult, but notationally a bit messy.

Proposition 7.4. *Given a context-free grammar $G = (V, \Sigma, P, S)$, one can construct a context-free grammar $G' = (V', \Sigma, P', S')$ such that $L(G') = L(G)$ and G' is in Chomsky Normal Form, that is, a grammar whose productions are of the form*

$$\begin{aligned}
&A \to BC,\\
&A \to a, \quad \text{or}\\
&S' \to \epsilon,
\end{aligned}$$

where $A, B, C \in N'$, $a \in \Sigma$, $S' \to \epsilon$ is in P' iff $\epsilon \in L(G)$, and S' does not occur on the right-hand side of any production in P'.

Proof. First, we apply Proposition 7.3, obtaining G_1. Let Σ_r be the set of terminals occurring on the right-hand side of rules $A \to \alpha \in P_1$, with $|\alpha| \geq 2$. For every $a \in \Sigma_r$, let X_a be a new nonterminal not in V_1. Let

$$P_2 = \{X_a \to a \mid a \in \Sigma_r\}.$$

Let $P_{1,r}$ be the set of productions

$$A \to \alpha_1 a_1 \alpha_2 \cdots \alpha_k a_k \alpha_{k+1},$$

where $a_1, \ldots, a_k \in \Sigma_r$ and $\alpha_i \in N_1^*$. For every production

$$A \to \alpha_1 a_1 \alpha_2 \cdots \alpha_k a_k \alpha_{k+1}$$

in $P_{1,r}$, let

$$A \to \alpha_1 X_{a_1} \alpha_2 \cdots \alpha_k X_{a_k} \alpha_{k+1}$$

be a new production, and let P_3 be the set of all such productions. Let $P_4 = (P_1 - P_{1,r}) \cup P_2 \cup P_3$. Now, productions $A \to \alpha$ in P_4 with $|\alpha| \geq 2$ do not contain terminals. However, we may still have productions $A \to \alpha \in P_4$ with $|\alpha| \geq 3$. We can perform some recoding using some new nonterminals. For every production of the form

$$A \to B_1 \cdots B_k,$$

where $k \geq 3$, create the new nonterminals

$$[B_1 \cdots B_{k-1}], [B_1 \cdots B_{k-2}], \cdots, [B_1 B_2 B_3], [B_1 B_2],$$

and the new productions

$$\begin{aligned}
A &\to [B_1 \cdots B_{k-1}] B_k, \\
[B_1 \cdots B_{k-1}] &\to [B_1 \cdots B_{k-2}] B_{k-1}, \\
\cdots &\to \cdots, \\
[B_1 B_2 B_3] &\to [B_1 B_2] B_3, \\
[B_1 B_2] &\to B_1 B_2.
\end{aligned}$$

All the productions are now in Chomsky Normal Form, and it is clear that the same language is generated. ☐

Applying the first phase of the method of Proposition 7.4 to the grammar G_3', we get the rules

$$
\begin{aligned}
E &\longrightarrow EX_+T,\\
E &\longrightarrow TX_*F,\\
E &\longrightarrow X_(EX_),\\
E &\longrightarrow a,\\
T &\longrightarrow TX_*F,\\
T &\longrightarrow X_(EX_),\\
T &\longrightarrow a,\\
F &\longrightarrow X_(EX_),\\
F &\longrightarrow a,\\
X_+ &\longrightarrow +,\\
X_* &\longrightarrow *,\\
X_(&\longrightarrow (,\\
X_) &\longrightarrow).
\end{aligned}
$$

After applying the second phase of the method, we get the following grammar in Chomsky Normal Form:

$$
\begin{aligned}
E &\longrightarrow [EX_+]T,\\
[EX_+] &\longrightarrow EX_+,\\
E &\longrightarrow [TX_*]F,\\
[TX_*] &\longrightarrow TX_*,\\
E &\longrightarrow [X_(E]X_),\\
[X_(E] &\longrightarrow X_(E,\\
E &\longrightarrow a,\\
T &\longrightarrow [TX_*]F,\\
T &\longrightarrow [X_(E]X_),\\
T &\longrightarrow a,\\
F &\longrightarrow [X_(E]X_),\\
F &\longrightarrow a,\\
X_+ &\longrightarrow +,\\
X_* &\longrightarrow *,\\
X_(&\longrightarrow (,\\
X_) &\longrightarrow).
\end{aligned}
$$

For large grammars, it is often convenient to use the abbreviation which consists in grouping productions having a common left-hand side, and listing the right-hand sides separated by the symbol $|$. Thus, a group of productions

$$\begin{aligned} A &\rightarrow \alpha_1, \\ A &\rightarrow \alpha_2, \\ \cdots &\rightarrow \cdots, \\ A &\rightarrow \alpha_k, \end{aligned}$$

may be abbreviated as

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_k.$$

An interesting corollary of the CNF is the following decidability result. There is an algorithm which, given a context-free grammar G, given any string $w \in \Sigma^*$, decides whether $w \in L(G)$. Indeed, we first convert G to a grammar G' in Chomsky Normal Form. If $w = \epsilon$, we can test whether $\epsilon \in L(G)$, since this is the case iff $S' \rightarrow \epsilon \in P'$. If $w \neq \epsilon$, letting $n = |w|$, note that since the rules are of the form $A \rightarrow BC$ or $A \rightarrow a$, where $a \in \Sigma$, any derivation for w has $n - 1 + n = 2n - 1$ steps. Thus, we enumerate all (leftmost) derivations of length $2n - 1$.

There are much better parsing algorithms than this naive algorithm. We now show that every regular language is context-free.

7.4 Regular Languages are Context-Free

The regular languages can be characterized in terms of very special kinds of context-free grammars, right-linear (and left-linear) context-free grammars.

Definition 7.6. A context-free grammar $G = (V, \Sigma, P, S)$ is *left-linear* iff its productions are of the form

$$\begin{aligned} A &\rightarrow Ba, \\ A &\rightarrow a, \\ A &\rightarrow \epsilon. \end{aligned}$$

where $A, B \in N$, and $a \in \Sigma$. A context-free grammar $G = (V, \Sigma, P, S)$ is *right-linear* iff its productions are of the form

$$\begin{aligned} A &\rightarrow aB, \\ A &\rightarrow a, \\ A &\rightarrow \epsilon. \end{aligned}$$

where $A, B \in N$, and $a \in \Sigma$.

The following proposition shows the equivalence between NFA's and right-linear grammars.

Proposition 7.5. *A language L is regular if and only if it is generated by some right-linear grammar.*

Proof. Let $L = L(D)$ for some DFA $D = (Q, \Sigma, \delta, q_0, F)$. We construct a right-linear grammar G as follows. Let $V = Q \cup \Sigma$, $S = q_0$, and let P be defined as follows:

$$P = \{p \to aq \mid q = \delta(p, a),\ p, q \in Q,\ a \in \Sigma\} \cup \{p \to \epsilon \mid p \in F\}.$$

It is easily shown by induction on the length of w that

$$p \overset{*}{\Longrightarrow} wq \quad \text{iff} \quad q = \delta^*(p, w),$$

and thus, $L(D) = L(G)$.

Conversely, let $G = (V, \Sigma, P, S)$ be a right-linear grammar. First, let $G = (V', \Sigma, P', S)$ be the right-linear grammar obtained from G by adding the new nonterminal E to N, replacing every rule in P of the form $A \to a$ where $a \in \Sigma$ by the rule $A \to aE$, and adding the rule $E \to \epsilon$. It is immediately verified that $L(G') = L(G)$. Next, we construct the NFA $M = (Q, \Sigma, \delta, q_0, F)$ as follows: $Q = N' = N \cup \{E\}$, $q_0 = S$, $F = \{A \in N' \mid A \to \epsilon\}$, and

$$\delta(A, a) = \{B \in N' \mid A \to aB \in P'\},$$

for all $A \in N$ and all $a \in \Sigma$. It is easily shown by induction on the length of w that

$$A \overset{*}{\Longrightarrow} wB \quad \text{iff} \quad B \in \delta^*(A, w),$$

and thus, $L(M) = L(G') = L(G)$. □

A similar proposition holds for left-linear grammars. It is also easily shown that the regular languages are exactly the languages generated by context-free grammars whose rules are of the form

$$\begin{aligned} A &\to Bu, \\ A &\to u, \end{aligned}$$

where $A, B \in N$, and $u \in \Sigma^*$.

7.5 Useless Productions in Context-Free Grammars

Given a context-free grammar $G = (V, \Sigma, P, S)$, it may contain rules that are useless for a number of reasons. For example, consider the grammar $G_3 = (\{E, A, a, b\}, \{a, b\}, P, E)$, where P is the set of rules

$$\begin{aligned} E &\longrightarrow aEb, \\ E &\longrightarrow ab, \\ E &\longrightarrow A, \\ A &\longrightarrow bAa. \end{aligned}$$

The problem is that the nonterminal A does not derive any terminal strings, and thus, it is useless, as well as the last two productions. Let us now consider the grammar $G_4 = (\{E, A, a, b, c, d\}, \{a, b, c, d\}, P, E)$, where P is the set of rules

$$\begin{aligned}
E &\longrightarrow aEb,\\
E &\longrightarrow ab,\\
A &\longrightarrow cAd,\\
A &\longrightarrow cd.
\end{aligned}$$

This time, the nonterminal A generates strings of the form $c^n d^n$, but there is no derivation $E \stackrel{+}{\Longrightarrow} \alpha$ from E where A occurs in α. The nonterminal A is not connected to E, and the last two rules are useless. Fortunately, it is possible to find such useless rules, and to eliminate them.

Let $T(G)$ be the set of nonterminals that actually derive some terminal string, i.e.

$$T(G) = \{A \in (V - \Sigma) \mid \exists w \in \Sigma^*,\ A \Longrightarrow^+ w\}.$$

The set $T(G)$ can be defined by stages. We define the sets T_n $(n \geq 1)$ as follows:

$$T_1 = \{A \in (V - \Sigma) \mid \exists (A \longrightarrow w) \in P,\ \text{with } w \in \Sigma^*\},$$

and

$$T_{n+1} = T_n \cup \{A \in (V - \Sigma) \mid \exists (A \longrightarrow \beta) \in P,\ \text{with } \beta \in (T_n \cup \Sigma)^*\}.$$

It is easy to prove that there is some least n such that $T_{n+1} = T_n$, and that for this n, $T(G) = T_n$.

If $S \notin T(G)$, then $L(G) = \emptyset$, and G is equivalent to the trivial grammar

$$G' = (\{S\}, \Sigma, \emptyset, S).$$

If $S \in T(G)$, then let $U(G)$ be the set of nonterminals that are actually useful, i.e.,

$$U(G) = \{A \in T(G) \mid \exists \alpha, \beta \in (T(G) \cup \Sigma)^*,\ S \Longrightarrow^* \alpha A \beta\}.$$

The set $U(G)$ can also be computed by stages. We define the sets U_n $(n \geq 1)$ as follows:

$$U_1 = \{A \in T(G) \mid \exists (S \longrightarrow \alpha A \beta) \in P,\ \text{with } \alpha, \beta \in (T(G) \cup \Sigma)^*\},$$

and

$$U_{n+1} = U_n \cup \{B \in T(G) \mid \exists (A \longrightarrow \alpha B \beta) \in P,\ \text{with } A \in U_n,\ \alpha, \beta \in (T(G) \cup \Sigma)^*\}.$$

It is easy to prove that there is some least n such that $U_{n+1} = U_n$, and that for this n, $U(G) = U_n \cup \{S\}$. Then, we can use $U(G)$ to transform G into an equivalent CFG in which every nonterminal is useful (i.e., for which $V - \Sigma = U(G)$). Indeed, simply delete all rules containing symbols not in $U(G)$. The details are left as an exercise. We say that a context-free grammar G is *reduced* if all its nonterminals are useful, i.e., $N = U(G)$.

It should be noted than although dull, the above considerations are important in practice. Certain algorithms for constructing parsers, for example, LR-parsers, may loop if useless rules are not eliminated!

We now consider another normal form for context-free grammars, the Greibach Normal Form.

7.6 The Greibach Normal Form

Every CFG G can also be converted to an equivalent grammar in *Greibach Normal Form (for short, GNF)*. A context-free grammar $G = (V, \Sigma, P, S)$ is in Greibach Normal Form iff its productions are of the form

$$\begin{aligned} &A \to aBC, \\ &A \to aB, \\ &A \to a, \quad \text{or} \\ &S \to \epsilon, \end{aligned}$$

where $A, B, C \in N$, $a \in \Sigma$, $S \to \epsilon$ is in P iff $\epsilon \in L(G)$, and S does not occur on the right-hand side of any production.

Note that a grammar in Greibach Normal Form does not have ϵ-rules other than possibly $S \to \epsilon$. More importantly, except for the special rule $S \to \epsilon$, every rule produces some terminal symbol.

An important consequence of the Greibach Normal Form is that every nonterminal is not left recursive. A nonterminal A is *left recursive* iff $A \overset{+}{\Longrightarrow} A\alpha$ for some $\alpha \in V^*$. Left recursive nonterminals cause top-down determinitic parsers to loop. The Greibach Normal Form provides a way of avoiding this problem.

There are no easy proofs that every CFG can be converted to a Greibach Normal Form. A particularly elegant method due to Rosenkrantz using least fixed-points and matrices will be given in section 7.9.

Proposition 7.6. *Given a context-free grammar $G = (V, \Sigma, P, S)$, one can construct a context-free grammar $G' = (V', \Sigma, P', S')$ such that $L(G') = L(G)$ and G' is in Greibach Normal Form, that is, a grammar whose productions are of the form*

$$\begin{aligned} &A \to aBC, \\ &A \to aB, \\ &A \to a, \quad \textit{or} \\ &S' \to \epsilon, \end{aligned}$$

where $A, B, C \in N'$, $a \in \Sigma$, $S' \to \epsilon$ is in P' iff $\epsilon \in L(G)$, and S' does not occur on the right-hand side of any production in P'.

7.7 Least Fixed-Points

Context-free languages can also be characterized as least fixed-points of certain functions induced by grammars. This characterization yields a rather quick proof that every context-free grammar can be converted to Greibach Normal Form. This characterization also reveals very clearly the recursive nature of the context-free languages.

We begin by reviewing what we need from the theory of partially ordered sets.

Definition 7.7. Given a partially ordered set $\langle A, \leq\rangle$, an *ω-chain* $(a_n)_{n\geq 0}$ is a sequence such that $a_n \leq a_{n+1}$ for all $n \geq 0$. The *least-upper bound* of an ω-chain (a_n) is an element $a \in A$ such that:

(1) $a_n \leq a$, for all $n \geq 0$;

(2) For any $b \in A$, if $a_n \leq b$, for all $n \geq 0$, then $a \leq b$.

A partially ordered set $\langle A, \leq\rangle$ is an *ω-chain complete poset* iff it has a least element $\perp$, and iff every ω-chain has a least upper bound denoted as $\bigsqcup a_n$.

Remark: The ω in ω-chain means that we are considering countable chains (ω is the ordinal associated with the order-type of the set of natural numbers). This notation may seem arcane, but is standard in denotational semantics.

For example, given any set X, the power set 2^X ordered by inclusion is an ω-chain complete poset with least element $\emptyset$. The Cartesian product $\underbrace{2^X \times \cdots \times 2^X}_{n}$ ordered such that

$$(A_1, \ldots, A_n) \leq (B_1, \ldots, B_n)$$

iff $A_i \subseteq B_i$ (where $A_i, B_i \in 2^X$) is an ω-chain complete poset with least element $(\emptyset, \ldots, \emptyset)$.

We are interested in functions between partially ordered sets.

Definition 7.8. Given any two partially ordered sets $\langle A_1, \leq_1\rangle$ and $\langle A_2, \leq_2\rangle$, a function $f\colon A_1 \to A_2$ is *monotonic* iff for all $x, y \in A_1$,

$$x \leq_1 y \quad \text{implies that} \quad f(x) \leq_2 f(y).$$

If $\langle A_1, \leq_1\rangle$ and $\langle A_2, \leq_2\rangle$ are ω-chain complete posets, a function $f\colon A_1 \to A_2$ is *ω-continuous* iff it is monotonic, and for every ω-chain (a_n),

$$f(\bigsqcup a_n) = \bigsqcup f(a_n).$$

Remark: Note that we are not requiring that an ω-continuous function $f\colon A_1 \to A_2$ preserve least elements, i.e., it is possible that $f(\perp_1) \neq \perp_2$.

We now define the crucial concept of a least fixed-point.

Definition 7.9. Let $\langle A, \leq\rangle$ be a partially ordered set, and let $f\colon A \to A$ be a function. A *fixed-point of* f is an element $a \in A$ such that $f(a) = a$. The *least fixed-point of* f is an element $a \in A$ such that $f(a) = a$, and for every $b \in A$ such that $f(b) = b$, then $a \leq b$.

The following proposition gives sufficient conditions for the existence of least fixed-points. It is one of the key propositions in denotational semantics.

Proposition 7.7. *Let $\langle A, \leq\rangle$ be an ω-chain complete poset with least element $\perp$. Every ω-continuous function $f\colon A \to A$ has a unique least fixed-point x_0 given by*

$$x_0 = \bigsqcup f^n(\perp).$$

Furthermore, for any $b \in A$ such that $f(b) \leq b$, then $x_0 \leq b$.

Proof. First, we prove that the sequence

$$\perp, f(\perp), f^2(\perp), \ldots, f^n(\perp), \ldots$$

is an ω-chain. This is shown by induction on n. Since $\perp$ is the least element of A, we have $\perp \leq f(\perp)$. Assuming by induction that $f^n(\perp) \leq f^{n+1}(\perp)$, since f is ω-continuous, it is monotonic, and thus we get $f^{n+1}(\perp) \leq f^{n+2}(\perp)$, as desired.

Since A is an ω-chain complete poset, the ω-chain $(f^n(\perp))$ has a least upper bound

$$x_0 = \bigsqcup f^n(\perp).$$

Since f is ω-continuous, we have

$$f(x_0) = f(\bigsqcup f^n(\perp)) = \bigsqcup f(f^n(\perp)) = \bigsqcup f^{n+1}(\perp) = x_0,$$

and x_0 is indeed a fixed-point of f.

Clearly, if $f(b) \leq b$ implies that $x_0 \leq b$, then $f(b) = b$ implies that $x_0 \leq b$. Thus, assume that $f(b) \leq b$ for some $b \in A$. We prove by induction of n that $f^n(\perp) \leq b$. Indeed, $\perp \leq b$, since $\perp$ is the least element of A. Assuming by induction that $f^n(\perp) \leq b$, by monotonicity of f, we get

$$f(f^n(\perp)) \leq f(b),$$

and since $f(b) \leq b$, this yields

$$f^{n+1}(\perp) \leq b.$$

Since $f^n(\perp) \leq b$ for all $n \geq 0$, we have

$$x_0 = \bigsqcup f^n(\perp) \leq b.$$

□

The second part of Proposition 7.7 is very useful to prove that functions have the same least fixed-point. For example, under the conditions of Proposition 7.7, if $g\colon A \to A$ is another ω-chain continuous function, letting x_0 be the least fixed-point of f and y_0 be the least fixed-point of g, if $f(y_0) \leq y_0$ and $g(x_0) \leq x_0$, we can deduce that $x_0 = y_0$. Indeed, since $f(y_0) \leq y_0$ and x_0 is the least fixed-point of f, we get $x_0 \leq y_0$, and since $g(x_0) \leq x_0$ and y_0 is the least fixed-point of g, we get $y_0 \leq x_0$, and therefore $x_0 = y_0$.

Proposition 7.7 also shows that the least fixed-point x_0 of f can be approximated as much as desired, using the sequence $(f^n(\perp))$. We will now apply this fact to context-free grammars. For this, we need to show how a context-free grammar $G = (V, \Sigma, P, S)$ with m nonterminals induces an ω-continuous map

$$\Phi_G\colon \underbrace{2^{\Sigma^*} \times \cdots \times 2^{\Sigma^*}}_{m} \to \underbrace{2^{\Sigma^*} \times \cdots \times 2^{\Sigma^*}}_{m}.$$

7.8 Context-Free Languages as Least Fixed-Points

Given a context-free grammar $G = (V, \Sigma, P, S)$ with m nonterminals $A_1, \ldots A_m$, grouping all the productions having the same left-hand side, the grammar G can be concisely written as

$$\begin{aligned}
A_1 &\to \alpha_{1,1} + \cdots + \alpha_{1,n_1},\\
\cdots &\to \cdots\\
A_i &\to \alpha_{i,1} + \cdots + \alpha_{i,n_i},\\
\cdots &\to \cdots\\
A_m &\to \alpha_{m,1} + \cdots + \alpha_{m,n_n}.
\end{aligned}$$

Given any set A, let $\mathcal{P}_{fin}(A)$ be the set of finite subsets of A.

Definition 7.10. Let $G = (V, \Sigma, P, S)$ be a context-free grammar with m nonterminals A_1, ..., A_m. For any m-tuple $\Lambda = (L_1, \ldots, L_m)$ of languages $L_i \subseteq \Sigma^*$, we define the function

$$\Phi[\Lambda]\colon \mathcal{P}_{fin}(V^*) \to 2^{\Sigma^*}$$

inductively as follows:

$$\begin{aligned}
\Phi[\Lambda](\emptyset) &= \emptyset,\\
\Phi[\Lambda](\{\epsilon\}) &= \{\epsilon\},\\
\Phi[\Lambda](\{a\}) &= \{a\}, \quad \text{if } a \in \Sigma,\\
\Phi[\Lambda](\{A_i\}) &= L_i, \quad \text{if } A_i \in N,\\
\Phi[\Lambda](\{\alpha X\}) &= \Phi[\Lambda](\{\alpha\})\Phi[\Lambda](\{X\}), \quad \text{if } \alpha \in V^+,\, X \in V,\\
\Phi[\Lambda](Q \cup \{\alpha\}) &= \Phi[\Lambda](Q) \cup \Phi[\Lambda](\{\alpha\}), \quad \text{if } Q \in \mathcal{P}_{fin}(V^*),\, Q \neq \emptyset,\, \alpha \in V^*,\, \alpha \notin Q.
\end{aligned}$$

Then, writing the grammar G as

$$\begin{aligned} A_1 &\to \alpha_{1,1} + \cdots + \alpha_{1,n_1},\\ \cdots &\to \cdots\\ A_i &\to \alpha_{i,1} + \cdots + \alpha_{i,n_i},\\ \cdots &\to \cdots\\ A_m &\to \alpha_{m,1} + \cdots + \alpha_{m,n_n},\end{aligned}$$

we define the map

$$\Phi_G \colon \underbrace{2^{\Sigma^*} \times \cdots \times 2^{\Sigma^*}}_{m} \to \underbrace{2^{\Sigma^*} \times \cdots \times 2^{\Sigma^*}}_{m}$$

such that

$$\Phi_G(L_1, \ldots L_m) = (\Phi[\Lambda](\{\alpha_{1,1}, \ldots, \alpha_{1,n_1}\}), \ldots, \Phi[\Lambda](\{\alpha_{m,1}, \ldots, \alpha_{m,n_m}\}))$$

for all $\Lambda = (L_1, \ldots, L_m) \in \underbrace{2^{\Sigma^*} \times \cdots \times 2^{\Sigma^*}}_{m}$.

One should verify that the map $\Phi[\Lambda]$ is well defined, but this is easy. The following proposition is easily shown:

Proposition 7.8. *Given a context-free grammar $G = (V, \Sigma, P, S)$ with m nonterminals A_1, …, A_m, the map*

$$\Phi_G \colon \underbrace{2^{\Sigma^*} \times \cdots \times 2^{\Sigma^*}}_{m} \to \underbrace{2^{\Sigma^*} \times \cdots \times 2^{\Sigma^*}}_{m}$$

is ω-continuous.

Now, $\underbrace{2^{\Sigma^*} \times \cdots \times 2^{\Sigma^*}}_{m}$ is an ω-chain complete poset, and the map Φ_G is ω-continous. Thus, by Proposition 7.7, the map Φ_G has a least-fixed point. It turns out that the components of this least fixed-point are precisely the languages generated by the grammars (V, Σ, P, A_i). Before proving this fact, let us give an example illustrating it.

Example. Consider the grammar $G = (\{A, B, a, b\}, \{a, b\}, P, A)$ defined by the rules

$$\begin{aligned} A &\to BB + ab,\\ B &\to aBb + ab.\end{aligned}$$

The least fixed-point of Φ_G is the least upper bound of the chain

$$(\Phi_G^n(\emptyset, \emptyset)) = ((\Phi_{G,A}^n(\emptyset, \emptyset), \Phi_{G,B}^n(\emptyset, \emptyset)),$$

where

$$\Phi_{G,A}^0(\emptyset, \emptyset) = \Phi_{G,B}^0(\emptyset, \emptyset) = \emptyset,$$

and

$$\begin{aligned}\Phi^{n+1}_{G,A}(\emptyset,\emptyset) &= \Phi^{n}_{G,B}(\emptyset,\emptyset)\Phi^{n}_{G,B}(\emptyset,\emptyset) \cup \{ab\},\\ \Phi^{n+1}_{G,B}(\emptyset,\emptyset) &= a\Phi^{n}_{G,B}(\emptyset,\emptyset)b \cup \{ab\}.\end{aligned}$$

It is easy to verify that

$$\begin{aligned}\Phi^{1}_{G,A}(\emptyset,\emptyset) &= \{ab\},\\ \Phi^{1}_{G,B}(\emptyset,\emptyset) &= \{ab\},\\ \Phi^{2}_{G,A}(\emptyset,\emptyset) &= \{ab, abab\},\\ \Phi^{2}_{G,B}(\emptyset,\emptyset) &= \{ab, aabb\},\\ \Phi^{3}_{G,A}(\emptyset,\emptyset) &= \{ab, abab, abaabb, aabbab, aabbaabb\},\\ \Phi^{3}_{G,B}(\emptyset,\emptyset) &= \{ab, aabb, aaabbb\}.\end{aligned}$$

By induction, we can easily prove that the two components of the least fixed-point are the languages

$$L_A = \{a^m b^m a^n b^n \mid m, n \geq 1\} \cup \{ab\} \quad \text{and} \quad L_B = \{a^n b^n \mid n \geq 1\}.$$

Letting $G_A = (\{A, B, a, b\}, \{a, b\}, P, A)$ and $G_B = (\{A, B, a, b\}, \{a, b\}, P, B)$, it is indeed true that $L_A = L(G_A)$ and $L_B = L(G_B)$.

We have the following theorem due to Ginsburg and Rice:

Theorem 7.9. *Given a context-free grammar $G = (V, \Sigma, P, S)$ with m nonterminals A_1, ..., A_m, the least fixed-point of the map Φ_G is the m-tuple of languages*

$$(L(G_{A_1}), \ldots, L(G_{A_m})),$$

where $G_{A_i} = (V, \Sigma, P, A_i)$.

Proof. Writing G as

$$\begin{aligned}A_1 &\to \alpha_{1,1} + \cdots + \alpha_{1,n_1},\\ \cdots &\to \cdots\\ A_i &\to \alpha_{i,1} + \cdots + \alpha_{i,n_i},\\ \cdots &\to \cdots\\ A_m &\to \alpha_{m,1} + \cdots + \alpha_{m,n_n},\end{aligned}$$

let $M = \max\{|\alpha_{i,j}|\}$ be the maximum length of right-hand sides of rules in P. Let

$$\Phi^{n}_{G}(\emptyset, \ldots, \emptyset) = (\Phi^{n}_{G,1}(\emptyset, \ldots, \emptyset), \ldots, \Phi^{n}_{G,m}(\emptyset, \ldots, \emptyset)).$$

Then, for any $w \in \Sigma^*$, observe that

$$w \in \Phi^1_{G,i}(\emptyset, \ldots, \emptyset)$$

iff there is some rule $A_i \to \alpha_{i,j}$ with $w = \alpha_{i,j}$, and that

$$w \in \Phi^n_{G,i}(\emptyset, \ldots, \emptyset)$$

for some $n \geq 2$ iff there is some rule $A_i \to \alpha_{i,j}$ with $\alpha_{i,j}$ of the form

$$\alpha_{i,j} = u_1 A_{j_1} u_2 \cdots u_k A_{j_k} u_{k+1},$$

where $u_1, \ldots, u_{k+1} \in \Sigma^*$, $k \geq 1$, and some $w_1, \ldots, w_k \in \Sigma^*$ such that

$$w_h \in \Phi^{n-1}_{G,j_h}(\emptyset, \ldots, \emptyset),$$

and

$$w = u_1 w_1 u_2 \cdots u_k w_k u_{k+1}.$$

We prove the following two claims.

Claim 1: For every $w \in \Sigma^*$, if $A_i \overset{n}{\Longrightarrow} w$, then $w \in \Phi^p_{G,i}(\emptyset, \ldots, \emptyset)$, for some $p \geq 1$.

Claim 2: For every $w \in \Sigma^*$, if $w \in \Phi^n_{G,i}(\emptyset, \ldots, \emptyset)$, with $n \geq 1$, then $A_i \overset{p}{\Longrightarrow} w$ for some $p \leq (M+1)^{n-1}$.

Proof of Claim 1. We proceed by induction on n. If $A_i \overset{1}{\Longrightarrow} w$, then $w = \alpha_{i,j}$ for some rule $A \to \alpha_{i,j}$, and by the remark just before the claim, $w \in \Phi^1_{G,i}(\emptyset, \ldots, \emptyset)$.

If $A_i \overset{n+1}{\Longrightarrow} w$ with $n \geq 1$, then

$$A_i \overset{n}{\Longrightarrow} \alpha_{i,j} \Longrightarrow w$$

for some rule $A_i \to \alpha_{i,j}$. If

$$\alpha_{i,j} = u_1 A_{j_1} u_2 \cdots u_k A_{j_k} u_{k+1},$$

where $u_1, \ldots, u_{k+1} \in \Sigma^*$, $k \geq 1$, then $A_{j_h} \overset{n_h}{\Longrightarrow} w_h$, where $n_h \leq n$, and

$$w = u_1 w_1 u_2 \cdots u_k w_k u_{k+1}$$

for some $w_1, \ldots, w_k \in \Sigma^*$. By the induction hypothesis,

$$w_h \in \Phi^{p_h}_{G,j_h}(\emptyset, \ldots, \emptyset),$$

for some $p_h \geq 1$, for every h, $1 \leq h \leq k$. Letting $p = \max\{p_1, \ldots, p_k\}$, since each sequence $(\Phi^q_{G,i}(\emptyset, \ldots, \emptyset))$ is an ω-chain, we have $w_h \in \Phi^p_{G,j_h}(\emptyset, \ldots, \emptyset)$ for every h, $1 \leq h \leq k$, and by the remark just before the claim, $w \in \Phi^{p+1}_{G,i}(\emptyset, \ldots, \emptyset)$. $\square$

Proof of Claim 2. We proceed by induction on n. If $w \in \Phi^1_{G,i}(\emptyset, \ldots, \emptyset)$, by the remark just before the claim, then $w = \alpha_{i,j}$ for some rule $A \to \alpha_{i,j}$, and $A_i \overset{1}{\Longrightarrow} w$.

If $w \in \Phi^n_{G,i}(\emptyset, \ldots, \emptyset)$ for some $n \geq 2$, then there is some rule $A_i \to \alpha_{i,j}$ with $\alpha_{i,j}$ of the form

$$\alpha_{i,j} = u_1 A_{j_1} u_2 \cdots u_k A_{j_k} u_{k+1},$$

where $u_1, \ldots, u_{k+1} \in \Sigma^*$, $k \geq 1$, and some $w_1, \ldots, w_k \in \Sigma^*$ such that

$$w_h \in \Phi^{n-1}_{G,j_h}(\emptyset, \ldots, \emptyset),$$

and

$$w = u_1 w_1 u_2 \cdots u_k w_k u_{k+1}.$$

By the induction hypothesis, $A_{j_h} \overset{p_h}{\Longrightarrow} w_h$ with $p_h \leq (M+1)^{n-2}$, and thus

$$A_i \Longrightarrow u_1 A_{j_1} u_2 \cdots u_k A_{j_k} u_{k+1} \overset{p_1}{\Longrightarrow} \cdots \overset{p_k}{\Longrightarrow} w,$$

so that $A_i \overset{p}{\Longrightarrow} w$ with

$$p \leq p_1 + \cdots + p_k + 1 \leq M(M+1)^{n-2} + 1 \leq (M+1)^{n-1},$$

since $k \leq M$. □

Combining Claim 1 and Claim 2, we have

$$L(G_{A_i}) = \bigcup_n \Phi^n_{G,i}(\emptyset, \ldots, \emptyset),$$

which proves that the least fixed-point of the map Φ_G is the m-tuple of languages

$$(L(G_{A_1}), \ldots, L(G_{A_m})).$$

□

We now show how theorem 7.9 can be used to give a short proof that every context-free grammar can be converted to Greibach Normal Form.

7.9 Least Fixed-Points and the Greibach Normal Form

The hard part in converting a grammar $G = (V, \Sigma, P, S)$ to Greibach Normal Form is to convert it to a grammar in so-called *weak Greibach Normal Form*, where the productions are of the form

$$\begin{aligned} &A \to a\alpha, \quad \text{or} \\ &S \to \epsilon, \end{aligned}$$

where $a \in \Sigma$, $\alpha \in V^*$, and if $S \to \epsilon$ is a rule, then S does not occur on the right-hand side of any rule. Indeed, if we first convert G to Chomsky Normal Form, it turns out that we will get rules of the form $A \to aBC$, $A \to aB$ or $A \to a$.

Using the algorithm for eliminating ϵ-rules and chain rules, we can first convert the original grammar to a grammar with no chain rules and no ϵ-rules except possibly $S \to \epsilon$, in which case, S does not appear on the right-hand side of rules. Thus, for the purpose of converting to weak Greibach Normal Form, we can assume that we are dealing with grammars without chain rules and without ϵ-rules. Let us also assume that we computed the set $T(G)$ of nonterminals that actually derive some terminal string, and that useless productions involving symbols not in $T(G)$ have been deleted.

Let us explain the idea of the conversion using the following grammar:

$$A \to AaB + BB + b.$$
$$B \to Bd + BAa + aA + c.$$

The first step is to group the right-hand sides α into two categories: those whose leftmost symbol is a terminal ($\alpha \in \Sigma V^*$) and those whose leftmost symbol is a nonterminal ($\alpha \in NV^*$). It is also convenient to adopt a matrix notation, and we can write the above grammar as

$$(A, B) = (A, B) \begin{pmatrix} aB & \emptyset \\ B & \{d, Aa\} \end{pmatrix} + (b, \{aA, c\})$$

Thus, we are dealing with matrices (and row vectors) whose entries are finite subsets of V^*. For notational simplicity, braces around singleton sets are omitted. The finite subsets of V^* form a semiring, where addition is union, and multiplication is concatenation. Addition and multiplication of matrices are as usual, except that the semiring operations are used. We will also consider matrices whose entries are languages over Σ. Again, the languages over Σ form a semiring, where addition is union, and multiplication is concatenation. The identity element for addition is $\emptyset$, and the identity element for multiplication is $\{\epsilon\}$. As above, addition and multiplication of matrices are as usual, except that the semiring operations are used. For example, given any languages $A_{i,j}$ and $B_{i,j}$ over Σ, where $i, j \in \{1, 2\}$, we have

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} = \begin{pmatrix} A_{1,1}B_{1,1} \cup A_{1,2}B_{2,1} & A_{1,1}B_{1,2} \cup A_{1,2}B_{2,2} \\ A_{2,1}B_{1,1} \cup A_{2,2}B_{2,1} & A_{2,1}B_{1,2} \cup A_{2,2}B_{2,2} \end{pmatrix}$$

Letting $X = (A, B)$, $K = (b, \{aA, c\})$, and

$$H = \begin{pmatrix} aB & \emptyset \\ B & \{d, Aa\} \end{pmatrix}$$

the above grammar can be concisely written as

$$X = XH + K.$$

More generally, given any context-free grammar $G = (V, \Sigma, P, S)$ with m nonterminals $A_1, \ldots, A_m$, assuming that there are no chain rules, no ϵ-rules, and that every nonterminal belongs to $T(G)$, letting

$$X = (A_1, \ldots, A_m),$$

we can write G as

$$X = XH + K,$$

for some appropriate $m \times m$ matrix H in which every entry contains a set (possibly empty) of strings in V^+, and some row vector K in which every entry contains a set (possibly empty) of strings α each beginning with a terminal ($\alpha \in \Sigma V^*$).

Given an $m \times m$ square matrix $A = (A_{i,j})$ of languages over Σ, we can define the matrix A^* whose entry $A^*_{i,j}$ is given by

$$A^*_{i,j} = \bigcup_{n \geq 0} A^n_{i,j},$$

where $A^0 = Id_m$, the identity matrix, and A^n is the n-th power of A. Similarly, we define A^+ where

$$A^+_{i,j} = \bigcup_{n \geq 1} A^n_{i,j}.$$

Given a matrix A where the entries are finite subset of V^*, where $N = \{A_1, \ldots, A_m\}$, for any m-tuple $\Lambda = (L_1, \ldots, L_m)$ of languages over Σ, we let

$$\Phi[\Lambda](A) = (\Phi[\Lambda](A_{i,j})).$$

Given a system $X = XH + K$ where H is an $m \times m$ matrix and X, K are row matrices, if H and K do not contain any nonterminals, we claim that the least fixed-point of the grammar G associated with $X = XH + K$ is KH^*. This is easily seen by computing the approximations $X^n = \Phi^n_G(\emptyset, \ldots, \emptyset)$. Indeed, $X^0 = K$, and

$$X^n = KH^n + KH^{n-1} + \cdots + KH + K = K(H^n + H^{n-1} + \cdots + H + I_m).$$

Similarly, if Y is an $m \times m$ matrix of nonterminals, the least fixed-point of the grammar associated with $Y = HY + H$ is H^+ (provided that H does not contain any nonterminals).

Given any context-free grammar $G = (V, \Sigma, P, S)$ with m nonterminals $A_1, \ldots, A_m$, writing G as $X = XH + K$ as explained earlier, we can form another grammar GH by creating m^2 new nonterminals $Y_{i,j}$, where the rules of this new grammar are defined by the system of two matrix equations

$$X = KY + K,$$
$$Y = HY + H,$$

where $Y = (Y_{i,j})$.

The following proposition is the key to the Greibach Normal Form.

Proposition 7.10. *Given any context-free grammar $G = (V, \Sigma, P, S)$ with m nonterminals $A_1, \ldots, A_m$, writing G as*

$$X = XH + K$$

as explained earlier, if GH is the grammar defined by the system of two matrix equations

$$X = KY + K,$$
$$Y = HY + H,$$

as explained above, then the components in X of the least-fixed points of the maps Φ_G and Φ_{GH} are equal.

Proof. Let U be the least-fixed point of Φ_G, and let (V, W) be the least fixed-point of Φ_{GH}. We shall prove that $U = V$. For notational simplicity, let us denote $\Phi[U](H)$ as $H[U]$ and $\Phi[U](K)$ as $K[U]$.

Since U is the least fixed-point of $X = XH + K$, we have

$$U = UH[U] + K[U].$$

Since $H[U]$ and $K[U]$ do not contain any nonterminals, by a previous remark, $K[U]H^*[U]$ is the least-fixed point of $X = XH[U] + K[U]$, and thus,

$$K[U]H^*[U] \leq U.$$

On the other hand, by monotonicity,

$$K[U]H^*[U]H\Big[K[U]H^*[U]\Big] + K\Big[K[U]H^*[U]\Big] \leq K[U]H^*[U]H[U] + K[U] = K[U]H^*[U],$$

and since U is the least fixed-point of $X = XH + K$,

$$U \leq K[U]H^*[U].$$

Therefore, $U = K[U]H^*[U]$. We can prove in a similar manner that $W = H[V]^+$.

Let $Z = H[U]^+$. We have

$$K[U]Z + K[U] = K[U]H[U]^+ + K[U] = K[U]H[U]^* = U,$$

and

$$H[U]Z + H[U] = H[U]H[U]^+ + H[U] = H[U]^+ = Z,$$

and since (V, W) is the least fixed-point of $X = KY + K$ and $Y = HY + H$, we get $V \leq U$ and $W \leq H[U]^+$.

We also have

$$V = K[V]W + K[V] = K[V]H[V]^{+} + K[V] = K[V]H[V]^{*},$$

and

$$VH[V] + K[V] = K[V]H[V]^{*}H[V] + K[V] = K[V]H[V]^{*} = V,$$

and since U is the least fixed-point of $X = XH + K$, we get $U \leq V$. Therefore, $U = V$, as claimed. $\square$

Note that the above proposition actually applies to any grammar. Applying Proposition 7.10 to our example grammar, we get the following new grammar:

$$\begin{aligned}(A, B) &= (b, \{aA, c\}) \begin{pmatrix} Y_1 & Y_2 \\ Y_3 & Y_4 \end{pmatrix} + (b, \{aA, c\}), \\ \begin{pmatrix} Y_1 & Y_2 \\ Y_3 & Y_4 \end{pmatrix} &= \begin{pmatrix} aB & \emptyset \\ B & \{d, Aa\} \end{pmatrix} \begin{pmatrix} Y_1 & Y_2 \\ Y_3 & Y_4 \end{pmatrix} + \begin{pmatrix} aB & \emptyset \\ B & \{d, Aa\} \end{pmatrix}\end{aligned}$$

There are still some nonterminals appearing as leftmost symbols, but using the equations defining A and B, we can replace A with

$$\{bY_1, aAY_3, cY_3, b\}$$

and B with

$$\{bY_2, aAY_4, cY_4, aA, c\},$$

obtaining a system in weak Greibach Normal Form. This amounts to converting the matrix

$$H = \begin{pmatrix} aB & \emptyset \\ B & \{d, Aa\} \end{pmatrix}$$

to the matrix

$$L = \begin{pmatrix} aB & \emptyset \\ \{bY_2, aAY_4, cY_4, aA, c\} & \{d, bY_1a, aAY_3a, cY_3a, ba\} \end{pmatrix}$$

The weak Greibach Normal Form corresponds to the new system

$$\begin{aligned} X &= KY + K, \\ Y &= LY + L. \end{aligned}$$

This method works in general for any input grammar with no ϵ-rules, no chain rules, and such that every nonterminal belongs to $T(G)$. Under these conditions, the row vector K

contains some nonempty entry, all strings in K are in ΣV^*, and all strings in H are in V^+. After obtaining the grammar GH defined by the system

$$\begin{aligned} X &= KY + K, \\ Y &= HY + H, \end{aligned}$$

we use the system $X = KY + K$ to express every nonterminal A_i in terms of expressions containing strings $\alpha_{i,j}$ involving a terminal as the leftmost symbol ($\alpha_{i,j} \in \Sigma V^*$), and we replace all leftmost occurrences of nonterminals in H (occurrences A_i in strings of the form $A_i\beta$, where $\beta \in V^*$) using the above expressions. In this fashion, we obtain a matrix L, and it is immediately shown that the system

$$\begin{aligned} X &= KY + K, \\ Y &= LY + L, \end{aligned}$$

generates the same tuple of languages. Furthermore, this last system corresponds to a weak Greibach Normal Form.

It we start with a grammar in Chomsky Normal Form (with no production $S \to \epsilon$) such that every nonterminal belongs to $T(G)$, we actually get a Greibach Normal Form (the entries in K are terminals, and the entries in H are nonterminals). Thus, we have justified Proposition 7.6. The method is also quite economical, since it introduces only m^2 new nonterminals. However, the resulting grammar may contain some useless nonterminals.

7.10 Tree Domains and Gorn Trees

Derivation trees play a very important role in parsing theory and in the proof of a strong version of the pumping lemma for the context-free languages known as Ogden's lemma. Thus, it is important to define derivation trees rigorously. We do so using Gorn trees.

Let $\mathbf{N}_+ = \{1, 2, 3, \ldots\}$.

Definition 7.11. A *tree domain* D is a nonempty subset of strings in $\mathbf{N}_+^*$ satisfying the conditions:

(1) For all $u, v \in \mathbf{N}_+^*$, if $uv \in D$, then $u \in D$.

(2) For all $u \in \mathbf{N}_+^*$, for every $i \in \mathbf{N}_+$, if $ui \in D$ then $uj \in D$ for every j, $1 \leq j \leq i$.

The tree domain

$$D = \{\epsilon, 1, 2, 11, 21, 22, 221, 222, 2211\}$$

is represented as follows:

$$\begin{tikzcd}
 & & \epsilon \arrow[dl] \arrow[dr] & & & \\
 & 1 \arrow[dl] & & 2 \arrow[dl] \arrow[dr] & & \\
11 & & 21 & & 22 \arrow[dl] \arrow[dr] & \\
 & & & 221 \arrow[d] & & 222 \\
 & & & 2211 & &
\end{tikzcd}$$

A tree labeled with symbols from a set Δ is defined as follows.

Definition 7.12. Given a set Δ of labels, a Δ-tree (for short, a *tree*) is a total function $t : D \to \Delta$, where D is a tree domain.

The domain of a tree t is denoted as $dom(t)$. Every string $u \in dom(t)$ is called a *tree address* or a *node*.

Let $\Delta = \{f, g, h, a, b\}$. The tree $t : D \to \Delta$, where D is the tree domain of the previous example and t is the function whose graph is

$$\{(\epsilon, f), (1, h), (2, g), (11, a), (21, a), (22, f), (221, h), (222, b), (2211, a)\}$$

is represented as follows:

$$\begin{tikzcd}
 & & f \arrow[dl] \arrow[dr] & & & \\
 & h \arrow[dl] & & g \arrow[dl] \arrow[dr] & & \\
a & & a & & f \arrow[dl] \arrow[dr] & \\
 & & & h \arrow[d] & & b \\
 & & & a & &
\end{tikzcd}$$

The *outdegree* (sometimes called *ramification*) $r(u)$ of a node u is the cardinality of the set

$$\{i \mid ui \in dom(t)\}.$$

Note that the outdegree of a node can be infinite. Most of the trees that we shall consider will be *finite-branching*, that is, for every node u, $r(u)$ will be an integer, and hence finite. If the outdegree of all nodes in a tree is bounded by n, then we can view the domain of the tree as being defined over $\{1, 2, \ldots, n\}^*$.

A node of outdegree 0 is called a *leaf*. The node whose address is ϵ is called the *root* of the tree. A tree is *finite* if its domain $dom(t)$ is finite. Given a node u in $dom(t)$, every node of the form ui in $dom(t)$ with $i \in \mathbf{N}_+$ is called a *son* (or *immediate successor*) of u.

Tree addresses are totally ordered *lexicographically*: $u \leq v$ if either u is a prefix of v or, there exist strings $x, y, z \in \mathbf{N}_+^*$ and $i, j \in \mathbf{N}_+$, with $i < j$, such that $u = xiy$ and $v = xjz$.

In the first case, we say that u is an *ancestor* (or *predecessor*) of v (or u *dominates* v) and in the second case, that u is *to the left* of v.

If $y = \epsilon$ and $z = \epsilon$, we say that xi is a *left brother* (or *left sibling*) of xj, $(i < j)$. Two tree addresses u and v are *independent* if u is not a prefix of v and v is not a prefix of u.

Given a finite tree t, the *yield* of t is the string

$$t(u_1)t(u_2)\cdots t(u_k),$$

where $u_1, u_2, \ldots, u_k$ is the sequence of leaves of t in lexicographic order.
For example, the yield of the tree below is $aaab$:

$$\begin{tikzcd}
 & & f \arrow[dl] \arrow[dr] & & & \\
 & h \arrow[dl] & & g \arrow[dl] \arrow[dr] & & \\
a & & a & & f \arrow[dl] \arrow[dr] & \\
 & & & h \arrow[d] & & b \\
 & & & a & &
\end{tikzcd}$$

Given a finite tree t, the *depth* of t is the integer

$$d(t) = \max\{|u| \mid u \in dom(t)\}.$$

Given a tree t and a node u in $dom(t)$, the *subtree rooted at* u is the tree t/u, whose domain is the set

$$\{v \mid uv \in dom(t)\}$$

and such that $t/u(v) = t(uv)$ for all v in $dom(t/u)$.

Another important operation is the operation of tree replacement (or tree substitution).

Definition 7.13. Given two trees t_1 and t_2 and a tree address u in t_1, the *result of substituting* t_2 at u in t_1, denoted by $t_1[u \leftarrow t_2]$, is the function whose graph is the set of pairs

$$\{(v, t_1(v)) \mid v \in dom(t_1),\ u \text{ is not a prefix of } v\} \cup \{(uv, t_2(v)) \mid v \in dom(t_2)\}.$$

Let t_1 and t_2 be the trees defined by the following diagrams:

Tree t_1

$$\begin{tikzcd}
& & f \arrow[dl] \arrow[dr] & & & \\
& h \arrow[dl] & & g \arrow[dl] \arrow[dr] & & \\
a & & a & & f \arrow[dl] \arrow[dr] & \\
& & & h \arrow[d] & & b \\
& & & a & &
\end{tikzcd}$$

Tree t_2

$$\begin{tikzcd}
& g \arrow[dl] \arrow[dr] & \\
a & & b
\end{tikzcd}$$

The tree $t_1[22 \leftarrow t_2]$ is defined by the following diagram:

$$\begin{tikzcd}
& & f \arrow[dl] \arrow[dr] & & & \\
& h \arrow[dl] & & g \arrow[dl] \arrow[dr] & & \\
a & & a & & g \arrow[dl] \arrow[dr] & \\
& & & a & & b
\end{tikzcd}$$

We can now define derivation trees and relate derivations to derivation trees.

7.11 Derivations Trees

Definition 7.14. Given a context-free grammar $G = (V, \Sigma, P, S)$, for any $A \in N$, an *A-derivation tree for G* is a $(V \cup \{\epsilon\})$-tree t (a tree with set of labels $(V \cup \{\epsilon\})$) such that:

(1) $t(\epsilon) = A$;

(2) For every nonleaf node $u \in dom(t)$, if $u1, \ldots, uk$ are the successors of u, then either there is a production $B \to X_1 \cdots X_k$ in P such that $t(u) = B$ and $t(ui) = X_i$ for all i, $1 \leq i \leq k$, or $B \to \epsilon \in P$, $t(u) = B$ and $t(u1) = \epsilon$. A *complete derivation (or parse tree)* is an S-tree whose yield belongs to Σ^*.

A derivation tree for the grammar

$$G_3 = (\{E, T, F, +, *, (,), a\}, \{+, *, (,), a\}, P, E),$$

where P is the set of rules

$$\begin{aligned} E &\longrightarrow E + T, \\ E &\longrightarrow T, \\ T &\longrightarrow T * F, \\ T &\longrightarrow F, \\ F &\longrightarrow (E), \\ F &\longrightarrow a, \end{aligned}$$

is shown in Figure 7.1. The yield of the derivation tree is $a + a * a$.

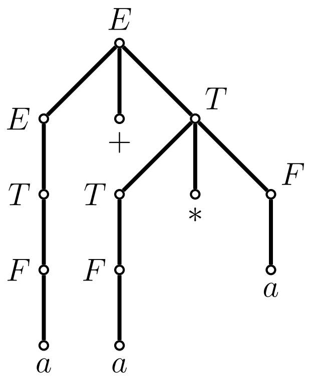


Figure 7.1: A complete derivation tree

Derivations trees are associated to derivations inductively as follows.

Definition 7.15. Given a context-free grammar $G = (V, \Sigma, P, S)$, for any $A \in N$, if $\pi : A \overset{n}{\Longrightarrow} \alpha$ is a derivation in G, we construct an *A-derivation tree t_π with yield α* as follows.

(1) If $n = 0$, then t_π is the one-node tree such that $dom(t_\pi) = \{\epsilon\}$ and $t_\pi(\epsilon) = A$.

(2) If $A \overset{n-1}{\Longrightarrow} \lambda B \rho \Longrightarrow \lambda\gamma\rho = \alpha$, then if t_1 is the A-derivation tree with yield $\lambda B \rho$ associated with the derivation $A \overset{n-1}{\Longrightarrow} \lambda B \rho$, and if t_2 is the tree associated with the production $B \to \gamma$ (that is, if

$$\gamma = X_1 \cdots X_k,$$

then $dom(t_2) = \{\epsilon, 1, \ldots, k\}$, $t_2(\epsilon) = B$, and $t_2(i) = X_i$ for all i, $1 \leq i \leq k$, or if $\gamma = \epsilon$, then $dom(t_2) = \{\epsilon, 1\}$, $t_2(\epsilon) = B$, and $t_2(1) = \epsilon$), then

$$t_\pi = t_1[u \leftarrow t_2],$$

where u is the address of the leaf labeled B in t_1.

The tree t_π is the *A-derivation tree associated with the derivation* $A \overset{n}{\Longrightarrow} \alpha$.

Given the grammar

$$G_2 = (\{E, +, *, (,), a\}, \{+, *, (,), a\}, P, E),$$

where P is the set of rules

$$\begin{aligned} E &\longrightarrow E + E, \\ E &\longrightarrow E * E, \\ E &\longrightarrow (E), \\ E &\longrightarrow a, \end{aligned}$$

the parse trees associated with two derivations of the string $a + a * a$ are shown in Figure 7.2:

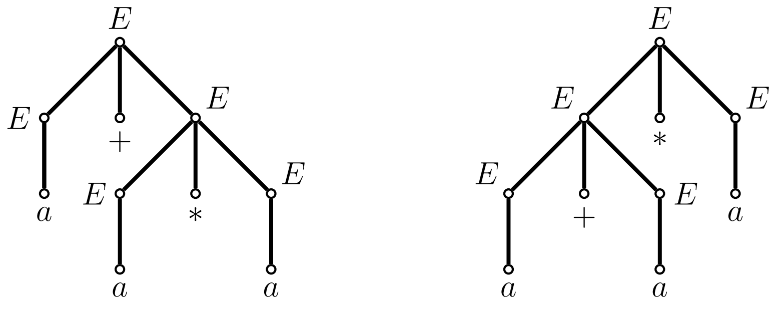


Figure 7.2: Two derivation trees for $a + a * a$

The following proposition is easily shown.

Proposition 7.11. *Let $G = (V, \Sigma, P, S)$ be a context-free grammar. For any derivation $A \overset{n}{\Longrightarrow} \alpha$, there is a unique A-derivation tree associated with this derivation, with yield α. Conversely, for any A-derivation tree t with yield α, there is a unique leftmost derivation $A \overset{*}{\underset{lm}{\Longrightarrow}} \alpha$ in G having t as its associated derivation tree.*

We will now prove a strong version of the pumping lemma for context-free languages due to Bill Ogden (1968).

7.12 Ogden's Lemma

Ogden's lemma states some combinatorial properties of parse trees that are deep enough. The yield w of such a parse tree can be split into 5 substrings u, v, x, y, z such that

$$w = uvxyz,$$

where u, v, x, y, z satisfy certain conditions. It turns out that we get a more powerful version of the lemma if we allow ourselves to *mark* certain occurrences of symbols in w before invoking the lemma. We can imagine that *marked occurrences* in a nonempty string w are occurrences of symbols in w in boldface, or red, or any given color (but one color only). For example, given $w = aaababbbaa$, we can mark the symbols of even index as follows:

$$a\mathbf{a}a\mathbf{b}a\mathbf{b}b\mathbf{b}a\mathbf{a}.$$

More rigorously, we can define a *marking* of a nonnull string $w\colon \{1, \ldots, n\} \to \Sigma$ as any function $m\colon \{1, \ldots, n\} \to \{0, 1\}$. Then, a letter w_i in w is a *marked occurrence* iff $m(i) = 1$, and an *unmarked occurrence* if $m(i) = 0$. The number of marked occurrences in w is equal to

$$\sum_{i=1}^{n} m(i).$$

Ogden's lemma only yields useful information for grammars G generating an infinite language. We could make this hypothesis, but it seems more elegant to use the precondition that the lemma only applies to strings $w \in L(D)$ such that w contains at least K marked occurrences, for a constant K large enough. If K is large enough, $L(G)$ will indeed be infinite.

Proposition 7.12. *For every context-free grammar G, there is some integer $K > 1$ such that, for every string $w \in \Sigma^+$, for every marking of w, if $w \in L(G)$ and w contains at least K marked occurrences, then there exists some decomposition of w as $w = uvxyz$, and some $A \in N$, such that the following properties hold:*

(1) There are derivations $S \overset{+}{\Longrightarrow} uAz$, $A \overset{+}{\Longrightarrow} vAy$, and $A \overset{+}{\Longrightarrow} x$, so that

$$uv^nxy^nz \in L(G)$$

for all $n \geq 0$ (the pumping property);

(2) x contains some marked occurrence;

(3) Either (both u and v contain some marked occurrence), or (both y and z contain some marked occurrence);

(4) vxy contains less than K marked occurrences.

Proof. Let t be any parse tree for w. We call a leaf of t a *marked leaf* if its label is a marked occurrence in the marked string w. The general idea is to make sure that K is large enough so that parse trees with yield w contain enough repeated nonterminals along some path from the root to some marked leaf. Let $r = |N|$, and let

$$p = \max\{2,\ \max\{|\alpha| \mid (A \to \alpha) \in P\}\}.$$

We claim that $K = p^{2r+3}$ does the job.

The key concept in the proof is the notion of a B-node. Given a parse tree t, a *B-node* is a node with at least two immediate successors u_1, u_2, such that for $i = 1, 2$, either u_i is a marked leaf, or u_i has some marked leaf as a descendant. We construct a path from the root to some marked leaf, so that for every B-node, we pick the leftmost successor with the maximum number of marked leaves as descendants. Formally, define a path $(s_0, \ldots, s_n)$ from the root to some marked leaf, so that:

(i) Every node s_i has some marked leaf as a descendant, and s_0 is the root of t;

(ii) If s_j is in the path, s_j is not a leaf, and s_j has a single immediate descendant which is either a marked leaf or has marked leaves as its descendants, let s_{j+1} be that unique immediate descendant of s_i.

(iii) If s_j is a B-node in the path, then let s_{j+1} be the leftmost immediate successors of s_j with the maximum number of marked leaves as descendants (assuming that if s_{j+1} is a marked leaf, then it is its own descendant).

(iv) If s_j is a leaf, then it is a marked leaf and $n = j$.

We will show that the path $(s_0, \ldots, s_n)$ contains at least $2r + 3$ B-nodes.

Claim: For every i, $0 \leq i \leq n$, if the path $(s_i, \ldots, s_n)$ contains b B-nodes, then s_i has at most p^b marked leaves as descendants.

Proof. We proceed by "backward induction", i.e., by induction on $n - i$. For $i = n$, there are no B-nodes, so that $b = 0$, and there is indeed $p^0 = 1$ marked leaf s_n. Assume that the claim holds for the path $(s_{i+1}, \ldots, s_n)$.

If s_i is not a B-node, then the number b of B-nodes in the path $(s_{i+1}, \ldots, s_n)$ is the same as the number of B-nodes in the path $(s_i, \ldots, s_n)$, and s_{i+1} is the only immediate successor of s_i having a marked leaf as descendant. By the induction hypothesis, s_{i+1} has at most p^b marked leaves as descendants, and this is also an upper bound on the number of marked leaves which are descendants of s_i.

If s_i is a B-node, then if there are b B-nodes in the path $(s_{i+1}, \ldots, s_n)$, there are $b + 1$ B-nodes in the path $(s_i, \ldots, s_n)$. By the induction hypothesis, s_{i+1} has at most p^b marked leaves as descendants. Since s_i is a B-node, s_{i+1} was chosen to be the leftmost immediate successor of s_i having the maximum number of marked leaves as descendants. Thus, since the outdegree of s_i is at most p, and each of its immediate successors has at most p^b marked leaves as descendants, the node s_i has at most $pp^d = p^{d+1}$ marked leaves as descendants, as desired. $\square$

Applying the claim to s_0, since w has at least $K = p^{2r+3}$ marked occurrences, we have $p^b \geq p^{2r+3}$, and since $p \geq 2$, we have $b \geq 2r + 3$, and the path $(s_0, \ldots, s_n)$ contains at least $2r + 3$ B-nodes (Note that this would not follow if we had $p = 1$).

Let us now select the lowest $2r+3$ B-nodes in the path, $(s_0, \ldots, s_n)$, and denote them $(b_1, \ldots, b_{2r+3})$. Every B-node b_i has at least two immediate successors $u_i < v_i$ such that u_i or v_i is on the path $(s_0, \ldots, s_n)$. If the path goes through u_i, we say that b_i is a *right B-node* and if the path goes through v_i, we say that b_i is a *left B-node*. Since $2r+3 = r+2+r+1$, either there are $r+2$ left B-nodes or there are $r+2$ right B-nodes in the path $(b_1, \ldots, b_{2r+3})$. Let us assume that there are $r+2$ left B-nodes, the other case being similar.

Let $(d_1, \ldots, d_{r+2})$ be the lowest $r+2$ left B-nodes in the path. Since there are $r+1$ B-nodes in the sequence $(d_2, \ldots, d_{r+2})$, and there are only r distinct nonterminals, there are two nodes d_i and d_j, with $2 \leq i < j \leq r+2$, such that $t(d_i) = t(d_j) = A$, for some $A \in N$. We can assume that d_i is an ancestor of d_j, and thus, $d_j = d_i\alpha$, for some $\alpha \neq \epsilon$.

If we prune out the subtree t/d_i rooted at d_i from t, we get an S-derivation tree having a yield of the form uAz, and we have a derivation of the form $S \overset{+}{\Longrightarrow} uAz$, since there are at least $r+2$ left B-nodes on the path, and we are looking at the lowest $r+1$ left B-nodes. Considering the subtree t/d_i, pruning out the subtree t/d_j rooted at α in t/d_i, we get an A-derivation tree having a yield of the form vAy, and we have a derivation of the form $A \overset{+}{\Longrightarrow} vAy$. Finally, the subtree t/d_j is an A-derivation tree with yield x, and we have a derivation $A \overset{+}{\Longrightarrow} x$. This proves (1) of the lemma.

Since s_n is a marked leaf and a descendant of d_j, x contains some marked occurrence, proving (2).

Since d_1 is a left B-node, some left sibbling of the immediate successor of d_1 on the path has some distinguished leaf in u as a descendant. Similarly, since d_i is a left B-node, some left sibbling of the immediate successor of d_i on the path has some distinguished leaf in v as a descendant. This proves (3).

$(d_j, \ldots, b_{2r+3})$ has at most $2r+1$ B-nodes, and by the claim shown earlier, d_j has at most p^{2r+1} marked leaves as descendants. Since $p^{2r+1} < p^{2r+3} = K$, this proves (4). ☐

Observe that condition (2) implies that $x \neq \epsilon$, and condition (3) implies that either $u \neq \epsilon$ and $v \neq \epsilon$, or $y \neq \epsilon$ and $z \neq \epsilon$. Thus, the pumping condition (1) implies that the set $\{uv^nxy^nz \mid n \geq 0\}$ is an infinite subset of $L(G)$, and $L(G)$ is indeed infinite, as we mentioned earlier. Note that $K \geq 3$, and in fact, $K \geq 32$. The "standard pumping lemma" due to Bar-Hillel, Perles, and Shamir, is obtained by letting all occurrences be marked in $w \in L(G)$.

Proposition 7.13. *For every context-free grammar G (without ϵ-rules), there is some integer $K > 1$ such that, for every string $w \in \Sigma^+$, if $w \in L(G)$ and $|w| \geq K$, then there exists some decomposition of w as $w = uvxyz$, and some $A \in N$, such that the following properties hold:*

(1) There are derivations $S \overset{+}{\Longrightarrow} uAz$, $A \overset{+}{\Longrightarrow} vAy$, and $A \overset{+}{\Longrightarrow} x$, so that

$$uv^nxy^nz \in L(G)$$

for all $n \geq 0$ (the pumping property);

(2) $x \neq \epsilon$*;*

(3) Either $v \neq \epsilon$ *or* $y \neq \epsilon$*;*

(4) $|vxy| \leq K$*.*

A stronger version could be stated, and we are just following tradition in stating this standard version of the pumping lemma.

Ogden's lemma or the pumping lemma can be used to show that certain languages are not context-free. The method is to proceed by contradiction, i.e., to assume (contrary to what we wish to prove) that a language L is indeed context-free, and derive a contradiction of Ogden's lemma (or of the pumping lemma). Thus, as in the case of the regular languages, it would be helpful to see what the negation of Ogden's lemma is, and for this, we first state Ogden's lemma as a logical formula.

For any nonnull string $w\colon \{1, \ldots, n\} \to \Sigma$, for any marking $m\colon \{1, \ldots, n\} \to \{0, 1\}$ of w, for any substring y of w, where $w = xyz$, with $|x| = h$ and $k = |y|$, the number of marked occurrences in y, denoted as $|m(y)|$, is defined as

$$|m(y)| = \sum_{i=h+1}^{i=h+k} m(i).$$

We will also use the following abbreviations:

$$\begin{aligned}
nat &= \{0, 1, 2, \ldots\},\\
nat32 &= \{32, 33, \ldots\},\\
A &\equiv w = uvxyz,\\
B &\equiv |m(x)| \geq 1,\\
C &\equiv (|m(u)| \geq 1 \wedge |m(v)| \geq 1) \vee (|m(y)| \geq 1 \wedge |m(z)| \geq 1),\\
D &\equiv |m(vxy)| < K,\\
P &\equiv \forall n\colon nat\ (uv^n xy^n z \in L(D)).
\end{aligned}$$

Ogden's lemma can then be stated as

$$\forall G\colon \text{CFG}\ \exists K\colon nat32\ \forall w\colon \Sigma^*\ \forall m\colon \text{marking}$$
$$\Big((w \in L(D) \wedge |m(w)| \geq K) \supset (\exists u, v, x, y, z\colon \Sigma^*\ A \wedge B \wedge C \wedge D \wedge P)\Big).$$

Recalling that

$$\neg(A \wedge B \wedge C \wedge D \wedge P) \equiv \neg(A \wedge B \wedge C \wedge D) \vee \neg P \equiv (A \wedge B \wedge C \wedge D) \supset \neg P$$

and

$$\neg(P \supset Q) \equiv P \wedge \neg Q,$$

the negation of Ogden's lemma can be stated as

$$\exists G\colon \mathrm{CFG}\, \forall K\colon nat32\, \exists w\colon \Sigma^*\, \exists m\colon \text{marking}$$
$$\Big((w \in L(D) \land |m(w)| \geq K) \land (\forall u, v, x, y, z\colon \Sigma^*\, (A \land B \land C \land D) \supset \neg P)\Big).$$

Since

$$\neg P \equiv \exists n\colon nat\, (uv^nxy^nz \notin L(D)),$$

in order to show that Ogden's lemma is contradicted, one needs to show that for some context-free grammar G, for every $K \geq 2$, there is some string $w \in L(D)$ and some marking m of w with at least K marked occurrences in w, such that for every possible decomposition $w = uvxyz$ satisfying the constraints $A \land B \land C \land D$, there is some $n \geq 0$ such that $uv^nxy^nz \notin L(D)$. When proceeding by contradiction, we have a language L that we are (wrongly) assuming to be context-free and we can use any CFG grammar G generating L. The creative part of the argument is to pick the right $w \in L$ and the right marking of w (not making any assumption on K).

As an illustration, we show that the language

$$L = \{a^nb^nc^n \mid n \geq 1\}$$

is not context-free. Since L is infinite, we will be able to use the pumping lemma.

The proof proceeds by contradiction. If L was context-free, there would be some context-free grammar G such that $L = L(G)$, and some constant $K > 1$ as in Ogden's lemma. Let $w = a^Kb^Kc^K$, and choose the $b's$ as marked occurrences. Then by Ogden's lemma, x contains some marked occurrence, and either both u, v or both y, z contain some marked occurrence. Assume that both u and v contain some b. We have the following situation:

$$\underbrace{a \cdots ab \cdots b}_{u}\underbrace{b \cdots b}_{v}\underbrace{b \cdots bc \cdots c}_{xyz}.$$

If we consider the string $uvvxyyz$, the number of a's is still K, but the number of b's is strictly greater than K since v contains at least one b, and thus $uvvxyyz \notin L$, a contradiction.

If both y and z contain some b we will also reach a contradiction because in the string $uvvxyyz$, the number of c's is still K, but the number of b's is strictly greater than K. Having reached a contradiction in all cases, we conclude that L is not context-free.

Let us now show that the language

$$L = \{a^mb^nc^md^n \mid m, n \geq 1\}$$

is not context-free.

Again, we proceed by contradiction. This time, let

$$w = a^Kb^Kc^Kd^K,$$

where the b's and c's are marked occurrences.

By Ogden's lemma, either both u, v contain some marked occurrence, or both y, z contain some marked occurrence, and x contains some marked occurrence. Let us first consider the case where both u, v contain some marked occurrence.

If v contains some b, since $uvvxyyz \in L$, v must contain only b's, since otherwise we would have a bad string in L, and we have the following situation:

$$\underbrace{a\cdots ab\cdots b}_{u}\underbrace{b\cdots b}_{v}\underbrace{b\cdots bc\cdots cd\cdots d}_{xyz}.$$

Since $uvvxyyz \in L$, the only way to preserve an equal number of b's and d's is to have $y \in d^+$. But then, vxy contains c^K, which contradicts (4) of Ogden's lemma.

If v contains some c, since x also contains some marked occurrence, it must be some c, and v contains only c's and we have the following situation:

$$\underbrace{a\cdots ab\cdots bc\cdots c}_{u}\underbrace{c\cdots c}_{v}\underbrace{c\cdots cd\cdots d}_{xyz}.$$

Since $uvvxyyz \in L$ and the number of a's is still K whereas the number of c's is strictly more than K, this case is impossible.

Let us now consider the case where both y, z contain some marked occurrence. Reasoning as before, the only possibility is that $v \in a^+$ and $y \in c^+$:

$$\underbrace{a\cdots a}_{u}\underbrace{a\cdots a}_{v}\underbrace{a\cdots ab\cdots bc\cdots c}_{x}\underbrace{c\cdots c}_{y}\underbrace{c\cdots cd\cdots d}_{z}.$$

But then, vxy contains b^K, which contradicts (4) of Ogden's Lemma. Since a contradiction was obtained in all cases, L is not context-free.

Ogden's lemma can also be used to show that the context-free language

$$\{a^m b^n c^n \mid m, n \geq 1\} \cup \{a^m b^m c^n \mid m, n \geq 1\}$$

is inherently ambiguous. The proof is quite involved.

Another corollary of the pumping lemma is that it is decidable whether a context-free grammar generates an infinite language.

Proposition 7.14. *Given any context-free grammar, G, if K is the constant of Ogden's lemma, then the following equivalence holds:*

$L(G)$ is infinite iff there is some $w \in L(G)$ such that $K \leq |w| < 2K$.

Proof. Let $K = p^{2r+3}$ be the constant from the proof of Proposition 7.12. If there is some $w \in L(G)$ such that $|w| \geq K$, we already observed that the pumping lemma implies that $L(G)$ contains an infinite subset of the form $\{uv^n xy^n z \mid n \geq 0\}$. Conversely, assume that $L(G)$ is infinite. If $|w| < K$ for all $w \in L(G)$, then $L(G)$ is finite. Thus, there is some $w \in L(G)$ such that $|w| \geq K$. Let $w \in L(G)$ be a minimal string such that $|w| \geq K$. By the pumping lemma, we can write w as $w = uvxyxz$, where $x \neq \epsilon$, $vy \neq \epsilon$, and $|vxy| \leq K$. By the pumping property, $uxz \in L(G)$. If $|w| \geq 2K$, then

$$|uxz| = |uvxyz| - |vy| > |uvxyz| - |vxy| \geq 2K - K = K,$$

and $|uxz| < |uvxyz|$, contradicting the minimality of w. Thus, we must have $|w| < 2K$. □

In particular, if G is in Chomsky Normal Form, it can be shown that we just have to consider derivations of length at most $4K - 3$.

7.13 Pushdown Automata

We have seen that the regular languages are exactly the languages accepted by DFA's or NFA's. The context-free languages are exactly the languages accepted by pushdown automata, for short, PDA's. However, although there are two versions of PDA's, deterministic and nondeterministic, contrary to the fact that every NFA can be converted to a DFA, nondeterministic PDA's are strictly more poweful than deterministic PDA's (DPDA's). Indeed, there are context-free languages that cannot be accepted by DPDA's.

Thus, the natural machine model for the context-free languages is nondeterministic, and for this reason, we just use the abbreviation PDA, as opposed to NPDA. We adopt a definition of a PDA in which the pushdown store, or stack, must not be empty for a move to take place. Other authors allow PDA's to make move when the stack is empty. Novices seem to be confused by such moves, and this is why we do not allow moves with an empty stack.

Intuitively, a PDA consists of an input tape, a nondeterministic finite-state control, and a stack.

Given any set X possibly infinite, let $\mathcal{P}_{fin}(X)$ be the set of all finite subsets of X.

Definition 7.16. A *pushdown automaton* is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where

- Q is a finite set of *states*;
- Σ is a finite *input alphabet*;
- Γ is a finite *pushdown store (or stack) alphabet*;
- $q_0 \in Q$ is the *start state* (or *initial state*);

- $Z_0 \in \Gamma$ is the *initial stack symbol* (or *bottom marker*);
- $F \subseteq Q$ is the set of *final (or accepting) states*;
- $\delta \colon Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to \mathcal{P}_{fin}(Q \times \Gamma^*)$ is the *transition function*.

A transition is of the form $(q, \gamma) \in \delta(p, a, Z)$, where $p, q \in Q$, $Z \in \Gamma$, $\gamma \in \Gamma^*$ and $a \in \Sigma \cup \{\epsilon\}$. A transition of the form $(q, \gamma) \in \delta(p, \epsilon, Z)$ is called an *ϵ-transition (or ϵ-move)*.

The way a PDA operates is explained in terms of *Instantaneous Descriptions*, for short *ID's*. Intuitively, an Instantaneous Description is a snapshot of the PDA. An ID is a triple of the form

$$(p, u, \alpha) \in Q \times \Sigma^* \times \Gamma^*.$$

The idea is that p is the current state, u is the remaining input, and α represents the stack.

It is important to note that we use the convention that the **leftmost** symbol in α represents the topmost stack symbol.

Given a PDA M, we define a relation $\vdash_M$ between pairs of ID's. This is very similar to the derivation relation $\Longrightarrow_G$ associated with a context-free grammar.

Intuitively, a PDA scans the input tape symbol by symbol from left to right, making moves that cause a change of state, an update to the stack (but only at the top), and either advancing the reading head to the next symbol, or not moving the reading head during an ϵ-move.

Definition 7.17. Given a PDA

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

the relation $\vdash_M$ is defined as follows:

(1) For any move $(q, \gamma) \in \delta(p, a, Z)$, where $p, q \in Q$, $Z \in \Gamma$, $a \in \Sigma$, for every ID of the form $(p, av, Z\alpha)$, we have

$$(p, av, Z\alpha) \vdash_M (q, v, \gamma\alpha).$$

(2) For any move $(q, \gamma) \in \delta(p, \epsilon, Z)$, where $p, q \in Q$, $Z \in \Gamma$, for every ID of the form $(p, u, Z\alpha)$, we have

$$(p, u, Z\alpha) \vdash_M (q, u, \gamma\alpha).$$

As usual, $\vdash_M^+$ is the transitive closure of $\vdash_M$, and $\vdash_M^*$ is the reflexive and transitive closure of $\vdash_M$.

A move of the form

$$(p, au, Z\alpha) \vdash_M (q, u, \alpha)$$

where $a \in \Sigma \cup \{\epsilon\}$, is called a *pop move*.

A move on a real input symbol $a \in \Sigma$ causes this input symbol to be consumed, and the reading head advances to the next input symbol. On the other hand, during an ϵ-move, the reading head stays put.

When

$$(p, u, \alpha) \vdash_M^* (q, v, \beta)$$

we say that we have a *computation*.

There are several equivalent ways of defining acceptance by a PDA.

Definition 7.18. Given a PDA

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

the following languages are defined:

(1) $T(M) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash_M^* (f, \epsilon, \alpha)$, where $f \in F$, and $\alpha \in \Gamma^*\}$.

We say that $T(M)$ is the *language accepted by M by final state*.

(2) $N(M) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash_M^* (q, \epsilon, \epsilon)$, where $q \in Q\}$.

We say that $N(M)$ is the *language accepted by M by empty stack*.

(3) $L(M) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash_M^* (f, \epsilon, \epsilon)$, where $f \in F\}$.

We say that $L(M)$ is the *language accepted by M by final state and empty stack*.

In all cases, note that the input w must be consumed entirely.

The following proposition shows that the acceptance mode does not matter for PDA's. As we will see shortly, it does matter for DPDAs.

Proposition 7.15. *For any language L, the following facts hold.*

(1) If $L = T(M)$ for some PDA M, then $L = L(M')$ for some PDA M'.

(2) If $L = N(M)$ for some PDA M, then $L = L(M')$ for some PDA M'.

(3) If $L = L(M)$ for some PDA M, then $L = T(M')$ for some PDA M'.

(4) If $L = L(M)$ for some PDA M, then $L = N(M')$ for some PDA M'.

In view of Proposition 7.15, the three acceptance modes T, N, L are equivalent.

The following PDA accepts the language

$$L = \{a^n b^n \mid n \geq 1\}$$

by empty stack.

$Q = \{1, 2\}$, $\Gamma = \{Z_0, a\}$;

$(1, a) \in \delta(1, a, Z_0)$,

$(1, aa) \in \delta(1, a, a)$,

$(2, \epsilon) \in \delta(1, b, a)$,

$(2, \epsilon) \in \delta(2, b, a)$.

The following PDA accepts the language

$$L = \{a^n b^n \mid n \geq 1\}$$

by final state (and also by empty stack).

$Q = \{1, 2, 3\}$, $\Gamma = \{Z_0, A, a\}$, $F = \{3\}$;

$(1, A) \in \delta(1, a, Z_0)$,

$(1, aA) \in \delta(1, a, A)$,

$(1, aa) \in \delta(1, a, a)$,

$(2, \epsilon) \in \delta(1, b, a)$,

$(2, \epsilon) \in \delta(2, b, a)$,

$(3, \epsilon) \in \delta(1, b, A)$,

$(3, \epsilon) \in \delta(2, b, A)$.

DPDA's are defined as follows.

Definition 7.19. A PDA

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

is a *deterministic PDA* (for short, *DPDA*), iff the following conditions hold for all $(p, Z) \in Q \times \Gamma$: either

(1) $|\delta(p, a, Z)| = 1$ for all $a \in \Sigma$, and $\delta(p, \epsilon, Z) = \emptyset$, or

(2) $\delta(p, a, Z) = \emptyset$ for all $a \in \Sigma$, and $|\delta(p, \epsilon, Z)| = 1$.

A DPDA *operates in realtime* iff it has no ϵ-transitions.

It turns out that for DPDA's the most general acceptance mode is by final state. Indeed, there are language that can only be accepted deterministically as $T(M)$. The language

$$L = \{a^m b^n \mid m \geq n \geq 1\}$$

is such an example. The problem is that $a^m b$ is a prefix of all strings $a^m b^n$, with $m \geq n \geq 2$.

A language L is a *deterministic context-free language* iff $L = T(M)$ for some DPDA M.

It is easily shown that if $L = N(M)$ (or $L = L(M)$) for some DPDA M, then $L = T(M')$ for some DPDA M' easily constructed from M.

A PDA is *unambiguous* iff for every $w \in \Sigma^*$, there is at most one computation

$$(q_0, w, Z_0) \vdash^* ID_n,$$

where ID_n is an accepting ID.

There are context-free languages that are not accepted by any DPDA. For example, it can be shown that the languages

$$L_1 = \{a^n b^n \mid n \geq 1\} \cup \{a^n b^{2n} \mid n \geq 1\},$$

and

$$L_2 = \{ww^R \mid w \in \{a, b\}^*\},$$

are not accepted by any DPDA.

Also note that unambiguous grammars for these languages can be easily given.

We now show that every context-free language is accepted by a PDA.

7.14 From Context-Free Grammars To PDA's

We show how a PDA can be easily constructed from a context-free grammar. Although simple, the construction is not practical for parsing purposes, since the resulting PDA is horribly nondeterministic.

Given a context-free grammar $G = (V, \Sigma, P, S)$, we define a one-state PDA M as follows:

$Q = \{q_0\}$; $\Gamma = V$; $Z_0 = S$; $F = \emptyset$;

For every rule $(A \to \alpha) \in P$, there is a transition

$$(q_0, \alpha) \in \delta(q_0, \epsilon, A).$$

For every $a \in \Sigma$, there is a transition

$$(q_0, \epsilon) \in \delta(q_0, a, a).$$

The intuition is that a computation of M mimics a leftmost derivation in G. One might say that we have a "pop/expand" PDA.

Proposition 7.16. *Given any context-free grammar $G = (V, \Sigma, P, S)$, the PDA M just described accepts $L(G)$ by empty stack, i.e., $L(G) = N(M)$.*

Proof sketch. The following two claims are proved by induction.

Claim 1:

For all $u, v \in \Sigma^*$ and all $\alpha \in NV^* \cup \{\epsilon\}$, if $S \overset{*}{\underset{lm}{\Longrightarrow}} u\alpha$, then

$$(q_0, uv, S) \vdash^* (q_0, v, \alpha).$$

Claim 2:

For all $u, v \in \Sigma^*$ and all $\alpha \in V^*$, if

$$(q_0, uv, S) \vdash^* (q_0, v, \alpha)$$

then $S \overset{*}{\underset{lm}{\Longrightarrow}} u\alpha$. $\square$

We now show how a PDA can be converted to a context-free grammar

7.15 From PDA's To Context-Free Grammars

The construction of a context-free grammar from a PDA is not really difficult, but it is quite messy. The construction is simplified if we first convert a PDA to an equivalent PDA such that for every move $(q, \gamma) \in \delta(p, a, Z)$ (where $a \in \Sigma \cup \{\epsilon\}$), we have $|\gamma| \leq 2$. In some sense, we form a kind of PDA in Chomsky Normal Form.

Proposition 7.17. *Given any PDA*

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

another PDA

$$M' = (Q', \Sigma, \Gamma', \delta', q'_0, Z'_0, F')$$

can be constructed, such that $L(M) = L(M')$ and the following conditions hold:

(1) There is a one-to-one correspondence between accepting computations of M and M';

(2) If M has no ϵ-moves, then M' has no ϵ-moves; If M is unambiguous, then M' is unambiguous;

(3) For all $p \in Q'$, all $a \in \Sigma \cup \{\epsilon\}$, and all $Z \in \Gamma'$, if $(q, \gamma) \in \delta'(p, a, Z)$, then $q \neq q'_0$ and $|\gamma| \leq 2$.

The crucial point of the construction is that accepting computations of a PDA accepting by empty stack and final state can be decomposed into *subcomputations* of the form

$$(p, uv, Z\alpha) \vdash^* (q, v, \alpha),$$

where for every intermediate ID (s, w, β), we have $\beta = \gamma\alpha$ for some $\gamma \neq \epsilon$.

The nonterminals of the grammar constructed from the PDA M are triples of the form $[p, Z, q]$ such that

$$(p, u, Z) \vdash^{+} (q, \epsilon, \epsilon)$$

for some $u \in \Sigma^*$.

Given a PDA

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

satisfying the conditions of Proposition 7.17, we construct a context-free grammar $G = (V, \Sigma, P, S)$ as follows:

$$V = \{[p, Z, q] \mid p, q \in Q, Z \in \Gamma\} \cup \Sigma \cup \{S\},$$

where S is a new symbol, and the productions are defined as follows: for all $p, q \in Q$, all $a \in \Sigma \cup \{\epsilon\}$, all $X, Y, Z \in \Gamma$, we have:

(1) $S \to \epsilon \in P$, if $q_0 \in F$;

(2) $S \to a \in P$, if $(f, \epsilon) \in \delta(q_0, a, Z_0)$, and $f \in F$;

(3) $S \to a[p, X, f] \in P$, for every $f \in F$, if $(p, X) \in \delta(q_0, a, Z_0)$;

(4) $S \to a[p, X, s][s, Y, f] \in P$, for every $f \in F$, for every $s \in Q$, if $(p, XY) \in \delta(q_0, a, Z_0)$;

(5) $[p, Z, q] \to a \in P$, if $(q, \epsilon) \in \delta(p, a, Z)$ and $p \neq q_0$;

(6) $[p, Z, s] \to a[q, X, s] \in P$, for every $s \in Q$, if $(q, X) \in \delta(p, a, Z)$ and $p \neq q_0$;

(7) $[p, Z, t] \to a[q, X, s][s, Y, t] \in P$, for every $s, t \in Q$, if $(q, XY) \in \delta(p, a, Z)$ and $p \neq q_0$.

Proposition 7.18. *Given any PDA*

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

satisfying the conditions of Proposition 7.17, the context-free grammar $G = (V, \Sigma, P, S)$ constructed as above generates $L(M)$, i.e., $L(G) = L(M)$. Furthermore, G is unambiguous iff M is unambiguous.

Proof skecth. We have to prove that

$$\begin{aligned} L(G) = \{w \in \Sigma^{+} \mid (q_0, w, Z_0) \vdash^{+} (f, \epsilon, \epsilon),\ f \in F\} \\ \cup\ \{\epsilon \mid q_0 \in F\}. \end{aligned}$$

For this, the following claim is proved by induction.

Claim:

For all $p, q \in Q$, all $Z \in \Gamma$, all $k \geq 1$, and all $w \in \Sigma^*$,

$$[p, Z, q] \overset{k}{\underset{lm}{\Longrightarrow}} w \quad \text{iff} \quad (p, w, Z) \vdash^+ (q, \epsilon, \epsilon).$$

Using the claim, it is possible to prove that $L(G) = L(M)$. $\square$

In view of Propositions 7.16 and 7.18, the family of context-free languages is exactly the family of languages accepted by PDA's. It is harder to give a grammatical characterization of the deterministic context-free languages. One method is to use Knuth *$LR(k)$-grammars*.

Another characterization can be given in terms of *strict deterministic grammars* due to Harrison and Havel.

7.16 The Chomsky-Schutzenberger Theorem

Unfortunately, there is no characterization of the context-free languages analogous to the characterization of the regular languages in terms of closure properties ($R(\Sigma)$).

However, there is a famous theorem due to Chomsky and Schutzenberger showing that every context-free language can be obtained from a special language, the *Dyck set*, in terms of homomorphisms, inverse homomorphisms and intersection with the regular languages.

Definition 7.20. Given the alphabet $\Sigma_2 = \{a, b, \overline{a}, \overline{b}\}$, define the relation $\simeq$ on Σ_2^* as follows: For all $u, v \in \Sigma_2^*$,

$$\begin{aligned} u \simeq v \quad \text{iff} \quad \exists x, y \in \Sigma_2^*, \qquad & u = xa\overline{a}y, \qquad v = xy \quad \text{or} \\ & u = xb\overline{b}y, \qquad v = xy. \end{aligned}$$

Let $\simeq^*$ be the reflexive and transitive closure of $\simeq$, and let $D_2 = \{w \in \Sigma_2^* \mid w \simeq^* \epsilon\}$. This is the *Dyck set* on two letters.

It is not hard to prove that D_2 is context-free.

Theorem 7.19. *(Chomsky-Schutzenberger) For every PDA, $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, there is a regular language R and two homomorphisms g, h such that*

$$L(M) = h(g^{-1}(D_2) \cap R).$$

Observe that Theorem 7.19 yields another proof of the fact that the language accepted a PDA is context-free.

Indeed, the context-free languages are closed under, homomorphisms, inverse homomorphisms, intersection with the regular languages, and D_2 is context-free.

From the characterization of a-transducers in terms of homomorphisms, inverse homomorphisms, and intersection with regular languages, we deduce that every context-free language is the image of D_2 under some a-transduction.

Chapter 8

A Survey of *LR*-Parsing Methods

In this chapter, we give a brief survey on *LR*-parsing methods. We begin with the definition of characteristic strings and the construction of Knuth's *LR*(0)-characteristic automaton. Next, we describe the shift/reduce algorithm. The need for lookahead sets is motivated by the resolution of conflicts. A unified method for computing FIRST, FOLLOW and LALR(1) lookahead sets is presented. The method uses a same graph algorithm *Traverse* which finds all nodes reachable from a given node and computes the union of predefined sets assigned to these nodes. Hence, the only difference between the various algorithms for computing FIRST, FOLLOW and LALR(1) lookahead sets lies in the fact that the initial sets and the graphs are computed in different ways. The method can be viewed as an efficient way for solving a set of simultaneously recursive equations with set variables. The method is inspired by DeRemer and Pennello's method for computing LALR(1) lookahead sets. However, DeRemer and Pennello use a more sophisticated graph algorithm for finding strongly connected components. We use a slightly less efficient but simpler algorithm (a depth-first search). We conclude with a brief presentation of *LR*(1) parsers.

8.1 *LR*(0)-Characteristic Automata

The purpose of *LR-parsing*, invented by D. Knuth in the mid sixties, is the following: Given a context-free grammar G, for any terminal string $w \in \Sigma^*$, find out whether w belongs to the language $L(G)$ generated by G, and if so, construct a rightmost derivation of w, in a deterministic fashion. Of course, this is not possible for all context-free grammars, but only for those that correspond to languages that can be recognized by a *deterministic* PDA (DPDA). Knuth's major discovery was that for a certain type of grammars, the *LR*(k)-grammars, a certain kind of DPDA could be constructed from the grammar (*shift/reduce parsers*). The k in *LR*(k) refers to the amount of *lookahead* that is necessary in order to proceed deterministically. It turns out that $k = 1$ is sufficient, but even in this case, Knuth construction produces very large DPDA's, and his original *LR*(1) method is not practical. Fortunately, around 1969, Frank DeRemer, in his MIT Ph.D. thesis, investigated a practical restriction of Knuth's method, known as *SLR*(k), and soon after, the *LALR*(k) method was discovered. The $SLR(k)$ and the $LALR(k)$ methods are both based on the construction of the $LR(0)$-*characteristic automaton* from a grammar G, and we begin by explaining this construction. The additional ingredient needed to obtain an $SLR(k)$ or an $LALR(k)$ parser from an $LR(0)$ parser is the computation of lookahead sets. In the SLR case, the FOLLOW sets are needed, and in the $LALR$ case, a more sophisticated version of the FOLLOW sets is needed. We will consider the construction of these sets in the case $k = 1$. We will discuss the shift/reduce algorithm and consider briefly ways of building $LR(1)$-parsing tables.

For simplicity of exposition, we first assume that grammars have no ϵ-rules. This restriction will be lifted in Section 8.10. Given a reduced context-free grammar $G = (V, \Sigma, P, S')$ augmented with start production $S' \to S$, where S' does not appear in any other productions, the set C_G of *characteristic strings of* G is the following subset of V^* (watch out, not Σ^*):

$$\begin{aligned} C_G = \{\alpha\beta \in V^* \mid S' \overset{*}{\underset{rm}{\Longrightarrow}} \alpha B v \underset{rm}{\Longrightarrow} \alpha\beta v,\\ \alpha, \beta \in V^*, v \in \Sigma^*, B \to \beta \in P\}. \end{aligned}$$

In words, C_G is a certain set of prefixes of sentential forms obtained in rightmost derivations: Those obtained by truncating the part of the sentential form immediately following the rightmost symbol in the righthand side of the production applied at the last step.

The fundamental property of LR-parsing, due to D. Knuth, is that C_G is a *regular language*. Furthermore, a DFA, DCG, accepting C_G, can be constructed from G.

Conceptually, it is simpler to construct the DFA accepting C_G in two steps:

(1) First, construct a nondeterministic automaton with ϵ-rules, NCG, accepting C_G.

(2) Apply the subset construction (Rabin and Scott's method) to NCG to obtain the DFA DCG.

In fact, careful inspection of the two steps of this construction reveals that it is possible to construct DCG directly in a single step, and this is the construction usually found in most textbooks on parsing.

The nondeterministic automaton NCG accepting C_G is defined as follows:

The states of N_{C_G} are "marked productions", where a marked production is a string of the form $A \to \alpha``."\beta$, where $A \to \alpha\beta$ is a production, and "." is a symbol not in V called the "dot" and which can appear anywhere within $\alpha\beta$.

The start state is $S' \to ``."S$, and the transitions are defined as follows:

(a) For every terminal $a \in \Sigma$, if $A \to \alpha``."a\beta$ is a marked production, with $\alpha, \beta \in V^*$, then there is a transition on input $a$ from state $A \to \alpha``."a\beta$ to state $A \to \alpha a``."\beta$ obtained by "shifting the dot." Such a transition is shown in Figure 8.1.

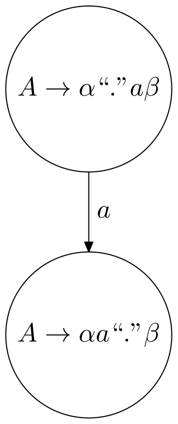


Figure 8.1: Transition on terminal input a

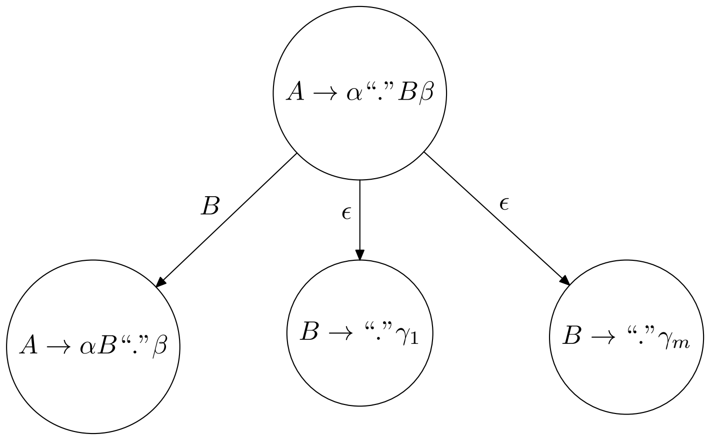


Figure 8.2: Transitions from a state $A \to \alpha$“.”$B\beta$

(b) For every nonterminal $B \in N$, if $A \to \alpha$“.”$B\beta$ is a marked production, with $\alpha, \beta \in V^*$, then there is a transition on input B from state $A \to \alpha$“.”$B\beta$ to state $A \to \alpha B$“.”β (obtained by “shifting the dot”), and transitions on input ϵ (the empty string) to all states $B \to$ “.”γ_i, for all productions $B \to \gamma_i$ with left-hand side B. Such transitions are shown in Figure 8.2.

(c) A state is *final* if and only if it is of the form $A \to \beta$“.” (that is, the dot is in the rightmost position).

The above construction is illustrated by the following example:

Example 1. Consider the grammar G_1 given by:

$$\begin{aligned} S &\longrightarrow E \\ E &\longrightarrow aEb \\ E &\longrightarrow ab \end{aligned}$$

The NFA for C_{G_1} is shown in Figure 8.3. The result of making the NFA for C_{G_1} deterministic is shown in Figure 8.4 (where transitions to the "dead state" have been omitted). The internal structure of the states $1, \ldots, 6$ is shown below:

$$\begin{aligned} 1: S &\longrightarrow .E \\ E &\longrightarrow .aEb \\ E &\longrightarrow .ab \\ 2: E &\longrightarrow a.Eb \\ E &\longrightarrow a.b \\ E &\longrightarrow .aEb \\ E &\longrightarrow .ab \\ 3: E &\longrightarrow aE.b \\ 4: S &\longrightarrow E. \\ 5: E &\longrightarrow ab. \\ 6: E &\longrightarrow aEb. \end{aligned}$$

The next example is slightly more complicated.

Example 2. Consider the grammar G_2 given by:

$$\begin{aligned} S &\longrightarrow E \\ E &\longrightarrow E + T \\ E &\longrightarrow T \\ T &\longrightarrow T * a \\ T &\longrightarrow a \end{aligned}$$

The result of making the NFA for C_{G_2} deterministic is shown in Figure 8.5 (where transitions to the "dead state" have been omitted). The internal structure of the states $1, \ldots, 8$

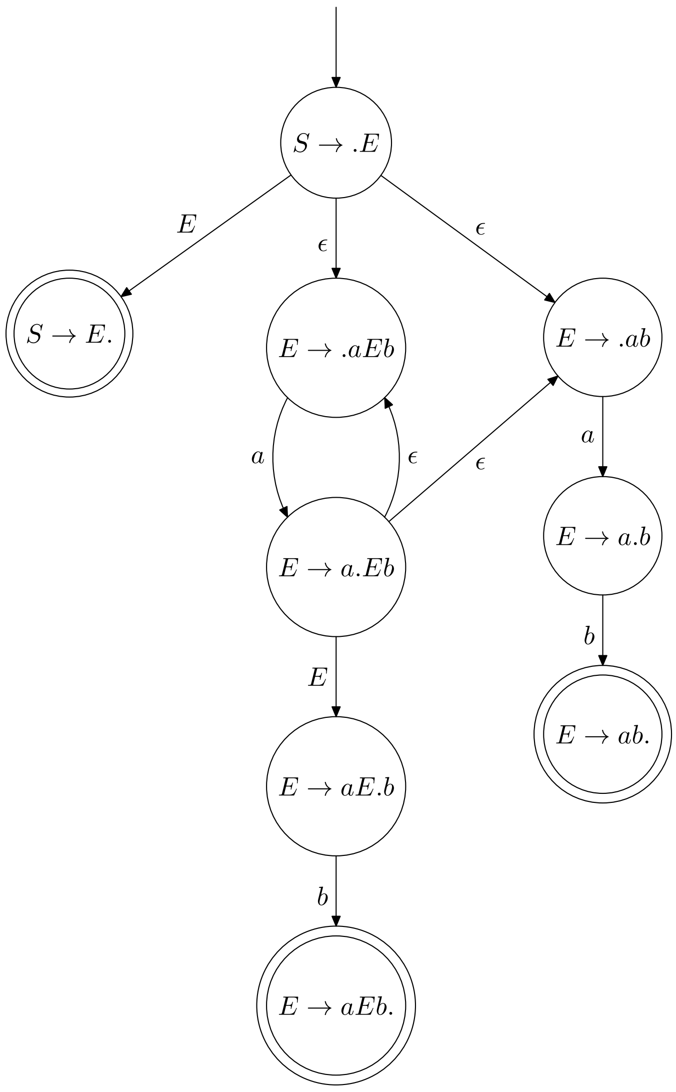


Figure 8.3: NFA for C_{G_1}

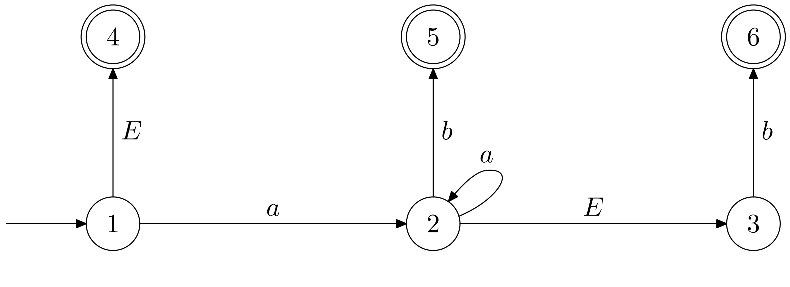


Figure 8.4: DFA for C_{G_1}

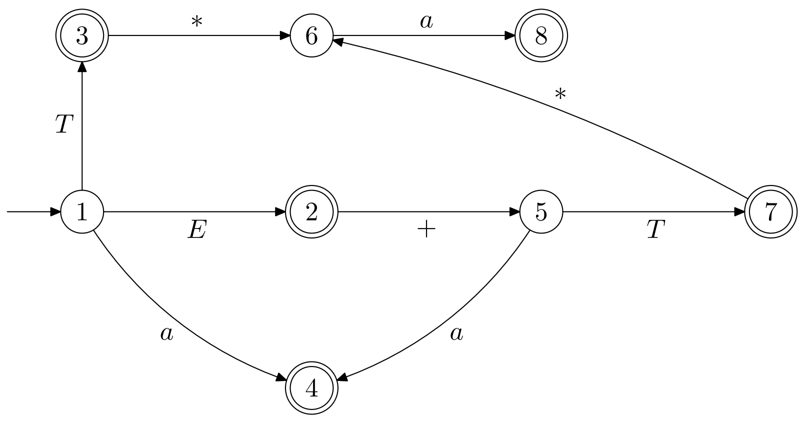


Figure 8.5: DFA for C_{G_2}

is shown below:

$$\begin{array}{rcl}
1: S & \longrightarrow & .E \\
E & \longrightarrow & .E + T \\
E & \longrightarrow & .T \\
T & \longrightarrow & .T * a \\
T & \longrightarrow & .a \\
2: E & \longrightarrow & E. + T \\
S & \longrightarrow & E. \\
3: E & \longrightarrow & T. \\
T & \longrightarrow & T. * a \\
4: T & \longrightarrow & a. \\
5: E & \longrightarrow & E + .T \\
T & \longrightarrow & .T * a \\
T & \longrightarrow & .a \\
6: T & \longrightarrow & T * .a \\
7: E & \longrightarrow & E + T. \\
T & \longrightarrow & T. * a \\
8: T & \longrightarrow & T * a.
\end{array}$$

Note that some of the marked productions are more important than others. For example, in state 5, the marked production $E \longrightarrow E + .T$ determines the state. The other two items $T \longrightarrow .T * a$ and $T \longrightarrow .a$ are obtained by ϵ-closure.

We call a marked production of the form $A \longrightarrow \alpha.\beta$, where $\alpha \neq \epsilon$, a *core item*. A marked production of the form $A \longrightarrow \beta.$ is called a *reduce item*. Reduce items only appear in final states.

If we also call $S' \longrightarrow .S$ a core item, we observe that every state is completely determined by its subset of core items. The other items in the state are obtained via ϵ-closure. We can take advantage of this fact to write a more efficient algorithm to construct in a single pass the $LR(0)$-automaton.

Also observe the so-called *spelling property*: All the transitions entering any given state have the same label.

Given a state s, if s contains both a reduce item $A \longrightarrow \gamma.$ and a shift item $B \longrightarrow \alpha.a\beta$, where $a \in \Sigma$, we say that there is a *shift/reduce conflict* in state s on input a. If s contains two (distinct) reduce items $A_1 \longrightarrow \gamma_1.$ and $A_2 \longrightarrow \gamma_2.$, we say that there is a *reduce/reduce conflict* in state s.

A grammar is said to be $LR(0)$ if the DFA DCG has no conflicts. This is the case for the grammar G_1. However, it should be emphasized that this is extremely rare in practice. The grammar G_1 is just very nice, and a toy example. In fact, G_2 is not $LR(0)$.

To eliminate conflicts, one can either compute $SLR(1)$-lookahead sets, using FOLLOW sets (see Section 8.6), or sharper lookahead sets, the $LALR(1)$ sets (see Section 8.9). For example, the computation of $SLR(1)$-lookahead sets for G_2 will eliminate the conflicts.

We will describe methods for computing $SLR(1)$-lookahead sets and $LALR(1)$-lookahead sets in Sections 8.6, 8.9, and 8.10. A more drastic measure is to compute the $LR(1)$-automaton, in which the states incoporate lookahead symbols (see Section 8.11). However, as we said before, this is not a practical methods for large grammars.

In order to motivate the construction of a shift/reduce parser from the DFA accepting C_G, let us consider a rightmost derivation for $w = aaabbb$ in reverse order for the grammar

$$\begin{aligned} 0&\colon S \longrightarrow E\\ 1&\colon E \longrightarrow aEb\\ 2&\colon E \longrightarrow ab \end{aligned}$$

$aaabbb$	$\alpha_1\beta_1v_1$		
$aaEbb$	$\alpha_1B_1v_1$		$E \longrightarrow ab$
$aaEbb$	$\alpha_2\beta_2v_2$		
aEb	$\alpha_2B_2v_2$		$E \longrightarrow aEb$
aEb	$\alpha_3\beta_3v_3$	$\alpha_3 = v_3 = \epsilon$	
E	$\alpha_3B_3v_3$	$\alpha_3 = v_3 = \epsilon$	$E \longrightarrow aEb$
E	$\alpha_4\beta_4v_4$	$\alpha_4 = v_4 = \epsilon$	
S	$\alpha_4B_4v_4$	$\alpha_4 = v_4 = \epsilon$	$S \longrightarrow E$

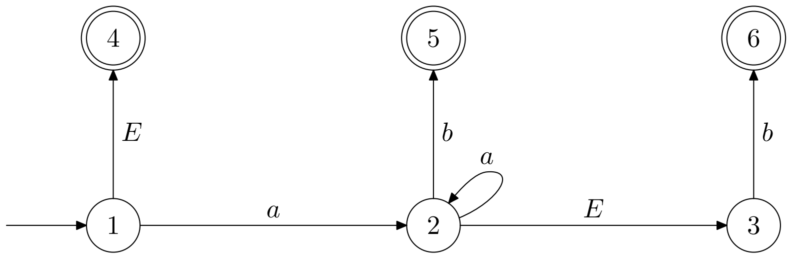


Figure 8.6: DFA for C_G

Observe that the strings $\alpha_i\beta_i$ for $i = 1, 2, 3, 4$ are all accepted by the DFA for C_G shown in Figure 8.6.

Also, every step from $\alpha_i\beta_i v_i$ to $\alpha_i B_i v_i$ is the inverse of the derivation step using the production $B_i \longrightarrow \beta_i$, and the marked production $B_i \longrightarrow \beta_i$" ." is one of the reduce items in the final state reached after processing $\alpha_i\beta_i$ with the DFA for C_G.

This suggests that we can parse $w = aaabbb$ by recursively running the DFA for C_G.

The first time (which correspond to step 1) we run the DFA for C_G on w, some string $\alpha_1\beta_1$ is accepted and the remaining input in v_1.

Then, we "reduce" β_1 to B_1 using a production $B_1 \longrightarrow \beta_1$ corresponding to some reduce item $B_1 \longrightarrow \beta_1$" ." in the final state s_1 reached on input $\alpha_1\beta_1$.

We now run the DFA for C_G on input $\alpha_1 B_1 v_1$. The string $\alpha_2\beta_2$ is accepted, and we have

$$\alpha_1 B_1 v_1 = \alpha_2\beta_2 v_2.$$

We reduce β_2 to B_2 using a production $B_2 \longrightarrow \beta_2$ corresponding to some reduce item $B_2 \longrightarrow \beta_2$" ." in the final state s_2 reached on input $\alpha_2\beta_2$.

We now run the DFA for C_G on input $\alpha_2 B_2 v_2$, and so on.

At the $(i+1)$th step ($i \geq 1$), we run the DFA for C_G on input $\alpha_i B_i v_i$. The string $\alpha_{i+1}\beta_{i+1}$ is accepted, and we have

$$\alpha_i B_i v_i = \alpha_{i+1}\beta_{i+1} v_{i+1}.$$

We reduce β_{i+1} to B_{i+1} using a production $B_{i+1} \longrightarrow \beta_{i+1}$ corresponding to some reduce item $B_{i+1} \longrightarrow \beta_{i+1}$" ." in the final state s_{i+1} reached on input $\alpha_{i+1}\beta_{i+1}$.

The string β_{i+1} in $\alpha_{i+1}\beta_{i+1}v_{i+1}$ if often called a *handle*.

Then we run again the DFA for C_G on input $\alpha_{i+1}B_{i+1}v_{i+1}$.

Now, because the DFA for C_G is *deterministic* there is no need to rerun it on the entire string $\alpha_{i+1}B_{i+1}v_{i+1}$, because *on input* α_{i+1} it will take us to *the same state*, say p_{i+1}, that it reached on input $\alpha_{i+1}\beta_{i+1}v_{i+1}$!

The trick is that we can use a *stack* to keep track of the sequence of states used to process $\alpha_{i+1}\beta_{i+1}$.

Then, to perform the reduction of $\alpha_{i+1}\beta_{i+1}$ to $\alpha_{i+1}B_{i+1}$, we simply *pop* a number of states equal to $|\beta_{i+1}|$, encovering a new state p_{i+1} on top of the stack, and from state p_{i+1} we perform the transition on input B_{i+1} to a state q_{i+1} (in the DFA for C_G), so we *push* state q_{i+1} on the stack which now contains the sequence of states on input $\alpha_{i+1}B_{i+1}$ that takes us to q_{i+1}.

Then we resume scanning v_{i+1} using the DGA for C_G, *pushing* each state being traversed on the stack until we hit a final state.

At this point we find the new string $\alpha_{i+2}\beta_{i+2}$ that leads to a final state and we continue as before.

The process stops when the remaining input v_{i+1} becomes empty and when the reduce item $S' \longrightarrow S.$ (here, $S \longrightarrow E.$) belongs to the final state s_{i+1}.

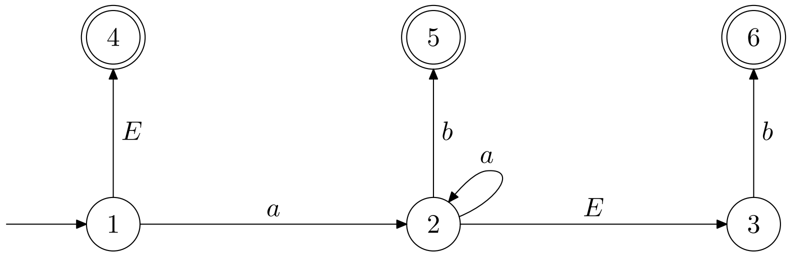


Figure 8.7: DFA for C_G

For example, on input $\alpha_2\beta_2 = aaEbb$, we have the sequence of states:

$$1\ 2\ 2\ 3\ 6$$

State 6 contains the marked production $E \longrightarrow aEb$“.”, so we pop the three topmost states 2 3 6 obtaining the stack

$$1\ 2$$

and then we make the transition from state 2 on input E, which takes us to state 3, so we push 3 on top of the stack, obtaining

$$1\ 2\ 3$$

We continue from state 3 on input b.

Basically, the recursive calls to the DFA for C_G are implemented using a stack.

What is not clear is, during step $i+1$, when reaching a final state s_{i+1}, how do we know which production $B_{i+1} \longrightarrow \beta_{i+1}$ to use in the reduction step?

Indeed, state s_{i+1} could contain several reduce items $B_{i+1} \longrightarrow \beta_{i+1}$"$.$".

This is where we assume that we were able to compute some *lookahead information*, that is, for every final state s and every input a, we know which unique production $n\colon B_{i+1} \longrightarrow \beta_{i+1}$ applies. This is recorded in a table name "action," such that action$(s, a) = rn$, where "r" stands for reduce.

Typically we compute SLR(1) or LALR(1) lookahead sets. Otherwise, we could pick some reducing production nondeterministicallly and use backtracking. This works but the running time may be exponential.

The DFA for C_G and the action table giving us the reductions can be combined to form a bigger action table which specifies completely how the parser using a stack works.

This kind of parser called a *shift-reduce parser* is discussed in the next section.

In order to make it easier to compute the reduce entries in the parsing table, we assume that the end of the input w is signalled by a special endmarker traditionally denoted by \$.

8.2 Shift/Reduce Parsers

A shift/reduce parser is a modified kind of DPDA. Firstly, push moves, called *shift moves*, are restricted so that exactly one symbol is pushed on top of the stack. Secondly, more powerful kinds of pop moves, called *reduce moves*, are allowed. During a reduce move, a finite number of stack symbols may be popped off the stack, and the last step of a reduce move, called a *goto move*, consists of pushing one symbol on top of new topmost symbol in the stack. Shift/reduce parsers use *parsing tables* constructed from the $LR(0)$-characteristic automaton DCG associated with the grammar. The shift and goto moves come directly from the transition table of DCG, but the determination of the reduce moves requires the computation of *lookahead sets*. The $SLR(1)$ lookahead sets are obtained from some sets called the FOLLOW sets (see Section 8.6), and the $LALR(1)$ lookahead sets LA$(s, A \longrightarrow \gamma)$ require fancier FOLLOW sets (see Section 8.9).

The construction of shift/reduce parsers is made simpler by assuming that the end of input strings $w \in \Sigma^*$ is indicated by the presence of an *endmarker*, usually denoted \$, and assumed not to belong to Σ.

Consider the grammar G_1 of Example 1, where we have numbered the productions $0, 1, 2$:

$$\begin{aligned} 0: S &\longrightarrow E \\ 1: E &\longrightarrow aEb \\ 2: E &\longrightarrow ab \end{aligned}$$

The parsing tables associated with the grammar G_1 are shown below:

	a	b	$\$$	E
1	$s2$			4
2	$s2$	$s5$		3
3		$s6$		
4			acc	
5	$r2$	$r2$	$r2$	
6	$r1$	$r1$	$r1$	

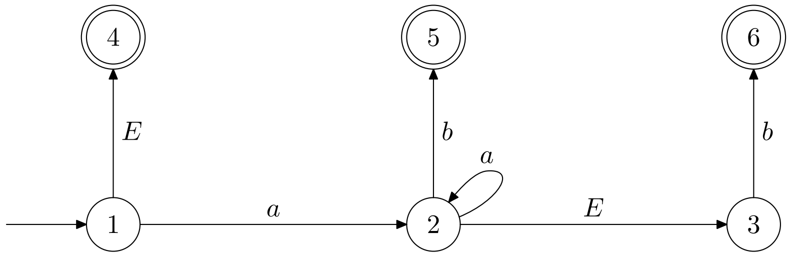


Figure 8.8: DFA for C_G

Entries of the form si are *shift actions*, where i denotes one of the states, and entries of the form rn are *reduce actions*, where n denotes a production number (*not* a state). The special action acc means accept, and signals the successful completion of the parse. Entries of the form i, in the rightmost column, are *goto actions*. All blank entries are **error** entries, and mean that the parse should be aborted.

We will use the notation action(s, a) for the entry corresponding to state s and terminal $a \in \Sigma \cup \{\$\}$, and goto$(s, A)$ for the entry corresponding to state s and nonterminal $A \in N - \{S'\}$.

Assuming that the input is $w\$$, we now describe in more detail how a shift/reduce parser proceeds. The parser uses a stack in which states are pushed and popped. Initially, the stack contains state 1 and the cursor pointing to the input is positioned on the leftmost symbol. There are four possibilities:

(1) If action$(s, a) = sj$, then push state j on top of the stack, and advance to the next input symbol in $w\$$. This is a *shift move*.

(2) If action$(s, a) = rn$, then do the following: First, determine the length $k = |\gamma|$ of the righthand side of the production n: $A \longrightarrow \gamma$. Then, pop the topmost k symbols off the stack (if $k = 0$, no symbols are popped). If p is the new top state on the stack (after the k pop moves), push the state goto(p, A) on top of the stack, where A is the lefthand side of the "reducing production" $A \longrightarrow \gamma$. Do not advance the cursor in the current input. This is a *reduce move*.

(3) If action$(s, \$)$ = acc, then accept. The input string w belongs to $L(G)$.

(4) In all other cases, **error**, abort the parse. The input string w does not belong to $L(G)$.

Observe that no explicit state control is needed. The current state is always the current topmost state in the stack. We illustrate below a parse of the input $aaabbb\$$.

stack	remaining input	action
1	$aaabbb\$$	$s2$
12	$aabbb\$$	$s2$
122	$abbb\$$	$s2$
1222	$bbb\$$	$s5$
12225	$bb\$$	$r2$
1223	$bb\$$	$s6$
12236	$b\$$	$r1$
123	$b\$$	$s6$
1236	$\$$	$r1$
14	$\$$	acc

Observe that the sequence of reductions read from bottom-up yields a rightmost derivation of $aaabbb$ from E (or from S, if we view the action acc as the reduction by the production $S \longrightarrow E$). This is a general property of LR-parsers.

The $SLR(1)$ reduce entries in the parsing tables are determined as follows: For every state s containing a reduce item $B \longrightarrow \gamma.$, if $B \longrightarrow \gamma$ is the production number n, enter the action rn for state s and every terminal $a \in \mathrm{FOLLOW}(B)$. If the resulting shift/reduce parser has no conflicts, we say that the grammar is $SLR(1)$. For the $LALR(1)$ reduce entries, enter the action rn for state s and production $n\colon B \longrightarrow \gamma$, for all $a \in \mathrm{LA}(s, B \longrightarrow \gamma)$. Similarly, if the shift/reduce parser obtained using $LALR(1)$-lookahead sets has no conflicts, we say that the grammar is $LALR(1)$.

8.3 Computation of FIRST

In order to compute the FOLLOW sets, we first need to to compute the FIRST sets! For simplicity of exposition, we first assume that grammars have no ϵ-rules. The general case will be treated in Section 8.10.

Given a context-free grammar $G = (V, \Sigma, P, S')$ (augmented with a start production $S' \longrightarrow S$), for every nonterminal $A \in N = V - \Sigma$, let

$$\mathrm{FIRST}(A) = \{a \mid a \in \Sigma,\ A \overset{+}{\Longrightarrow} a\alpha, \text{ for some } \alpha \in V^*\}.$$

For a terminal $a \in \Sigma$, let $\mathrm{FIRST}(a) = \{a\}$. The key to the computation of FIRST(A) is the following observation: a is in FIRST(A) if either a is in

$$\mathrm{INITFIRST}(A) = \{a \mid a \in \Sigma,\ A \longrightarrow a\alpha \in P, \text{ for some } \alpha \in V^*\},$$

or a is in

$$\{a \mid a \in \mathrm{FIRST}(B),\ A \longrightarrow B\alpha \in P, \text{ for some } \alpha \in V^*,\ B \neq A\}.$$

Note that the second assertion is true because, if $B \stackrel{+}{\Longrightarrow} a\delta$, then $A \Longrightarrow B\alpha \stackrel{+}{\Longrightarrow} a\delta\alpha$, and so, $\mathrm{FIRST}(B) \subseteq \mathrm{FIRST}(A)$ whenever $A \longrightarrow B\alpha \in P$, with $A \neq B$. Hence, the FIRST sets are the least solution of the following set of recursive equations: For each nonterminal A,

$$\mathrm{FIRST}(A) = \mathrm{INITFIRST}(A) \cup \bigcup \{\mathrm{FIRST}(B) \mid A \longrightarrow B\alpha \in P,\ A \neq B\}.$$

In order to explain the method for solving such systems, we will formulate the problem in more general terms, but first, we describe a "naive" version of the shift/reduce algorithm that hopefully demystifies the "'optimized version" described in Section 8.2.

8.4 The Intuition Behind the Shift/Reduce Algorithm

Let $DCG = (K, V, \delta, q_0, F)$ be the DFA accepting the regular language C_G, and let δ^* be the extension of δ to $K \times V^*$. Let us assume that the grammar G is either $SLR(1)$ or $LALR(1)$, which implies that it has no shift/reduce or reduce/reduce conflicts. We can use the DFA DCG accepting C_G recursively to parse $L(G)$. The function CG is defined as follows: Given any string $\mu \in V^*$,

$$CG(\mu) = \begin{cases} error & \text{if } \delta^*(q_0, \mu) = error; \\ (\delta^*(q_0, \theta), \theta, v) & \text{if } \delta^*(q_0, \theta) \in F,\ \mu = \theta v \text{ and } \theta \text{ is the} \\ & \text{shortest prefix of } \mu \text{ s.t. } \delta^*(q_0, \theta) \in F. \end{cases}$$

The naive shift-reduce algorithm is shown below:

```
begin
  $accept$ := true;
  $stop$ := false;
  $\mu := w\$$;   {input string}
  while $\neg stop$ do
    if $CG(\mu) = error$ then
      $stop$ := true; $accept$ := false
    else
      Let $(q, \theta, v) = CG(\mu)$
      Let $B \rightarrow \beta$ be the production so that
      action$(q, \mathrm{FIRST}(v)) = B \rightarrow \beta$ and let $\theta = \alpha\beta$
      if $B \rightarrow \beta = S' \rightarrow S$ then
        $stop$ := true
      else
```

```
          $\mu := \alpha B v$   {reduction}
        endif
      endif
    endwhile
end
```

The idea is to recursively run the DFA DCG on the sentential form μ, until the first final state q is hit. Then, the sentential form μ must be of the form $\alpha\beta v$, where v is a terminal string ending in \$, and the final state q contains a reduce item of the form $B \longrightarrow \beta$, with $\text{action}(q, \text{FIRST}(v)) = B \longrightarrow \beta$. Thus, we can reduce $\mu = \alpha\beta v$ to $\alpha B v$, since we have found a rightmost derivation step, and repeat the process.

Note that the major inefficiency of the algorithm is that when a reduction is performed, the prefix α of μ is reparsed entirely by DCG. Since DCG is deterministic, the sequence of states obtained on input α is uniquely determined. If we keep the sequence of states produced on input θ by DCG in a stack, then it is possible to avoid reparsing α. Indeed, all we have to do is update the stack so that just before applying DCG to $\alpha A v$, the sequence of states in the stack is the sequence obtained after parsing α. This stack is obtained by popping the $|\beta|$ topmost states and performing an update which is just a goto move. This is the standard version of the shift/reduce algorithm!

8.5 The Graph Method for Computing Fixed Points

Let X be a finite set representing the domain of the problem (in Section 8.3 above, $X = \Sigma$), let $F(1), \ldots, F(N)$ be N sets to be computed and let $I(1), \ldots, I(N)$ be N given subsets of X. The sets $I(1), \ldots, I(N)$ are the initial sets. We also have a directed graph G whose set of nodes is $\{1, \ldots, N\}$ and which represents relationships among the sets $F(i)$, where $1 \leq i \leq N$. The graph G has no parallel edges and no loops, but it may have cycles. If there is an edge from i to j, this is denoted by iGj (note that the absense of loops means that iGi never holds). Also, the existence of a path from i to j is denoted by iG^+j. The graph G represents a relation, and G^+ is the graph of the transitive closure of this relation. The existence of a path from i to j, including the null path, is denoted by iG^*j. Hence, G^* is the reflexive and transitive closure of G. We want to solve for the least solution of the system of recursive equations:

$$F(i) = I(i) \cup \{F(j) \mid iGj,\, i \neq j\}, \quad 1 \leq i \leq N.$$

Since $(2^X)^N$ is a complete lattice under the inclusion ordering (which means that every family of subsets has a least upper bound, namely, the union of this family), it is an ω-complete poset, and since the function $F\colon (2^X)^N \to (2^X)^N$ induced by the system of equations is easily seen to preserve least upper bounds of ω-chains, the least solution of the system can be computed by the standard fixed point technique (as explained in Section 3.7

of the class notes). We simply compute the sequence of approximations $(F^k(1), \ldots, F^k(N))$, where

$$F^0(i) = \emptyset, \quad 1 \leq i \leq N,$$

and

$$F^{k+1}(i) = I(i) \cup \bigcup \{F^k(j) \mid iGj,\, i \neq j\}, \quad 1 \leq i \leq N.$$

It is easily seen that we can stop at $k = N - 1$, and the least solution is given by

$$F(i) = F^1(i) \cup F^2(i) \cup \cdots \cup F^N(i), \quad 1 \leq i \leq N.$$

However, the above expression can be simplified to

$$F(i) = \bigcup \{I(j) \mid iG^*j\}, \quad 1 \leq i \leq N.$$

This last expression shows that in order to compute $F(i)$, it is necessary to compute the union of all the initial sets $I(j)$ reachable from i (including i). Hence, any transitive closure algorithm or graph traversal algorithm will do. For simplicity and for pedagogical reasons, we use a depth-first search algorithm.

Going back to FIRST, we see that all we have to do is to compute the INITFIRST sets, the graph GFIRST, and then use the graph traversal algorithm. The graph GFIRST is computed as follows: The nodes are the nonterminals and there is an edge from A to B ($A \neq B$) if and only if there is a production of the form $A \longrightarrow B\alpha$, for some $\alpha \in V^*$.

Example 1. Computation of the FIRST sets for the grammar G_1 given by the rules:

$$\begin{array}{lcl} S & \longrightarrow & E\$ \\ E & \longrightarrow & E + T \\ E & \longrightarrow & T \\ T & \longrightarrow & T * F \\ T & \longrightarrow & F \\ F & \longrightarrow & (E) \\ F & \longrightarrow & -T \\ F & \longrightarrow & a. \end{array}$$

We get

$$\text{INITFIRST}(E) = \emptyset, \quad \text{INITFIRST}(T) = \emptyset, \quad \text{INITFIRST}(F) = \{(, -, a\}.$$

The graph GFIRST is shown in Figure 8.9.

We obtain the following FIRST sets:

$$\text{FIRST}(E) = \text{FIRST}(T) = \text{FIRST}(F) = \{(, -, a\}.$$

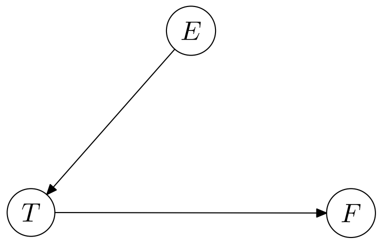


Figure 8.9: Graph GFIRST for G_1

8.6 Computation of FOLLOW

Recall the definition of FOLLOW(A) for a nonterminal A:

$$\text{FOLLOW}(A) = \{a \mid a \in \Sigma,\ S \overset{+}{\Longrightarrow} \alpha A a \beta, \quad \text{for some } \alpha, \beta \in V^*\}.$$

Note that a is in FOLLOW(A) if either a is in

$$\text{INITFOLLOW}(A) = \{a \mid a \in \Sigma,\ B \longrightarrow \alpha A X \beta \in P,\ a \in \text{FIRST}(X),\ \alpha, \beta \in V^*\}$$

or a is in

$$\{a \mid a \in \text{FOLLOW}(B),\ B \longrightarrow \alpha A \in P,\ \alpha \in V^*,\ A \neq B\}.$$

Indeed, if $S \overset{+}{\Longrightarrow} \lambda B a \rho$, then $S \overset{+}{\Longrightarrow} \lambda B a \rho \Longrightarrow \lambda \alpha A a \rho$, and so,

$$\text{FOLLOW}(B) \subseteq \text{FOLLOW}(A)$$

whenever $B \longrightarrow \alpha A$ is in P, with $A \neq B$. Hence, the FOLLOW sets are the least solution of the set of recursive equations: For all nonterminals A,

$$\text{FOLLOW}(A) = \text{INITFOLLOW}(A) \cup \bigcup \{\text{FOLLOW}(B) \mid B \longrightarrow \alpha A \in P,\ \alpha \in V^*,\ A \neq B\}.$$

According to the method explained above, we just have to compute the INITFOLLOW sets (using FIRST) and the graph GFOLLOW, which is computed as follows: The nodes are the nonterminals and there is an edge from A to B ($A \neq B$) if and only if there is a production of the form $B \longrightarrow \alpha A$ in P, for some $\alpha \in V^*$. Note the duality between the construction of the graph GFIRST and the graph GFOLLOW.

Example 2. Computation of the FOLLOW sets for the grammar G_1.

$$\text{INITFOLLOW}(E) = \{+,), \$\},\ \text{INITFOLLOW}(T) = \{*\},\ \text{INITFOLLOW}(F) = \emptyset.$$

The graph GFOLLOW is shown in Figure 8.10. We have

$$\begin{aligned}
\text{FOLLOW}(E) &= \text{INITFOLLOW}(E),\\
\text{FOLLOW}(T) &= \text{INITFOLLOW}(T) \cup \text{INITFOLLOW}(E) \cup \text{INITFOLLOW}(F),\\
\text{FOLLOW}(F) &= \text{INITFOLLOW}(F) \cup \text{INITFOLLOW}(T) \cup \text{INITFOLLOW}(E),
\end{aligned}$$

and so

$$\text{FOLLOW}(E) = \{+,), \$\}, \quad \text{FOLLOW}(T) = \{+, *,), \$\}, \quad \text{FOLLOW}(F) = \{+, *,), \$\}.$$

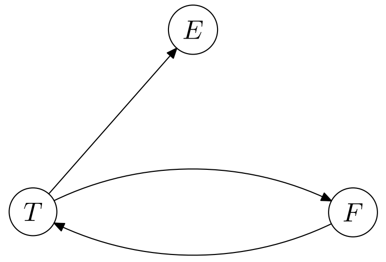


Figure 8.10: Graph GFOLLOW for G_1

8.7 Algorithm *Traverse*

The input is a directed graph Gr having N nodes, and a family of initial sets $I[i]$, $1 \leq i \leq N$. We assume that a function *successors* is available, which returns for each node n in the graph, the list $successors[n]$ of all immediate successors of n. The output is the list of sets $F[i]$, $1 \leq i \leq N$, solution of the system of recursive equations of Section 8.5. Hence,

$$F[i] = \bigcup \{I[j] \mid iG^*j\}, \quad 1 \leq i \leq N.$$

The procedure *Reachable* visits all nodes reachable from a given node. It uses a stack $STACK$ and a boolean array $VISITED$ to keep track of which nodes have been visited. The procedures *Reachable* and *traverse* are shown in Figure 8.11.

8.8 More on $LR(0)$-Characteristic Automata

Let $G = (V, \Sigma, P, S')$ be an augmented context-free grammar with augmented start production $S' \longrightarrow S\$$ (where S' only occurs in the augmented production). The righmost derivation relation is denoted by $\underset{rm}{\Longrightarrow}$.

Recall that the *set* C_G *of characteristic strings* for the grammar G is defined by

$$C_G = \{\alpha\beta \in V^* \mid S' \overset{*}{\underset{rm}{\Longrightarrow}} \alpha A v \underset{rm}{\Longrightarrow} \alpha\beta v,\ \alpha\beta \in V^*,\ v \in \Sigma^*\}.$$

The fundamental property of LR-parsing, due to D. Knuth, is stated in the following theorem:

Theorem 8.1. *Let G be a context-free grammar and assume that every nonterminal derives some terminal string. The language C_G (over V^*) is a regular language. Furthermore, a deterministic automaton DCG accepting C_G can be constructed from G.*

The construction of DCG can be found in various places, including the book on Compilers by Aho, Sethi and Ullman. We explained this construction in Section 8.1. The proof that the

```
**Procedure** $Reachable(Gr : graph; startnode : node; I : listofsets;$
                                 **var**$F : listofsets);$
**var** $currentnode, succnode, i : node; STACK : stack;$
                                       $VISITED$ : **array**$[1..N]$ **of boolean**;
  **begin**
    **for** $i := 1$ **to** $N$ **do**
      $VISITED[i] :=$ **false**;
    $STACK := EMPTY;$
    $push(STACK, startnode);$
    **while** $STACK \neq EMPTY$ **do**
      **begin**
        $currentnode := top(STACK); pop(STACK);$
        $VISITED[currentnode] :=$ **true**;
        **for each** $succnode \in successors(currentnode)$ **do**
          **if** $\neg VISITED[succnode]$ **then**
            **begin**
              $push(STACK, succnode);$
              $F[startnode] := F[startnode] \cup I[succnode]$
            **end**
      **end**
  **end**
```

The sets $F[i]$, $1 \leq i \leq N$, are computed as follows:

```
**begin**
  **for** $i := 1$ **to** $N$ **do**
    $F[i] := I[i];$
  **for** $startnode := 1$ **to** $N$ **do**
    $Reachable(Gr, startnode, I, F)$
**end**
```

Figure 8.11: Algorithm *traverse*

NFA NCG constructed as indicated in Section 8.1 is correct, i.e., that it accepts precisely C_G, is nontrivial, but not really hard either. This will be the object of a homework assignment! However, note a subtle point: The construction of NCG is only correct under the assumption that every nonterminal derives some terminal string. Otherwise, the construction could yield an NFA NCG accepting strings **not in** C_G.

Recall that the states of the characteristic automaton CGA are sets of *items* (or *marked productions*), where an item is a production with a dot anywhere in its right-hand side. Note that in constructing CGA, it is not necessary to include the state $\{S' \longrightarrow S\$.\}$ (the endmarker \$ is only needed to compute the lookahead sets). If a state p contains a marked production of the form $A \longrightarrow \beta.$, where the dot is the rightmost symbol, state p is called a *reduce state* and $A \longrightarrow \beta$ is called a *reducing production* for p. Given any state q, we say that a string $\beta \in V^*$ *accesses* q if there is a path from some state p to the state q on input β in the automaton CGA. Given any two states $p, q \in CGA$, for any $\beta \in V^*$, if there is a sequence of transitions in CGA from p to q on input β, this is denoted by

$$p \xrightarrow{\beta} q.$$

The initial state which is the closure of the item $S' \longrightarrow .S\$$ is denoted by 1. The LALR(1)-lookahead sets are defined in the next section.

8.9 LALR(1)-Lookahead Sets

For any reduce state q and any reducing production $A \longrightarrow \beta$ for q, let

$$\mathrm{LA}(q, A \longrightarrow \beta) = \{a \mid a \in \Sigma,\ S' \underset{rm}{\overset{*}{\Longrightarrow}} \alpha A a v \underset{rm}{\Longrightarrow} \alpha\beta a v,\ \alpha, \beta \in V^*,\ v \in \Sigma^*,\ \alpha\beta \textit{ accesses } q\}.$$

In words, $LA(q, A \longrightarrow \beta)$ consists of the terminal symbols for which the reduction by production $A \longrightarrow \beta$ in state q is the correct action (that is, for which the parse will terminate successfully). The LA sets can be computed using the FOLLOW sets defined below.

For any state p and any nonterminal A, let

$$\mathrm{FOLLOW}(p, A) = \{a \mid a \in \Sigma,\ S' \underset{rm}{\overset{*}{\Longrightarrow}} \alpha A a v,\ \alpha \in V^*,\ v \in \Sigma^* \text{ and } \alpha \text{ accesses } p\}.$$

Since for any derivation

$$S' \underset{rm}{\overset{*}{\Longrightarrow}} \alpha A a v \underset{rm}{\Longrightarrow} \alpha\beta a v$$

where $\alpha\beta$ accesses q, there is a state p such that $p \xrightarrow{\beta} q$ and α accesses p, it is easy to see that the following result holds:

Proposition 8.2. *For every reduce state q and any reducing production $A \longrightarrow \beta$ for q, we have*

$$\mathrm{LA}(q, A \longrightarrow \beta) = \bigcup \{\mathrm{FOLLOW}(p, A) \mid p \xrightarrow{\beta} q\}.$$

Also, we let

$$\mathrm{LA}(\{S' \longrightarrow S.\$\}, S' \longrightarrow S\$) = \mathrm{FOLLOW}(1, S).$$

Intuitively, when the parser makes the reduction by production $A \longrightarrow \beta$ in state q, each state p as above is a possible top of stack after the states corresponding to β are popped. Then the parser must read A in state p, and the next input symbol will be one of the symbols in FOLLOW(p, A).

The computation of FOLLOW(p, A) is similar to that of FOLLOW(A). First, we compute INITFOLLOW(p, A), given by

$$\mathrm{INITFOLLOW}(p, A) = \{a \mid a \in \Sigma,\ \exists q, r,\ p \xrightarrow{A} q \xrightarrow{a} r\}.$$

These are the terminals that can be read in CGA after the "goto transition" on nonterminal A has been performed from p. These sets can be easily computed from CGA.

Note that for the state p whose core item is $S' \longrightarrow S.\$$, we have

$$\mathrm{INITFOLLOW}(p, S) = \{\$\}.$$

Next, observe that if $B \longrightarrow \alpha A$ is a production and if

$$S' \overset{*}{\underset{rm}{\Longrightarrow}} \lambda B a v$$

where λ accesses p', then

$$S' \overset{*}{\underset{rm}{\Longrightarrow}} \lambda B a v \underset{rm}{\Longrightarrow} \lambda \alpha A a v$$

where λ accesses p' and $p' \xrightarrow{\alpha} p$. Hence $\lambda\alpha$ accesses p and

$$\mathrm{FOLLOW}(p', B) \subseteq \mathrm{FOLLOW}(p, A)$$

whenever there is a production $B \longrightarrow \alpha A$ and $p' \xrightarrow{\alpha} p$. From this, the following recursive equations are easily obtained: For all p and all A,

$$\begin{aligned}\mathrm{FOLLOW}(p, A) = {} & \mathrm{INITFOLLOW}(p, A) \cup \\ & \bigcup \{\mathrm{FOLLOW}(p', B) \mid B \longrightarrow \alpha A \in P,\ \alpha \in V^* \text{ and } p' \xrightarrow{\alpha} p\}.\end{aligned}$$

From Section 8.5, we know that these sets can be computed by using the algorithm *traverse*. All we need is to compute the graph GLA.

The nodes of the graph GLA are the pairs (p, A), where p is a state and A is a nonterminal. There is an edge from (p, A) to (p', B) if and only if there is a production of the form $B \longrightarrow \alpha A$ in P for some $\alpha \in V^*$ and $p' \xrightarrow{\alpha} p$ in CGA. Note that it is only necessary to consider nodes (p, A) for which there is a nonterminal transition on A from p. Such pairs can be obtained from the parsing table. Also, using the *spelling property*, that is, the fact that all transitions entering a given state have the same label, it is possible to compute the relation *lookback* defined as follows:

$$(q, A)\ \textit{lookback}\ (p, A) \quad \text{iff} \quad p \xrightarrow{\beta} q$$

for some reduce state q and reducing production $A \longrightarrow \beta$. The above considerations show that the FOLLOW sets of Section 8.6 are obtained by ignoring the state component from FOLLOW(p, A). We now consider the changes that have to be made when ϵ-rules are allowed.

8.10 Computing FIRST, FOLLOW, etc. in the Presence of ϵ-Rules

[Computing FIRST, FOLLOW and LA$(q, A \longrightarrow \beta)$ in the Presence of ϵ-Rules] First, it is necessary to compute the set E of *erasable nonterminals*, that is, the set of nonterminals A such that $A \overset{+}{\Longrightarrow} \epsilon$.

We let E be a boolean array and *change* be a boolean flag. An algorithm for computing E is shown in Figure 8.12. Then, in order to compute FIRST, we compute

$$\begin{aligned}\text{INITFIRST}(A) = \{a \mid{} & a \in \Sigma,\ A \longrightarrow a\alpha \in P, \text{ or} \\ & A \longrightarrow A_1 \cdots A_k a\alpha \in P, \text{ for some } \alpha \in V^*, \text{ and } E(A_1) = \cdots = E(A_k) = \textbf{true}\}.\end{aligned}$$

The graph GFIRST is obtained as follows: The nodes are the nonterminals, and there is an edge from A to B if and only if either there is a production $A \longrightarrow B\alpha$, or a production $A \longrightarrow A_1 \cdots A_k B\alpha$, for some $\alpha \in V^*$, with $E(A_1) = \cdots = E(A_k) = \textbf{true}$. Then, we extend FIRST to strings in V^+, in the obvious way. Given any string $\alpha \in V^+$, if $|\alpha| = 1$, then $\beta = X$ for some $X \in V$, and

$$\text{FIRST}(\beta) = \text{FIRST}(X)$$

as before, else if $\beta = X_1 \cdots X_n$ with $n \geq 2$ and $X_i \in V$, then

$$\text{FIRST}(\beta) = \text{FIRST}(X_1) \cup \cdots \cup \text{FIRST}(X_k),$$

where k, $1 \leq k \leq n$, is the largest integer so that

$$E(X_1) = \cdots = E(X_k) = \textbf{true}.$$

To compute FOLLOW, we first compute

$$\text{INITFOLLOW}(A) = \{a \mid a \in \Sigma,\ B \longrightarrow \alpha A\beta \in P,\ \alpha \in V^*,\ \beta \in V^+, \text{ and } a \in \text{FIRST}(\beta)\}.$$

The graph GFOLLOW is computed as follows: The nodes are the nonterminals. There is an edge from A to B if either there is a production of the form $B \longrightarrow \alpha A$, or $B \longrightarrow \alpha A A_1 \cdots A_k$, for some $\alpha \in V^*$, and with $E(A_1) = \cdots = E(A_k) = \textbf{true}$.

```
begin
    for each nonterminal $A$ do
      $E(A)$ := false;
    for each nonterminal $A$ such that $A \longrightarrow \epsilon \in P$ do
      $E(A)$ := true;
    change := true;
    while change do
      begin
        change := false;
        for each $A \longrightarrow A_1 \cdots A_n \in P$
            s.t. $E(A_1) = \cdots = E(A_n) =$ true do
          if $E(A) =$ false then
            begin
              $E(A)$ := true;
              change := true
            end
      end
end
```

Figure 8.12: Algorithm for computing E

The computation of the LALR(1) lookahead sets is also more complicated because another graph is needed in order to compute INITFOLLOW(p, A). First, the graph GLA is defined in the following way: The nodes are still the pairs (p, A), as before, but there is an edge from (p, A) to (p', B) if and only if either there is some production $B \longrightarrow \alpha A$, for some $\alpha \in V^*$ and $p' \xrightarrow{\alpha} p$, or a production $B \longrightarrow \alpha A\beta$, for some $\alpha \in V^*$, $\beta \in V^+$, $\beta \overset{+}{\Longrightarrow} \epsilon$, and $p' \xrightarrow{\alpha} p$. The sets INITFOLLOW$(p, A)$ are computed in the following way: First, let

$$\mathrm{DR}(p, A) = \{a \mid a \in \Sigma,\ \exists q, r,\ p \xrightarrow{A} q \xrightarrow{a} r\}.$$

The sets DR(p, A) are the *direct read* sets. Note that for the state p whose core item is $S' \longrightarrow S.\$$, we have

$$\mathrm{DR}(p, S) = \{\$\}.$$

Then,

$$\begin{aligned}\mathrm{INITFOLLOW}(p, A) = \mathrm{DR(p, A)} \cup \\ \bigcup \{a \mid a \in \Sigma,\ S' \underset{rm}{\overset{*}{\Longrightarrow}} \alpha A\beta av \underset{rm}{\Longrightarrow} \alpha Aav,\ \alpha \in V^*,\ \beta \in V^+,\ \beta \overset{+}{\Longrightarrow} \epsilon,\ \alpha\, accesses\, p\}.\end{aligned}$$

The set INITFOLLOW(p, A) is the set of terminals that can be read before any handle containing A is reduced. The graph $GREAD$ is defined as follows: The nodes are the pairs (p, A), and there is an edge from (p, A) to (r, C) if and only if $p \xrightarrow{A} r$ and $r \xrightarrow{C} s$, for some s, with $E(C) = \mathbf{true}$.

Then, it is not difficult to show that the INITFOLLOW sets are the least solution of the set of recursive equations:

$$\mathrm{INITFOLLOW}(p, A) = DR(p, A) \cup \bigcup \{\mathrm{INITFOLLOW}(r, C) \mid (p, A)\, GREAD\, (r, C)\}.$$

Hence the INITFOLLOW sets can be computed using the algorithm traverse on the graph $GREAD$ and the sets $DR(p, A)$, and then, the FOLLOW sets can be computed using traverse again, with the graph GLA and sets INITFOLLOW. Finally, the sets LA$(q, A \longrightarrow \beta)$ are computed from the FOLLOW sets using the graph *lookback*.

From section 8.5, we note that $F(i) = F(j)$ whenever there is a path from i to j and a path from j to i, that is, whenever i and j are *strongly connected*. Hence, the solution of the system of recursive equations can be computed more efficiently by finding the maximal strongly connected components of the graph G, since F has a same value on each strongly connected component. This is the approach followed by DeRemer and Pennello in: Efficient Computation of LALR(1) Lookahead sets, by F. DeRemer and T. Pennello, *TOPLAS*, Vol. 4, No. 4, October 1982, pp. 615-649.

We now give an example of grammar which is LALR(1) but not SLR(1).

Example 3. The grammar G_2 is given by:

$$\begin{aligned}
S' &\longrightarrow S\$ \\
S &\longrightarrow L = R \\
S &\longrightarrow R \\
L &\longrightarrow *R \\
L &\longrightarrow id \\
R &\longrightarrow L
\end{aligned}$$

The states of the characteristic automaton CGA_2 are:

$$\begin{aligned}
1: S' &\longrightarrow .S\$ \\
S &\longrightarrow .L = R \\
S &\longrightarrow .R \\
L &\longrightarrow .*R \\
L &\longrightarrow .id \\
R &\longrightarrow .L \\
2: S' &\longrightarrow S.\$ \\
3: S &\longrightarrow L. = R \\
R &\longrightarrow L. \\
4: S &\longrightarrow R. \\
5: L &\longrightarrow *.R \\
R &\longrightarrow .L \\
L &\longrightarrow .*R \\
L &\longrightarrow .id \\
6: L &\longrightarrow id. \\
7: S &\longrightarrow L = .R \\
R &\longrightarrow .L \\
L &\longrightarrow .*R \\
L &\longrightarrow .id \\
8: L &\longrightarrow *R. \\
9: R &\longrightarrow L. \\
10: S &\longrightarrow L = R.
\end{aligned}$$

We find that

$$\begin{aligned}
\text{INITFIRST}(S) &= \emptyset \\
\text{INITFIRST}(L) &= \{*, id\} \\
\text{INITFIRST}(R) &= \emptyset.
\end{aligned}$$

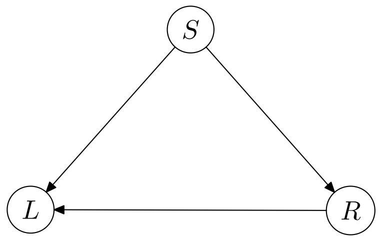


Figure 8.13: The graph GFIRST

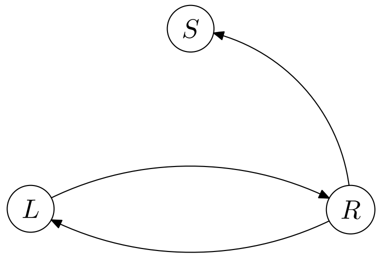


Figure 8.14: The graph GFOLLOW

The graph GFIRST is shown in Figure 8.13.
Then, we find that

$$\begin{aligned}
\text{FIRST}(S) &= \{*, id\}\\
\text{FIRST}(L) &= \{*, id\}\\
\text{FIRST}(R) &= \{*, id\}.
\end{aligned}$$

We also have

$$\begin{aligned}
\text{INITFOLLOW}(S) &= \{\$\}\\
\text{INITFOLLOW}(L) &= \{=\}\\
\text{INITFOLLOW}(R) &= \emptyset.
\end{aligned}$$

The graph GFOLLOW is shown in Figure 8.14.
Then, we find that

$$\begin{aligned}
\text{FOLLOW}(S) &= \{\$\}\\
\text{FOLLOW}(L) &= \{=, \$\}\\
\text{FOLLOW}(R) &= \{=, \$\}.
\end{aligned}$$

Note that there is a shift/reduce conflict in state 3 on input $=$, since there is a shift on input $=$ (since $S \longrightarrow L. = R$ is in state 3), and a reduce for $R \to L$, since $=$ is in

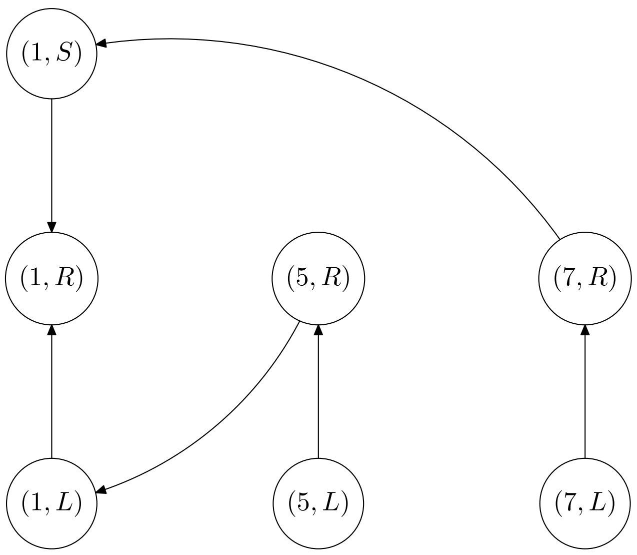


Figure 8.15: The graph GLA

FOLLOW(R). However, as we shall see, the conflict is resolved if the LALR(1) lookahead sets are computed.

The graph GLA is shown in Figure 8.15.

We get the following INITFOLLOW and FOLLOW sets:

$$\begin{array}{rcrc}
\text{INITFOLLOW}(1, S) = & \{\$\} & \text{INITFOLLOW}(1, S) = & \{\$\} \\
\text{INITFOLLOW}(1, R) = & \emptyset & \text{INITFOLLOW}(1, R) = & \{\$\} \\
\text{INITFOLLOW}(1, L) = & \{=\} & \text{INITFOLLOW}(1, L) = & \{=, \$\} \\
\text{INITFOLLOW}(5, R) = & \emptyset & \text{INITFOLLOW}(5, R) = & \{=, \$\} \\
\text{INITFOLLOW}(5, L) = & \emptyset & \text{INITFOLLOW}(5, L) = & \{=, \$\} \\
\text{INITFOLLOW}(7, R) = & \emptyset & \text{INITFOLLOW}(7, R) = & \{\$\} \\
\text{INITFOLLOW}(7, L) = & \emptyset & \text{INITFOLLOW}(7, L) = & \{\$\}.
\end{array}$$

Thus, we get

$$\begin{array}{rcl}
\text{LA}(2, S' \longrightarrow S\$) & = & \text{FOLLOW}(1, S) = \{\$\} \\
\text{LA}(3, R \longrightarrow L) & = & \text{FOLLOW}(1, R) = \{\$\} \\
\text{LA}(4, S \longrightarrow R) & = & \text{FOLLOW}(1, S) = \{\$\} \\
\text{LA}(6, L \longrightarrow id) & = & \text{FOLLOW}(1, L) \cup \text{FOLLOW}(5, L) \cup \text{FOLLOW}(7, L) = \{=, \$\} \\
\text{LA}(8, L \longrightarrow *R) & = & \text{FOLLOW}(1, L) \cup \text{FOLLOW}(5, L) \cup \text{FOLLOW}(7, L) = \{=, \$\} \\
\text{LA}(9, R \longrightarrow L) & = & \text{FOLLOW}(5, R) \cup \text{FOLLOW}(7, R) = \{=, \$\} \\
\text{LA}(10, S \longrightarrow L = R) & = & \text{FOLLOW}(1, S) = \{\$\}.
\end{array}$$

Since LA($3, R \longrightarrow L$) does not contain =, the conflict is resolved.

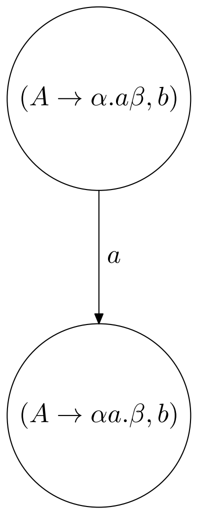


Figure 8.16: Transition on terminal input a

8.11 $LR(1)$-Characteristic Automata

We conclude this brief survey on LR-parsing by describing the construction of $LR(1)$-parsers. The new ingredient is that when we construct an NFA accepting C_G, we incorporate lookahead symbols into the states. Thus, a state is a pair $(A \longrightarrow \alpha.\beta, b)$, where $A \longrightarrow \alpha.\beta$ is a marked production, as before, and $b \in \Sigma \cup \{\$\}$ is a *lookahead symbol*. The new twist in the construction of the nondeterministic characteristic automaton is the following:

The start state is $(S' \to .S, \$)$, and the transitions are defined as follows:

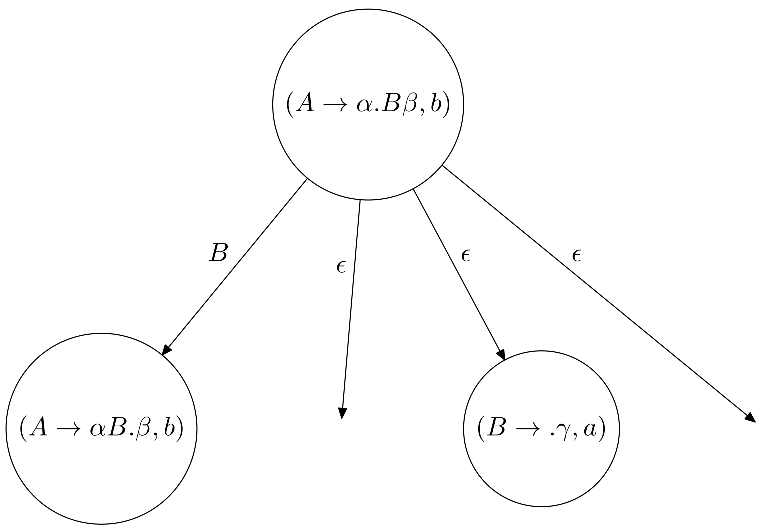
(a) For every terminal $a \in \Sigma$, then there is a transition on input a from state $(A \to \alpha.a\beta, b)$ to the state $(A \to \alpha a.\beta, b)$ obtained by "shifting the dot" (where $a = b$ is possible). Such a transition is shown in Figure 8.16.

(b) For every nonterminal $B \in N$, there is a transition on input B from state $(A \to \alpha.B\beta, b)$ to state $(A \to \alpha B.\beta, b)$ (obtained by "shifting the dot"), and transitions on input ϵ (the empty string) to all states $(B \to .\gamma, a)$, for all productions $B \to \gamma$ with left-hand side B and all $a \in \text{FIRST}(\beta b)$. Such transitions are shown in Figure 8.17.

(c) A state is *final* if and only if it is of the form $(A \to \beta., b)$ (that is, the dot is in the rightmost position).

Example 3. Consider the grammar G_3 given by:

$$\begin{aligned} 0\colon S &\longrightarrow E \\ 1\colon E &\longrightarrow aEb \\ 2\colon E &\longrightarrow \epsilon \end{aligned}$$



$(A \rightarrow \alpha.B\beta, b)$
B
ϵ
ϵ
ϵ
$(A \rightarrow \alpha B.\beta, b)$
$(B \rightarrow .\gamma, a)$

Figure 8.17: Transitions from a state $(A \rightarrow \alpha.B\beta, b)$

The result of making the NFA for C_{G_3} deterministic is shown in Figure 8.18 (where transitions to the "dead state" have been omitted). The internal structure of the states $1, \ldots, 8$ is shown below:

$$\begin{array}{rcl}
1: S & \longrightarrow & .E, \$ \\
E & \longrightarrow & .aEb, \$ \\
E & \longrightarrow & ., \$ \\
2: E & \longrightarrow & a.Eb, \$ \\
E & \longrightarrow & .aEb, b \\
E & \longrightarrow & ., b \\
3: E & \longrightarrow & a.Eb, b \\
E & \longrightarrow & .aEb, b \\
E & \longrightarrow & ., b \\
4: E & \longrightarrow & aE.b, \$ \\
5: E & \longrightarrow & aEb., \$ \\
6: E & \longrightarrow & aE.b, b \\
7: E & \longrightarrow & aEb., b \\
8: S & \longrightarrow & E., \$
\end{array}$$

The $LR(1)$-shift/reduce parser associated with DCG is built as follows: The shift and goto entries come directly from the transitions of DCG, and for every state s, for every item

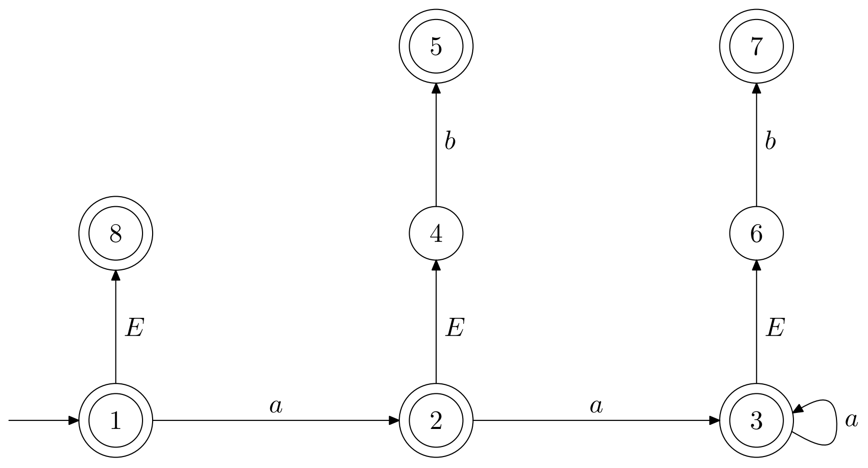


Figure 8.18: DFA for C_{G_3}

$(A \longrightarrow \gamma, b)$ in s, enter an entry rn for state s and input b, where $A \longrightarrow \gamma$ is production number n. If the resulting parser has no conflicts, we say that the grammar is an $LR(1)$ grammar. The $LR(1)$-shift/reduce parser for G_3 is shown below:

	a	b	$\$$	E
1	$s2$		$r2$	8
2	$s3$	$r2$		4
3	$s3$	$r2$		6
4		$r5$		
5			$r1$	
6	$r1$	$s7$		
7		$r1$		
8			acc	

Observe that there are three pairs of states, $(2,3)$, $(4,6)$, and $(5,7)$, where both states in a common pair only differ by the lookahead symbols. We can merge the states corresponding to each pair, because the marked items are the same, but now, we have to allow lookahead sets. Thus, the merging of $(2,3)$ yields

$$\begin{array}{rcl}
2'\colon E & \longrightarrow & a.Eb, \{b, \$\} \\
E & \longrightarrow & .aEb, \{b\} \\
E & \longrightarrow & ., \{b\},
\end{array}$$

the merging of $(4,6)$ yields

$$3'\colon E \longrightarrow aE.b, \{b, \$\},$$

the merging of $(5,7)$ yields

$$4'\colon E \longrightarrow aEb., \{b, \$\}.$$

We obtain a merged DFA with only five states, and the corresponding shift/reduce parser is given below:

	a	b	$\$$	E
1	$s2'$		$r2$	8
$2'$	$s2'$	$r2$		$3'$
$3'$		$s4'$		
$4'$		$r1$	$r1$	
8			acc	

The reader should verify that this is the $LALR(1)$-parser. The reader should also check that that the $SLR(1)$-parser is given below:

	a	b	$\$$	E
1	$s2$	$r2$	$r2$	5
2	$s2$	$r2$	$r2$	3
3		$s4$		
4		$r1$	$r1$	
5			acc	

The difference between the two parsing tables is that the $LALR(1)$-lookahead sets are sharper than the $SLR(1)$-lookahead sets. This is because the computation of the $LALR(1)$-lookahead sets uses a sharper version of FOLLOW sets. It can also be shown that if a grammar is $LALR(1)$, then the merging of states of an $LR(1)$-parser always succeeds and yields the $LALR(1)$ parser. Of course, this is a very inefficient way of producing $LALR(1)$ parsers, and much better methods exist, such as the graph method described in these notes. However, there are cases where the merging fails. Sufficient conditions for successful merging have been investigated, but there is still room for research in this area.

Bibliography

[1] Pierre Brémaud. *Markov Chains, Gibbs Fields, Monte Carlo Simulations, and Queues.* TAM, Vol. 31. Springer Verlag, third edition, 2001.

[2] Erhan Cinlar. *Introduction to Stochastic Processes.* Dover, first edition, 2014.

[3] Geoffrey Grimmett and David Stirzaker. *Probability and Random Processes.* Oxford University Press, third edition, 2001.

[4] John G. Kemeny, Snell J. Laurie, and Anthony W. Knapp. *Denumerable Markov Chains.* GTM, Vol. No 40. Springer-Verlag, second edition, 1976.

[5] Michael Mitzenmacher and Eli Upfal. *Probability and Computing. Randomized Algorithms and Probabilistic Analysis.* Cambridge University Press, first edition, 2005.

[6] Lawrence R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[7] Elaine Rich. *Automata, Computability, and Complexity. Theory and Applications.* Prentice Hall, first edition, 2007.

[8] Mark Stamp. A revealing introduction to hidden markov models. Technical report, San Jose State University, Department of Computer Science, San Jose, California, 2015.