Introduction to the Theory of Computation
Some Slides

Jean Gallier

July 8, 2017

Chapter 1

Part I: Basics of Formal Language Theory

1.1 Generalities, Motivations, Problems

In this part of the course we want to understand

- What is a language?
- How do we define a language?
- How do we manipulate languages, combine them?
- What is the complexity of a language?

Roughly, there are two dual views of languages:

(A) The *recognition* point view.

(B) The *generation* point of view.

No matter how we view a language, we are typically considering two things:

(1) The *syntax*, i.e., what are the "legal" strings in that language (what are the "grammar rules"?).

(2) The *semantics* of strings in the language, i.e., what is the *meaning* (or *interpretation*) of a string.

The semantics is usually a lot more interesting than the syntax but unfortunately much more difficult to deal with!

Therefore, sorry, we will only be dealing with syntax!

In (A), we typically assume some kind of "black box", M, (an *automaton*) that takes a string, w, as input and returns two possible answers:

Yes, the string w is *accepted*, which means that w belongs to the language, L, that we are trying to define.

No, the string w is *rejected*, which means that w *does not* belong to the language, L.

Usually, the black box M gives a definite answer for every input after a finite number of steps, but not always.

For example, a Turing machine may go on computing forever and not give any answer for certain strings not in the language. This is an example of *undecidability*.

The black box may compute *deterministically* or *non-deterministically*, which means roughly that on input w, the machine M is allowed to try different computations and to ignore failing computations as long as there is some successful computation on input w.

This affects greatly the *complexity* of recognition, i.e,. how many steps it takes to process w.

Sometimes, a nondeterministic version of an automaton turns out to be equivalent to the deterministic version (although, with different complexity).

This tends to happen for very restrictive models—where nondeterminism does not help, or for very powerful models—where again, nondeterminism does not help, but because the deterministic model is already very powerful!

We will investigate automata of increasing power of recognition:

(1) Deterministic and nondeterministic finite automata (DFA's and NFA's, their power is the same).

(2) Pushdown automata (PDA's) and determinstic pushdown automata (DPDA's), here PDA $>$ DPDA.

(3) Deterministic and nondeterministic Turing machines (their power is the same).

(4) If time permits, we will also consider some restricted type of Turing machine known as LBA (linear bounded automaton).

In (B), we are interested in formalisms that specify a language in terms of *rules* that allow the generation of "legal" strings. The most common formalism is that of a formal *grammar*.

Remember:

- An automaton *recognizes* (or *accepts*) a language,
- a grammar *generates* a language.
- gramm*a*r is spelled with an "a" (not with an "e").
- The plural of automat*on* is automat*a* (not automat*ons*).

For "good" classes of grammars, it is possible to build an automaton, M_G, from the grammar, G, in the class, so that M_G recognizes the language, $L(G)$, generated by the grammar G.

However, grammars are nondeterministic in nature. Thus, even if we try to avoid nondeterministic automata, we usually can't escape having to deal with them.

We will investigate the following types of grammars (the so-called *Chomsky hierarchy*) and the corresponding families of languages:

(1) Regular grammars (type 3-languages).

(2) Context-free grammars (type 2-languages).

(3) The recursively enumerable languages or r.e. sets (type 0-languages).

(4) If time permit, context-sensitive languages (type 1-languages).

Miracle: The grammars of type (1), (2), (3), (4) correspond exactly to the automata of the corresponding type!

Furthermore, there are *algorithms* for converting grammars to the corresponding automata (and backward), although some of these algorithms are not practical.

Building an automaton from a grammar is an important practical problem in language processing. A lot is known for the regular and the context-free grammars, but there is still room for improvements and innovations!

There are other ways of defining families of languages, for example

Inductive closures.

In this style of definition, a collection of basic (atomic) languages is specified, some operations to combine languages are also specified, and the family of languages is defined as the smallest one containing the given atomic languages and closed under the operations.

Investigating closure properties (for example, union, intersection) is a way to assess how "robust" (or complex) a family of languages is.

Well, it is now time to be precise!

1.2 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 1.1. 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 1.2. 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 denote the string u as

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

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 1.3. 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:

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

Clearly, $u^1 = u$, and it is an easy exercise to show that

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

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 1.4. 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 1.5. 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:

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

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

$$reillag = gallier^R.$$

It can be shown that

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

Thus,

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

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 1.6. 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.

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.

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

If X is not the empty set, then X is countable iff there is a *surjection* from $\mathbb{N}$ onto X.

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

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.

If $\Sigma \neq \emptyset$, then the set Σ^* of all strings over Σ is infinite and countable.

Suppose $|\Sigma| = k$ with $\Sigma = \{a_1, \ldots, a_k\}$.

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 $(k^n - 1)/(k - 1)$,

$$\begin{aligned} \nu(u) &= i_1 k^{n-1} + i_2 k^{n-2} + \cdots + i_{n-1} k + i_n \\ &= \frac{k^n - 1}{k - 1} + (i_1 - 1)k^{n-1} + \cdots + (i_{n-1} - 1)k + i_n - 1. \end{aligned}$$

(with $\nu(\epsilon) = 0$).

We leave it as an exercise to show that $\nu \colon \Sigma^* \to \mathbb{N}$ is a bijection.

In fact, ν correspond to the enumeration of Σ^* where u precedes v if $|u| < |v|$, and u precedes v in the lexicographic ordering if $|u| = |v|$.

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

$$\epsilon,\ a,\ b,\ aa,\ ab,\ ba,\ bb.$$

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 show that there is no surjection from Σ^* onto 2^{Σ^*}.

But, if $\Sigma \neq \emptyset$, then Σ^* is infinite and countable, thus in bijection with $\mathbb{N}$, so there is no surjection from $\mathbb{N}$ onto 2^{Σ^*} either.

We use a *diagonalization* argument. This is an instance of *Cantor's Theorem*.

Assume there is a surjection $h\colon \Sigma^* \to 2^{\Sigma^*}$, and consider the set

$$D = \{u \in \Sigma^* \mid u \notin h(u)\}.$$

By definition, for any u we have $u \in D$ iff $u \notin h(u)$, so D can't be in the range of h, contradicting the fact that h is surjective.

Therefore, if $\Sigma \neq \emptyset$, then 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.

1.3 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 1.7. 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:

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

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 1.8. 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: $\Sigma = \{a, b, c\}$, $\Delta = \{0, 1\}$, 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 2

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.

2.1 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 make 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 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:

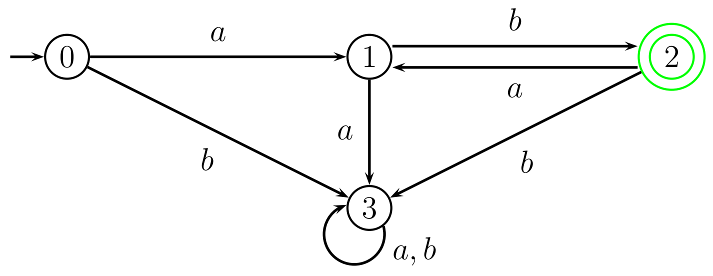


Figure 2.1: DFA for $\{ab\}^+$

Example 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: $F_2 = \{0\}$.

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:

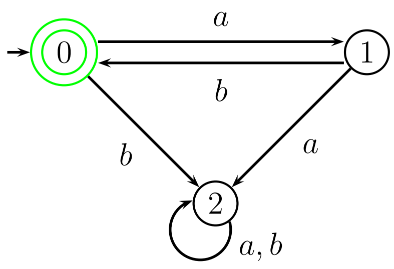


Figure 2.2: DFA for $\{ab\}^*$

Example 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:

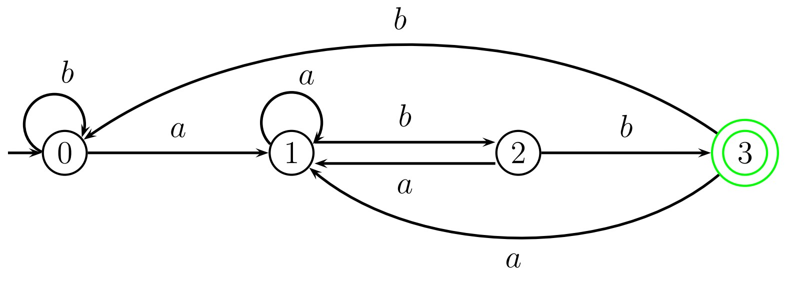


Figure 2.3: DFA for $\{a, b\}^*\{abb\}$

Is this a minimal DFA?

Definition 2.1. 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 2.2. 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 $u \in \Sigma^*$.

It is immediate that $\delta^*(p, a) = \delta(p, a)$ 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 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 2.3. 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.

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 set Q_r of *reachable* (or *accessible*) states as

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

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.

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
$D_r = (Q_r, \Sigma, \delta_r, q_0, Q_r \cap F)$, 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.

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

It can be shown that $L(D_r) = L(D)$ (see the homework problems).

A DFA D such that $Q = Q_r$ is said to be *trim* (or *reduced*).

Observe that the DFA D_r is trim. A minimal DFA must be trim.

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

$$L(D) \neq \emptyset \quad \text{iff} \quad Q_r \cap F \neq \emptyset$$

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

Regular Languages, Version 1

Definition 2.4. 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.

2.2 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, 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 $w \in \Sigma^*$.

Now, we have $w \in L(D_1) \cap L(D_2)$

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

Thus, $L(D) = L(D_1) \cap L(D_2)$.

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 $(F_1 \times Q_2) \cup (Q_1 \times F_2)$.

Indeed, $w \in L(D_1) \cup L(D_2)$

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

Thus, $L(D) = L(D_1) \cup L(D_2)$.

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 $F_1 \times (Q_2 - F_2)$.

Indeed, $w \in L(D_1) - L(D_2)$

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

Thus, $L(D) = L(D_1) - L(D_2)$.

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.

2.3 Morphisms, F-Maps, B-Maps and Homomorphisms of DFA's

A map between DFA's is a certain kind of graph homomorphism. The following Definition is adapted from Eilenberg.

Definition 2.5. Given any two DFA's $D_1 = (Q_1, \Sigma, \delta_1, q_{0,1}, F_1)$ and $D_2 = (Q_2, \Sigma, \delta_2, q_{0,2}, F_2)$, a *morphism $h\colon D_1 \to D_2$ of DFA's* is a function $h\colon Q_1 \to Q_2$ satisfying the following conditions:

(1)
$$h(\delta_1(p, a)) = \delta_2(h(p), a),$$
for all $p \in Q_1$ and all $a \in \Sigma$;

(2) $h(q_{0,1}) = q_{0,2}$.

Condition (1) can be expressed by the commutativity of the following diagram:

$$\begin{tikzcd}
p \arrow[r, "h"] \arrow[d, "a"'] & h(p) \arrow[d, "a"] \\
\delta_1(p, a) \arrow[r, "h"] & \delta_2(h(p), a)
\end{tikzcd}$$

An *F-map of DFA's*, for short, a *map*, is a morphism of DFA's $h\colon D_1 \to D_2$ that satisfies the condition

(3a) $h(F_1) \subseteq F_2$.

A *B-map of DFA's* is a morphism of DFA's $h\colon D_1 \to D_2$ that satisfies the condition

(3b) $h^{-1}(F_2) \subseteq F_1$.

A *proper homomorphism of DFA's*, for short, a *homomorphism*, is an F-map of DFA's that is also a B-map of DFA's.

Now, for any function $f\colon X \to Y$ and any two subsets $A \subseteq X$ and $B \subseteq Y$, recall that

$$\begin{aligned} f(A) &= \{f(a) \in Y \mid a \in A\} \\ f^{-1}(B) &= \{x \in X \mid f(x) \in B\} \end{aligned}$$

and

$$f(A) \subseteq B \quad \text{iff} \quad A \subseteq f^{-1}(B).$$

Thus, (3a) & (3b) is equivalent to the condition
(3c) $h^{-1}(F_2) = F_1$.

Note that the condition for being a proper homomorphism of DFA's is **not** equivalent to

$$h(F_1) = F_2.$$

Condition (3c) forces $h(F_1) = F_2 \cap h(Q_1)$, and furthermore, for every $p \in Q_1$, whenever $h(p) \in F_2$, then $p \in F_1$.

The reader should check that if $f\colon D_1 \to D_2$ and $g\colon D_2 \to D_3$ are morphisms (resp. F-maps, resp. B-maps), then $g \circ f\colon D_1 \to D_3$ is also a morphism (resp. an F-map, resp. a B-map).

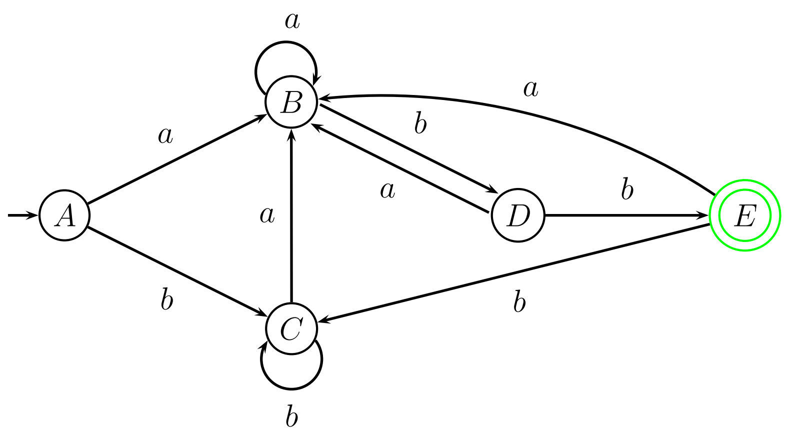
Note that an F-map or a B-map is a special case of the concept of *simulation* of automata. A proper homomorphism is a special case of a *bisimulation*.

Bisimulations play an important role in real-time systems and in concurrency theory.

Figure 2.3 shows a map, h, of DFA's, with

$$\begin{aligned} h(A) = h(C) &= 0 \\ h(B) &= 1 \\ h(D) &= 2 \\ h(E) &= 3. \end{aligned}$$

It is easy to check that h is actually a (proper) homomorphism.



$A \longrightarrow 0;\ B \longrightarrow 1;\ C \longrightarrow 0;\ D \longrightarrow 2;\ E \longrightarrow 3$

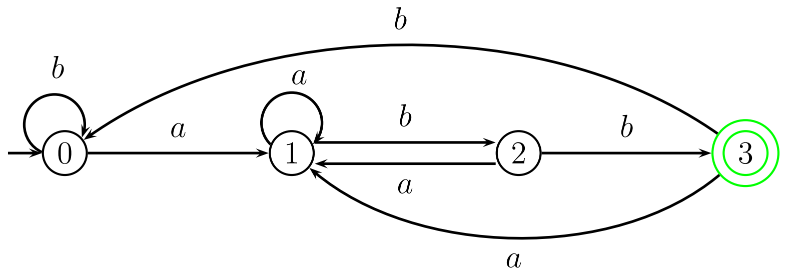


Figure 2.4: A map of DFA's

The DFA's of Figure 2.4 accept the same language.

The main motivation behind these definitions is that when there is an F-map $h\colon D_1 \to D_2$, somehow, D_2 simulates D_1, and it turns out that $L(D_1) \subseteq L(D_2)$.

When there is a B-map $h\colon D_1 \to D_2$, somehow, D_1 simulates D_2, and it turns out that $L(D_2) \subseteq L(D_1)$.

When there is a proper homomorphism $h\colon D_1 \to D_2$, somehow, D_1 bisimulates D_2, and it turns out that $L(D_2) = L(D_1)$.

A DFA morphism, $f\colon D_1 \to D_2$, is an *isomorphism* iff there is a DFA morphism, $g\colon D_2 \to D_1$, so that

$$g \circ f = \mathrm{id}_{D_1} \quad \text{and} \quad f \circ g = \mathrm{id}_{D_2}.$$

Similarly, an F-map $f\colon D_1 \to D_2$ is an *isomorphism* iff there is an F-map $g\colon D_2 \to D_1$, so that

$$g \circ f = \mathrm{id}_{D_1} \quad \text{and} \quad f \circ g = \mathrm{id}_{D_2}.$$

A B-map $f\colon D_1 \to D_2$ is an *isomorphism* iff there is a B-map $g\colon D_2 \to D_1$, so that

$$g \circ f = \mathrm{id}_{D_1} \quad \text{and} \quad f \circ g = \mathrm{id}_{D_2}.$$

The map g is unique and it is denoted f^{-1}.

The reader should prove that if a DFA F-map is an isomorphism, then it is also a proper homomorphism, and if a DFA B-map is an isomorphism, then it is also a proper homomorphism.

If $h\colon D_1 \to D_2$ is a morphism of DFA's, it is easily shown by induction on the length of w that

$$h(\delta_1^*(p, w)) = \delta_2^*(h(p), w),$$

for all $p \in Q_1$ and all $w \in \Sigma^*$.

As a consequence, we have the following proposition:

Proposition 2.1. *If $h\colon D_1 \to D_2$ is an F-map of DFA's, then $L(D_1) \subseteq L(D_2)$. If $h\colon D_1 \to D_2$ is a B-map of DFA's, then $L(D_2) \subseteq L(D_1)$. Finally, if $h\colon D_1 \to D_2$ is a proper homomorphism of DFA's, then $L(D_1) = L(D_2)$.*

A DFA is *accessible, or trim,* if every state is reachable from the start state.

A morphism (resp. F-map, B-map) $h\colon D_1 \to D_2$ is *surjective* if $h(Q_1) = Q_2$.

It can be shown that if D_1 is trim, then there is at most one morphism $h\colon D_1 \to D_2$ (resp. F-map, B-map). If D_2 is also trim and we have a morphism $h\colon D_1 \to D_2$, then h is surjective.

It can also be shown that a minimal DFA D_L for L is characterized by the property that there is unique surjective proper homomorphism $h\colon D \to D_L$ from any trim DFA D accepting L to D_L.

Another useful notion is the notion of a congruence on a DFA.

Definition 2.6. Given any DFA
$D = (Q, \Sigma, \delta, q_0, F)$, a *congruence $\equiv$ on D* is an equivalence relation $\equiv$ on Q satisfying the following conditions: For all $p, q \in Q$ and all $a \in \Sigma$,

(1) If $p \equiv q$, then $\delta(p, a) \equiv \delta(q, a)$.

(2) If $p \equiv q$ and $p \in F$, then $q \in F$.

It can be shown that a proper homomorphism of DFA's $h\colon D_1 \to D_2$ induces a congruence $\equiv_h$ on D_1 defined as follows:

$$p \equiv_h q \quad \text{iff} \quad h(p) = h(q).$$

Given a congruence $\equiv$ on a DFA D, we can define the *quotient DFA $D/\equiv$*, and there is a surjective proper homomorphism $\pi\colon D \to D/\equiv$.

We will come back to this point when we study minimal DFA's.

2.4 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 4. 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	$\emptyset$	$\emptyset$

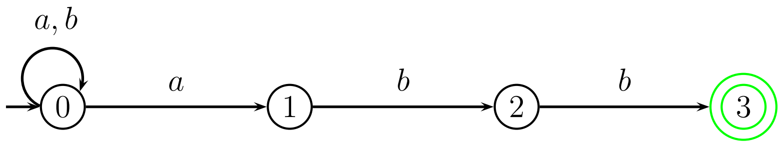


Figure 2.5: NFA for $\{a, b\}^*\{abb\}$

Example 5. Let $\Sigma = \{a_1, \ldots, a_n\}$, 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 2.7. 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*.

For example, given the NFA shown below,

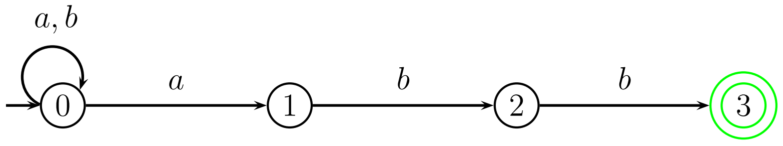


Figure 2.6: NFA for $\{a, b\}^*\{abb\}$

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

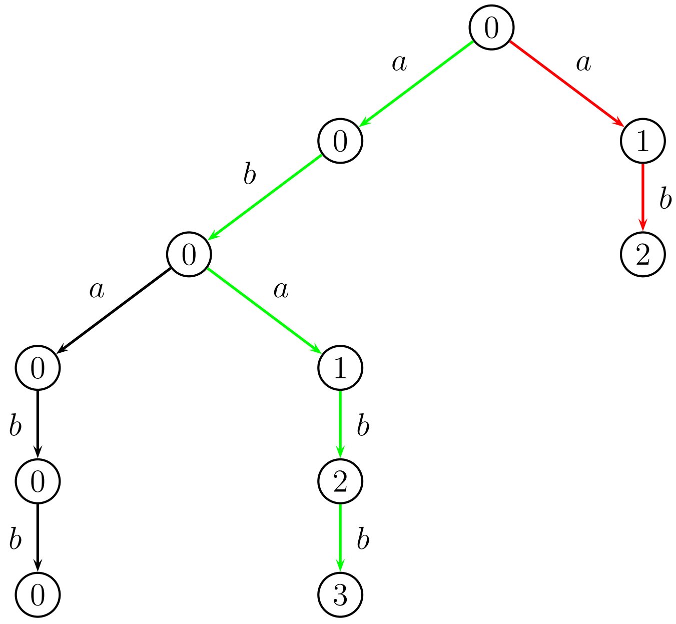


Figure 2.7: A tree of computation paths on input $ababb$

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.

2.5 ϵ-Closure

Definition 2.8. 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:

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

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 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).

Assume by contradiction that

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

Then, I claim that $|\epsilon\text{-clo}_i(p)| \geq i + 1$ for all $i \geq 0$.

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

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 hypothesis.

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

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

by proving 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 $\leq i$.

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 ϵ-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

$eclotype = \mathbf{array}[1..n]$ **of boolean**

```
**function** $eclosure[N\colon NFA, p\colon \textbf{integer}]\colon eclotype$;
    **begin**
        **var** $eclo\colon eclotype$, $q, s\colon$ **integer**, $st\colon stack$;
        **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).

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

2.6 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 2.9. 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 $a \in \Sigma$,

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

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*.

We can 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: 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

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

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 $\mathcal{Q}$):

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

with

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

Then, the DFA D is defined as follows:

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

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

It is not difficult to show that $L(D) = L(N)$, that is, D is a DFA accepting $L(N)$. For this, we show that

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

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 Rabin and Scott.

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 $a \in \Sigma$.

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

Let us illustrate the subset construction on the NFA of Example 4.

A NFA for the language

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

Transition table δ_4:

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

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

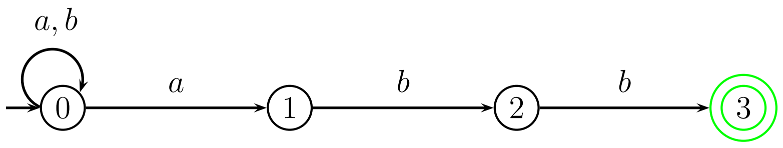


Figure 2.8: NFA for $\{a, b\}^*\{abb\}$

The pointer $\Rightarrow$ corresponds to *marked* and the pointer $\rightarrow$ to *total*.

Initial transition table Δ.

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

Just after entering the while loop

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

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	$\{0,1\}$		

After the second round through the while loop.

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

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	$\{0,3\}$		

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 2.3, except that in that example A, B, C, D are renamed $0, 1, 2, 3$.

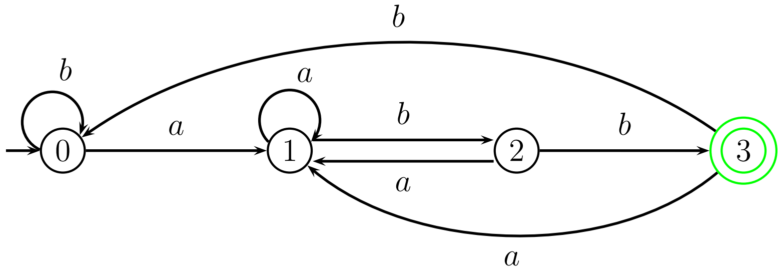


Figure 2.9: DFA for $\{a, b\}^*\{abb\}$