Regular languages, regular expressions and finite automata

“Regular” languages are relatively simple languages.

We’ll study means for “generating” regular languages and also for “recognizing” them.

All finite languages are regular.

Some infinite languages are regular.

Each regular language can be characterized by a (finite!) regular expression: which says how to generate the strings in the language.

It can also be characterized by a (finite!) finite state automaton: which provides a mechanism for recognizing the strings in the language.

Later we’ll study the larger class of context-free languages and the associated generation and recognition tools: namely, context-free grammar and pushdown automata.

Regular expressions

We can similarly define the set of regular expressions over Σ, and what each of them stands for.

Definition  The set of regular expressions over Σ is the smallest set s.t.

1. Λ is a regular expression, and stands for the regular language Λ.
   Also, Λ is a regular expression, and stands for the regular language {Λ}.

2. If a ∈ Σ, then a is a regular expression, and stands for the regular language {a}.

3. If r₁ is regular expression that stands for L₁, then r₁ is a regular expression that stands for the regular language L₁.

4. If r₁ is regular expression that stands for the regular language L₁ and r₂ is regular expression that stands for the regular language L₂, then (r₁ + r₂) is a regular expression that stands for the regular language L₁ ∪ L₂, and (r₁ r₂) is a regular expression that stands for the regular language L₁ L₂.

For example, ((aa) + (ab) + (ba)) is a regular expression that stands for the regular language {aa, ab, ba, bb}.

Compare

((aa) [aa]) ∪ ([a] [a]) ∪ ([b] [a]) ∪ ([b] [b])

Another regular expression for this language is ((a + b)(a + b))².

Regular languages

A language over a finite alphabet Σ is regular if it can be constructed from the empty language, the language {Λ} and the singleton languages {a} (a ∈ Σ) by a finite number of applications of union, concatenation and Kleene star.

Sounds like a recursive definition of a set!

Definition  The set R of regular languages over Σ is the smallest subset of 2Σ* s.t.

1. Λ ∈ R
2. If a ∈ Σ, {a} ∈ R
3. If L₁ ∈ R, then L₁ ∈ R
4. If L₁, L₂ ∈ R, then L₁ ∪ L₂, L₁ L₂ ∈ R.

Example  {aa, ab, ba, bb} is a regular language. (It is the set of all strings over {a, b} of even length.)

L₁ = {a} and L₂ = {b} are regular.

Hence, L₁ L₁ = {aa}, L₁ L₂ = {ab}, L₂ L₁ = {ba} and L₂ L₂ = {bb} are regular.

It follows that {aa} ∪ {ab} = {aa, ab} is regular, as are

{aa, ab} ∪ {ba} = {aa, ab, ba} and

{aa, ab, ba} ∪ {bb} = {aa, ab, ba, bb}.

So {aa, ab, ba, bb} is regular.

Often we can leave out some of the parentheses in regular expressions.

For instance, outermost parentheses can be safely dropped. So

((a + b) + c) = (a + b) + c.

Notice that when we write “~” here, we are saying that the expressions stand for the same language (not that the two expressions are identical viewed as strings).

Also, concatenation and set union are associative, and accordingly

(a + b) + c = a + (b + c) = a + b + c

and

(ab)c = a(bc) = alc.

Under the convention that the Kleene star operator has the highest precedence, + the lowest, with concatenation in between, we also have

(a + b)c ≠ a + b c = a + (b c).

3

4
We also use exponentiation, much as before, to make the notation more convenient. Hence,
\[(a + b)(a + b) = (a + b)^2\]
and so forth. Of course,
\[(a + b)^0 = \Lambda\]
where the regular expression \(\Lambda\) stands for the language \(\{\Lambda\}\).

The precedence of exponentiation is also higher than that of concatenation and union, so
\[ab^2 = ab^2\]
and
\[a + b^2 = a + b^2\]

We also use the “exponent” operator \(+\), much as before. So, for instance, \((a + b)^+\) stands for the language
\[(\{a\} \cup \{b\})^+ = \{a, b\}^+\].

We have at hand many identities that allow us to simplify regular expressions. Here are some. (They are easily verified in light of the fact that regular expressions stand for languages — i.e., sets of strings.)

**Examples** Let \(r_1, r_2, r_3\) be regular expressions.

\[
\begin{align*}
r_1 + 0 &= r_1 - r_1 + r_1 \\
r_1 + r_2 - r_2 + r_1 &= (r_1 + r_2) + r_1 = r_1 + (r_2 + r_1) = r_1 + r_2 + r_2 \\
0r_1 &= r_1 - r_10 \\
\Lambda r_1 &= r_1\Lambda \\
(r_1r_2)r_3 &= r_1(r_2r_3) = r_1r_2r_3 \\
r_1(r_2 + r_3) &= r_1r_2 + r_1r_3 \\
(r_1 + r_2)r_3 &= r_1r_3 + r_2r_3 \\
r_1r_1 &= r_1 \\
r_1^2 + r_1^3 &= r_1^2 \\
\Lambda + r_1^2 &= r_1^2 + \Lambda + r_1^2 \\
r_1^2 + r_1^3 + r_1^2 &= r_1^2 \\
\Lambda^0 &= \Lambda
\end{align*}
\]

**Claim** \((0^*10^*)^+\) is the set of strings over \(\{0, 1\}\) with an odd number of 1’s.

Let \(x\) be a string over \(\{0, 1\}\) with one 1. Hence \(x\) can be written \(0^m10^n\) for some \(m, n \in \mathbb{N}\). Since \(0\), \(1 \in 0^*\), we have \(0^*10^n - x \in 0^*10^n\). We can conclude that every string over \(\{0, 1\}\) with one 1 belongs to \(0^*10^n\). Similar reasoning establishes inclusion in the other direction. So \(0^*10^n\) stands for the language that consists of the strings over \(\{0, 1\}\) with one 1.

Now consider the regular expression \(10^*10^n\). Similar reasoning shows that it stands for the set of all strings over \(\{0, 1\}\) that begin with 1 and have two 1’s.

Notice that \((10^*10^n)^+\) can be considered in two parts:
\((10^*10^n)^n = \{\Lambda\}\) and \((10^*10^n)^+\). Reasoning much as before, we see that \((10^*10^n)^+\) stands for the set of all strings that start with 1 and have an even number of 1’s.

Any string with an odd number of 1’s can be obtained by concatenating a string \(x\) with one 1 and a string \(y\) s.t. either \(y = \Lambda\) or \(y\) starts with 1 and has an even number of 1’s. Moreover, no string with an even number of 1’s can be obtained in this fashion.

**Example** An identifier in C is any string that begins with a letter or underscore and includes only letters, digits and underscores.

Let \(l\) stand for the regular expression
\[
g + b + \ldots + Z + A + B + \ldots + Z
\]
and let \(d\) stand for
\[
0 + 1 + \ldots + 9.
\]

Then the language of C identifiers is
\[
(l + _)\!(l + d + \!^*).
\]
Recognizing a regular language

For now, we address the language recognition problem roughly as follows. To decide whether a given string is in the language of interest...

- Allow a single pass over the string, left to right.
- Rather than waiting until the end of the string to make a decision, we make a tentative decision at each step. That is, for each prefix we decide whether it is in the language.

**Question:** How much do we need to remember at each step about what we have seen previously?

**Everything?** (If so we’re in trouble — our memory is finite, and strings can be arbitrarily long.)

**Nothing?** (Fine for $\emptyset$ and $\Sigma$.)

In general, we can expect there to be strings $x, y$ s.t. $x \in L$ and $y \notin L$. As we read these strings from left to right, we will need to remember enough to allow us to distinguish $x$ from $y$.

---

**Definition** A finite automaton (FA) is a 5-tuple

$$(Q, \Sigma, q_0, A, \delta)$$

where

- $Q$ is a finite set of “states”
- $\Sigma$ is a finite alphabet of “input symbols”
- $q_0 \in Q$ is the “initial state”
- $A \subseteq Q$ is the set of “accepting states”
- $\delta : Q \times \Sigma \rightarrow Q$ is the “transition function”

Recall that, intuitively, an FA reads a string from left to right, always remembering only a “limited amount” about the prefix it has seen so far, and always making a decision about whether that prefix belongs to the language recognized by the automaton.

Intuitively, the FA “remembers” and also “decides” about a prefix by changing state as it reads the input string.

Initially, (i.e., after having read the empty prefix of the input string), the FA is in state $q_0$.

The FA “accepts” the empty string iff $q_0 \in A$.

---

$M = (Q, \Sigma, q_0, A, \delta)$, where

- $Q$ is a finite set of “states”
- $\Sigma$ is a finite alphabet of “input symbols”
- $q_0 \in Q$ is the “initial state”
- $A \subseteq Q$ is the set of “accepting states”
- $\delta : Q \times \Sigma \rightarrow Q$ is the “transition function”

If $a \in \Sigma$ is the first character in the input string, then the state of $M$ after reading that first character is given by $\delta(q_0, a)$.

$M$ “accepts” the string $a$ iff $\delta(q_0, a) \in A$.

Similarly, if $b \in \Sigma$ is the second character in the input string, then the state of $M$ after reading that second character is given by $\delta(\delta(q_0, a), b)$.

$M$ “accepts” the string $ab$ iff $\delta(\delta(q_0, a), b) \in A$.

Notice that this notation grows cumbersome as we move along the input string. We’d like a more convenient way to describe the state that $M$ is in after reading an input string $x$.

We’ll use the transition function $\delta$ to give a recursive definition of the function $\delta'$ that maps each state $q \in Q$ and each string $x \in \Sigma^*$ to the state that the FA is in after reading $x$ starting in $q$.

**Definition** Given any FA $M = (Q, \Sigma, q_0, A, \delta)$, we define the function

$$\delta' : Q \times \Sigma^* \rightarrow Q$$

as follows:

1. For any $q \in Q$, $\delta'(q, \Lambda) = q$.
2. For any $q \in Q$, $y \in \Sigma^*$ and $a \in \Sigma$,

$$\delta'(q, ya) = \delta(\delta'(q, y), a).$$

**Claim** For any $q \in Q$ and $a \in \Sigma$,

$$\delta'(q, a) = \delta(q, a).$$

**Proof.**

\[
\delta'(q, a) = \delta'(q, \Lambda a) = \delta(\delta'(q, \Lambda), a) (\text{def } \delta') = \delta(q, a) (\text{def } \delta')
\]
Function $\delta^*: Q \times \Sigma^* \to Q$ defined in terms of $\delta: Q \times \Sigma \to Q$:

1. For any $q \in Q$, $\delta^*(q, \Lambda) = q$.
2. For any $q \in Q$, $y \in \Sigma^*$ and $a \in \Sigma$, $\delta^*(q, ya) = \delta(\delta^*(q, y), a)$.

Also, recall: For any alphabet $\Sigma$, $\Sigma^*$ is the smallest set s.t.

1. $\Lambda \in \Sigma^*$
2. For all $a \in \Sigma$ and $x \in \Sigma^*$, $xa \in \Sigma^*$.

Claim. For any $q \in Q$ and $x, y \in \Sigma^*$,

$$\delta(q, xy) = \delta^*(\delta(q, x), y).$$

Proof. By structural induction on $y$.

Case 1: $y = \Lambda$

$$\delta(q, x\Lambda) = \delta(q, x) \quad (x\Lambda = x)$$
$$= \delta^*(\delta(q, x), \Lambda) \quad (\text{def } \delta^*)$$

Case 2: $y \in \Sigma^*$, $a \in \Sigma$

III: $\delta(q, xy) = \delta^*(\delta(q, x), y)$

To show: $\delta(q, xya) = \delta^*(\delta(q, x), ya)$

$$\delta(q, xya) = \delta(\delta(q, xy), a) \quad (\text{def } \delta^*)$$
$$= \delta(\delta^*(\delta(q, x), y), a) \quad (\text{III})$$
$$= \delta^*(\delta(q, x), ya) \quad (\text{def } \delta^*)$$

FA diagrams

It is convenient to represent an FA as a diagram . . .

Recall: An FA is a 5-tuple $(Q, \Sigma, q_0, A, \delta)$.

For every state $q \in Q$ there is a corresponding node, represented by the symbol $q$ with a circle around it:

We indicate the initial state by an incoming arrow (with no "source"):  

We indicate that $q \in A$ by adding a concentric circle:

For any $q, r \in Q$ and $a \in \Sigma$, there is a directed arc labeled $a$ from $q$ to $r$ if $\delta(q, a) = r$.

Accepting a Language

Now that we have a convenient way to describe the state of an FA after it reads an input string, we can define when an FA “accepts” a string . . .

Definition. Given an FA $M = (Q, \Sigma, q_0, A, \delta)$, we say a string $x \in \Sigma^*$ is accepted by $M$ if $\delta^*(q_0, x) \in A$. If a string is not accepted by $M$, we say that it is rejected by $M$. The language accepted by $M$ (or recognized by $M$) is

$$L(M) = \{ x \in \Sigma^* \mid x \text{ is accepted by } M \}.$$  

Notice that if $L \subseteq L(M)$ we still do not say that $L$ is accepted by $M$ unless $L = L(M)$.

One of our goals will be to prove the following characterization theorem.

Theorem (Kleene’s Theorem). A language is regular iff there is an FA that accepts it.

An FA for $01^*$
An FA for \((0^*1)(0^*10^*)^*\)

---

Distinguishing Strings

The use of an FA to recognize an infinite language depends on the ability to adequately distinguish strings from one another without remembering everything about them.

**Definition** Given a language \(L \subseteq \Sigma^*\) and a string \(x \in \Sigma^*\),

\[L/x = \{ z \in \Sigma^* \mid xz \in L \}.\]

We say that \(x, y \in \Sigma^*\) are **distinguishable with respect to \(L\)** if

\[L/x \neq L/y.\]

Similarly, if \(L/x = L/y\), then we say that \(x\) and \(y\) are **indistinguishable with respect to \(L\)**.

Finally, any string that belongs to exactly one of \(L/x\) and \(L/y\) is said to **distinguish** \(x\) and \(y\) with respect to \(L\).

So \(x\) and \(y\) are **indistinguishable wrt \(L\)** if for all \(z \in \Sigma^*\),

\[xz \in L \Leftrightarrow yz \in L.\]

**Example**

\[L = (0^*1)(0^*1)\]

The strings 0 and 01 are distinguishable wrt \(L\). In fact, any string in \(0^*1\) distinguishes them. The strings \(A\) and 01 are indistinguishable wrt \(L\).
Example Consider the language

\[ L = (0 + 1)^*0. \]

The strings 0 and 01 are distinguishable wrt \( L \). They are distinguished by only one string: 0.

Observation For any language \( L \) over alphabet \( \Sigma \), the following is an equivalence relation on \( \Sigma^* \):

\[ \{ (x, y) \in \Sigma^* \times \Sigma^* \mid x \text{ and } y \text{ are indistinguishable wrt } L \} \]

(This would be easy to show.)

---

Distinguishability Theorem

Theorem 3.2 For any language \( L \) over \( \Sigma \), if there are \( n \) strings over \( \Sigma \) s.t. each is distinguishable from all the others wrt \( L \), then any FA that recognizes \( L \) has at least \( n \) states.

Proof. Assume that \( x_1, x_2, \ldots, x_n \) are all distinguishable from one another wrt \( L \). Assume that FA \( M = (Q, \Sigma, \delta) \) recognizes \( L \).

By Lemma 3.1, since any two distinct strings from

\[ x_1, x_2, \ldots, x_n \]

are distinguishable wrt \( L \), we can conclude that each of the states

\[ \delta(q_0, x_1), \delta(q_0, x_2), \ldots, \delta(q_0, x_n) \]

are distinct. Hence, \( M \) has at least \( n \) states.

Distinguishability Corollary For any language \( L \) over \( \Sigma \), if there are infinitely many strings over \( \Sigma \) s.t. each is distinguishable from all the others wrt \( L \), then there is no FA that recognizes \( L \).

Notice that it follows by Kleene’s Theorem (not proved yet) that such an \( L \) is not regular!
Claim. For any $n \in N$, any FA that recognizes

$$L_n = \{0 + 1\}^1 (0 + 1)^n$$

has at least $2^{n+1}$ states.

Proof. First we show that any two distinct strings over $\{0, 1\}$ of length $n + 1$ are distinguishable wrt $L_n$. Any two such strings $x, y$ differ on the $(k + 1)$st character, for some $k$ ($0 \leq k \leq n$). Assume wlog that the $(k + 1)$st character of $x$ is 1 and the $(k + 1)$st character of $y$ is 0. So

$$x \in (0 + 1)^k (0 + 1)^{n-k}$$

and

$$y \in (0 + 1)^k (0 + 1)^{n-k}.$$  

Consequently, $x$ and $y$ are distinguished by the string $1^k$, since

$$x1^k \in (0 + 1)^k (0 + 1)^{n-k}1^k \subseteq (0 + 1)^k (0 + 1)^n = L_n$$

while

$$y1^k \in (0 + 1)^k (0 + 1)^{n-k}1^k \subseteq (0 + 1)^k (0 + 1)^n \subseteq L_n'.$$

Finally, since there are $2^{n+1}$ distinct strings over $\{0, 1\}$ of length $n + 1$, we conclude by Theorem 3.2 that any FA that recognizes $L_n$ has at least $2^{n+1}$ states.

---

The language $\{a^n b^n \mid n \in N\}$ is not regular

Claim. The language $L = \{a^n b^n \mid n \in N\}$ over $\{a, b\}$ is not regular.

Proof. For any $m, n \in N$, if $m \neq n$, then $a^m$ and $a^n$ are distinguished wrt $L$ by $b^n$, since $a^m b^n \notin L$ while $a^n b^n \in L$. Since there are infinitely many strings of the form $a^n$ ($n \in N$), and all are distinguishable from one another wrt $L$, there is no finite automaton that recognizes $L$ (by the corollary to Theorem 3.2). Hence, by Kleene’s Theorem, $L$ is not regular.

(We still need to prove Kleene’s Theorem. That will take some work. In the meantime we’ll continue to use it.)
Complements

By the recursive definition of regular languages, the set of regular languages is closed under (finite) union, concatenation and Kleene star. (Recall: the set of regular languages over $\Sigma$ is the smallest subset of $\Sigma^*$ that: (1) is closed under these three operations, and (2) contains $\emptyset$, $[A]$, and $[a]$ for all $a \in \Sigma$.)

Claim: The regular languages are closed under set complement.

Proof. Consider any regular language $L$ over $\Sigma$. By Kleene’s Theorem, $L$ is accepted by some FA

$$M = (Q, \Sigma, q_0, A, \delta).$$

Let

$$M' = (Q, \Sigma, q_0, Q - A, \delta).$$

Notice that these FA’s differ only on their sets of accepting states, which are complements. In particular, both FA’s have the same multi-step transition function $\delta$. Hence, for any $x \in \Sigma^*$,

- $M$ accepts $x$ iff $\delta'(q_0, x) \in A$
- $M'$ accepts $x$ iff $\delta'(q_0, x) \notin A$.

So we see that $M'$ accepts the language $L'$ (that is, $\Sigma^* - L$).

By Kleene’s Theorem we can conclude that the regular languages are closed under set complement.

Intersections, Differences

Claim. The regular languages are closed under set intersection and set difference.

Proof. By the recursive definition of regular languages, we know that the set of regular languages is closed under union, and we have shown by FA construction that the regular languages are also closed under complement.

Since

$$A \cap B = (A' \cup B')'$$

and

$$A - B = A \cap B',$$

we can conclude that the set of regular languages is also closed under set intersection and set difference.

Question: Given FA’s $M_1$ and $M_2$ that accept languages $L_1$ and $L_2$ over $\Sigma$, respectively, can we construct FA’s to accept

- $L_1 \cup L_2'$
- $L_1 L_2'$
- $L_1'$
- $L_1 \cap L_2'$
- $L_1 - L_2'$

The short answer is “yes” for all of these.

But it will be convenient to postpone the constructions for concatenation and Kleene star until later, when we will have a generalized version of FA’s and a number of associated results that make the construction more straightforward.

On the other hand, the construction for union is already easy, and by slightly altering this construction, we obtain FA’s for $L_1 \cap L_2$, $L_1 - L_2$.

Example. Before looking at the general construction, let’s try an example. Given FA’s for $0^*1^*$ and $1^*0^*$, construct an FA for the intersection of the two languages...
Theorem 3.4 Assume that $M_1 = (Q_1, \Sigma, q_0, A_1, \delta_1)$ and $M_2 = (Q_2, \Sigma, q_2, A_2, \delta_2)$ accept languages $L_1$ and $L_2$ respectively. Let

$$M = (Q, \Sigma, q_0, A, \delta)$$

where

- $Q = Q_1 \times Q_2$,
- $q_0 = (q_0, q_2)$, and
- for all $(p, q) \in Q$ and $a \in \Sigma$,

$$\delta((p, q), a) = (\delta_1(p, a), \delta_2(q, a)) .$$

Then

- if $A = \{ (p, q) \in Q \mid p \in A_1 \text{ or } q \in A_2 \}$, then $L(M) = L_1 \cup L_2$,
- if $A = \{ (p, q) \in Q \mid p \in A_1 \text{ and } q \in A_2 \}$, then $L(M) = L_1 \cap L_2$,
- if $A = \{ (p, q) \in Q \mid p \in A_1 \text{ and } q \not\in A_2 \}$, then $L(M) = L_1 - L_2$.

Of course this construction is not guaranteed to produce a minimal FA accepting $L_1 \cup L_2$, $L_1 \cap L_2$ or $L_1 - L_2$ even when given minimal FAs for $L_1$ and $L_2$.

Recall: Given any FA $M = (Q, \Sigma, q_0, A, \delta)$, we define the function $\delta^*: Q \times \Sigma^* \rightarrow Q$ as follows:

1. For any $q \in Q$, $\delta^*(q, \Lambda) = q$.
2. For any $q \in Q$, $y \in \Sigma^*$ and $a \in \Sigma$, $\delta^*(q, ya) = \delta(\delta^*(q, y), a)$.

Lemma Given FAs $M_1 = (Q_1, \Sigma, q_0, A_1, \delta_1)$ and $M_2 = (Q_2, \Sigma, q_2, A_2, \delta_2)$, let $M = (Q, \Sigma, q_0, A, \delta)$, where $Q = Q_1 \times Q_2$, $q_0 = (q_0, q_2)$, and for all $(p, q) \in Q$ and $a \in \Sigma$,

$$\delta((p, q), a) = (\delta_1(p, a), \delta_2(q, a)) .$$

For all $(p, q) \in Q$ and $x \in \Sigma^*$,

$$\delta^*((p, q), x) = (\delta_1^*(p, x), \delta_2^*(q, x)) .$$

Proof. By structural induction on $x$ (that is, on the recursive definition of $\Sigma^*$).

Case 1: $x = \Lambda$.
To show: $\delta^*((p, q), \Lambda) = (\delta_1^*(p, \Lambda), \delta_2^*(q, \Lambda)) .$

$$\delta^*((p, q), \Lambda) = (p, q) \quad \text{(def $\delta^*$)}$$

$$= (\delta_1^*(p, \Lambda), \delta_2^*(q, \Lambda)) \quad \text{(def $\delta_1^*$, def $\delta_2^*$)}$$

Case 2: $x \in \Sigma$, $a \in \Sigma$.
III: $\delta^*((p, q), xa) = (\delta_1^*(p, x), \delta_2^*(q, xa))$.
To show: $\delta^*((p, q), xa) = (\delta_1^*(p, xa), \delta_2^*(q, xa))$.

$$\delta^*((p, q), xa) = \delta(\delta^*((p, q), x), a) \quad \text{(def $\delta^*$)}$$

$$= \delta_1^*(p, x), \delta_1^*(q, xa) \quad \text{(III)}$$

$$= (\delta_1^*(p, x), \delta_1^*(q, xa)) \quad \text{(def $\delta$)}$$

$$= (\delta_1^*(p, xa), \delta_2^*(q, xa)) \quad \text{(def $\delta_1^*$, $\delta_2^*$)}$$