1 Regular Languages and Equivalent Forms

A language can be thought of a set $L$ of strings $w$, where each $w$ is some sequence (possibly empty and possibly with repetition) of letters in an alphabet. Generally, we denote that alphabet by the symbol $\Sigma$. It is VERY IMPORTANT to remember that languages are simply sets. You can create new languages from existing languages by the union/alternation operator (e.g. $L_1 \cup L_2 = \{w \mid w \in L_1 \cup L_2\}$), the intersection operator (e.g. $L_1 \cap L_2 = \{w \mid w \in L_1 \cap L_2\}$), concatenation (e.g. $L_1 \circ L_2 = \{xy \mid x \in L_1, y \in L_2\}$), the Kleene closure/star operator (e.g. $L_1^* = \{w_1w_2\ldots w_k \mid k \geq 0, w_i \in L_1, 1 \leq i \leq k\}$), and complement (e.g. $\overline{L_1} = \{w \mid w \notin L_1\}$).

However, not all languages are created equal; some are “harder” than others. One of the main goals of automata theory is to classify, group and compare these languages. The Chomsky hierarchy provides four classifications, in ascending order of complexity: regular languages, context-free languages, context-sensitive languages, and recursively enumerable languages. In this class you will only study in depth regular languages and context-free languages.

A note before proceeding: Many of these definitions can be found in

- *Introduction to the Theory of Computation* by Michael Sipser
- *Introduction to Automata Theory, Languages, and Computation* by J.E. Hopcroft and Jeffrey D. Ullman. 2nd edition. 1979. (NOTE: There is a more 3rd edition, with A. Aho added as a co-author. If possible go to the 2nd edition. There are some copies in Carlson.)

You may want to look at one (or both) of these books. There are also countless tutorials online. Some of the examples come from those sources, some I made up, some I found online.

In the Chomsky hierarchy, regular languages can be thought of as very simple pattern matchers. It can be shown that a language is regular if and only if there is some regular expression to describe it. Despite their simplicity, regular expressions can be extremely useful: anyone who has used Perl, or the command line program grep knows this.\footnote{In fact, some languages are not even decidable!}

\footnote{Technically, the “regular expressions” provided by many common/modern utilities, such as Perl and grep, are slightly more powerful than actual regular expressions. We’ll gloss over this fact though.}
Briefly, a regular expression can be defined inductively:

**Definition 1.** A string $R$ is called a **regular expression** if $R$ is

- $a \in \Sigma$,
- $\epsilon$ (the empty string),
- $\emptyset$ (the empty language),
- $R_1 \cup R_2$, where $R_1$ and $R_2$ are regular expressions (union/alternation),
- $R_1 \circ R_2$, where $R_1$ and $R_2$ are regular expressions (concatenation),
- $R_1^*$ where $R_1$ is a regular expression.

Concatenation is normally shown by juxtaposition and alternation is normally shown with a vertical pipe ($|$). You may think of regular expressions as sets. There is a difference between $\epsilon$ and $\emptyset$. What is it?

Examples of regular expressions include: $(0^*1^*), \Sigma^*, (0^*1011^*) \setminus (1^*001(000)^*)$ (where $\setminus$ is the difference of sets operator).

There can be an issue with the precedence rules. However, the simplest thing is just to remember order of operations of arithmetic (PEMDAS). You can think of alternation ($|$) as being like addition; concatenation (juxtaposition) as being like multiplication, and Kleene star as being like exponentiation. For instance, how would you read

$$x^2y^3 + x(y + z)^4?$$

Now, try the same thing with the regular expression

$$a^*b^*|a(b|c)^*.$$  

(1)

**Example 1.** Some practice problems.

1. All words over $\Sigma$ whose length is a multiple of three.
   $$(\Sigma\Sigma\Sigma)^*$$

2. $(01^+)^*$
   The set of all words $w$ where every 0 in $w$ is followed by at least one 1.

3. Simplify: $1^*\emptyset$
   $\emptyset$

4. Simplify: $\emptyset^*$ (This can be tricky! What happens if you put together 0 strings?)
   $\{\epsilon\}$

5. Over $\Sigma = [0,9] \cap \mathbb{Z}$, write a regular expression for valid IPv4 addresses.

6. Write a regular expression for valid email addresses.
Example 2. Come up with a regular expression to describe all even binary, non-negative integers.

An even number is any which can be divided (without remainder) by 2. Any binary nonnegative integer \( x \) of \( n \) bits can be represented as

\[
x = \sum_{i=n-1}^{0} a_i 2^i,
\]

where \( a_i \in \{0, 1\} \). But for all \( i > 0 \), \( 2^i \) is even. The even integers form an additive group (that is, even plus even yields even), so we only must be concerned with the least significant bit — that is, when \( i = 0 \), \( 2^i = 2^0 = 1 \). The only way to get an odd binary positive integer is for \( a_0 = 1 \). So, we want to accept only those strings that end in 0 (remember, strings are read left-to-right).

Let \( \Sigma = \{0, 1\} \). Then the regular expression \( R = \Sigma^* 0 \) will do.

Having regular expressions is nice, but it would be really nice if there were some way to recognize them. We would really like some machine that could have some sort of internal state and accept or reject a regular expression.

Definition 2. A (deterministic) finite automaton is a 5-tuple \( \langle Q, \Sigma, \delta, q_0, F \rangle \) where

- \( Q \) is a finite set, called states,
- \( \Sigma \) is a finite set, called the alphabet,
- \( \delta : Q \times \Sigma \to Q \) is the transition function,
- \( q_0 \in Q \) is a special start state, and
- \( F \subseteq Q \) is a set of accepting states.

Formally, acceptance can be defined in terms of a graph, but we can think of it more informally as whether the current state \( q_e \) is an accepting state once all the input has been processed; that is, if when the input has been read once (and only one), \( q_e \in F \). If so, we accept and if not we reject.

Further, \( \delta \) should be a total function (that is, it should be defined for every element in its domain). Note that in many cases, it will not be easy to write \( \delta \) in a succinct expression; rather, the transitions will have to be enumerated. To make describing FAs easier, a diagram may also suffice. Though \( \delta \) is a total function, it can generally also be assumed that any input tokens not explicitly defined for a given state result in automatic failure. Generally, the language generated by a FA \( M \) is defined as \( \mathcal{L}(M) = \{ w \mid M \text{ accepts } w \} \).

Note that DFAs are equivalent to regular expressions. That is, given a regular expression \( R \), there exists some DFA \( M \) that recognizes \( R \) (that is, \( x \in \mathcal{L}(M) \iff x \in R \), where \( R \) is the regular language represented by \( R \)).

Example 3. Given the DFA \( M = \langle Q, \Sigma, \delta, q_0, F \rangle \), what can you say about \( \mathcal{L}(M) \) if

(i) \( q_0 \in F \)?

In such a case, we know that \( M \) accepts at least one string: the empty string \( \epsilon \). However, that is all we can say without more information.
(ii) $F = Q$?
Every state is an accepting state. Since $\delta$ is a total function, then $\mathcal{L}(M) = \Sigma^*$.

(iii) $F = \emptyset$?
In such a case, no state is accepting, so $\mathcal{L}(M) = \emptyset$.

(iv) $|Q| = 1$?
Since we must define an initial state $q_0$, then we know that $Q = \{q_0\}$. $F \subseteq Q$, so either $F = \emptyset$ or $F = Q$. If $F = \emptyset$, then no strings are accepted, so $\mathcal{L}(M) = \emptyset$; on the other hand, if $F = Q$, then $\mathcal{L}(M) = \Sigma^*$ (since $\delta$ is a total function).

1. Give the state diagram of a DFA $M$ which recognizes $L$.

![Figure 1: DFA for Ex3.1](image)

2. Give a regular expression which represents $L$.

$0^* (0^*10^*10^*10^*)^*$

**Example 4.** Consider the language $L = \{s \mid$ the number of 1s in $s$ is a multiple of $3\}$. See Figure [4].

**Example 5.** Describe a machine that accepts even binary non-negative integers.
Assume our alphabet is $\Sigma = \{0, 1\}$. We can describe the language that is required by

$L = \{w \in \Sigma^* \mid w \equiv 0 \mod 2, w \in \mathbb{Z}_{\geq 0}\}$.

However, that may not help when trying to come up with a machine to recognize $L$. However, we already came up with a regular expression $R$ from Example 2: $R = \Sigma^*0$. See Figure 2 for the machine.

![Figure 2: DFA to accept even binary non-negative integers.](image)

**Example 6.** Describe a machine that accepts even binary non-negative integers with an even number of 1s.
Example 7. (This may be very tricky. Read only if you feel confident with DFAs; attempt only if you feel very confident with DFAs. The question is from Sipser, the answer is my own.)

Let $B = \{1^k y \mid y \in \{0,1\}^* \text{ and } y \text{ contains at least } k \text{ 1s, for } k \geq 1\}$. Show that $B$ is regular; that is, give some (N)DFA accepting $B$.

Let $B$ be as described, and let $A = \{1y \mid y \in \Sigma^*, y \text{ has at least one } 1\}$. Note that if $x \in B$, then there is some positive integer $k$ such that $x = 1^k y$ where $y$ is over $\Sigma$ and has at least $k$ 1s. So $x = 1^k (0^*1)^k \Sigma^* \Rightarrow x = 11^{k-1} (0^*1)^k \Sigma^* \Rightarrow x = 1z$ where $z$ over $\Sigma$ has at least $2k - 1$ 1s $\Rightarrow x = 1z'$ where $z'$ has at least one 1 $\Rightarrow x \in A$.

So let $M = \langle Q, \Sigma, \delta, q_0, F \rangle$. Let $Q = \{q_0, q_1, q_2, q_3\}$, where $q_0$ is the initial state and $F = \{q_2\}$. For $\delta$, consider the following figure (note that $q_3$ is the “fail state”).

As proven above, if $x \in B \iff x \in A$, which happens if and only if $x$ begins with a 1 and then, prior to being accepted, has a suffix $z$ over $\Sigma^*$ with at least one 1. However, the only path from $q_0$ to $q_2$ (the only accepting state) is $(q_0, q_1, q_2)$, which requires an initial 1 and a second 1, following an arbitrary number of 0s. So $L(M) = B$.

We may define a similar type of machine, called a non-deterministic finite automaton:

**Definition 3.** A **non-deterministic finite automaton** is a 5-tuple $\langle Q, \Sigma, \delta, q_0, F \rangle$ where

- $Q$ is a finite set, called states,
- $\Sigma$ is a finite set, called the alphabet,
- $\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$ is the transition function,
- $q_0 \in Q$ is a special start state, and
- $F \subseteq Q$ is a set of accepting states.

Here $\mathcal{P}(-)$ represents the power set. You may think of NFAs as for every state in $\delta(q, \sigma)$, an exact copy of the input and current state of machine is generated, where the current state corresponds to the state in $\delta(q, \sigma)$. These multiple/alternative paths are traversed simultaneously. We may think of each possible path as a branch. Then, we may informally define acceptance as whether there exists some path/branch such that the ending state (the state once all input has been processed once and only once) of that branch is an accepting state.

Example 8. Describe an NFA that accepts even binary non-negative integers.

Assume our alphabet is $\Sigma = \{0,1\}$. Recall that we already came up with a regular expression $R$ from Example 2: $R = \Sigma^*0$. See Figure 5 for the machine.
On input $w = 100$, the machine from Figure 3 would yield the execution:

1. Reading $w = 100$, there is only one option: to be in $q_0$.
2. Reading $w = 1\underline{0}0$, there are two options: either $q_0q_0$ or $q_0q_1$.
3. Reading $w = 1\underline{0}0$, there are three options (use $X$ to mark auto-reject):
   - $q_0q_0q_0$ (failure),
   - $q_0q_1X$ (failure),
   - $q_0q_0q_1$ (accept).

It may initially seem that NFAs are more powerful than DFAs. However, the subset construction algorithm creates a DFA $M_2$ that is equivalent to any NFA $M_1$; that is, $L(M_1) = L(M_2)$.

If there are $n$ states in an NFA $N$, what is the supremum on the number of states possible in an equivalent DFA $M$? $2^n$. Do you see how/why? (Hint: Power set)

**Example 9.** Using the subset construction algorithm, construct a DFA equivalent to the NFA in Figure 3. Try it, and then compare what you get to Figure 2. In this case, they should be identical.

We may extend the definition of the NFA to include “instantenous jumps:”

**Definition 4.** A **non-deterministic finite automaton with $\epsilon$-transitions** is a 5-tuple $\langle Q, \Sigma, \delta, q_0, F \rangle$ where

- $Q$ is a finite set, called states,
- $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ is a finite set, called the alphabet with $\epsilon$ appended,
- $\delta : Q \times \Sigma_\epsilon \rightarrow \mathcal{P}(Q)$ is the transition function,
- $q_0 \in Q$ is a special start state, and
- $F \subseteq Q$ is a set of accepting states.

By using the alphabet $\Sigma_\epsilon$, we are giving ourselves the option of simulatenously transitioning from one state to another. Acceptance is defined similarly as above.

**Example 10.** Construct an NFA that is equivalent to the regular expression given in (1). Recall that to be $R = a^*b^* \mid a(b|c)^*$. See the following figure for the machine and its construction.
Note that you can probably come up with a much simpler NFA (or possibly DFA) that will do the job. This is just a very simple example to show how you can build an NFA from a regular expression.

**Example 11.** Construct a DFA equivalent to the NFA in Figure 4.

Now, it really seems as though an NFA with \( \epsilon \)-transitions has to be more power than either a DFA or a regular NFA. But that isn’t so! NFAs with \( \epsilon \)-transitions are no more powerful than regular NFAs. However, that means that they’re equivalent. We therefore have a very beautiful picture coming together, which we may sum up in a main point:

**Main Point:** A language \( L \) is regular iff there is some regular expression \( R \) generating \( L \), iff there is some NFA-\( \epsilon \) recognizing \( L \), iff there is some NFA recognizing \( L \), iff there is some DFA recognizing \( L \).

**Example 12.** Is every finite language regular? Why or why not?