Automata Theory- Introduction to Regular Languages

Michael Levet

1 Introduction

This tutorial is the first in a sequence covering automata theory and formal languages. This particular entry will introduce regular languages, which is a good starting point for automata theory. The reason for this will become clear later in this entry, but I will start with some context. Theoretical computer science is divided into three sections: automata theory, computability theory, and complexity theory. Automata theory deals with defining computation machines in a mathematically precise manner. Computability theory then seeks to determine the power and limits of these computation machines. In essence, automata and computability theory seek to define the notion of an algorithm. Complexity theory deals with algorithm efficiency and which problems can be effectively solved by computers.

Regular languages are related to one of the simplest computation machines: the finite state machine or finite state automata.

2 Defining Regular Languages

In order to talk about regular languages, we need to formally define a language. A formal language has an alphabet, which is a finite set of symbols. By convention, the alphabet is denoted as Σ. The symbols can be words, characters, integers, or anything else you like so long as they are discrete. Some examples of alphabets include the English alphabet $\{A, B, C, ..., Z, a, b, c, ..., z\}$, the binary alphabet $\{0, 1\}$, and a standard deck of 52 playing cards. A language over an alphabet Σ contains strings formed from characters in the alphabet. Consider the following example:

- Let $\Sigma = \{0, 1\}$. Some example strings over Σ include $010, 000,$ and 1.
- 121 is not a string over Σ as $2 \notin \Sigma$.

In order to more concisely define a language, we need to introduce the Kleene Closure of an alphabet. Let $n \in \mathbb{N} \cup \{0\}$, and define Σ^n is the set of all strings of length n chosen from Σ. By convention, $\Sigma^0 = \{\epsilon\}$, where ϵ is the empty string or the string of length 0. Note that some authors use λ to denote the empty string.

Kleene Closure: Let Σ be an alphabet. The Kleene Closure of Σ is denoted Σ^*, where $\Sigma^* := \bigcup_{i=0}^{\infty} \Sigma^i$. That is, Σ^* contains all the finite strings over Σ.

Consider an example:

- Let $\Sigma = \{0, 1\}$ be the alphabet. Then Σ^* contains strings such as $\epsilon, 0101, 1111111111,$ and 10000. Any finite binary string is in Σ^*.
- The strings $121, z11, 345 \notin \Sigma^*$ as $2, z, 3, 4, 5 \notin \Sigma$.

Now a language will be more formally defined.

Language: Let Σ be an alphabet. The set L is said to be a language over Σ if $L \subset \Sigma^*$.

Consider the following examples of languages:

- Let $\Sigma = \{0, 1\}$. Then $L = \Sigma^*$ is the language consisting of all finite binary strings.
- Let $\Sigma = \{0, 1\}$ with the language L defined as the set of binary strings ending in 0.
- Let $\Sigma = \{0, 1\}$. Then $L = \{00, 01, 10, 11\}$ is the language consisting of all binary strings of length 2.

We now delve into regular languages, starting with a definition. This definition for regular languages is rather difficult to work with and offers little intuition or insights into computation. Kleene's Theorem (which will be discussed later) provides an alternative definition for regular languages which is much more intuitive and useful for studying computation. However, the definition of a regular language provides some nice syntax for regular expressions, which are useful in pattern matching.

Regular Language: Let Σ be an alphabet. The language $\emptyset$ is regular. If $a \in \Sigma$, then $\{a\}$ is regular. If A, B are regular languages over Σ, then $A \cup B$, $A \cdot B := \{ab : a \in A, b \in B\}$ and A^* are all regular. These are the only regular languages over Σ.

The first operation is set union, while the second is referred to as string concatenation. Consider an example of string concatenation. Let the strings $\text{abc}, \text{def} \in \{a, b, c, d, e, f\}^*$. Then $\text{abc} \cdot \text{def} = \text{abcdef}$.

Below are a couple examples of regular expressions, which describe regular languages using the syntax in the above definition:

- Let $\Sigma = \{0, 1\}$. Then $01 \cdot \Sigma^*$ is the set of strings over Σ beginning with 01. The $\cdot$ operator is commonly dropped when referring to string concatenation, so the regular expression is more commonly written as $01\Sigma^*$.
- Let $\Sigma = \{0, 1\}$. Then $0(10)^* \cup 0(10)^*1$ is the set of strings over Σ beginning with a 0 and alternating between 0 and 1. The union operation is usually indicated with a + operation, so the regular expression is usually written as $0(10)^* + 0(10)^*1$.

There is a reason that the operations of union and concatenation are written using the addition and multiplication operation symbols. The set of regular languages with the operations of union and concatenation has algebraic properties closely related to the integers. These properties will be explored more in a later tutorial. The relation between regular expressions and computation machines will also be explored in more depth in a later tutorial.

This section will be concluded with Kleene's Theorem.

Kleene's Theorem: Let Σ be an alphabet. A language L over Σ is regular if and only if it can be accepted by a finite state automaton.

The finite state automaton is one of the simplest computation machines. The next sections will be used to introduce finite state automata and explore the implications of Kleene's Theorem.

3 Finite State Automata

The term "finite state automaton" has been introduced as a computation machine, but with no intuition as to what it does or how it works. This notion of language acceptance needs to be defined as well.

A finite state automaton, as its name suggests, has a finite set of states. This set is denoted as Q. A single state $q_0 \in Q$ is selected as the initial or starting state. We now select a string $\omega \in \Sigma^*$ as the input string. From the initial state, we transfer to another state in Q based on the first character in ω. The second character in ω is examined and another state transition is executed based on this second character and the current state. We repeat this for each character in the string.

A finite state automaton has a set of accepting states $F \subset Q$. Some definitions provide for only a single accepting state. These two definitions are equivalent, and this point will be expanded upon later.

A string ω is said to be accepted by the finite state automaton if, when started on q_0 with ω as the input, the finite state automaton terminates on a state in F. The language of a finite state automaton M is defined as follows:

$$L(M) = \{\omega \in \Sigma^* : M \text{ when started on } \omega \text{ in state } q_0 \text{ halts on a state in } F\}$$

For now, the deterministic finite state automaton will be introduced. There is a non-deterministic variant which will be discussed in greater depth later.

Finite State Automaton (Deterministic): A finite state automaton is a five-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

- Q is the finite set of states.
- Σ is the alphabet for the input strings.
- $\delta : Q \times \Sigma \to Q$ is the state transition function. It examines the given state and character, and then returns a new state.
- q_0 is the initial state.
- $F \subset Q$ is the set of accepting halt states.

Consider an example. Let $\Sigma = \{0, 1\}$, and let $Q = \{q_0, q_1, q_{reject}\}$. We define $\delta(q_0, 0) = q_0$, $\delta(q_0, 1) = q_1$, $\delta(q_1, 1) = 1$, and $\delta(q_1, 0) = q_{reject}$. Finally, we have $\delta(q_{reject}, 0) = \delta(q_{reject}, 1) = q_{reject}$. We have the accepting set of states (or in this case, state) $F = \{q_0, q_1\}$. Observe that this finite state automata accepts the language 0^*1^*. We start at state q_0. For each 0 read in, we simply stay at state q_0. Then when we finish reading in 0's, we transition to q_1 if any 1's follow the sequence of 0's. At q_1, we simply eat away at the 1's. If a 0 is read after any 1's have been recognized, then we transition to a "trap" state or a "reject" state.

So now we have some notion of a finite state automaton, but the concept is still fairly abstract. Visually, a finite state automaton can be thought as a labeled multigraph $G(V, E, L)$ where $V(G) := Q$. There is a directed edge $(q_i, q_j) \in E(G)$ if and only if there exists a character $a \in \Sigma$ such that $\delta(q_i, a) = q_j$. The labeling function is defined $L : E(G) \to 2^{\Sigma \cup \{\epsilon\}}$ (the power set of Σ) where $L((q_i, q_j)) = \{a \in \Sigma : \delta(q_i, a) = q_j\}$.

Intuitively, if multiple characters will cause the same state change, a single directed edge is used on the diagram and each of these characters will be used in labeling the edge.

So now let's construct a finite state automaton. Consider the alphabet $\Sigma = \{0, 1\}$ and the language $L = \{\omega \in \Sigma^* : |\omega| \text{ is even }\}$, where $|\omega|$ denotes the length of the string.

Recall from the introduction that we are moving towards the notion of an algorithm. This is actually a good starting place. Observe that a finite state automaton has no memory beyond the current state. It also has no capabilities to write to memory. Conditional statements and loops can all be reduced to

state transitions, so this is a good place to start.

Consider the following algorithm to recognize exactly the strings in L:

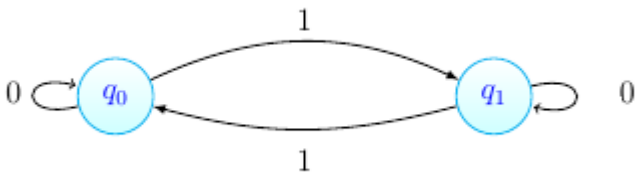
```
function parity(string input):
    parity := 0

    for i = 0 to input.length - 1:
        parity := (parity + input[i]) mod 2

    return parity == 0
end function
```

So this algorithm accepts a binary string as input and examines each character. If it is a 1, then parity moves from $0 \to 1$ if it is 0, or from $1 \to 0$ if its current value is 1. So if there are an even number of 1's in input, then parity will be 0. Otherwise, parity will be 1.

The following diagram models the algorithm as a finite state automaton. Here, we have $Q = \{q_0, q_1\}$ as our set of states with q_0 as the initial state. Observe in the algorithm above that parity only changes value when a 1 is processed. This is expressed in the finite state automata below, with the directed edges indicating that $\delta(q_i, \epsilon) = \delta(q_i, 0) = q_i$, $\delta(q_0, 1) = q_1$, and $\delta(q_1, 1) = q_0$. A string is accepted if and only if it has parity $= 0$, so $F = \{q_0\}$.



From this finite state automaton and algorithm above, it is relatively easy to guess that the corresponding regular expression is $(0^*10^*1)^*$. Consider 0^*10^*1. Recall that 0^* can have zero or more 0 characters. As we are starting on q_0 and $\delta(q_0, 0) = q_0$, 0^* will leave the finite state automaton on state q_0. So then the 1 transitions the finite state automaton to state q_1. By similar analysis, the second 0^* term keeps the finite state automaton at state q_1, with the second 1 term sending the finite state automaton back to state q_0. The Kleene closure of 0^*10^*1 captures all such strings that will cause the finite state automaton to halt at the accepting state q_0.

In this case, the method of judicious guessing worked nicely. For more complicated finite state automata, there are algorithms to produce the corresponding regular expressions. Some of these algorithms may be explored in later tutorials.

The next section will explore some of the closure properties of regular languages. Some of the closure properties are easier to prove using non-deterministic finite state automata instead of the deterministic variants. It will be shown in a later tutorial that the non-deterministic variant can be reduced to the deterministic variant. For now, we take this as fact. The non-deterministic finite state automata will now be formally defined:

Finite State Automata (Non-Deterministic): A finite state automaton is a five-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

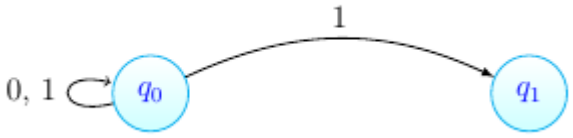
- Q is the finite set of states.

- Σ is the alphabet for the input strings.
- $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$ is the state transition function. It examines the given state and character, and then returns a new set of states.
- q_0 is the initial state.
- $F \subset Q$ is the set of accepting halt states.

Remark: The only difference between the non-deterministic and deterministic finite state automata is that the non-deterministic variant's transition function returns a subset of Q, while the deterministic variant's transition function returns a single state.

A non-deterministic finite state automata is said to accept a string ω if there is a sequence of state transitions (which can be thought of as walks on the graph diagram) in which the automata will halt in a state in F when started at q_0 on ω. Note that this is an existence quantifier, not a universal quantifier.

Example: Consider the simple non-deterministic finite state automata below, which accepts the language $(01)^*1$. Observe that q_0 can transition to either q_0, q_1 on an input character of 1. This could not be the case in the deterministic variant. That is, if the finite state automata was deterministic, q_0 could transition only to q_0 or q_1 on an input of 1. Both options would not be available.



4 IV. Closure Properties

Finally, we will examine some closure properties of regular languages. The closure properties that will be examined include set union, set intersection, set complementation, concatenation, and Kleene closure. So what exactly does closure mean? Informally, it means if we take two elements in a set and do something to them, we get an element in the set. This section focuses on operations on which regular languages are closed; however, we also have closure in other mathematical operations. Consider the integers, which are closed over addition. This means that if we take two integers and add them, we get an integer back.

Similarly, if we take two real numbers and multiply them, the product is also a real number. The real numbers are not closed under the square root operation, however. Consider $\sqrt{-1} = i$, which is a complex number but not a real number. This is an important point to note- operations on which a set is closed will never give us an element outside of the set. So adding two real numbers will never give us a complex number of the form $a + bi$ where $b \neq 0$.

Now let us look at operations on which regular languages are closed. Let Σ be an alphabet and let $RE(\Sigma)$ be the set of regular languages over Σ. A binary operator is closed on the set $RE(\Sigma)$ if it is defined as: $\odot : RE(\Sigma) \times RE(\Sigma) \to RE(\Sigma)$. In other words, each of these operations takes either one or two (depending on the operation) regular languages and returns a regular language. Note that the list of operations including set union, set intersection, set complementation, concatenation, and Kleene

closure is by no means an extensive or complete list of closure properties.

Recall from the definition of a regular language that if A, B are regular languages over the alphabet Σ, then $A \cup B$ is also regular. More formally, we can write $\cup : RE(\Sigma) \times RE(\Sigma) \to RE(\Sigma)$, which says that the set union operator takes two regular languages over a fixed alphabet Σ and returns a regular language over Σ. Similarly, string concatenation is a closed binary operator on $RE(\Sigma)$ where $A \cdot B = \{a \cdot b : a \in A, b \in B\}$. The set complementation and Kleene closure operations are closed, unary operators. Set complementation is defined as $- : RE(\Sigma) \to RE(\Sigma)$ where for a language $A \in RE(\Sigma)$, $\overline{A} = \Sigma^* \setminus A$. Similarly, the Kleene closure operator takes a regular language A and returns A^*.

Some of these closure properties follow from the definition of a regular language. Kleene's Theorem will instead be assumed to derive these closure properties. That is, we simply suppose that a language is regular if and only if it is accepted by some finite state automaton. This fact is sufficient to prove the closure properties.

The idea in the proof of the closure properties is to use the machines for each language to derive the machine for the closure language. So let A, B be regular languages, let $M(A) = (Q_A, \Sigma, \delta_A, q_A, F_A), M(B) = (Q_B, \Sigma, \delta_B, q_B, F_B)$ be the finite state automata accepting A, B respectively. Suppose $Q_A \cap Q_B = \emptyset$. Let $\omega \in \Sigma^*$.

Set Union: To show that $\omega \in A \cup B$, at least one of $M(A)$ or $M(B)$ must accept ω.

The formal construction of a finite state automata for $A \cup B$ is best done with the use of a non-deterministic finite state automata. So the given construction of $M(A \cup B)$ takes essentially the union of $M(A)$ and $M(B)$.

$M(A \cup B) := (Q_A \cup Q_B \cup \{q_0\}, \Sigma, \delta, q_0, F_A \cup F_B)$, where $\delta : (Q_A \cup Q_B \cup \{q_0\}) \times (\Sigma \cup \{\epsilon\}) \to 2^Q$ given by $\delta(q_i, a) = \delta_A(q_i, a)$ if $q_i \in Q_A$ and $\delta(q_i, a) = \delta_B(q_i, a)$ if $q_i \in Q_B$. We define $\delta(q_0, \epsilon) = \{q_A, q_B\}$. By convention, $\delta(q_0, a) = q_0$ for all $a \in \Sigma$.

The non-deterministic aspect deals with whether to simulate the $M(A)$ component or the $M(B)$ component. In this manner, there exists a transition sequence on ω starting at q_0 if and only if at least one of $M(A), M(B)$ accepts ω.

Set Complementation: This closure property is pretty easy to see. If $M(A)$ is the finite state automata accepting A, then $M(\overline{A}) = \{Q_A, \Sigma, \delta_A, q_A, Q_A \setminus F_A)$. In this way, $M(\overline{A})$ accepts a string ω if and only if $M(A)$ rejects ω.

Set Intersection: This is perhaps the most difficult closure property to prove. In order to show $A \cap B$ is regular, a product machine $M(A) \cap M(B)$ is constructed. The idea is that the product machine will run $M(A)$ and $M(B)$ in parallel. So $M(A \cap B) = (Q_A \times Q_B, \Sigma, \delta_A \times \delta_B, (q_A, q_B), F_A \times F_B)$. Observe then that the states of $M(A \cap B)$ are elements (q_x, q_y) where $q_x \in Q_A, q_y \in Q_B$.

The transition function δ also applies δ_A, δ_B component-wise. That is, $\delta((q_x, q_y), a) = (\delta_A(q_x, a), \delta_B(q_y, a))$. So a string $\omega \in \Sigma^*$ is accepted by $M(A \cap B)$ if the state (q_f, q_g) on which the automaton halts satisfies $q_f \in F_A, q_g \in F_B$. If $q_f \in F_A$ but $q_g \notin F_B$ (or vice-versa), then ω will not be accepted.

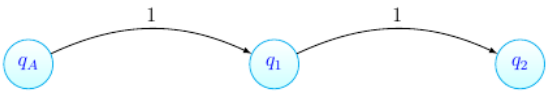
Concatenation: Showing that regular languages are closed over the concatenation operation is straightforward if it is assumed that the machines are non-deterministic. So consider $M(A), M(B)$ again, as non-deterministic finite state automata. We take each state $f \in F_A$ and add a transition $\delta(f, \epsilon) = q_B$. So $M(A \cdot B) = (Q_A \cup Q_B, \Sigma, \delta_A \cup \delta_B \cup \delta(f, \epsilon) = q_B(\text{for all } f \in F), q_A, F_B)$.

The idea is that the prefix will first be accepted, then the suffix evaluated by the second machine. The

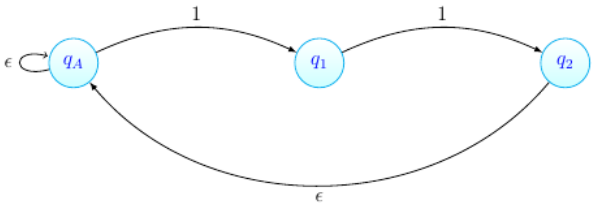
non-determinism is important, as it is possible to cut off the prefix prematurely, which would then cause the second machine to reject the string. Consider the language $(11)^* \cdot 10$. So let $M(A)$ accept $(11)^*$ and $M(B)$ accept 10. So the second 1 is parsed on the $M(A)$ component of $M(A \cdot B)$ and the third 1 is read in. Should $M(A \cdot B)$ have used the ϵ transition to q_B on the $M(B)$ component prior to evaluating the third 1? Or should it have returned to q_A? As $M(A \cdot B)$ is non-deterministic, there exists a sequence of transitions accepting $(11)^*10$, so the concern about prematurely truncating the prefix is resolved. We can also reduce the non-deterministic variant to a deterministic finite state automata.

Kleene Closure: The Kleene closure is another closure property proven with a non-deterministic finite state automata similar to concatenation. Observe that $A^* = A \cdot A \cdot A ... \cdot A$, for a finite amount of concatenations (or no concatenations, as $\epsilon \in A^*$). The construction takes $M(A)$ for regular language A, and adds $\delta(f, \epsilon)$ transitions for each $f \in F$. Essentially, once a substring is accepted, we start over and consider the rest of the string. The transition $\delta(q_A, \epsilon) = q_A$ is also added if not already present to account for $\epsilon \in A^*$. The accepting halt states of $M(A)$ are the same accepting halt states of $M(A^*)$, with the addition that q_A is an accepting halt state of $M(A^*)$. An alternative formulation is to think of $F_{A^*} = \{q_A\}$, as the ϵ transition would send a state in F_A to q_A on $M(A^*)$.

Consider the language $(11)^*$ where $A = \{11\}$. So $M(A)$ starts at q_A. It reads the first 1 and transitions to q_1. From q_1, reading in a 1 moves $M(A)$ to $q_2 \in F$. So the machine accepting A is pretty simple. In order to modify it to accept A^*, we add a transition $\delta(q_2, \epsilon) = q_A$, causing the machine to start over after having accepted a substring. We also add $\delta(q_A, \epsilon) = q_A$ to account for the empty string being accepted. The finite state automata diagrams are provided below:



M(A), the FSA accepting the language A = {11}.



M(A*), the FSA accepting the language A*.

5 Using Closure Properties

So why do we care about closure properties of regular languages? The reason is that they provide tools to easily check if a given language is regular. So let L be a language and we wish to check if it is regular. Then if we apply set intersection, for example, with a known regular language, L is regular if and only if the resulting language from the intersection is regular. In order for the closure properties to be useful, an initial non-regular language is needed. Consider the language $\{0^n1^n : n \in \mathbb{N}\}$. This language is not regular, and the proof of this fact uses machinery we don't have (specifically, the Pumping Lemma for Regular Languages). The Pumping Lemma is involved, so for the sake of example let's assume $\{0^n1^n : n \in \mathbb{N}\}$ is not regular.

Now consider $L = \{w\overline{w} : w \in \{0,1\}^*\}$. Closure properties will be used to show L is not regular. Consider $L \cap 0^*1^*$, where 0^*1^* is regular. Observe that the resulting set from $L \cap 0^*1^* = \{0^n1^n : n \in \mathbb{N}\}$, which we know not to be regular. Therefore, L cannot be regular.

Now suppose M is a regular language. Let $M' = \{xy : x \in M, y \in \overline{M}\}$. Recall that since M is regular, $\overline{M}$ is regular by the closure of set complementation. So $M' = M \cdot \overline{M}$. Thus, M' is regular, as it is the concatenation of two regular languages.