# READING NOTES ON
# *AUTOMATA THEORY LANGUAGE AND COMPUTATION*

JIECAO CHEN
JIECA001@UMN.EDU

ABSTRACT. This will be a reading note on the book *Introduction to Automata Theory Languages and Computation*. It consists of both the content from the book and my own thinking based on those content.

CONTENTS

## 1. AUTOMATA: THE METHODS AND THE MADNESS

1.1. **Why study Automata Theory.** This section serves to introduce the principal motivation and also outlines the major topics covered in book [1].

1.1.1. *Introduction to Finite Automata.* Here are some examples that Finite Automata will be used

- Software for designing and checking the behavior of digital circuits.
- The ”lexical analyzer” of a typical compiler, that is, the compiler component that breaks the input text into logical units, such as identifiers, keywords, and punctuation.
- Software for scanning large bodies of text, such as collections of Web pages, to find occurrences of words, phrases, or other patterns.
- Software for verifying system of all types that have a finite number of distinct states, such as communications protocols or protocols for secure exchange of information.

1.1.2. *Structural Representations.* There are two important notations that are not automation-like, but play an important role in the study of automata and their applications.

- *Grammars* are useful models when designing software that processes data with a recursive structure. The ”best-know” example is a ”parser”, the component of a compiler that deals with the recursively nested features of the typical programming language, such as expressions – arithmetic, conditional, and so on. For instance, a grammatical rule like $E \Rightarrow E+E$ states that an expression can be formed by taking any two expressions and connecting them by a plus sign; this rule is a typical of how expressions of real programming language are formed. Context-free grammars may be covered later (which was first formulated by one of the best-known scholar, *Avram Noam Chomsky*).
- *Regular Expressions* also denote the structure of data, especially text strings. The text described by *regular expressions* is exactly same as what can be described by finite automata.

1.1.3. *Automata and Complexity.* Automata are essential for the study of the limits of computation. There are two important issues:

- What can a computer do at all? This study is called ”decidability”, and the problems that can be solved by computer are called ”decidable”.
- What can a computer do efficiently? This study is called ”intractability”, and the problems that can be solved by a computer using no more time than some slowly growing function of the size of the input are called ”tractable”. Often, full polynomial functions are considered to be ”slowly growing” while function that grow faster than than any polynomial are deemed to grow too fast.

1.2. **The Central Concepts of Automata Theory.** This section will give the definitions of most important terms that pervade the theory of automata.

1.3. **Alphabets.** An alphabet is a finite, nonempty set of symbols. Conventionally, we use the symbol $\Sigma$ for an alphabet. Common alphabets include:

- $\Sigma = \{0, 1\}$, the binary alphabet
- $\Sigma = \{a, b, \ldots, z\}$, the set of all lower-case letters.
- The set of all ASCII characters, or the set of all printable ASCII characters.

1.3.1. *Strings.* A *string* (or sometimes *word*) is a finite sequence of symbols chosen from some alphabet. For example, $01010$ is a string from the binary alphabet $\Sigma = \{0, 1\}$.

**The Empty String** The *empty string* is the string with zero occurrences of symbols. This string is normally denoted as $\epsilon$.

**Length of a String** Then *length* of string $w$ Normally denoted as $|w|$.

**Power of an Alphabet** $\Sigma^k = \{w \| |w| = k$ and $w$ is a string from $\Sigma\}$.

Here are several related concepts:

- $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots$
- $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \ldots$

**Concatenation of Strings** Let $x$ and $y$ be strings, then $xy$ denotes the *concatenation* of $x$ and $y$.

**Languages** If $\Sigma$ is an alphabet, and $L \subseteq \Sigma^*$, then $L$ is a *language over* $\Sigma$.

1.3.2. *Problems.* If $\Sigma$ is an alphabet, and $L$ is a language over $\Sigma$, then the problem $L$ is:

Give a string $w$ in $\Sigma^*$, decide whether or not $w$ is in $L$.

It turns out that anything we more colloquially called a ”problem” can be expressed as membership in a language.

## 2. Finite Automata

2.1. **Definition of a Deterministic Finite Automaton.** A *deterministic finite automaton* consists of:

- A finite set of states, often denoted $Q$
- A finite set of *input symbols*, often denoted as $\Sigma$
- A *transition function* that takes as arguments a state and an input symbol and returns a state. The function will commonly be denoted $\delta$.
- A *start state*, one of the states in $Q$
- A set of *final* or accepting states $F$. The set is a subset of $Q$.

A deterministic finite automaton will often be referred to by its acronym: DFA. The most succinct representation of a DFA is a listing of the five components above. In proofs we often talk about a DFA in ”five-tuple” notation:

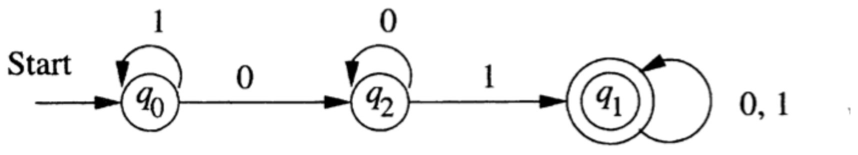
$$A = (Q, \Sigma, \delta, q_0, F)$$

where $A$ is the name of the DFA, $Q$ is its set of states, $\Sigma$ its input symbols, $\delta : Q \times \Sigma \mapsto Q$ its transition function, $q_0$ its start state, and $F$ its set of accepting states. A example:

**Example 2.1.** $A = (\{q_0, q_1, q_3\}, \{0, 1\}, \delta, q_0, \{q_1\})$

One thing should be mentioned is that the transition could also be extended to $\hat{\delta} : Q \times \Sigma^+ \mapsto Q$, i.e. it takes a state and a string as input then gives a state as a output.

Following picture will be a informal illustration of a DFA.



*An example of a DFA*

2.1.1. *The Language of a DFA.* The *language* of a DFA $A = (Q, \Sigma, \delta, q_0, F)$, denoted $L(A)$, is defined by:

$$L(A) = \{w | \hat{\delta}(q_0, w) \text{ is in } F\}$$

If $L$ is $L(A)$ for some DFA $A$, then we say $L$ is *regular language*.

2.1.2. *Definition of Nondeterministic Finite Automata.* Let us introduce the formal notions associated with nondeterministic finite automamta. The differences between DFA's and NFA's will be pointed out as we do. An NFA is represented essentially like a DFA:

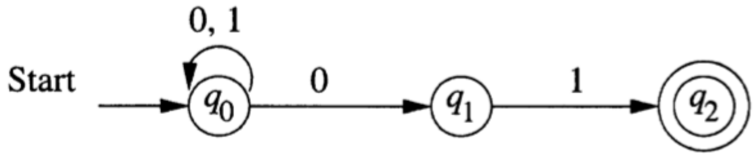
$$A = (Q, \Sigma, \delta, q_0, F)$$

Where:

- $Q$ is a finite set of *states*.
- $\Sigma$ is a finite set of *input state*.
- $q_0$, a member of $Q$, is the *start state*.
- $F$, a subset of $Q$, is the set of *final* (or accepting) states.

2.2. **Nondeterministic Finite Automata.** A ”nondeterministic” finite automaton (NFA) has the power to be in several states at once. This ability is often expressed as an ability to ”guess” something about its input. For instance, when the automation is used to search for certain sequences of characters (e.g., keywords) in a long text string, it is helpful to ”guess” that we are at the beginning of one of those strings and use a sequence of states to do nothing but check that string appears, character by character.

A informal example:



*An NFA accepting all strings that end in* 01

2.2.1. *The Extended Transition Function.* We need to extend the transition function $\delta$ to $\hat{\delta}$ to make it has the ability to take a state and a string as an input and give a column of states as an output. Let's use INDUCTION to construct such functions.

**BASIS**: $\hat{\delta}(q,\epsilon) = \{q\}$

**INDUCTION**: Suppose $w$ is of the form $w = xa$, where $a$ is the final symbol of $w$ and $x$ is the rest of $w$. Also suppose that $\hat{\delta}(q,x) = \{p_1, p_2, \dots, p_k\}$. Let

$$\cup_{i=1}^{k}\delta(p_i,a) = \{r_1, r_2, \dots, r_m\}$$

Then $\hat{\delta}(q,w) = \{r_1, r_2, \dots, r_m\}$.

2.2.2. *The Language of an NFA.* Formally, if $A = (Q,\Sigma,\delta,q_0,F)$ is an NFA, then

$$L(A) = \{w|\hat{\delta}(q_0,w)\bigcap F \neq \emptyset\}$$

2.2.3. *Equivalence of Deterministic and Nondeterministic Finite Automata.* Although there are many languages for which an NFA is easier to construct than a DFA, such as the language of strings that end in $01$, it is a surprising fact that **every language that can be described by some NFA can also be described by some DFA**, the proof involves a technique called *subset construction* which constructs all subsets of the set of stats of the NFA.

**Theorem 2.2.** *If $D = (Q_D,\Sigma,\delta_D,\{q_0\},F_D)$ is the DFA constructed from NFA $N = (Q_N,\Sigma,\delta_N,q_0,F_N)$ by the subset construction, then $L(D) = L(N)$.*

Further, we have

**Theorem 2.3.** *A language $L$ is accepted by some $DFA$ if and only if $L$ is accepted by some $NFA$.*

2.3. **Finite Automata With Epsilon-Transitions.** A $\epsilon-$NFA (or $\epsilon$-DFA) is an extended NFA such that it also accepts $\epsilon$ as input of the transition function, i.e. the input set will be $\{\epsilon\}\cup\Sigma$.

**Theorem 2.4.** *A language $L$ is accepted by some $\epsilon-$NFA if and only if $L$ is accepted by some DFA.*

## 3. Regular Expressions and Languages

”Regular Expression” are another type of language-defining notation, which may be thought of as a ”programming language”. It has various applications such as text-search application or compiler components. Regular expressions are closely related to nondeterministic finite automata and can be thought of as a ”user-friendly” alternative to the NFA notation for describing software components.

3.1. **Regular Expressions.** ”Regular Expressions” is regarded as a kind of algebraic description of language, comparing with the machine-like description: NFA and DFA. We shall find that regular expressions can define exactly the same languages that the various forms of automata describe: the regular languages. However, regular expressions offer something that automata do not: a declarative way to express the strings we want to accept. Thus, regular expressions serve as the input language for many systems that process strings. Examples:

- Search command such as UNIX **grep**. Different search engines convert the regular expressions to DFA or NFA, and simulate that automaton on the file being searched.
- Lexical-analyzer, such as **Lex** or **Flex**. Both are extremely useful to construct compilers.

3.1.1. *The Operators of Regular Expressions.* Regular expressions denote languages. For example, $01^* + 10^*$ denotes the language consisting of all strings that are either a single $0$ followed by any number of 1’s or a single $1$ followed by any number of 0’s. Here are the operations:

- The *union* of two languages $L$ and $M$, denoted $L \cup M$
- The *concatenation* of language $L$ and $M$ is the set of strings that can be formed by taking any string in $L$ and concatenating it with any string in M, denoted $L.M$ or just $LM$.
- The *closure* (or *star*, or *Kleene closure*) of a language $L$ is denoted $L^*$ and represents the set of those strings that can be formed by taking any number of strings from $L$, possibly with repetitions and concatenating all of them. Formally, $L^* = \cup_{i \geq 0} L^i$ where $L^0 = \{\epsilon\}$, $L^1 = L$, for $i > 1$ is $L^i = LL \ldots L$ (the concatenation of $i$ copies of $L$) .

## References

[1] Hopcroft, John E. ***Introduction to Automata Theory, Languages, and Computation***, 3/E. Pearson Education India, 2008.

[2] Wadler, Philip. ***The essence of functional programming***. Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM, 1992.