Chapter 2

Introduction to Languages and Finite State Automata

Computer language designers must formally specify the set of legal programs in the designed language to users (programmers) and implementors of that language. Formally, a programming language is a set of strings. Since the set may be infinite, a formal way of generating the set in a finite way is needed. To do this, formal grammars are used to generate the languages. Checking whether a program is syntactically correct is tantamount to implementing an algorithm realizing set membership of strings in a possibly infinite set. To do this, implementors also use computational models, such as finite state automata, push-down automata, or Turing machines to implement the membership test.

In this chapter and the next, formal languages, formal grammars, automata, and related concepts are covered. In addition, their application to building and engineering the front-end of translators (interpreters and compilers)is covered. As presented in the previous chapter the first two phases of a translator usually include lexical analysis (scanning) and parsing. Together these phases are usually referred to as the front-end of a translator.

To model lexical analysis, the lexemes are usually viewed as comprising the "sentences" of this lexical language. The lexemes are categorized into groups called tokens. These tokens makes up the vocabulary of the programming language being translated. To recognize lexemes, the computational model class known as a finite state automata are utilized. There are different kinds of finite state automata: nondeterministic, nondeterministic-with-ϵ-moves, and deterministic. Each are useful for different aspects of lexical analysis, but, it turns out, all of these are equivalent in the sense that they recognize the same family of languages, called *regular* languages. None of these finite automata recognize more languages than the other.

To help language designers express these lexemes of regular languages, regular expressions are used. We say that regular expressions *denote* regular languages and finite state automata *recognize* regular languages. This chapter looks at the formal definition of finite state automata and the simulation or execution of them to test string membership in regular languages. We also examine how to express regular languages with regular expressions along with the algebraic identities of regular expressions and the construction of finite

state automata from regular expressions. This latter translation is used by compiler building tools, such as lex or antlr4.

The concepts presented in this chapter are intended to give the theoretical basis for building lexical analyzers. The lexical analysis of a compiler not only involves recognizing the lexemes of a programming language, but categorizing the lexeme as a token kind and returning the token and, where necessary, the corresponding lexeme to a parser. In the next chapter grammars and automata related to parsing is examined. The parser realizes the string membership function for a given language whose vocabulary is comprised of the tokens recognized by the lexical analyzed (scanner).

2.1 Formal Languages

Computer scientists use finite state automata to model many types of hardware and software systems. The primary focus here is to introduce finite state automata as regular language recognizers. This Chapter ultimately provides the foundation of writing software for scanners in compilers and front ends for the command and data entry components of user interfaces. It can be shown that given a set of regular expression, an finite state automata can be constructed that recognizes the regular language specified by the set of regular expressions. We will study an algorithm that can translate a regular expression into a deterministic finite state automata. Being able to do this, allows one to develop compiler-writing tools that allows a compiler writer specify the lexemes as a set of regular expression and a recognizer automatically generated. Lex is an example that does this.

Regular expressions and the role as a denotation for regular languages are covered first. The syntax of regular expressions and their meanings (semantics) is given. To make regular expressions easier to express and make more concise extensions to meta-syntax and operators are shown. Also, identities surrounding regular expressions are also shown for the purpose of simplifying or transforming a regular expression into a particular form.

A class of finite state automata, called deterministic finite automata, are defined first. Following their definition and behavior, non-deterministic automata are introduced. It will be shown that non-deterministic automata and deterministic automata recognize the same class of regular languages.

Following the presentation of the automata and their behavior, the theorems stating the equivalence of finite automata to regular grammars, in sense of the class of languages recognized by finite automata is the same as the class of languages generated, is shown. First, the transformation from a regular language that generates a language L to a non-deterministic automata that recognizes that language L and the inverse transformation are shown. In addition similar transformations between regular grammars and deterministic automata are also established.

2.2 Overview of Formal Languages

Since the advent of the electronic digital computer the representation of data and instructions along with the properties of the representations has been studied. In particular, the text representation of programming

languages and data along with the automatic translation of text from one natural or programming language to another has been researched quite extensively by both computer scientists and linguists. Chomsky grammars have played a central and foundational role in describing the syntax of these languages and the framework for the formal semantics of these languages.

Although electronic digital computers are finite, the sets of finite strings representing programming languages and abstract data are modeled as potentially infinite sets. Designers of a programming language need a notation or metalanguage for formally specifying the potentially infinite number of legal strings of their new language. Developers writing translators for the language must know the designer's intent of what the legal strings are comprising the new programming language.

A particular family of formal grammars, first formulated by Noam Chomsky [1], are widely used and studied by computer scientists for describing formal languages. Chomsky grammars also allow one to formally describe the syntactic structure of a language, which can be useful in describing the semantics of that language. This can help with proper use of the language and also translation of strings of that language to strings of other languages.

Many notations and constructions have been introduced to describe infinite sets. In mathematical literature, an author might describe an infinite set of non-negative, even integers to a reader through establishing a pattern sequence or through elementary arithmetic operations over constants and numbers chosen from a given set. For example, one can describe the set of even non-negative integers using these two ways as:

$$\mathbb{N}_e \quad = \quad \{0, 2, 4, \ldots\} = \{2k | k \in \mathbb{N} = \{0, 1, 2, \ldots\}\}.$$

To test membership of a number n in $\mathbb{N}_e$ one can either begin enumerating the members of $\mathbb{N}_e$ until n is found. Another algorithmic approach is to test whether 2 divides n without remainder; if so, then n is in $\mathbb{N}_e$ and if not, then it is not in the set.

Just as set construction notation involving functions and predicates is used to describe the numbers comprising an infinite set of those numbers, strings comprising the elements of an infinite set be described similarly. However, the operations and objects that comprise the functions and predicates must be operations over string objects. In this chapter we examine what these operations are and how formal grammars serve as a finite way for constructing infinite (and finite) languages of strings and for as a means for constructing procedure or algorithms for testing membership of strings. This latter aspect of language theory deals primarily with the study and design of algorithms and models that are often referred to as language recognizers and automata. We shall concentrate the discussion first on the representation or description of languages and postpone the study of recognizers, parsers, etc. until later.

The scope of this chapter is to introduce the reader to the fundamental concepts underlying formal language theory so that it may be applied later to translator writing systems, such as interpreters and compilers. Section 2.3 introduces definitions of concepts related to strings, string operations, and string properties, each of which provide the foundational concepts necessary for discussing formal grammars. Next, we cover Chomsky grammars and the four hierarchical families of Chomsky languages along with the corresponding families of Chomsky grammars. Context Free grammars, including regular grammars, and related concepts such as

[1] Noam Chomsky, a linguist, has provided the framework for the subject matter in this chapter. Most of his original work was done in the late 1950's.

leftmost and rightmost derivations, parse trees, ambiguity and properties of these grammars are introduced. Finally, the relationship to parsing is introduced. A more complete discussion of parsing of context free languages and related recognizers are discussed in a later chapter.

2.3 Strings

Within formal language theory, an alphabet (or vocabulary) is a finite set of symbols. A finite *string* is a finite sequence over an alphabet A. Recall that a finite sequence is a total function restricted to an index set domain: $[n] = \{0, 1, \ldots, n-1\}$ where $n \geq 0$; $[0]$ represents the empty set. A string, $\alpha : [n] \rightarrow A$, has *length* n and is denoted $|\alpha| = n$, where $n \geq 0$. For each alphabet, A, there is only one string with the empty index set; this string is called the empty string, denoted in this text as ϵ_A. Often the subscript A is dropped when the alphabet A is clear.

Notation: Although strings are finite sequences, we often replace the sequence delimiters $<$, $>$, with quote " symbol delimiters and leave out commas separating the symbols (terms). Where confusion will not arise, the quote " delimiters may be dropped.

Example 2.1 ***Using the* String *definition***

Let $A = \{a, b, c\}$. Example of strings are α_1 = "a" and α_2 = "aac". More formally, we have:

> *$\alpha_1 : [1] \rightarrow A$, formally defined as: $\{(0 \mapsto a)\}$ and $\alpha_2 : [3] \rightarrow A$ formally defined as: $\{(0 \mapsto a), (1 \mapsto a), (2 \mapsto c)\}$.*

■

Substrings, String Prefixes and String Suffixes

Given strings α, β, and ω, (each possibly empty) such that $\phi = \alpha\beta\omega$, α is the *prefix* string of ϕ and *proper prefix* iff $\alpha \neq \phi$ and $\alpha \neq \epsilon$; similarly, ω (ω possibly ϵ) is the *suffix* string of ϕ and the proper suffix iff $\omega \neq \phi$ and $\omega \neq \epsilon$; and α, β, and ω are all *substrings* of ϕ.

Example 2.2 ***Prefixes and Suffixes***

Let $\phi = cabb$ be a string over an alphabet that contains $\{a, b, c\}$.

- Prefixes of *cabb*: ϵ, *c*, *ca*, *cab*, and *cabb*;
- Proper Prefixes of *cabb*: *c*, *ca*, *cab*;
- Suffixes of *cabb*: *cabb*, *abb*,*bb*, *b* and ϵ;
- Proper Suffixes of *cabb*: *abb*,*bb*, and *b* .
- Substrings of *cabb*: All prefixes and suffixes of *cabb* along with *a*, *b*, *ab*.

■

Concatenation

The *concatenation* of strings $\alpha : [m] \to A$ and $\beta : [n] \to A$ is the string $\alpha\beta : [m+n] \to A$, where $\alpha\beta(i) = \alpha(i)$ for $0 \leq i \leq m$ and $\alpha\beta(j+m) = \beta(j)$ for $0 \leq j \leq n$. It follows that $|\alpha\beta| = |\alpha| + |\beta|$. Note that for all strings α, $\alpha\epsilon = \epsilon\alpha = \alpha$. Also note that concatenation is not always commutative; that is, it does not always hold that $\alpha\beta = \beta\alpha$.

Example 2.3 ***Concatenation***

Again, let the alphabet $A = \{a, b, c\}$ and let α_1 = "a" and α_2 = "aac". The concatenation, $\alpha_1\alpha_2$ = "aaac". We have $\alpha_1\alpha_2 : [4] \to A$. Formally, we have:

$$\alpha_1\alpha_2 \quad = \quad \{(0 \mapsto a), (1 \mapsto a), (2 \mapsto a), (3 \mapsto c)\}$$

■

Exponentiation and String Reverse Notation

It is convenient to use exponentiation notation to represent repeated symbols in a string. For each symbol $a \in A$, we may write a^n $(n \geq 0)$ as a shorthand notation for $\underbrace{aa \cdots a}_{n\ a's}$. For example, it will be convenient to write *aaac* as a^3b^1 For each symbol $a \in A$, $a^1 = a$. Usually, exponent of value 1, the exponent is left implicit. The strings of length 1 are not only interpreted as strings, but as single symbols from an alphabet; the context of usage is used for understanding the proper interpretation or explicit quotes are used: "a". Also note that for any $a \in A$, $a^0 = \epsilon$ (or more precisely, ϵ_A).

For $a \in A$ and $m, n \geq 0$:

$$a^m a^n \quad = a^{m+n}.$$

It is also convenient to use exponentiation notation to represent repeated concatenation of repeated strings over an alphabet. To denote the concatenation of $n \geq 0$ copies of a string α, we write α^n as shorthand notation for $\underbrace{\alpha\alpha \cdots \alpha}_{n\ \alpha's}$. Again, $\alpha^0 = \epsilon$.

It is sometimes convenient to refer to the reverse of a string. The *reverse* of a string $\alpha = a_1 a_2 \ldots a_n$ is a string denoted and defined as $\alpha^R = a_n \ldots a_2 a_1$. A string α is a palindrome iff $\alpha = \alpha^R$. The reverse of *ababc*, written $ababc^R = cbaba$. The string *abba* is a palindrome.

Concatenation over Sets of Strings

Concatenation can also be extended over *sets* of strings. Let S and T each be sets of strings. The set $S \bullet T = \{\alpha\beta | \alpha \in S \text{ and } \beta \in T\}$. For an alphabet A, A^n $(n \geq 0)$ is recursively defined as follows:

$$A^n = \begin{cases} \{\epsilon\} & \text{if } n = 0 \\ A^{n-1} \bullet A & \text{if } n > 0. \end{cases}$$

Kleene Closure

Note that A^n is the set of all strings of length n over A. The *Kleene closure of an alphabet A* is defined as:

$$A^* = A^0 \cup A^1 \cup A^2 \cup \cdots = \bigcup_{i=0}^{\infty} A^i.$$

A^* is the infinite set of all possible finite length strings over alphabet A. Any set $L \subseteq A^*$ is a *language* over an alphabet A. Each string in language L is called a *sentence*.

It also will be convenient to use the following set and corresponding notation:

$$A^+ = A^* - \{\epsilon\}$$

An Algebraic Structure using String Concatenation

In summary, the concatenation operation over strings of A^* forms an algebraic structure called a monoid. Such algebraic structures possess the following properties:

1. (*Closure*) For all α, $\beta \in A^*$, $\alpha\beta \in A^*$;
2. (*Associative*) For all α, β, $\gamma \in A^*$, $(\alpha\beta)\gamma = \alpha(\beta\gamma)$;
3. (*Identity Element:* ϵ) For each $\alpha \in A^*$, $\alpha\epsilon = \epsilon\alpha = \alpha$;

These properties show that A^* with concatenation forms an algebraic structure called an monoid. Because a monoid is formed, properties of monoids prove to be important and useful in more theoretical studies of language theory.

Examples of Languages

Example 2.4 ***Application of String and Language Definitions***

$$\begin{aligned} \textit{Let } A &= \{0, 1\}. \\ A^* &= \{\epsilon, 0, 1, 00, 01, 10, 11, \\ &\qquad 000, 010, 100, 110, 001, 011, \ldots\} \end{aligned}$$

The set of binary digit strings with no leading zeroes is one possible language.
The set $L = \{0^n 1^m \mid n, m \geq 0\}$ is another example of a language over this alphabet A. Some sentences of L are $0 \in L$ since $0^1 1^0 = 0\epsilon = 0$, where $n = 1$ and $m = 0$. Similarly, $1 \in L$ since $0^0 1^1 = \epsilon 1 = 1$, where $n = 0$ and $m = 1$. For $n = 1$, $m = 1$, we have $0^1 1^1 \in L$. For $n = 5$, $m = 2$, we have $0^5 1^2 = 0000011 \in L$. Note that $\epsilon \in L$ for $n = m = 0$. Languages, L, described in English is: any number of 0's (possibly none), followed by any number of 1's (possibly none), but once a 1 is encountered (read from left to right) no 0 may occur. ■

As shown in the previous example some languages may contain the empty string. For each alphabet, is a language that contains only the empty string, ϵ. Note that there is a difference between an empty language and the language containing only the empty string. $L = \emptyset = \{\ \}$ and $L = \{\epsilon\}$ are both languages given any alphabet.

Example 2.5 ***Language of Latin Letter Alphabet***

$$
\begin{aligned}
\textit{Let } A &= \{a, b, \ldots, z\}. \\
A^* &= \{\epsilon, a, b, \ldots, z, \\
&\quad aa, ab, ac, \ldots, az, ba, bb, \ldots bz, \\
&\quad \vdots \\
&\quad \ldots, za, zb, zc, \ldots, zz, \\
&\quad aaa, aab, aac, \ldots, aaz, aba, abb, \ldots, abz, \\
&\quad \vdots \\
&\quad zza, zzb, zzc, \ldots, zzz, \ldots, \\
&\quad \ldots\}
\end{aligned}
$$

Here the alphabet is the set of lower-case Latin letters. One possible subset of A^* is the set of English words in Webster's dictionary. ■

By using the terminology defined thus far, this example language has sentences that are words; this is not commonly used terminology when discussing natural languages, such as English. It is more common to think of the words of a dictionary as serving as the building blocks of the language as alphabet sets have been defined above. We shall see in the development of formal grammars, the sentences of one language may serve the role of an alphabet for another language. Such is the case here for English. When using alphabets in this hierarchical way, the sentences forming the dictionary and, in turn, used as an alphabet for another language, the latter alphabet is often called a *vocabulary*. This is shown in the next example.

Example 2.6 ***Language Over a Dictionary Alphabet (or Vocabulary)***

Let the vocabulary V equal the set of words in Webster's dictionary along with the usual punctuation marks and a space. The set of syntactically legal English sentences is one possible language. Note that some of the sentences might not make any sense semantically. ■

Note that most natural languages, such as English, are dynamic in the sense that the set of syntactically legal sentences change; moreover, the vocabulary, which comprises an English dictionary, is also ever-changing.

Programming languages, like many natural languages, can be viewed as being built from a vocabulary, which can be viewed as a language built from an alphabet. In contrast to natural languages each of which

are based upon a dictionary of finite size that serves as its vocabulary or alphabet, programming language's alphabet may be theoretically infinite if there are no restrictions on the length of identifier (e.g. variable) names. Each identifier can serve as an element in the alphabet and sentences are legally correct programs in that language.

Example 2.7 ***Java Vocabulary***

$$
\begin{aligned}
\textit{Let } V \quad = \quad & \{ \textit{ keywords: } \texttt{abstract} \text{ , } \texttt{boolean, } \ldots \text{ , } \texttt{while} \ \} \\
& \cup \{ \textit{ operators and/or delimiters: } \quad \{ \ \text{ , } \ \} \text{ , } \ \texttt{(} \text{ , } \ \texttt{)} \text{ , } \texttt{,} \text{ , } \ \texttt{+} \text{ , } \ \texttt{-} \text{ , } \ \texttt{*} \text{ , } \ \texttt{/} \text{ , } \ \texttt{++} \text{ , } \ \texttt{--} \text{ , } \ \texttt{==} \text{ , } \ \texttt{=} \text{ , } \ldots \} \\
& \cup \{ \textit{ identifiers: } \quad \texttt{x} \text{ , } \ \texttt{X} \text{ , } \ \texttt{x1} \text{ , } \ \texttt{Max} \text{ , } \ \texttt{MAX} \text{ , } \ldots \text{ , } \ \texttt{sort} \text{ , } \ldots \text{ , } \} \\
& \cup \{ \textit{ literals: } \texttt{abc, } \ \texttt{a} \text{ , } \ \texttt{'a'} \text{, } \ \texttt{123} \text{ , } \ \texttt{-123} \text{ , } \ \texttt{+123} \text{ , } \ \texttt{123L} \text{ , } \ \texttt{12.3} \text{ , } \ \texttt{12.3f} \text{ , } \ \texttt{12.3e04} \text{ , } \ldots \} \\
V^* \quad = \quad & \{ \ \epsilon, \ \texttt{abstract} \text{ , } \ \texttt{class x} \ \{ \ \} \text{ , } \ldots \text{ , } \ \texttt{abstract class, if} \text{ , } \ldots \}
\end{aligned}
$$

The Java language is one possible subset of V^*. Listed above are strings that are not in the Java language. ■

2.4 Regular Expressions

Regular expressions serve as a convenient way to describe string patterns that can be searched in a larger string, such as a text file. In language design and compiler engineering, regular expressions are used to describe lexemes and, with these regular expressions, FSAs can be constructed and, in turn, can represent scanners. We now look at the syntax and semantics of regular expressions.

2.4.1 Regular Expression Syntax and Semantics

Definition 2.4.1 Regular Expression Syntax

Let V_T be an alphabet. Regular expressions over the alphabet, $V_T \cup \{\underline{\emptyset}, \underline{\epsilon}, |, *, (,)\}$, with $V_T \cap \{\underline{\emptyset}, \underline{\epsilon}, |, *, (,)\} = \emptyset$ are defined as follows:

- $\underline{\emptyset}$ is a regular expression;
- $\underline{\epsilon}$ is a regular expression;
- for each $a \in V_T$, $\underline{a}$ is a regular expression;
- if r_1 and r_2 are regular expressions, then $r_1|r_2$ is a regular expression;
- if r_1 and r_2 are regular expressions, then $r_1 r_2$ is a regular expression;
- if r is a regular expression, then r^* is a regular expression;
- if r is a regular expression, the (r) is a regular expression.

Definition 2.4.2 Regular Expression Semantics Let V_T be an alphabet and r_1, r_2, and r be regular expressions.

- $\underline{\emptyset}$ is denotes $\emptyset$;
- $\underline{\epsilon}$ is denotes $\{\epsilon\}$;
- each regular expression $\underline{a}$ is denotes $\{a\}$;
- if r_1 and r_2 denotes sets, R_1 and R_2, $r_1|r_2$ denotes $R_1 \cup R_2 = \{\alpha | \alpha \in R_1 \text{or } \alpha \in R_2\}$;
- if r_1 and r_2 denotes sets, R_1 and R_2, $r_1 r_2$ denotes $R_1 R_2 = \{\alpha\beta \mid \alpha \in R_1, \beta \in R_2\}$;
- if r denotes set R, then r^* denotes R^*;
- if r denotes set R, the (r) denotes R.

The operators precedence levels are highest to lowest as follows: Kleene closure ($*$), Concatenation, and then Union ($|$). The parentheses can alter the precedence rules.

Example 2.8 ***Regular Expression***

Identifiers in C/C++ can be denoted as follows. Let letter $\rightarrow \underline{a}|\underline{b}|\ldots\underline{z}|\underline{A}|\underline{B}|\ldots\underline{Z}$

Let digit $\rightarrow \underline{0}|\underline{1}|\ldots\underline{9}$

Let ident $\rightarrow (letter|\underline{\_})(letter|\underline{\_}|digit)^*$

■

Example 2.9 ***Regular Expression***

Consider this regular expression: $\underline{abc}^*$ *is denoted by the set* $\{a\}\{b\}\{c\}^* = \{a\}\{b, bc, bcc, bccc, \ldots\} = \{ab, abc, abcc, abccc, \ldots\}$.

■

Example 2.10 ***Regular Expression***

$\underline{a}|\underline{bc}^*$ *is denoted by the set* $\{a\} \cup \{b\}\{c\}^* = \{a\} \cup \{b, bc, bcc, bccc, \ldots\} = \{a, b, bc, bcc, bccc, \ldots\}$ ■

2.4.2 Regular Expression Identities

In Figure 2.1 is a list of identities.

2.4.3 Extended Notation for Regular Expressions

The *class of regular languages* is the family or set of languages that can be described with the set constructions operations used for defining the semantics of regular expressions. The sets of lexemes for most programming languages form small regular languages. Since many languages have are based upon fairly large character sets, regular expressions used for denoting these regular languages can become quite long and cumbersome.

Let r, s, and t represent regular expressions in the following. For a regular expression, r, the following notation is introduced for convenience: $r^+ = rr^* = r^*r$.

$$
\begin{array}{lcll}
r|s & = & s|t & \text{(Commutative)} \\
\\
(r|s)|t & = & r|(s|t) & \text{(Associative)} \\
(rs)t & = & r(st) & \\
\\
r(s|t) & = & rs|rt & \text{(Distributive)} \\
(s|t)r & = & sr|tr & \\
\\
r|\underline{\emptyset} & = & \underline{\emptyset}|r = r & \text{(Identity)} \\
r\underline{\epsilon} & = & \underline{\epsilon}r = r & \\
\\
r\underline{\emptyset} & = & \underline{\emptyset}r = \underline{\emptyset} & \text{(Zero)} \\
\\
r|r & = & r & \text{(Idempotent)} \\
\\
(r^*)^* & = & r^* & \text{(Closure-related Identities)} \\
r^* & = & \underline{\epsilon} + r^+ & \\
\underline{\epsilon}^* & = & \underline{\epsilon} & \\
\underline{\epsilon}^+ & = & \underline{\epsilon} & \\
\underline{\emptyset}^* & = & \underline{\epsilon} &
\end{array}
$$

Figure 2.1: Regular Expression Identities

For example, using the standard ASCII character set, a regular expression for denoting a string of upper-case or lower-case letters, requires a regular expression like:

```
[a | b | c | ... | z | A | B | C | ... | Z]*
```

2.5 Finite State Automata

Abstractly, a finite state automaton(FSA)is a computational model for checking membership of strings in a language over a given alphabet. Given a string over an alphabet, an FSA reads a string from left to right and indicates whether that string is accepted or not as a member of the language. An FSA, includes a read-head, a finite set of states and a (finite) set of transitions between any two states (possibly the same state). A transition is dependent on a state and also a specific alphabet symbol for determining what state becomes the current state. A subset of states are designated as final (or accepting) states. When "running" an FSA, an FSA is always in one state and over time the machine changes states based upon the specification of the transitions and what symbol is being read. As each symbol is read the read head is moved to the

next symbol. When the end of a input string is reached and the automaton is in a final state, the string is considered accepted. If not, the string is rejected. Depending on how an FSA's transitions are specified, there may be a case where in a given state, there are no transitions for a particular alphabet symbol; in this case, the machine is considered "stuck" and, in such as case, the string is rejected.

An abstract graphical depiction of is shown in Figure 2.3.

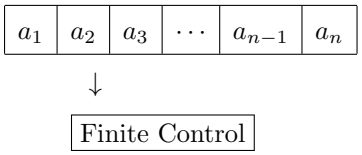


Figure 2.2: Abstract view of a Finite State Automaton

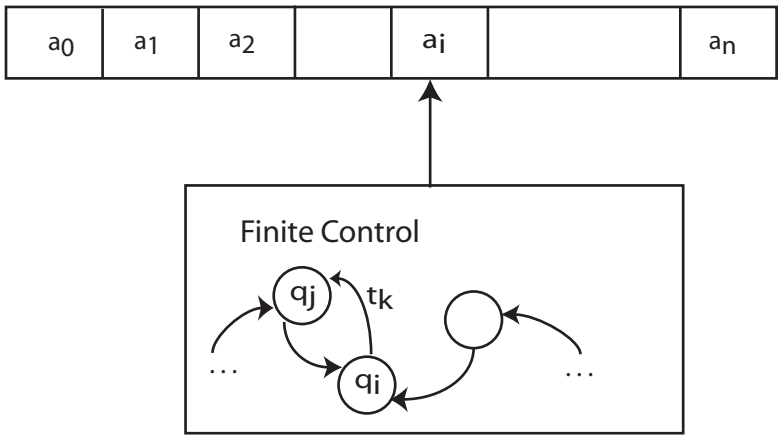


Figure 2.3: Abstract view of a Finite State Automata

As a recognizer, a finite state automaton may recognize many prefixes in a given string. It does this informally as follows. The machine starts in a start state. The machine transitions between state q_i to q_j if the symbol that the read-head is reading matches the symbol label of the directed edge from q_i to q_j. The entire string is recognized only when the last symbol is read and the state of the automaton is in a final accepting state. If not in a final state, the string is not recognized as being in the language. We now formally define a finite state automaton, give a graphical and matrix representation, and define the meaning of a FSA as a language recognizer.

Definition 2.5.1 Deterministic Finite Automaton A finite state automaton (FSA) is a 5-tuple $< Q, T, \delta, q_0, F >$ where

- Q is a non-empty finite set of *states*;
- T is an *alphabet*;
- $\delta : Q \times T \to Q$ is a (possibly partial) function, called the *transition function*;
- $q_0 \in Q$ is **the** *start state*; and
- $F \subseteq Q$ is a non-empty (finite) set of *final states*.

■

2.5.1 Representations of FSAs

Each FSA $= M = < Q, T, \delta, q_0, F >$ is usually presented or expressed with a matrix or an edge-labeled and vertex-labeled directed graph with added notations for the start and final states. Assuming that $|Q| = n$ and $|T| = m$, a transition function δ can be specified with a $n \times m$-matrix as follows:

- rows are bijectively labeled with states $q \in Q$;
- columns are bijectively labeled with alphabet symbols $t \in T$;
- each (row-q,column-t) entry is q' when δ is defined and $q^{`} = \delta(q, t)$; and
- each(row-q,column-t)entry has a blank entry or the symbol $\perp$ when $\delta(q, t)$ is undefined.

δ	$\cdots$	t	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
q	$\cdots$	q'	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$

and when undefined

δ	$\cdots$	t	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
q	$\cdots$	$\perp$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$

One can also graphically present instances of an FSA as follows:

a state | the start state | a final state | a transition

In order to define what is meant by a string α being *recognized* by an FSA, we extend the δ transition function to $\hat{\delta} : Q \times T^* \to Q$ and define $\hat{\delta}$ as follows:

$$\begin{aligned} \hat{\delta}(q, \epsilon) &= q \\ \hat{\delta}(q, \beta a) &= \delta(\hat{\delta}(q, \beta), a) \end{aligned}$$

Definition 2.5.2 String Recognition by an FSA A string α is *recognized* by an FSA, $M =< Q, T, \delta, q_0, F >$ when $\delta(q_0, \alpha) = q_f$, where $q_f \in F$. A language L recognized by an FSA, M, denoted $L(M)$, is defined as follows:

$$L(M) = \{\alpha \mid \hat{\delta}(q_0, \alpha) = q_f,\ q_f \in F\}$$

■

A string α is *not recognized (rejected)* when either at some point in the trace, reading symbol a while in state q, the transition function, δ, is undefined for that state-alphabet pair, (q, a), or the end of the string is reached and the machine is in not in a final state.

Example 2.11 $M_{2.11}$

Let $M_{2.11}$ be a finite state automaton: $Q = \{q_0, q_1\}$. Start state = q_0. $T = \{a, b\}$. $F = \{q_1\}$.

δ	a	b
q_0	q_0	q_1
q_1	q_1	q_0

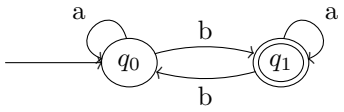


Figure 2.4: State diagram of $M_{2.11}$

We trace the steps for $\hat{\delta}(q_0, aba)$.

$$\begin{aligned}
\hat{\delta}(q_0, aba) &= \delta(\hat{\delta}(q_0, ab), a) \\
&= \delta(\delta(\hat{\delta}(q_0, a), b), a) \\
&= \delta(\delta(\delta(\hat{\delta}(q_0, \epsilon), a), b), a) \\
&= \delta(\delta(\delta(q_0, a), b), a) \\
&= \delta(\delta(q_0, b), a) \\
&= \delta(q_1, a) \\
&= q_1
\end{aligned}$$

Since $q_1 \in F$, the string aba is recognized by M. To simplify the tracing of the acceptance or rejection of strings the following notation is introduced. For each step $\delta(q_i, a) = q_j$ we use ${q_i}^a{q_j}$. So, to denote the trace of aba using machine M, we have the ${q_0}^a{q_0}^b{q_1}^a{q_1}$. Shown below are several example traces of strings using M:

1. *abaa $\in L(M)$:*
   ${q_0}^a{q_0}^b{q_1}^a{q_1}^a{q_1}$

2. *bbbab is not in $L(M)$:*
   ${q_0}^b{q_1}^b{q_0}^b{q_1}^a{q_1}^b{q_0}$

3. *bababa $\in L(M)$*
   ${q_0}^b{q_1}^a{q_1}^b{q_0}^a{q_0}^b{q_1}^a{q_1}$

4. *bababab is not in $L(M)$:*
   ${q_0}^b{q_1}^a{q_1}^b{q_0}^a{q_0}^b{q_1}^a{q_1}^b{q_0}$

$L(M)$ consists of strings over $\{a,b\}^+$ that contains an odd number of bs. Intuitively, we could use a phrase to represent a state that would convey a more meaningful name than say, q_0 or q_1. When in state q_0, we are in a state of having read an even number of b's; in state q_1 we are in a state of having read an odd number of b's. Although this gives more insight into the motivation for machine and structure and intended semantics of the language recognized by the machine, it is too cumbersome to label the states with such phrases or assertions. Often, if such assertions prove useful legends indicating the associations of the state labels with the assertions can be included. ■

Example 2.12 $M_{2.12}$

Let $M_{2.12}$ be a finite state automaton: $Q = \{q_0, q_1, q_2\}$. Start state = q_0. $T = \{a, b, c\}$. $F = \{q_2\}$.

δ	a	b	c
q_0	q_0	q_1	$\perp$
q_1	q_1	q_2	$\perp$
q_2	q_2	$\perp$	q_0

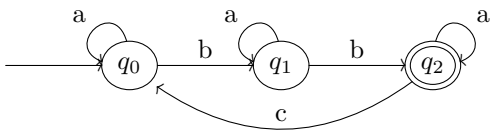


Figure 2.5: State diagram for $M_{2.12}$

Shown below are several example traces of strings using M:

1. *$ababa \in L(M)$:*
   $q_0{}^a q_0{}^b q_1{}^a q_1{}^b q_2{}^a q_2$

2. *$bb \in L(M)$:*
   $q_0{}^b q_1{}^b q_2$

3. *$ababacababa \in L(M)$:*
   $q_0{}^a q_0{}^b q_1{}^a q_1{}^b q_2{}^a q_2{}^c q_0{}^a q_0{}^b q_1{}^a q_1{}^b q_2{}^a q_2$

4. *ac is not in $L(M)$:*
   $q_0{}^a q_0{}^c$

5. *$ababaca$ is not in $L(M)$:*
   $q_0{}^a q_0{}^b q_1{}^a q_1{}^b q_2{}^a q_2{}^c q_0{}^a q_0$ *and* $q_0 \notin F$

$L(M)$ is the set of strings that is comprised of one or more substrings separated (delimited) by symbol c and with each of these substrings being a string over $\{a,b\}^+$ with exactly two bs (not necessarily consecutive). ■

2.5.2 An alternative approach to DFA transition functions: Total transition functions

An alternative to using partial transition function is to introduce an *error* state, q_e, where q_e is required not to be a final state and to modify the transition function, δ into a transition function, δ_e, as follows. δ_e is identical to δ except where δ is undefined. For all $\delta : (q, t) \mapsto \perp$ define $\delta_e : (q, t) \mapsto q_e$. and add another $|T|$ transitions, $\delta(q_e, t) = q_e$, for all $t \in T$. This introduces many more transitions and complicates the graphical representation.

Example 2.13 ***Alternative DFA:*** $M_{2.12}$

Consider the DFA, $M_{2.12}$, from Example 2.12. Using the alternative approach we have the alternative DFA, $M^e_{2.12}$, graphically presented below. It can be shown that $L(M_{2.12}) = L(M^e_{2.12})$. $M^e_{2.12}$ is graphically depicted in Figure 2.6. Tracing a string, $\alpha \in L(M^e_{2.12})$ is identical to that shown above for $L(M_{2.12})$ in Example 2.12.

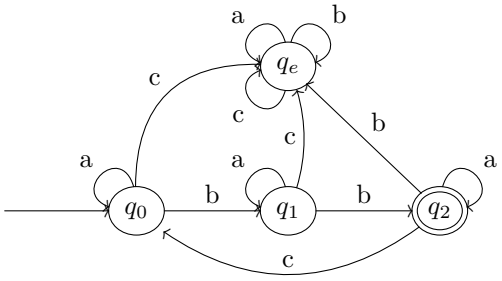


Figure 2.6: State diagram for alternative, total, transition function for $M^e_{2.12}$

However, tracing a string that is not in the language can be different. When encountering the

1. *ac is not in $L(M)$:*
   $q_0{}^a q_0{}^c q_e$
2. *ababaca is not in $L(M)$:*
   $q_0{}^a q_0{}^b q_1{}^a q_1{}^b q_2{}^a q_2{}^c q_0{}^a$

■

2.5.3 Deterministic Finite Automata as Language Recognizers

We present an algorithm as a solution to the decision problem: Given a DFA M and a string α, does $\alpha \in L(M)$?

Algorithm 2.1: Algorithm for a DFA as a Recognizer

Input : Deterministic Finite Automaton: $M =< Q, T, \delta, \text{Start State}, F >$,
$\alpha\$$, where $\$ \notin T$ and $\alpha \in T^*$.

Output: $\alpha \in L(M)$?: Yes/No

```
Q currentState;
T currentSymbol;
currentState ← Start State;
currentSymbol ← getSymbol();
while currentSymbol ≠ $ and currentState ≠ ⊥ do
    currentState ← δ[currentState ][currentSymbol ];
    currentSymbol ← getSymbol();
if currentSymbol = $ and currentState ∈ F then
    print("Yes")
else
    print("No")
;
```

2.6 Non-deterministic Finite Automata (NFAs)

As FSA are defined above, at most one transition can be taken for each symbol read from the input string. This class of FSAs are collectively referred to as *Deterministic Finite Automata (DFA)*. Another class of FSA, where this constraint is lifted, are *Non-deterministic Finite State Automata (NFA)*. At each state there may be more than one transition applicable with each input symbol. A nondeterministic finite automaton is defined as follows:

Definition 2.6.1 Non-deterministic Finite State Automaton A non-deterministic finite state automaton (NFA) is a 5-tuple $< Q, T, \delta, q_0, F >$ where

- Q is a non-empty finite set of *states*;
- T is an *alphabet*;
- $\delta : (Q \times (T \cup \{\epsilon\})) \rightarrow 2^Q$ is a total function, called the *transition function*;
- $q_0 \in Q$ is **the** *start state*; and
- $F \subseteq Q$ is a non-empty (finite) set of *final states*.

■

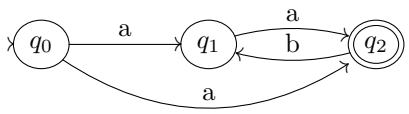


Figure 2.7: State diagram for $M_{2.14}$

The transition matrix contains *sets* of states rather than states. Note δ is a total function and that $\emptyset$ is used instead of $\perp$.

Example 2.14 $M_{2.14}$

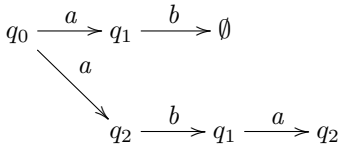
Let $M_{2.14}$ be specified as follows:

$$Q = \{q_0, q_1, q_2\}.\ T = \{a, b\}.\ \mathit{Start} = q_0.\ F = \{q_2\}.$$

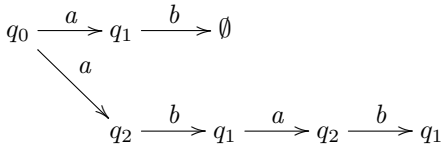
δ	a	b
q_0	$\{q_1, q_2\}$	$\emptyset$
q_1	$\{q_2\}$	$\emptyset$
q_2	$\emptyset$	$\{q_1\}$

Tracing strings for NFAs can be graphically depicted via trees. This representation is also referred to as as the proliferation of states. A string α is recognized *if at least on path may be found from the start state (root) to a final state labeling a leaf. If all leaves are either labeled with $\emptyset$ or a state that is a non-final state, the string α is* not accepted (rejected)*.*

1. *aba is* ***accepted****. A trace via a proliferation of states is shown below:*



2. *abab is not accepted by $L(M)$. Trace:*



■

2.7 Equivalence of DFAs and NFAs

Let $\mathcal{DFA}$ and $\mathcal{NFA}$ be the set of DFA and NFA machines, respectively. Let $\mathcal{L}_{\mathcal{DFA}}$ be the set of languages accepted by at least one machine $M \in DFA$ and let $\mathcal{L}_{\mathcal{NFA}}$ be the set of languages accepted by at least one machine $M \in \mathcal{NFA}$. It can be shown that $\mathcal{L}_{\mathcal{DFA}} = \mathcal{L}_{\mathcal{NFA}}$. This means that the non-determinism with finite automata recognize the same class of languages deterministic finite automata recognize.

Theorem 2.7.1 $\mathcal{L}_{\mathcal{DFA}} \subseteq \mathcal{L}_{\mathcal{NFA}}$.

Proof: It must be shown that if a DFA, M_D, recognizes language, $L(M_D)$, then an NFA, M_N, can be constructed such that $L(M_D) = L(M_N)$. If M_D is a DFA, one could convert it to a NFA, M_N, simply by changing the states in the range of δ to singleton sets and changing $\perp$ to $\emptyset$. From this construction we have $L(M_D) = L(M_N)$. ■.

It can be shown that for regular language $L(M_N)$ where M_N is an NFA, then there is a DFA, M_D, such that $L(M_D) = L(M_N)$. This is stated in Theorem 2.7.2. The proof given below is a partial proof. An algorithm is given for constructing a DFA from an NFA. For a full proof, a correctness proof of the algorithm is needed.

Theorem 2.7.2 $\mathcal{L}_{\mathcal{NFA}} \subseteq \mathcal{L}_{\mathcal{DFA}}$.

The proof is based upon subset-construction algorithm. The DFA constructed simulates moving through the distinct NFA transitions with the same label in "parallel." All the next states reached on the same symbol from one state form a subset that will become a state in the DFA. In the worst case, all subsets of the NFA state set could be generated. If the NFA state set, Q_N, has n states, then this could result in 2^N states for the constructed DFA. However, in practice, the DFAs used for scanners in translators have nearly the same number of states of a corresponding NFA.

Given these two theorems, we have the following corollary, which means that NFAs and DFAs recognize or accept the same family of languages, namely the regular languages.

Corollary 2.7.1 $\mathcal{L}_{\mathcal{DFA}} = \mathcal{L}_{\mathcal{NFA}}$.

Before the subset-construction algorithm is shown, we need some supporting algorithms to handle ϵ transitions and for moving through multiple transitions in parallel.

Algorithm 2.2: ϵ-*closure*(q)

```
/* This function returns the set of states reachable using only ε transitions from
   state q in NFA N.                                                              */
```

Input : $q \in Q_N$, where NFA $N =< Q_N, T_N, \delta_N, q_{N_0}, F_N >$.

Output: ClosureSet_Of_q

begin

- ClosureSet_Of_$q \leftarrow \{q\}$;
- **while** *There exists a* $q \in$ ClosureSet_Of_q *and* $((q, \epsilon) \mapsto q') \in \delta_N$ *and* $q' \notin$ ClosureSet_Of_q **do**
  - ClosureSet_Of_q.insert(q')
- **return** ClosureSet_Of_q;

Algorithm 2.3: ϵ-$closure(J)$

```
/* This function returns the union of all subsets of states reachable using only ϵ
   transitions from each state q in J:  ∪_{q∈J} ϵ-closure(q).                    */
```

Input : $J \subseteq Q_N$, where Q_N is a set of states of an NFA, $N =< Q_N, T_N, \delta_N, q_{N_0}, F_N >$.

Output: ClosureSet_Of_J

begin

- push all states $j \in J$ on Stack S;
- ClosureSet_Of_$J = J$;
- **while** *Stack S is not empty* **do**
  - q = top of S;
  - pop S;
  - **foreach** $q' = \delta_N((q, \epsilon))$ **do**
    - **if** $q' \notin$ ClosureSet_Of_J **then**
      - ClosureSet_Of_$J \leftarrow$ ClosureSet_Of_$J \cup \epsilon$-$Closure(q')$;
      - push q' on S ;
- **return** ClosureSet_Of_J

Algorithm 2.4: $move(J, t)$

```
/* This function returns the union of all states reachable from each state q in J
   on a symbol t in NFA N.                                                      */
```

Input : $J \subseteq Q_N$, where Q_N is a set of states of an NFA, $N =< Q_N, T_N, \delta_N, q_{N_0}, F_N >$

and

Input : $t \in T$

Output: ClosureSet_Of_J_t

begin

- ClosureSet_Of_$J_t \leftarrow \emptyset$;
- **foreach** $q \in J$ **do**
  - ClosureSet_Of_$J_t \leftarrow$ ClosureSet_Of_$J_t \cup \delta_N((q, t))$
- **return** ClosureSet_Of_J_t

Constructive part of proof of Theorem 2.7.2:

Algorithm 2.5: NFA to DFA Subset Construction

```
/* This function constructs a DFA D from an NFA N.                    */
```

Input : $N =< Q_N, T_N, \delta_N, q_{0_N}, F_N >$

Output: $D =< Q_D, T_D, \delta_D, q_{0_D}, F_D >$

```
/* Each state in Q_D of DFA is a unique set from 2^(Q_N).              */
```

begin

- $T_D = T_N$;
- $Q_D \leftarrow \emptyset$;
- $q_{0_D} \leftarrow \epsilon\text{-}closure(q_{0_N})$;
- Mark q_{0_D} as unvisited;
- Q_D.insert(q_{0_D});
- **while** *there is an* unvisited *state* $q_D \in Q_D$ **do**
  - Mark q_D as visited;
  - **foreach** $t \in T_N$ **do**
    - $X \leftarrow \epsilon\text{-}closure(move(q_D, t))$;
    - **if** $X \notin Q_D$ **then**
      - Set X as unvisited;
      - Q_D.insert(X);
    - δ_D.insert($((q_D, t) \mapsto X)$);
- $F_D \leftarrow \emptyset$;
- **foreach** $Y \in Q_D$ **do**
  - **if** $F_N \cap Y \neq \emptyset$ **then**
    - F_D.insert(Y);
- **return** D.

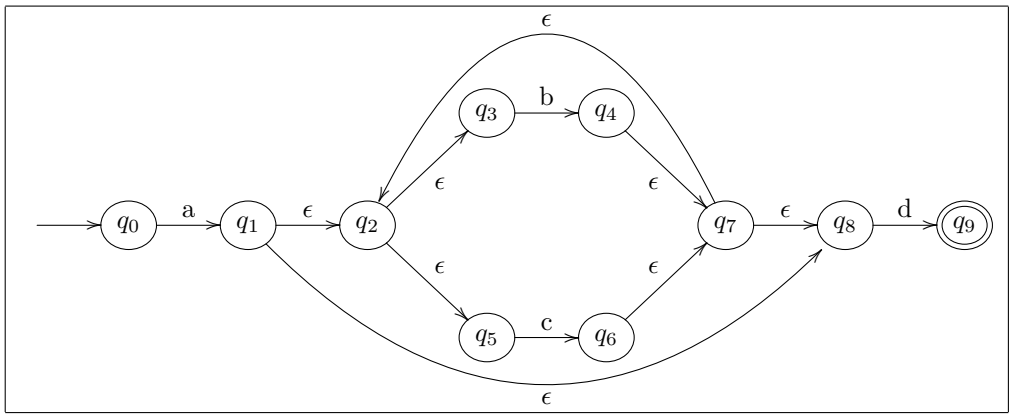


Figure 2.8: An NFA machine, M_N.

Example 2.15 ***Constructing a DFA from an NFA***

Let M_N be the NFA shown in Figure 2.8. We construct the DFA M_D using Algorithm 2.5. For reference in tracing the algorithm, each ϵ-closure(q) for all $q \in M_N$ is given in the table immediately following.

$\epsilon\text{-closure}(q_0) = \{q_0\}$	$\epsilon\text{-closure}(q_1) = \{q_1, q_2, q_3, q_5, q_8\}$	$\epsilon\text{-closure}(q_2) = \{q_3, q_5\}$
$\epsilon\text{-closure}(q_3) = \{q_3\}$	$\epsilon\text{-closure}(q_4) = \{q_2, q_3, q_4, q_5, q_7, q_8\}$	$\epsilon\text{-closure}(q_5) = \{q_5\}$
$\epsilon\text{-closure}(q_6) = \{q_2, q_3, q_5, q_6, q_7, q_8\}$	$\epsilon\text{-closure}(q_7) = \{q_2, q_3, q_5, q_7, q_8\}$	$\epsilon\text{-closure}(q_8) = \{q_8\}$
$\epsilon\text{-closure}(q_9) = \{q_9\}$		

Get start state. $q_{0_D} = \epsilon\text{-closure}(q_{0_N}) = \{q_0\}$. *Next compute* Q_D *and* δ_D *through constructing subsets. The overbar notation means the DFA state* $\overline{q_D}$, *representing the subset of* M_N *states, is visited. Upper-case Latin letters are used as labels of the subsets and are used as the name of states in the constructed DFA.*

$Q_D = \{\}$, $\delta_D = \{\}$.

Consider $q_{0_N} = A = \{q_0\}$. *Mark* A *as visited.*

$Q_D = \{\overline{A = \{q_0\}}\}$

$\epsilon\text{-closure}(move(\{q0\}, a)) = \epsilon\text{-closure}(q_1) = B = \{q_1, q_2, q_3, q_5, q_8\}$

$\epsilon\text{-closure}(move(\{q0\}, b)) = \epsilon\text{-closure}(\emptyset) = \emptyset$

$\epsilon\text{-closure}(move(\{q0\}, c)) = \epsilon\text{-closure}(\emptyset) = \emptyset$

$\epsilon\text{-closure}(move(\{q0\}, d)) = \epsilon\text{-closure}(\emptyset) = \emptyset$

$Q_D = \{\overline{A = \{q_0\}},\ B = \{q_1, q_2, q_3, q_5, q_8\}\ \}$

$\delta_D = \{((A, a) \mapsto B)\}$

Consider $B = \{q_1, q_2, q_3, q_5, q_8\}$. *Mark* B *as visited.*

$Q_D = \{\overline{A = \{q_0\}}, \overline{B = \{q_1, q_2, q_3, q_5, q_8\}}\}$

$\epsilon\text{-closure}(move(\{q_1, q_2, q_3, q_5, q_8\}, a)) = \emptyset$

$\epsilon\text{-closure}(move(\{q_1, q_2, q_3, q_5, q_8\}, b)) = \epsilon\text{-closure}(\{q_4\}) = C = \{q_2, q_3, q_4, q_5, q_7, q_8\}$

$\epsilon\text{-closure}(move(\{q_1, q_2, q_3, q_5, q_8\}, c)) = \epsilon\text{-closure}(\{q_6\}) = D = \{q_2, q_3, q_5, q_6, q_7, q_8\}$

$\epsilon\text{-closure}(move(\{q_1, q_2, q_3, q_5, q_8\}, d)) = \epsilon\text{-closure}(\{q_9\}) = E = \{q_9\}$

$Q_D = \{\overline{A = \{q_0\}},\ \overline{B = \{q_1, q_2, q_3, q_5\}},\ C = \{q_2, q_3, q_4, q_5, q_7, q_8\},\ D = \{q_2, q_3, q_5, q_6, q_7, q_8\},\ E = \{q_9\}\ \}$

$\delta_D = \{\ ((A, a) \mapsto B),\ ((B, b) \mapsto C),\ ((B, c) \mapsto D),\ ((B, d) \mapsto E)\ \}$

Consider $C = \{q_2, q_3, q_4, q_5, q_7, q_8\}$. *Mark* C *as visited.*

$Q_D = \{\ \overline{A = \{q_0\}},\ \overline{B = \{q_1, q_2, q_3, q_5, q_8\}},\ \overline{C = \{q_2, q_3, q_4, q_5, q_7, q_8\}},\ D = \{q_2, q_3, q_5, q_6, q_7, q_8\},\ E = \{q_9\}\ \}$

$\epsilon\text{-}closure(move(\{q_2, q_3, q_4, q_5, q_7, q_8\}, a)) = \emptyset$

$\epsilon\text{-}closure(move(\{q_2, q_3, q_4, q_5, q_7, q_8\}, b)) = \epsilon\text{-}closure(\{q_4\}) = C = \{q_2, q_3, q_4, q_5, q_7, q_8\}$

$\epsilon\text{-}closure(move(\{q_2, q_3, q_4, q_5, q_7, q_8\}, c)) = \epsilon\text{-}closure(\{q_6\}) = D = \{q_2, q_3, q_5, q_6, q_7, q_8\}$

$\epsilon\text{-}closure(move(\{q_2, q_3, q_4, q_5, q_7, q_8\}, d)) = \epsilon\text{-}closure(\{q_9\}) = E = \{q_9\}$

$Q_D = \{\ \overline{A = \{q_0\}},\ \overline{B = \{q_1, q_2, q_3, q_5\}},\ \overline{C = \{q_2, q_3, q_4, q_5, q_7, q_8\}},\ D = \{q_2, q_3, q_5, q_6, q_7, q_8\},\ E = \{q_9\}\ \}$

$\delta_D = \{\ ((A, a) \mapsto B),\ ((B, b) \mapsto C),\ ((B, c) \mapsto D),\ ((B, d) \mapsto E),\ ((C, b) \mapsto C),\ ((C, c) \mapsto D)$
$((C, d) \mapsto E),\ \}$

Consider $D = \{q_2, q_3, q_5, q_6, q_7, q_8\}$. *Mark* D *as visited.*

$Q_D = \{\ \overline{A = \{q_0\}},\ \overline{B = \{q_1, q_2, q_3, q_5, q_8\}},\ \overline{C = \{q_2, q_3, q_4, q_5, q_7, q_8\}},$
$\overline{D = \{q_2, q_3, q_5, q_6, q_7, q_8\}},\ E = \{q_9\}\ \}$

$\epsilon\text{-}closure(move(\{q_2, q_3, q_5, q_6, q_7, q_8\}, a)) = \emptyset$

$\epsilon\text{-}closure(move(\{q_2, q_3, q_5, q_6, q_7, q_8\}, b)) = \epsilon\text{-}closure(\{q_4\}) = C = \{q_2, q_3, q_4, q_5, q_7, q_8\}$

$\epsilon\text{-}closure(move(\{q_2, q_3, q_5, q_6, q_7, q_8\}, c)) = \epsilon\text{-}closure(\{q_6\}) = D = \{q_2, q_3, q_5, q_6, q_7, q_8\}$

$\epsilon\text{-}closure(move(\{q_2, q_3, q_5, q_6, q_7, q_8\}, d)) = \epsilon\text{-}closure(\{q_9\}) = E = \{q_9\}$

$Q_D = \{\overline{A = \{q_0\}},\ \overline{B = \{q_1, q_2, q_3, q_5\}},\ \overline{C = \{q_2, q_3, q_4, q_5, q_7, q_8\}},$
$\overline{D = \{q_2, q_3, q_5, q_6, q_7, q_8\}},\ E = \{q_9\}\ \}$

$\delta_D = \{\ ((A, a) \mapsto B),\ ((B, b) \mapsto C),\ ((B, c) \mapsto D),\ ((B, d) \mapsto E),\ ((C, b) \mapsto C),((C, c) \mapsto D)$
$((C, d) \mapsto E),\ ((D, b) \mapsto C),\ ((D, c) \mapsto D),\ ((D, d) \mapsto E)\ \}$

Consider $E = \{q_9\}$. *Mark* E *as visited.*

$Q_D = \{\ \overline{A = \{q_0\}},\ \overline{B = \{q_1, q_2, q_3, q_5, q_8\}},\ \overline{C = \{q_2, q_3, q_4, q_5, q_7, q_8\}},\ \overline{D = \{q_2, q_3, q_5, q_6, q_7, q_8\}},\ \overline{E = \{q_9\}}\ \}$

$\epsilon\text{-}closure(move(\{q_9\}a)) = \emptyset$

$\epsilon\text{-}closure(move(\{q_9\}b)) = \emptyset$

$\epsilon\text{-}closure(move(\{q_9\}c)) = \emptyset$

$\epsilon\text{-}closure(move(\{q_9\}d)) = \emptyset$

$Q_D = \{\overline{A = \{q_0\}},\ \overline{B = \{q_1, q_2, q_3, q_5, q_8\}},\ \overline{C = \{q_2, q_3, q_4, q_5, q_7, q_8\}},$
$\overline{D = \{q_2, q_3, q_5, q_6, q_7, q_8\}},\ \overline{E = \{q_9\}}\ \}$

$\delta_D = \{\ ((A, a) \mapsto B),\ ((B, b) \mapsto C),\ ((B, c) \mapsto D),\ ((B, d) \mapsto E),\ ((C, b) \mapsto C),$
$((C, d) \mapsto E)\ ((D, c) \mapsto D),\ ((D, d) \mapsto E)\ \}$

E is the only final state for M_D since that is the only set that has a non-empty intersection with F_N for the M_N.

The state diagram for M_D is shown in Figure 2.9. ***End of Example 2.15.***■

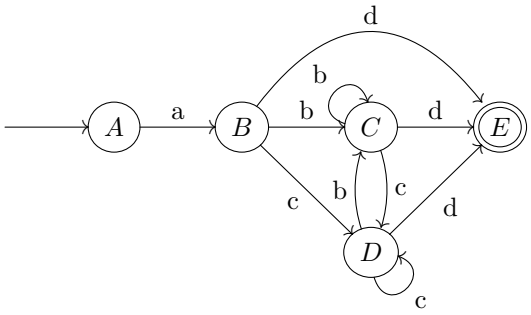


Figure 2.9: DFA constructed from NFA in Figure 2.8

2.8 REs to DFAs

We will use an algorithm based upon a proof that show all languages defined by a RE can be recognized by a DFA. The algorithm is based McNaughton-Yamada-Thompson algorithm first presented in [7]. First we show how to construct a NFA from an regular expression. Once that is created we can use the subset construction of Algorithm 2.5.

Theorem 2.8.1 *For each for each regular expression r, there is a DFA, M_D, such that $L(r) = L(M_D)$.*

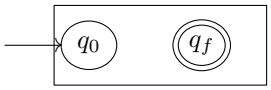
Proof: *This is a proof by using an algorithm for constructing a DFA from a regular expression. This is a partial proof. A complete proof would require a correctness argument and a proof that it terminates.*
(Algorithm):

INPUT: Regular Expression, r.

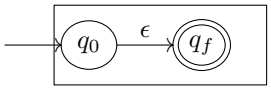
OUTPUT: NFA, M, such that $L(M) = L(r)$.

We have start with the fundamental expressions, $\underline{\epsilon}$, $\underline{\emptyset}$ and $\underline{a}$.

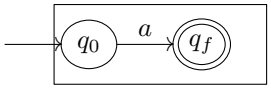
For $\underline{\emptyset}$ we have a machine:



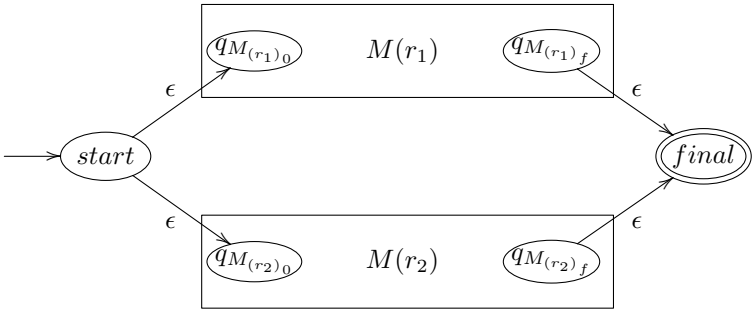
For $\underline{\epsilon}$ we have a machine:



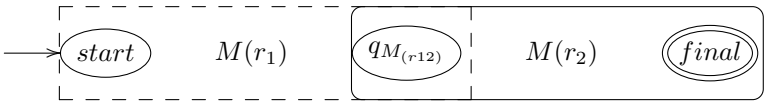
For $\underline{a}$ we have a machine:



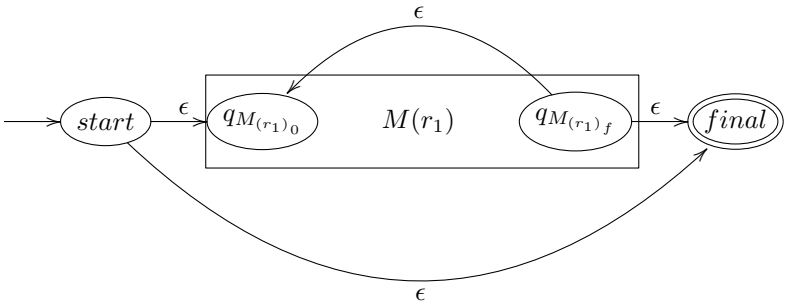
For $r_1|r_2$ we have a machine:



For $r_1 r_2$ we have a machine:



For $r_1{}^*$ we have a machine:



End of construction of an NFA using recursive definition of regular expressions.

Consider the regular expression: **a (b | c)* d**. The syntax tree is shown in Figure 2.10.

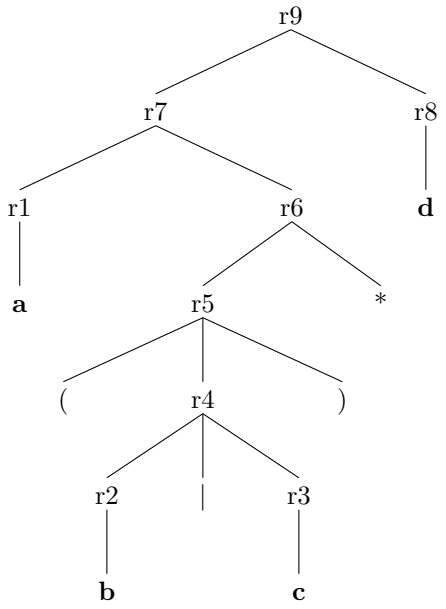


Figure 2.10: Syntax Tree for **a (b | c)* d**.

In Example 2.16 an NFA is constructed from this regular expression.

Example 2.16 ***Constructing an NFA from a Regular Expression: a (b | c)* d***

Using the tree in Figure 2.10 and using the construction of an NFA N from a regular expression r shown in the proof of Theorem 2.8.1 This last NFA machine is the same one that was used to construct a DFA in Example 2.15. See Figure 2.9.

By using the construction of a NFA, N, from regular expression, r, described in the proof in Algorithm 2.8.1 and then constructing a DFA, D, from NFA, N, using the subset-construction shown in the proof of Algorithm 2.5, we have an overall algorithm for constructing a DFA, D, from a regular expression, r. This is just one of the algorithms used for building DFAs from regular expressions. This algorithm has the time and

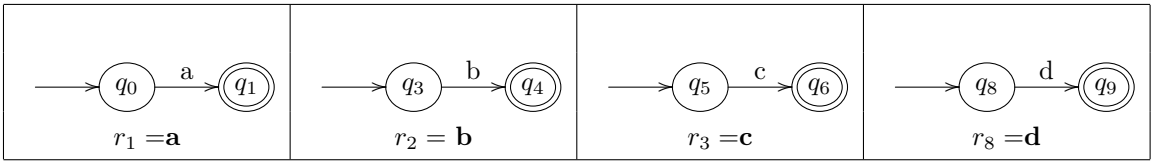


Figure 2.11: NFA machines for single symbol regular sub-expressions of **a (b | c)* d**.

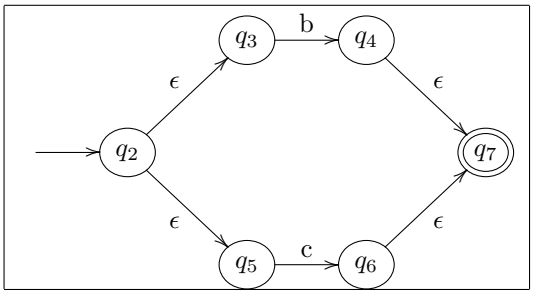


Figure 2.12: NFA machine for $r_4 =$ **b** | **c** and $r_5 =$ (**b** | **c**) .

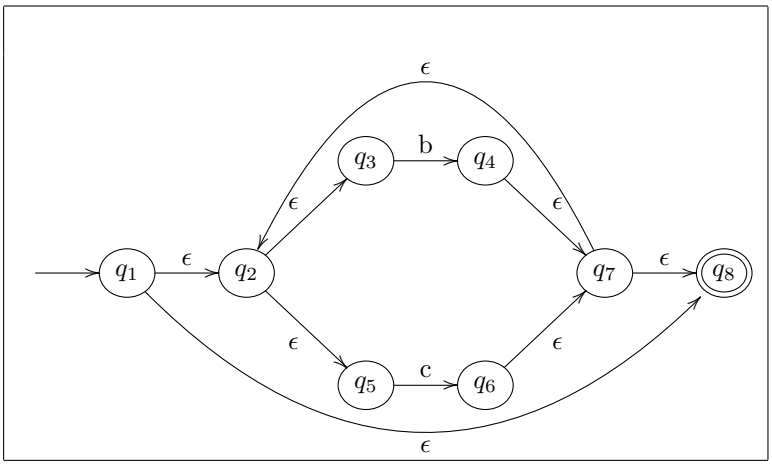


Figure 2.13: NFA machine for $r_6 =$ **(b | c)***.

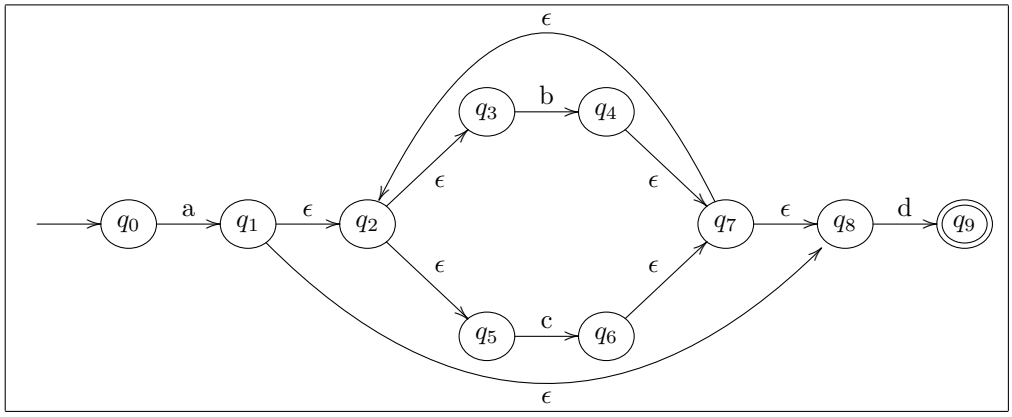


Figure 2.8: NFA machine for $r_9 = r_1 r_6 r_8 =$ **a(b|c)*d**.

space complexity of $O(2^n)$ where n is the number of states in the NFA, N. In the worst case, the number of states in the constructed DFA is 2^n. However, this is usually not the case for the DFAs used for scanners in translators for most programming languages. When using regular expressions for multiple pattern matching instances, this algorithm is acceptable. If the regular expression will be used once or is temporary, it's best to convert to the NFA and then run the NFA with a backtracking algorithm on the proliferation of states. ■

2.8.1 Regular Expressions Constructed from FSAs

A regular expression, r, can be constructed from a DFAs, M_D, or NFA, M_N, such that $L(r) = L(M_D) = L(M_N)$. Of course can construct a DFA M_D from an NFA, M_N and then construct a regular expression. This construction can be useful when given an FSA in a specification and one needs to implement it in software. With an equivalent regular expression we can use a software tool like Flex or Antrl4 to construct the implementation of the equivalent DFA.

Theorem 2.8.2 *For each FSM, M, such that $L(r) = L(M_D)$.*

We do not show a proof here or given an algorithm for constructing regular expressions from FSAs. Instead, examples of constructing regular expressions from FSAs are given.

Example 2.17 ***Regular Expression for*** $L(M_{2.11})$

The regular expression for the language recognized by the DFA first shown in Example 2.11 (and whose state diagram is presented in Figure 2.15) is

$$\underline{a}^*\underline{b}\underline{a}^*(\underline{b}\underline{a}^*\underline{b}\underline{a}^*)^*$$

The state diagram of $M_{2.11}$ is shown below. ■

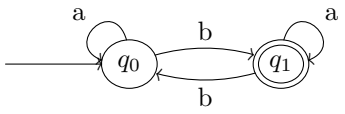


Figure 2.15: State diagram of $M_{2.11}$

Example 2.18 ***Regular Expression for*** $L(M_{2.12})$

The regular expression for the DFA first shown in Example 2.12 (and whose state diagram is presented in Figure 2.16) is

$$\underline{a}^*\underline{b}\underline{a}^*\underline{b}\underline{a}^*(\underline{c}\underline{a}^*\underline{b}\underline{a}^*\underline{b}\underline{a}^*)^*$$

■

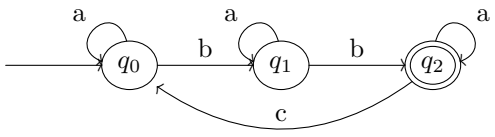


Figure 2.16: State diagram for $M_{2.12}$

Example 2.19 ***Two Regular Expressions for a NFA:*** $L(M_{2.19})$

The state diagram from Example 2.14 is shown below along with two (equivalent) regular expressions that describe the language recognized by $M_{2.14}$*. The state diagram for* $M_{2.19}$ *is presented in Figure 2.17.*

$$\underline{a}(\underline{ba})^*|\underline{aa}(\underline{ba})^* = (\underline{a}|\underline{aa})(\underline{ba})^*$$

■

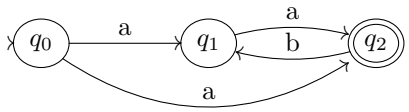


Figure 2.17: State diagram for $M_{2.14}$

From Example 2.19 we see that regular expression identities can be applied to give alternative expressions or, in some cases, simpler expressions in the sense that there are fewer operators.