

## 2.1 Preface

To start things off, let's look at a familiar problem for most programmers. The problem is to check whether an input is a correctly formed input. We will state the problem as follows.

**Problem 1.** [THE RECOGNITION PROBLEM] *Write an algorithm to recognize input strings that have a certain property.*

**Example 1.** *Suppose we represent the set of file names satisfying some property using the DOS format. For example, to represent all files with name started with characters `cs` and ended with characters `html`, we write them as `cs*.html`, where `*` represents any sequence of characters. Then the question is how to write a program that recognize the input with format `cs*.html`?*

In this Chapter, we will study a special class of languages, namely, the *Regular language*. Remember that, this course is about the relationships between languages, machines, and grammars. Thus, we will also study the machines that can recognize the *Regular language*, the grammars that can generate the *Regular language*. Here a *language* is a set of strings over a finite alphabet, where a *string* is the concatenation of zero or more symbols from the alphabet.

## 2.2 Basic Concepts and Terminologies

### 2.2.1 Basic Definitions

Some terminologies used in this course include the follows. For more, please see the textbook from page 14 to page 29.

**Alphabet:** It is finite set of symbols, often denoted by  $\Sigma$ . For example, the alphabet of English is  $a, b, c, \dots, x, y, z$ , and  $A, B, C, \dots, X, Y, Z$ , punctuation, space and number digits  $0, 1, \dots, 8, 9$ . For the sake of convenience, we often use  $a - z$  to denote the alphabets  $a, b, c, \dots, x, y, z$ . Similarly to use  $A - Z$  to denote all capital English alphabets, use  $0 - 9$  to denote all decimal digits.

For computer, the alphabets are often binary  $0, 1$ .

**String:** It is finite sequence of symbols taken from alphabet  $\Sigma$ . For example, in English: *Hello world*. A sequence of binary bits  $011001$  is a string. Notice that the set of all strings is denoted by  $\Sigma^*$ . For example,  $\{0, 1\}^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots\}$ . It is countably infinite.

**Language:** It is set of strings, i.e., subset of  $\Sigma^*$ , distinguishing strings into those in/out of the language. A language consists of strings over an alphabet. Usually some restrictions are placed on the strings that comprise the language. The English language consists of those strings of words form sentences in a language, only those satisfying certain conditions on the order and type of the constituent's words. Consequently, a language consists of a subset of all possible strings over he alphabet.

**Machine:** It is formal abstraction of a computer based on states and transitions between states. It computes some output from input. Typically, there are three uses of machine, which can be related to each other.

**acceptance:** input = string, output = yes/no membership in language

**enumeration:** input = nothing, output = all strings in language

**function:** input = string, output = string

**Grammar:** The rules for deriving and parsing strings, akin to high school grammar rules for natural languages.

**Expression:** A meta-string that "denotes" a language.

## 2.2.2 Operations on Alphabet, Languages, or Strings

- Length:  $\|S\|$  is number of characters in  $S$ .
- The number of times a particular character  $x$  occurs in a word  $w$  is denoted by  $N_x(w)$
- Concatenation: The concatenation of two strings  $w$  and  $u$  is the string obtained by appending the symbols of  $u$  to the right end of  $w$ .

$w = a_1a_2 \dots a_n$ , and  $u = b_1b_2 \dots b_m$  then  $wu = a_1a_2 \dots a_nb_1b_2 \dots b_m$ .

The concatenation of two languages is the concatenation of any two strings from these two languages.

$$L_1L_2 = \{xy \mid x \in L_1, y \in L_2\}.$$

- Reverse: The reverse of a string is obtained by the symbols in reverse order. For example, if  $w = a_1a_2 \dots a_n$  then  $w^R = a_na_{n-1} \dots a_1$ .
- Power:

$$\Sigma^n = \{w \mid |w| = n \}.$$

$$\Sigma^* = \bigcup_{n \geq 0} \Sigma^n.$$

$$\Sigma^+ = \bigcup_{n \geq 1} \Sigma^n.$$

The second is also called *Kleene Star*.

$$L^n = \{w = w_1w_2 \dots w_m \mid w_i \in L, 1 \leq i \leq n \}.$$

$$L^* = \bigcup_{n \geq 0} L^n.$$

$$L^+ = \bigcup_{n \geq 1} L^n.$$

- Complement: The complement of a language  $L$  is the set of strings that are not in  $L$ . In other words,

$$\bar{L} = \Sigma^* - L.$$

- Suffix, prefix: A string  $y$  is a *suffix* of a string  $w$  if there exist a string (maybe empty) such that  $w = xy$ . String  $x$  is also called a prefix of  $w$ .

**Theorem 2.2.1.**  $\Sigma^*$  is countably infinite. Each  $L \in \Sigma^*$  is countable ( i.e., countably infinite or finite).

Therefore we can order the strings in a languages. Typically, we can use the lexicographic order. Given a language  $L$ , the *decision problem* of  $L$  is: given a word  $x$ , is  $x$  in  $L$ ?

### 2.2.3 Grammar

A grammar  $G$  is defined as quadruple

$$G = (V, T, S, P),$$

where

- $V$  is a finite set of symbols, called **variables**.
- $T$  is a finite set of symbols, called **terminal symbols**.
- $S \in V$  is a special symbol called **start symbol**.
- $P$  is finite set of productions.

We always assume that the productions are of the form

$$x \rightarrow y,$$

where  $x$  is an element of  $(V \cup T)^+$  and  $y$  is in  $(V \cup T)^*$ .

**Applying production rules:**

- $\Rightarrow$ : If we have  $w = uxv$  and we also have rule  $x \rightarrow y$ , then we have  $w \Rightarrow uyv$ , by replacing  $x$  by  $y$ .
- $\xRightarrow{*}$ : If  $w_1 \Rightarrow w_2 \Rightarrow \dots \Rightarrow w_n$ , we say that  $w_1$  deriving  $w_n$  and write it as

$$w_1 \xRightarrow{*} w_n.$$

Here  $*$  indicates that an unspecified number of steps can be taken to derive  $w_n$  from  $w_1$ .

**Definition 1.** Let  $G = (V, T, S, P)$  be a grammar. Then the set

$$L(G) = \{w \in T^* : S \xRightarrow{*} w\}$$

is the language generated by  $G$ .

**Example 2.** Consider the grammar  $G = (V, T, P, S)$ , where  $V = \{S, A\}$ ,  $T = \{0, 1\}$ , and the productions,  $P$ , are:

$$\begin{aligned} S &\rightarrow 0A|0 \\ A &\rightarrow 10A \end{aligned}$$

The grammar accepts all strings that starts with 0 followed by sequence of 10. This grammar corresponds to the regular expression  $0(10)^*$ , which in turn corresponds to the deterministic finite acceptor that is explained later.

**Problem 2.** Consider the grammar  $G_1 = (\{S\}, \{a, b\}, S, P)$ , where  $P$  is given by

$$S \rightarrow aSb|\lambda.$$

What is the language generated by grammar  $G_1$ ?

**Problem 3.** Consider the grammar  $G_2 = (\{S\}, \{a, b\}, S, P)$ , where  $P$  is given by

$$S \rightarrow aSb | bSa | \lambda.$$

What is the language  $L(G_2)$  generated by grammar  $G_2$ ?

**Example 3.** Give the grammar that generates the following language

$$L = \{ww^R : w \in \{a, b\}^+\}.$$

SOLUTION. One grammar is  $G_3 = (\{S\}, \{a, b\}, S, P)$ , where  $P$  is given by

$$S \rightarrow aa | bb | aSa | bSb.$$

The other grammar that generates the same language is  $G_4 = (\{S\}, \{a, b\}, S, P)$ , where  $P$  is given by

$$\begin{aligned} S &\rightarrow aAa \mid bAb \\ A &\rightarrow aAa \mid bAb \mid \lambda \end{aligned}$$

□

This grammar generates the set of all *palindromes* over the alphabet  $\{a, b\}$ . A palindrome is a string that is the same forward and backward. Example of a palindrome is

*Madam, I'm Adam*

**Definition 2.** Let  $n_a(w)$  denote the number of  $a$ 's in the string  $w$ .

**Problem 4.** Is  $L(G_2)$  equivalent to the following language  $L = \{w \mid n_a(w) = n_b(w)\}$ .

Two grammars are *equivalent* if they generate the same language.

## 2.3 Regular Languages

The collection of regular languages over  $\Sigma$  is defined inductively as follows:

**Basis** Empty set  $\phi$ , the set  $\{\phi\}$ , and  $\{a\}$  are regular languages for all  $a \in \Sigma$ ;

**Induction** If  $L_1$  and  $L_2$  are regular languages, then the following languages are also regular:  $L_1 \cup L_2$ ,  $L_1 L_2$ , and  $L_1^*$ .

## 2.4 Finite Automata

An automaton is an abstract model of a digital computer. As such, every automaton includes some essential features. It has a mechanism for reading input. It is assumed that the input is a string over a given alphabet, written on an input file, which the automaton can read but not change. The input file is divided into cells, each of which can hold one symbol. The input mechanism can read the input file left to right, one symbol at a time. The input mechanism can also detect the end of the input string. The automaton can produce output of some form. It may have temporary storage device, consisting of an unlimited number of cells, each capable of holding a single symbol from an alphabet. Finally, the automaton has a control unit, which can be in any one of a finite number of internal states, and which can change state in some defined manner. See the textbook for a schematic representation of a general automaton.

An automaton operates in a discrete time frame. At any given time, the control unit is in some internal state and the input mechanism is scanning a particular symbol on the input file. The internal state of the control unit at the next step is determined by the next-state or transition function. The transition of the automaton from one configuration to the next is called move.

Automata can be either deterministic or nondeterministic. A deterministic automaton is one in which each move is uniquely determined by current configuration. If we know the internal state,

the input, and the contents of temporary storage we can predict the future behavior of automaton exactly. Whereas a nondeterministic automata may have several possible moves at each point, so we can only predict a set of possible actions.

An automaton whose output response is limited to a simple 'yes' or 'no' is called accepter. Presented with an input string, an accepter either accepts or rejects it.

### 2.4.1 Finite Automata

An effective procedure that determines whether an input string is in a language or not is called a language acceptor or recognizer. As discussed above an automaton is a simple model of a computer. The Finite Automaton (FA) is a graph-based way of specifying patterns. The FA computes a Boolean function and is, therefore, classified as an acceptor. Acceptors are used as the membership criterion of a regular language. Regular expressions are an algebra for describing the same kinds of patterns that can be described by automata and can be converted to an automaton and vice versa.

There is no formal definition for "automaton"—instead, there are formal definitions for each kind of automaton. Generally, an automaton

- has some form of input,
- has some form of output,
- has internal states,
- may or may not have some form of storage
- is hard-wired rather than programmable.

Properties common to all machines includes the processing of input string and the generation of output. A vending machine takes coins as input and returns food or beverages as output. A combination lock expects a sequence of numbers and opens the lock if the input sequence is correct. The input to the machines introduced in this chapter consists of a string over an alphabet. The result of the computation indicates whether the input string is acceptable or not.

The previous examples exhibits a property that we take for granted in mechanical computation, determinism. When the appropriate amount of money is inserted into a vending machine, we are upset if nothing is forthcoming. Similarly, we expect the combination to open the lock and all other sequences to fail. Initially we require machines to be deterministic. This condition will be relaxed to examine the effects of nondeterminism on the capabilities of computation.

A formal definition of machine is not concerned with the hardware involved in the operation of the machine, but rather with a description as the internal operations, as the machine processes the input.

### 2.4.2 A Finite State Machine Example

A simple newspaper vending machine, similar to those found on many street corners, is used to illustrate the components of a finite state machine, assuming that the machine charges 30 cents for a newspaper. The input to the machine consists of *nickels*, *dimes*, and *quarters*. When 30 cents is

inserted, the cover of the machine may be opened and a newspaper removed. If the total of coins exceeds 30 cents, the machine graciously accepts the overpayment and does not give change.

The newspaper vending machine on the street corner has no memory, at least not as we usually conceive of memory in a computing machine. However the machine "knows" that an additional 5 cents will unlatch the cover when 25 cents has previously been inserted. The machine altering its internal state whenever input is received and processed acquires this knowledge.

A machine state represents the status of the ongoing computation. The internal operation of the vending machine can be described by the interactions of the following seven states. The names of the states, given in italics, indicate the progress made towards opening the cover.

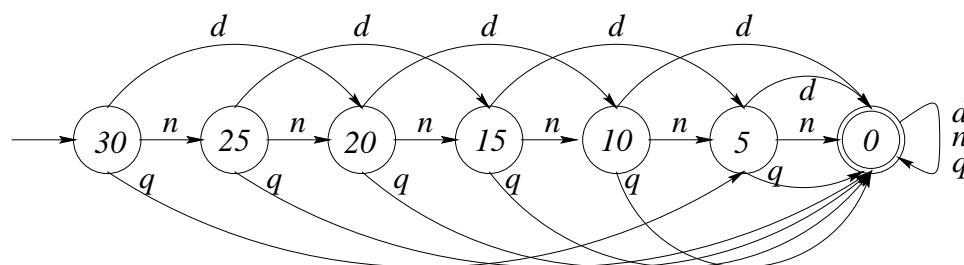
- needs 30 cents - the state of the machine before any coins are inserted.
- needs 25 cents - the state after a nickel has been input.
- needs 20 cents - the state after two nickels or a dime have been input.
- needs 15 cents - the state after three nickels or a dime and a nickel have been input.
- needs 10 cents - the state after four nickels, a dime and two nickels, or two dimes have been input.
- needs 5 cents - the state after a quarter, five nickels, two dimes and a nickel, or one dime and three nickels have been input.
- needs 0 cents - the state that represents having at least 30 cents input.

The insertion of coins causes the machine to alter its state. When 30 cents or more are input, the state needs 0 cents is entered and the latch is opened. Such a state is called accepting state since it indicates the correctness of the input.

The design of the machine must represent each of components symbolically. Rather than a sequence of coins, the input to the abstract machine is a string of symbols. A labeled directed graph is known as a state diagram or a transition graph is often used to represent the transformation of the internal state of machine. The nodes of the state diagram are the states described above. The needs  $m$  cents node is represented by  $m$  in the state diagram. The initial state for the newspaper vending machine is the node 30.

The arcs are labeled  $n$ ,  $d$ , or  $q$ , representing the input of a nickel, dime, or quarter. An arc from node  $x$  to node  $y$  labeled  $v$  specifies that processing input  $v$  when the machine as in state  $x$  causes the machine to enter state  $y$ . Figure gives the state diagram for the news paper vending machine. The arc labeled  $d$  from node 15 to 5 represents the change of state of machine when 15 cents has previously been processed and a dime is input. The cycles of length one from node 0 to itself indicate that any input that increases the total past 30 cents leaves the latch unlocked. See the following Figure 2.1 for an illustration of the vending machine.

Input to the machine consists of strings from  $\{n, d, q\}^*$ . The sequence of states entered during the processing of an input string can be traced by following the arcs in the state diagram. The machine is in its initial state at the beginning of a computation. The arc labeled by the first input symbol is traversed, specifying the subsequent machine state. The next symbol of the input string is processed by traversing the appropriate arc from the current node, the node reached by traversal



**Figure 2.1.** State Diagram of Newspaper Vending Machine.

of the previous arc. This procedure is repeated until the entire input string has been processed. The string is accepted if the computation terminates in the accepting state (which is 0 here). For example, the string  $dndn$  is accepted by the vending machine while the string  $nndn$  is not accepted since the computation terminates in state 5, which is not an accepting state.

As we will see this vending machine is a case of deterministic finite accepter as there is no choice of move provided at any state, i.e. the automaton has just one possible move for a given input at every state. It can't move to any of two next states with a given input at any intermediate or initial state. Also all the paths are clearly defined for all possible inputs at all the states, which eliminates nondeterminism.

### 2.4.3 Deterministic Finite Automata

The first type of automaton that we consider here is finite accepter that are deterministic in their operation. A *deterministic finite accepter* or *DFA*  $M$  is defined as quintuple

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

where

- $Q$  is a finite set of **internal states**.
- $\Sigma$  is a finite set of symbols, called **input alphabet**.
- $\delta : Q \times \Sigma \rightarrow Q$  is a total function called the **transition function**.
- $q_0$  is the **initial state**.
- $F \subseteq Q$  is a set of **final states**.

A deterministic finite accepter operates in the following manner. At the initial time, it is assumed to be in the initial state  $q_0$  with its input mechanism on the leftmost symbol of the input string. When the end of input string is reached, the string is accepted if the automaton is in one of its final states. Otherwise the string is rejected. The input mechanism can move only from left to right and reads exactly one symbol on each step. The transitions from one internal state to another are governed by the transition function  $\delta$ . For example, if

$$\delta(q_0, a) = q_1,$$

then if DFA is in state  $q_0$  and the current input symbol is  $a$ , the DFA will go into state  $q_1$ .

As said earlier to visualize and represent finite automata, we use transition graphs, in which the vertices represents states and the edges represent transitions. The labels on the vertices are the names of the states, while the labels on the edges are current values of the input symbol. For example, if  $q_0$  and  $q_1$  are the internal states of some DFA  $M$  then the graph associated with  $M$  will have one vertex labeled  $q_0$  and another labeled  $q_1$ . An edge  $(q_0, q_2)$  labeled  $a$  represents the transition  $\delta(q_0, a) = q_1$ . An incoming unlabeled arrow not originating at any vertex identifies the initial state. Final states are drawn with a double circle.

It is convenient to introduce the *extended transition function*  $\delta^*$

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

The second argument of  $\delta^*$  is a string rather than a single symbol, and its value gives the state the automaton will be in after the automaton reads that string. For example, if

$$\delta(q_0, a) = q_1$$

and

$$\delta(q_1, b) = q_2$$

then

$$\delta^*(q_0, ab) = q_2$$

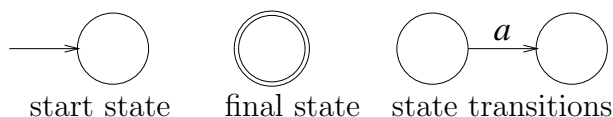
The language accepted by a DFA  $M = (Q, \Sigma, \delta, q_0, F)$  is the set of all strings on  $\Sigma$  accepted by  $M$ . Formally,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F\}.$$

DFAs are:

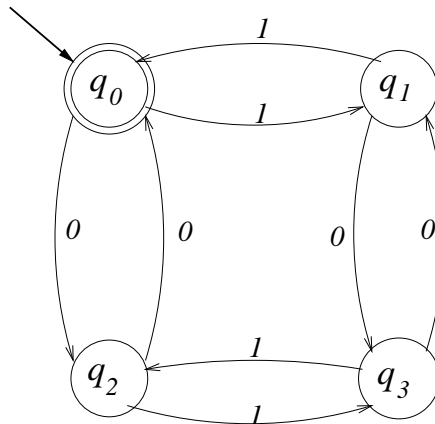
- Deterministic—there is no element of choice
- Finite—only a finite number of states and arcs
- Acceptors—produce only a yes/no answer

A DFA is drawn as a graph, with each state represented by a circle. One designated state is the *start* state. Some states (possibly including the start state) can be designated as *final* states. Arcs between states represent state *transitions* – each such arc is labeled with the symbol that triggers the transition. See the following Figure 2.2 for an illustration.



**Figure 2.2.** The start state, final state and the states transitions.

**Example 4.** The following figure gives an example of DFA. For example, this DFA accepts the string 10011100.



The basic steps to decide whether a string can be accepted by a DFA is as follows

- Start with the "current state" set to the *start* state and a "read head" at the beginning of the input string;
- While there are still characters in the string:
  - Read the next character and advance the read head;
  - From the current state, follow the arc that is labeled with the character just read; the state that the arc points to becomes the next current state;
- When all characters have been read, accept the string if the current state is a final state, otherwise reject the string.

For example, for the above DFA, given the string 10011100, the trace is  $q_01q_10q_30q_11q_01q_11q_00q_20q_0$ . Since  $q_0$  is a final state, the string is accepted.

### Implementing a DFA

If you don't object to the go to statement, there is an easy way to implement a DFA:

- $q_0$  : read char;
  - if eof then accept string;
  - if  $char = 0$  then go to  $q_2$ ;
  - if  $char = 1$  then go to  $q_1$ ;
- $q_1$  : read char;
  - if eof then reject string;
  - if  $char = 0$  then go to  $q_3$ ;
  - if  $char = 1$  then go to  $q_0$ ;

- $q_2$  : read char;
  - if eof then reject string;
  - if  $char = 0$  then go to  $q_0$ ;
  - if  $char = 1$  then go to  $q_3$ ;
- $q_3$  : read char;
  - if eof then reject string;
  - if  $char = 0$  then go to  $q_1$ ;
  - if  $char = 1$  then go to  $q_2$ ;

If you are not allowed to use a go to statement, you can fake it with a combination of a loop and a case statement as follows:

```
state :=  $q_0$ ;
loop
case state of
```

- $q_0$  : read char;
  - if eof then accept string;
  - if  $char = 0$  then  $state = q_2$ ;
  - if  $char = 1$  then  $state = q_1$ ;
- $q_1$  : read char;
  - if eof then reject string;
  - if  $char = 0$  then  $state = q_3$ ;
  - if  $char = 1$  then  $state = q_0$ ;
- $q_2$  : read char;
  - if eof then reject string;
  - if  $char = 0$  then  $state = q_0$ ;
  - if  $char = 1$  then  $state = q_3$ ;
- $q_3$  : read char;
  - if eof then reject string;
  - if  $char = 0$  then  $state = q_1$ ;
  - if  $char = 1$  then  $state = q_2$ ;

```
end case;
end loop;
```

**Problem 5.** What is the language accepted by the above DFA?

### 2.4.4 Nondeterministic Finite Automata

Nondeterminism means a choice of move for an automaton. Rather than prescribing a unique move in each situation, we allow a set of possible moves. Formally, we achieve this by defining the transition function so that its range is a set of possible states.

A *nondeterministic finite acceptor* or *NFA*  $M$  is defined as quintuple

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

where

- $Q$  is a finite set of .
- $\Sigma$  is a finite set of symbols, called .
- $\delta : Q \times (\Sigma \cup \{\lambda\}) \rightarrow 2^Q$  is a total function called the **trans**.
- $q_0$  is the **initial** state.
- $F \subseteq Q$  is a set of .

In nondeterministic acceptor, the range of  $\delta$  is in the power set  $2^Q$ , so that its value is not a single element of  $Q$ , but a subset of it. This subset defines the set of all possible states that can be reached by the transition.

Notice that the definition of  $\delta^*$  is not same as DFA.  $\delta^*(q_i, w)$  contains state  $q_j$  if and only if there is a walk in the transition graph from  $q_i$  to  $q_j$  labelled by  $w$ .

The language accepted by a NFA  $M = (Q, \Sigma, \delta, q_0, F)$  is the set of all strings on  $\Sigma$  accepted by  $M$ . Formally,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \cap F \neq \Phi\}.$$

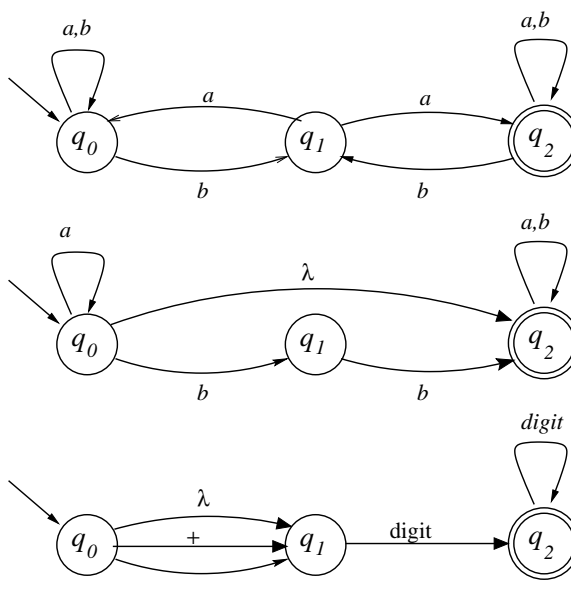
A finite-state automaton can be nondeterministic in either or both of two ways:

- A state may have two or more arcs emanating from it labeled with the same symbol. When the symbol occurs in the input, either arc may be followed.
- A state may have one or more arcs emanating from it labeled with (the empty string) . These arcs may optionally be followed without looking at the input or consuming an input symbol.

Due to nondeterminism, the same string may cause an NFA to end up in one of several different states, some of which may be final states while others are not. The string is accepted if there is one possible ending state that is a final state.

**Example 5.** *The following Figure 2.3 demonstrates some NFAs*

**Example 6.** *Show that the language  $L = \{a^n \mid n = i + jk, i, k \text{ fixed}, j = 0, 1, 2, \dots\}$  is regular by constructing an automaton accepting this language.*



**Figure 2.3.** Three examples of NFA.

### 2.4.5 Implementing an NFA

If you think of an automaton as a computer, how does it handle nondeterminism? There are two ways that this could, in theory, be done:

- When the automaton is faced with a choice, it always (magically) chooses correctly. We sometimes think of the automaton as consulting an oracle which advises it as to the correct choice.
- When the automaton is faced with a choice, it spawns a new process, so that all possible paths are followed simultaneously.

The first of these alternatives, using an oracle, is sometimes attractive mathematically. But if we want to write a program to implement an NFA, that isn't feasible. There are three ways, two feasible and one not yet feasible, to simulate the second alternative:

- Use a *recursive backtracking algorithm*. Whenever the automaton has to make a choice, cycle through all the alternatives and make a recursive call to determine whether any of the alternatives leads to a solution (final state).
- Maintain a *state set* or a state vector, keeping track of all the states that the NFA could be in at any given point in the string.
- Use a *quantum computer*. Quantum computers explore literally all possibilities simultaneously. They are theoretically possible, but are at the cutting edge of physics. It may (or may not) be feasible to build such a device.

## Recursive Implementation of NFAs

An NFA can be implemented by means of a recursive search from the start state for a path (directed by the symbols of the input string) to a final state. Here is a rough outline of such an implementation:

**Algorithm 1.** *function NFA (state A) returns Boolean:*

```

local state B, symbol x;
for each transition from state A to some state B do
if NFA(B) then return True;
if there is a next symbol then {
    read next symbol (x);
    for each x transition from state A to some state B do
        if NFA(B) then return True;
    return False;
} else { if A is a final state then return True;
        else return False;
}

```

One problem with this implementation is that it could get into an infinite loop if there is a cycle of transitions. This could be prevented by maintaining a simple counter.

## State-Set Implementation of NFAs

Another way to implement an NFA is to keep either a state set or a bit vector of all the states that the NFA could be in at any given time. Implementation is easier if you use a bit-vector approach ( $v[i]$  is True if and only if state  $i$  is a possible state after processing current input), since most languages provide vectors, but not sets, as a built-in data type. However, it's a bit easier to describe the algorithm if you use a state-set approach, so that's what we will do. The logic is the same in either case.

**Algorithm 2.** *function NFA (state set A) returns Boolean:*

```

local state set B = {}, state a, state b, state c, symbol x;
while (there is new state added to A) {
for each a in A do
    for each λ transition from a to some state b do
        add b to A;
}
while (there is a next symbol) do {
    read next symbol (x);
    for each a in A do {
        for each x transition from a to some state b do
            add b to B;
        for each λ transition from a to some state b do
            add b to B;
    }
}

```

```

A := B;
}
if any element of A is a final state then
    return True;
else
    return False;

```

## 2.4.6 Equivalence

Though there is a difference in definition of DFA and NFA, but it does not imply that there is any essential distinction between them. To explore this matter, we introduce the notion of equivalence between automata.

**Definition 3.** *Two acceptors  $M_1$  and  $M_2$  are equivalent if they accept the same language.*

**Theorem 2.4.1.** *NFAs and DFAs are equivalent. In other words, given a NFA there is a DFA that accepts the exact same language; and given a DFA there is a NFA that accepts the exact same language.*

PROOF. A DFA is just a special case of an NFA that happens not to have any null transitions or multiple transitions on the same symbol. So DFAs are not more powerful than NFAs.

Assume that we have a NFA  $M = (Q, \Sigma, \delta, q_0, F)$ . We prove using construction to prove that there is a DFA that accepts the same language as the given NFA. The trick is how to define the states in the DFA. We use all subsets of states from the NFA as the states of DFA. Each state in the DFA summarizes all the states that the NFA might be in. If the NFA contains  $|Q|$  states, the resulting *DFA* could contain as many as  $2^{|Q|}$  states. (Usually far fewer states will be needed.) The transition function is defined as follows. Assume that  $q_1$  and  $q_2$  are two states of the constructed *DFA*,  $\delta'(q_1, x) = q_2$ , if and only if there is some state  $p_1 \in q_1$  and  $p_1 \in q_1$  such that  $\delta(p_1, x) = p_2$ .

We leave it as off-class question to prove that the constructed *DFA* does generate the same language as the given *NFA*.  $\square$

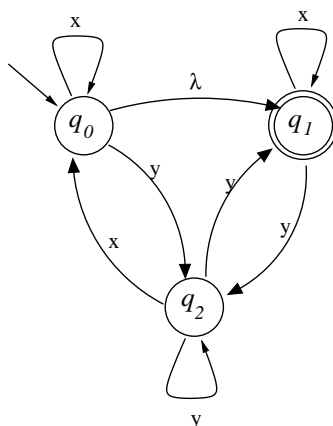
For any NFA, we can construct an equivalent DFA (see below). So NFAs are not more powerful than DFAs. DFAs and NFAs define the same class of languages – the regular languages, which is also proved later.

## 2.4.7 Reductions

Consider the NFA illustrated by the following Figure 2.4: What states can we be in (in the NFA) before reading any input? Obviously, the start state,  $q_0$ . Notice that there is a  $\lambda$ -transition from  $q_0$  to  $q_1$ , so we could also be in state  $q_1$ . For the DFA, we construct the composite state  $\{q_0, q_1\}$  as the start state.

For the NFA  $M = (Q, \Sigma, \delta, q_0, F)$ , Figure 2.1 illustrates the generalized state transitions for the NFA.

We then study the transition from state  $\{q_0, q_1\}$  when the input is  $x$ . From  $q_0$ ,  $x$  takes us to  $q_0$  (in the NFA), and the null transition might take us to  $q_1$ ; from  $q_1$ ,  $x$  takes us to  $q_1$ . So in the DFA,  $x$  takes us from  $\{q_0, q_1\}$  to  $\{q_0, q_1\}$ .



**Figure 2.4.** An NFA to be transformed to DFA.

$\delta^*(q_0, x) = \{q_0, q_1\}$	$\delta^*(q_0, y) = \{q_2\}$
$\delta^*(q_1, x) = \{q_1\}$	$\delta^*(q_1, y) = \{q_2\}$
$\delta^*(q_2, x) = \{q_0, q_1\}$	$\delta^*(q_2, y) = \{q_1\}$

**Table 2.1.** State transitions.

State  $\{q_0, q_1\}$  also needs a transition for  $y$ . In the NFA,  $\delta(q_0, y) = q_2$  and  $\delta(q_1, y) = q_2$ , so we need to add a state  $\{q_2\}$  and an arc  $y$  from  $\{q_0, q_1\}$  to  $\{q_2\}$ .

In the NFA,  $\delta(q_2, x) = q_0$ , but then a null transition might or might not take us to  $q_1$ , so we need to add an arc  $x$  from  $\{q_2\}$  to  $\{q_0, q_1\}$ .

Also, there are two arcs from  $q_2$  labeled  $y$ , going to states  $q_0$  and  $q_1$ . So in the DFA we need to add the state  $\{q_1, q_2\}$  and the arc  $y$  from  $\{q_2\}$  to this new state  $\{q_1, q_2\}$ .

In the NFA,  $\delta(q_1, x) = q_1$  and  $\delta(q_2, x) = q_0$  (and by a transition we might get back to  $q_1$ ), so we need an  $x$  arc from  $\{q_1, q_2\}$  to  $\{q_0, q_1\}$ .

$\delta(q_1, y) = q_2$ , while  $\delta(q_2, y)$  is either  $q_1$  or  $q_2$ , so we have an arc labeled  $y$  from  $\{q_1, q_2\}$  to  $\{q_1, q_2\}$ .

We now have a transition from every state for every symbol in the DFA. See the following Figure 2.5 for illustration.

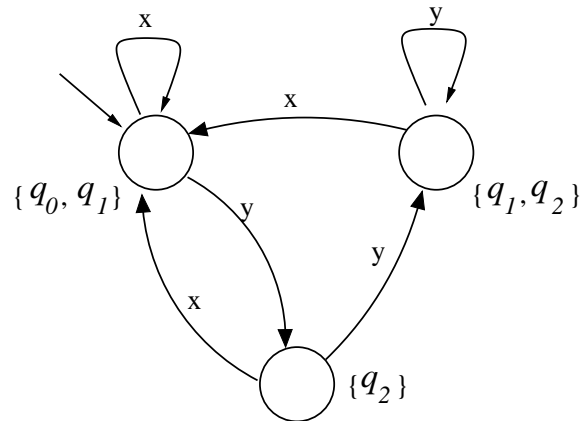
The only remaining chore is to mark all the final states. In the original NFA,  $q_1$  is a final state, so in the DFA, every state containing  $q_1$  is a final state. See the Figure 2.6 for the final DFA.

### 2.4.8 Reduce Number of States in DFA

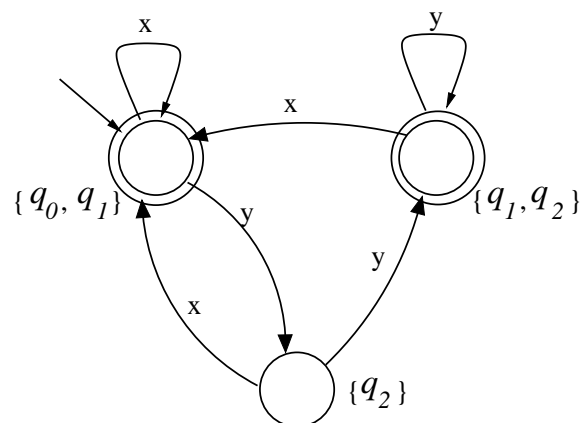
For a given language, there are many DFA's that accept it. There may be considerable difference in the number of states of such equivalent automata. Two DFA's may be accepting the same language but the selection of one acceptor for practical settings needs a close inspection and if needed then doing minimization of the acceptor, that is removing unnecessary features from the acceptor. These unnecessary features may be either inaccessible states, which can never be reached from initial state, or some indistinguishable pairs. Let's first define distinguishable and indistinguishable pairs.

**Definition 4.** Two states  $p$  and  $q$  of a DFA are called indistinguishable if

$$\delta^*(p, w) \in F \quad \text{implies} \quad \delta^*(q, w) \in F$$



**Figure 2.5.** The DFA transformed from NFA without selecting the final states.



**Figure 2.6.** The DFA transformed from NFA after selecting the final states.

$q_0$	$O$			
$q_1$	$O$	$O$		
$q_3$	$X$	$X$	$O$	
$q_5$	$X$	$X$	$O$	$O$
	$q_0$	$q_1$	$q_3$	$q_5$

**Table 2.2.** The indistinguishable and distinguishable relations of all pairs of states.

and

$$\delta^*(p, w) \notin F \quad \text{implies} \quad \delta^*(q, w) \notin F$$

for all  $w \in \Sigma^*$ . If, on the other hand, there exists some string  $w \in \Sigma^*$  such that

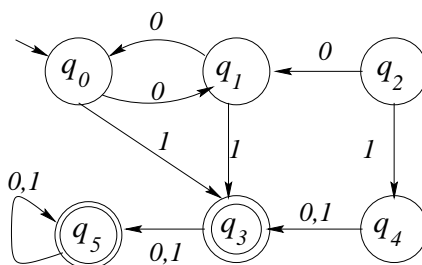
$$\delta^*(p, w) \in F \quad \text{and} \quad \delta^*(q, w) \notin F$$

or vice versa, then the states  $p$  and  $q$  are said to be distinguishable by a string  $w$ .

**Lemma 2.4.2.** Two states  $p$  and  $q$  of a DFA are indistinguishable if for any character  $x \in \Sigma$ ,  $\delta(p, x)$  and  $\delta(q, x)$  are indistinguishable.

**Lemma 2.4.3.** Two states  $p$  and  $q$  of a DFA are distinguishable if there exists a character  $x \in \Sigma$  such that  $\delta(p, x)$  and  $\delta(q, x)$  are distinguishable.

**Example 7.** Let's see how these unnecessary states can be removed with the help of one example illustrated by Figure 2.7.



**Figure 2.7.** The DFA with unnecessary states could be simplified.

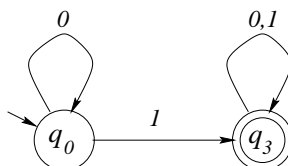
**SOLUTION.** First, remove inaccessible states  $q_2$  and  $q_4$ , as they can not be reached from initial state. Then find the pairs of distinguishable states. Here we use character  $O$  to denote two states indistinguishable,  $X$  for distinguishable. The indistinguishable and distinguishable relations of all pairs of states are illustrated by the following table. First, any non-final state and any final state is distinguishable by selecting  $w = \lambda$ . Then we use the above two lemmas to derive all the relations between pairs of states.

Consequently, we know that states  $q_0$  and  $q_1$  are not distinguishable; states  $q_3$  and  $q_5$  are not distinguishable.

Let  $M = (Q, \Sigma, \delta, q_0, F)$  denote the original DFA without simplification. The final simplified DFA  $M' = (\bar{Q}, \Sigma, \tilde{\delta}, \bar{q}_0, \bar{F})$  is constructed as follows. Here  $\bar{q}$  is the set of states that are not distinguishable from state  $q$ . The the simplified DFA has the following states

$$\bar{Q} = \{\bar{q} \mid q \in Q\}$$

The transition function  $\tilde{\delta}$  of the simplified DFA is defined as  $\tilde{\delta}(\bar{p}, x) = \overline{\delta(p, x)}$ . The final states set is defined as  $\bar{F} = \{\bar{q} \mid q \in F\}$ .  $\square$



**Figure 2.8.** The simplified DFA with the minimum number of states.

## 2.5 Regular expressions

### 2.5.1 Primitive Regular Expressions

A regular expression can be used to define a language. A regular expression represents a "pattern" strings that match the pattern are in the language, strings that do not match the pattern are not in the language. As usual, the strings are over some alphabet .

The following are primitive regular expressions:

- $x$ , for each  $x \in \Sigma$  ,
- $\lambda$ , the empty string, and
- $\phi$ , indicating no strings at all.

Thus, if the size of the alphabet is  $n$ , then there are  $n + 2$  primitive regular expressions defined over it.

Here are the languages defined by the primitive regular expressions:

- For each  $x$  , the primitive regular expression  $x$  denotes the language  $\{x\}$ . That is, the only string in the language is the string "x".
- The primitive regular expression  $\lambda$  denotes the language  $\{\lambda\}$ . The only string in this language is the empty string.
- The primitive regular expression  $\phi$  denotes the language  $\{\}$ . There are no strings in this language.

## 2.5.2 Regular Expressions

Every primitive regular expression is a regular expression. We can compose additional regular expressions by applying the following rules a finite number of times:

- If  $r_1$  is a regular expression, then so is  $(r_1)$ .
- If  $r_1$  is a regular expression, then so is  $r_1^*$ .
- If  $r_1$  and  $r_2$  are regular expressions, then so is  $r_1r_2$ .
- If  $r_1$  and  $r_2$  are regular expressions, then so is  $r_1 + r_2$ .

Here's what the above notation means:

- Parentheses are just used for grouping.
- The postfix star indicates zero or more repetitions of the preceding regular expression. Thus, if  $x$  is a regular expression, then the regular expression  $x^*$  denotes the language  $\{\lambda, x, xx, xxx, \dots\}$ .
- Concatenation of  $r_1$  and  $r_2$  indicates any string described by  $r_1$  immediately followed by any string described by  $r_2$ . For example the regular expression  $xy$  describes the language  $\{xy\}$ .
- The plus sign, read as "or," denotes the language containing strings described by either of the component regular expressions. For example the regular expression  $x + y$  describes the language  $\{x, y\}$ .

$\mathbb{P} \setminus \times : *$  binds most tightly, then concatenation, then  $+$ . For example,  $a + bc^*$  denotes the language  $\{a, b, bc, bcc, bccc, bcccc, \dots\}$ .

## 2.5.3 Languages Defined by Regular Expressions

There is a simple correspondence between regular expressions and the languages they denote:

Regular expression	$L(\text{regular expression})$
$x$ , for each $x \in \Sigma$	$\{x\}$
$\lambda$	$\{\lambda\}$
$\phi$	$\{\}$
$(r_1)$	$L(r_1)$
$r_1^*$	$L(r_1)^*$
$r_1r_2$	$L(r_1)L(r_2)$
$r_1 + r_2$	$L(r_1) \cup L(r_2)$

## 2.5.4 Building Regular Expressions

Here are some hints on building regular expressions. We will assume  $\Sigma = \{a, b, c\}$

- Zero or more.

$a^*$  means "zero or more  $a$ 's". To say "zero or more  $ab$ 's," that is,  $\{\lambda, ab, abab, ababab, \dots\}$ , you need to say  $(ab)^*$ . Don't write it as  $ab^*$ , because that denotes the language  $\{a, ab, abb, abbb, abbbb, \dots\}$ .

- One or more.

Since  $a^*$  means "zero or more  $a$ 's", you can use  $aa^*$  (or equivalently,  $a^*a$ ) to mean "one or more  $a$ 's." Similarly, to describe "one or more  $ab$ 's," that is,  $\{\lambda, ab, abab, ababab, \dots\}$ , you can use  $ab(ab)^*$ .

- Zero or one.

You can describe an optional  $a$  with  $(a + \lambda)$ .

- Any string at all.

To describe any string at all (with  $\Sigma = \{a, b, c\}$ ), you can use  $(a + b + c)^*$  or just simply  $\Sigma^*$ .

- Any nonempty string.

This can be written as any character from  $\Sigma$  followed by any string at all:  $(a+b+c)(a+b+c)^*$ .

- Any string not containing....

To describe any string at all that doesn't contain an  $a$  (with  $\Sigma = \{a, b, c\}$ ), you can use  $(b+c)^*$ .

- Any string containing exactly one...

To describe any string that contains exactly one  $a$ , put "any string not containing an  $a$ ," on either side of the  $a$ , like this:  $(b+c)^*a(b+c)^*$ .

## 2.5.5 Example Regular Expressions

**Example 8.** Give regular expressions for the following languages on  $\Sigma = \{a, b, c\}$ .

All strings containing exactly one  $a$ .

SOLUTION.  $(b+c)^*a(b+c)^*$  □

**Example 9.** All strings containing no more than three  $a$ 's.

SOLUTION. We can describe the string containing zero, one, two, or three  $a$ 's (and nothing else) as  $(\lambda + a)(\lambda + a)(\lambda + a)$ . Now we want to allow arbitrary strings not containing  $a$ 's at the places marked by  $X$ 's:

$$X(\lambda + a)X(\lambda + a)X(\lambda + a)X.$$

So we put in  $(b+c)^*$  for each  $X$ . Consequently, the following regular expression

$$(b+c)^*(\lambda + a)(b+c)^*(\lambda + a)(b+c)^*(\lambda + a)(b+c)^*$$

denotes the regular expression. □

**Example 10.** All strings which contain at least one occurrence of each symbol in .

SOLUTION. The problem here is that we cannot assume the symbols are in any particular order. We have no way of saying "in any order", so we have to list the possible orders:

$$abc + acb + bac + bca + cab + cba$$

To make it easier to see what's happening, let's put an  $X$  in every place we want to allow an arbitrary string:

$$XaXbXcX + XaXcXbX + XbXaXcX + XbXcXaX + XcXaXbX + XcXbXaX$$

Finally, replacing the  $X$ 's with  $(a + b + c)^*$  gives the final (unwieldy) answer:

$$\begin{aligned} &(a + b + c)^* a(a + b + c)^* b(a + b + c)^* c(a + b + c)^* + \\ &(a + b + c)^* a(a + b + c)^* c(a + b + c)^* b(a + b + c)^* + \\ &(a + b + c)^* b(a + b + c)^* a(a + b + c)^* c(a + b + c)^* + \\ &(a + b + c)^* b(a + b + c)^* c(a + b + c)^* a(a + b + c)^* + \\ &(a + b + c)^* c(a + b + c)^* a(a + b + c)^* b(a + b + c)^* + \\ &(a + b + c)^* c(a + b + c)^* b(a + b + c)^* a(a + b + c)^* \end{aligned}$$

□

**Example 11.** All strings which contain no runs of  $a$ 's of length greater than two.

SOLUTION. We can fairly easily build an expression containing no  $a$ , one  $a$ , or one  $aa$ :

$$(b + c)^*(\lambda + a + aa)(b + c)^*$$

but if we want to repeat this, we need to be sure to have at least one non- $a$  between repetitions:

$$(b + c)^*(\lambda + a + aa)(b + c)^*((b + c)(b + c)^*(\lambda + a + aa)(b + c)^*)^*$$

□

**Example 12.** All strings in which all runs of  $a$ 's have lengths that are multiples of three.

SOLUTION.  $(aaa + b + c)^*$

□

## 2.6 Regular Expressions Denote Regular Languages

Languages described by deterministic finite acceptors (DFAs) are called regular languages. For any nondeterministic finite acceptor (NFA) we can find an equivalent DFA. Thus NFAs also describe regular languages.

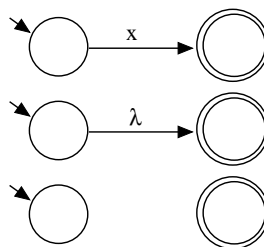
Regular expressions also describe regular languages. We will show that regular expressions are equivalent to NFAs by doing two things:

- For any given regular expression, we will show how to build an NFA that accepts the same language.
- For any given NFA, we will show how to construct a regular expression that describes the same language.

### 2.6.1 From Primitive Regular Expressions to NFAs

Every NFA we construct will have a single start state and a single final state. We will build more complex NFAs out of simpler NFAs, each with a single start state and a single final state. The simplest NFAs will be those for the primitive regular expressions.

For any  $x$  in  $\Sigma$ , the regular expression  $x$  denotes the language  $\{x\}$ . The left figure of the following Figure 2.9 represents exactly that language.



**Figure 2.9.** The NFAs for  $\{x\}$ ,  $\{\lambda\}$  and empty language  $\{\}$ .

Note that if this were a DFA, we would have to include arcs for all the other elements of  $\Sigma$ .

The regular expression  $\lambda$  denotes the language  $\{\lambda\}$ , that is, the language containing only the empty string.

The regular expression  $\phi$  denotes the language  $\{\}$ , i.e., no strings belong to this language, not even the empty string.

Since the final state is unreachable in the NFA for the language  $\{\}$ , why bother to have it at all? The answer is that it simplifies the construction if every NFA has exactly one start state and one final state. We could do without this final state, but we would have more special cases to consider, and it doesn't hurt anything to include it.

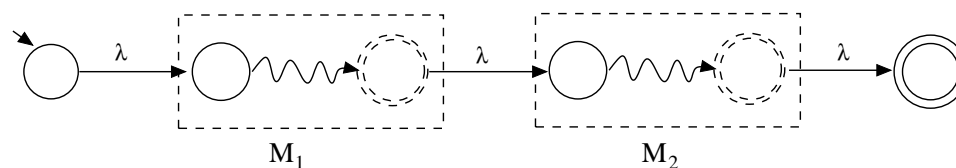
### 2.6.2 From Regular Expressions to NFAs

We will build more complex NFAs out of simpler NFAs, each with a single start state and a single final state. Since we have NFAs for primitive regular expressions, we need to compose them for the operations of grouping, concatenation, union, and Kleene star (\*).

For grouping (parentheses), we don't really need to do anything. The NFA that represents the regular expression  $(r_1)$  is the same as the NFA that represents  $r_1$ .

For concatenation (strings in  $L(r_1)$  followed by strings in  $L(r_2)$ ), we simply chain the NFAs together, as shown in the following Figure 2.10. Assume that automata  $M_1$  and  $M_2$  are the NFAs that accept the languages  $L(r_1)$  and  $L(r_2)$  respectively. We construct an NFA  $M$  to accept  $L(r_1)L(r_2)$  as follows. The initial and final states of the original NFAs  $M_1$  and  $M_2$  (boxed) stop being initial and final states. We add new initial and final states for the automaton  $M$  accepting the language  $L(r_1)L(r_2)$ . We add the following arcs:

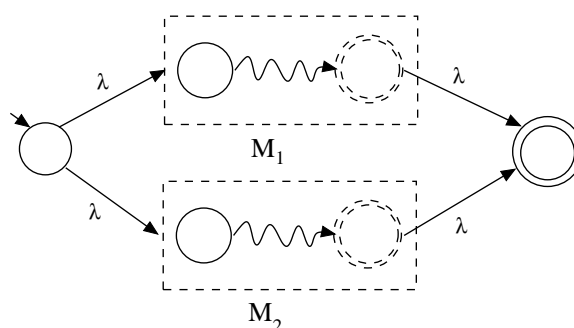
- an arc from the initial state of  $M$  to the initial state of  $M_1$  with label  $\lambda$ ;
- an arc from the final state of  $M_1$  to the initial state of  $M_2$  with label  $\lambda$ ;
- an arc from the final state of  $M_2$  to the final state of  $M$  with label  $\lambda$ .



**Figure 2.10.** The NFA for  $r_1r_2$ .

The  $+$  denotes "or" in a regular expression, so it makes sense that we would use an NFA with a choice of paths. (This is one of the reasons that it's easier to build an NFA than a DFA.) We construct an NFA  $M$  to accept  $L(r_1) \cup L(r_2)$  as follows. The initial and final states of the original NFAs  $M_1$  and  $M_2$  (boxed) stop being initial and final states. We add new initial and final states for the automaton  $M$  accepting the language  $L(r_1) \cup L(r_2)$ . (We could construct a NFA that accepting the language  $L(r_1) \cup L(r_2)$  with fewer states and fewer transitions here, but we aren't trying for the best construction; we're just trying to show that a construction is possible.) We add the following four arcs:

- an arc from the initial state of  $M$  to the initial state of  $M_1$  with label  $\lambda$ ;
- an arc from the initial state of  $M$  to the initial state of  $M_2$  with label  $\lambda$ ;
- an arc from the final state of  $M_1$  to the final state of  $M$  with label  $\lambda$ ;
- an arc from the final state of  $M_2$  to the final state of  $M$  with label  $\lambda$ .



**Figure 2.11.** The NFA for  $r_1 + r_2$ .

The star denotes zero or more applications of the regular expression, so we need to set up a loop in the NFA. We can do this with a backward-pointing arc. Since we might want to traverse the regular expression zero times (thus matching the null string), we also need a forward-pointing arc to bypass the NFA entirely. We construct an NFA  $M$  to accept  $L(r_1)^*$  as follows. The initial and final states of the original NFA  $M_1$  (boxed) stop being initial and final states. We add new initial and final states for the automaton  $M$  accepting the language  $L(r_1)^*$ . We add the following four arcs:

- an arc from the initial state of  $M$  to the initial state of  $M_1$  with label  $\lambda$ ;
- an arc from the initial state of  $M$  to the final state of  $M$  with label  $\lambda$ ;

- an arc from the final state of  $M_1$  to the final state of  $M$  with label  $\lambda$ ;
- an arc from the final state of  $M$  to the initial state of  $M$  with label  $\lambda$ .

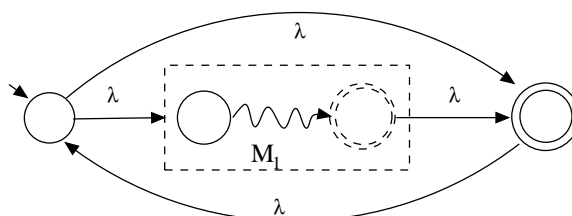


Figure 2.12. The NFA for  $r_1^*$ .

### 2.6.3 From NFAs to Regular Expressions

Creating a regular expression to recognize the same strings as an NFA is trickier than you might expect, because the NFA may have arbitrary loops and cycles. Here's the basic approach:

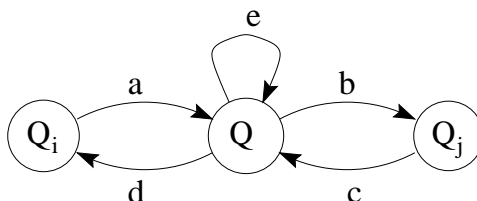
1. If the NFA has more than one final state, convert it to an NFA with only one final state as follows: add a new final state; make the original final states non-final; and add a  $\lambda$ -transition from each old final state to the new (single) final state.
2. Consider the NFA to be a *generalized transition graph*, which is just like an NFA except that the edges may be labeled with arbitrary regular expressions. Since the labels on the edges of an NFA may be either  $\lambda$  or members of  $\Sigma$ , each of these can be considered to be a regular expression.
3. Remove states one by one from the NFA, relabeling edges as you go, until only the initial and the final state remain.
4. Read the final regular expression from the two-states automaton that results.

The regular expression derived in the final step accepts the same language as the original NFA.

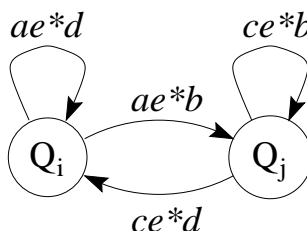
**Remark 1.** *Since we can convert an NFA to a regular expression, and we can convert a regular expression to an NFA, the two are equivalent formalisms—that is, they both describe the same class of languages, the regular languages.*

There are two complicated parts to extracting a regular expression from an NFA: *removing states*, and *reading the regular expression* off the resultant two-state generalized transition graph.

We first consider how to remove a state, say  $Q$  from the NFA. To delete state  $Q$ , where  $Q$  is neither the initial state nor the final state, we have to guarantee that the removing of  $Q$  does not change the language accepted by the NFA. Consider any two states  $Q_i$  and  $Q_j$  of all states  $\{Q_1, Q_2, \dots, Q_k\}$ . Assume that all transition functions (in generalized transition graph) using only  $Q$ ,  $Q_i$  and  $Q_j$  is as follows:



**Figure 2.13.** The state transitions using only  $Q$ ,  $Q_i$  and  $Q_j$ .



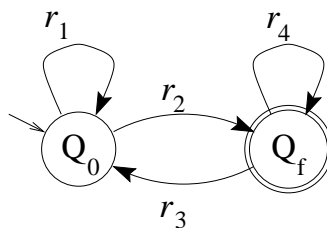
**Figure 2.14.** The state transitions using only  $Q_i$  and  $Q_j$  without using  $Q$ .

Here  $a, b, c, d, e$  are regular expressions. Some of these regular expressions may be  $\phi$ , or  $\lambda$ . We replace the above transition structures with the following structure.

You should convince yourself that this transformation is "correct", in the sense that paths which leave you in  $Q_i$  in the original NFA will leave you in  $Q_i$  in the replacement, and similarly for  $Q_j$ .

What if state  $Q$  has connections to more than two other states, say,  $Q_i, Q_j$ , and  $Q_k$ ? Then you have to consider these states pairwise:  $Q_i$  with  $Q_j$ ,  $Q_i$  with  $Q_k$  and  $Q_j$  with  $Q_k$ ?

What if some of the arcs in the original state are missing? There are too many cases to work this out in detail, but you should be able to figure it out for any specific case, using the above as a model. Actually, the above transformation gives the solutions to all cases. If the some arc is missing from the Figure 2.13, it means that the corresponding regular expression is  $\phi$ . Notice that  $\phi r$  ( $\phi$  concatenated by any regular expression  $r$ ) is still  $\phi$ .



**Figure 2.15.** The final NFA using only the start state  $Q_0$  and the final state  $Q_f$ .

You will end up with an NFA that looks like the Figure 2.15, where  $r_1, r_2, r_3$ , and  $r_4$  are (probably very complex) regular expressions. The resultant NFA represents the regular expression

$$r_1^* r_2 (r_4 + r_3 r_1^* r_2)^*$$

All you have to do is plug in the correct values for  $r_1, r_2, r_3$ , and  $r_4$ .

## 2.7 Regular Grammars

So far, we already know:

- A language defined by a DFA is a regular language.
- Any DFA can be regarded as a special case of an NFA.
- Any NFA can be converted to an equivalent DFA; thus, a language defined by an NFA is a regular language.
- A regular expression can be converted to an equivalent NFA; thus, a language defined by a regular expression is a regular language.
- An NFA can (with some effort!) be converted to a regular expression.

**Remark 2.** *So DFAs, NFAs, and regular expressions are all "equivalent," in the sense that any language you define with one of these could be defined by the others as well.*

We also know that languages can be defined by grammars. Now we will begin to classify grammars; and the first kinds of grammars we will look at are the regular grammars. As you might expect, regular grammars will turn out to be equivalent to DFAs, NFAs, and regular expressions.

### 2.7.1 Right-Linear Grammars

Recall that a grammar  $G$  is a quadruple  $G = (V, T, S, P)$ , where

- $V$  is a finite set of symbols, called **variables**.
- $T$  is a finite set of symbols, called **terminal symbols**.
- $S \in V$  is a special symbol called **start symbol**.
- $P$  is finite set of productions.

We always assume that the productions are of the form

$$x \rightarrow y,$$

where  $x$  is an element of  $(V \cup T)^+$  and  $y$  is in  $(V \cup T)^*$ .

The above is true for all grammars. We will distinguish among different kinds of grammars based on the form of the productions. If the productions of a grammar all follow a certain pattern, we have one kind of grammar. If the productions all fit a different pattern, we have a different kind of grammar.

Productions have the form  $(V \cup T)^+ \rightarrow (V \cup T)^*$ . Different types of grammars can be defined by putting additional restrictions on the left-hand side of productions, the right-hand side of productions, or both.

In general, productions have the form:

$$(V \cup T)^+ \rightarrow (V \cup T)^*$$

In a right-linear grammar, all productions have one of the two forms:

$$V \rightarrow T^*V$$

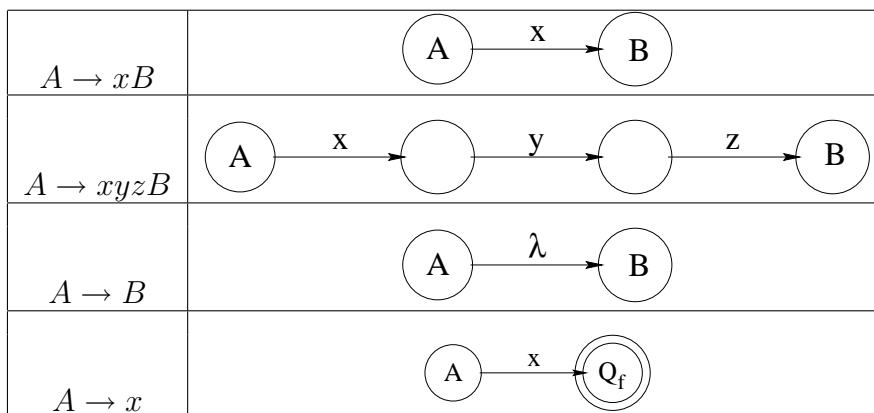
or

$$V \rightarrow T^*$$

That is, the left-hand side must consist of a single variable, and the right-hand side consists of any number of terminals (members of  $\Sigma$ ) optionally followed by a single variable. (The "right" in "right-linear grammar" refers to the fact that, following the arrow, a variable can occur only as the *rightmost* symbol of the production.)

### 2.7.2 Right-Linear Grammars and NFAs

There is a simple connection between right-linear grammars and NFAs, as suggested by the following diagrams:



**Figure 2.16.** The NFAs for various productions.

As an example of the correspondence between an NFA and a right-linear grammar, the following automaton and grammar (Figure 2.3) both recognize the set of strings consisting of an even number of 0's and an even number of 1's.

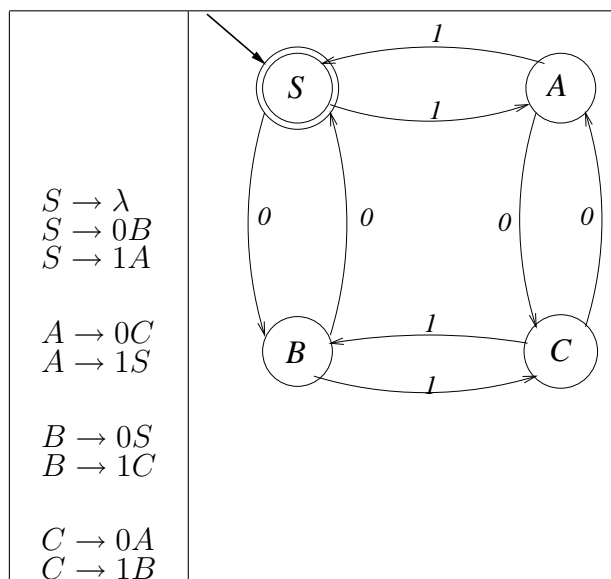
### 2.7.3 Left-Linear Grammars

In a left-linear grammar, all productions have one of the two forms:

$$V \rightarrow VT^*$$

or

$$V \rightarrow T^*$$



**Table 2.3.** The grammar that accepts the same language as the NFA.

That is, the left-hand side must consist of a single variable, and the right-hand side consists of an optional single variable followed by any number of terminals. This is just like a right-linear grammar except that, following the arrow, a variable can occur only on the left of the terminals, rather than only on the right.

We won't pay much attention to left-linear grammars, because they turn out to be equivalent to right-linear grammars. Given a left-linear grammar for language  $L$ , we can construct a right-linear grammar for the same language, as follows:

### 2.7.4 Regular Grammars

**Definition 5.** A regular grammar is either a right-linear grammar or a left-linear grammar.

Notice that, to be a right-linear grammar, every production of the grammar must have one of the two forms  $V \rightarrow T^*V$  or  $V \rightarrow T^*$ . To be a left-linear grammar, every production of the grammar must have one of the two forms  $V \rightarrow VT^*$  or  $V \rightarrow T^*$ . You do not get to mix the two. For example, consider a grammar with the following productions:

$$\begin{aligned} S &\rightarrow Xa \\ X &\rightarrow bS \\ X &\rightarrow c \end{aligned}$$

This grammar is neither right-linear nor left-linear, hence it is not a regular grammar. We have no reason to suppose that the language it generates is a regular language (one that is generated by a DFA). In fact, the grammar generates a language whose strings are of the form  $b^nca^n$ ,  $n > 0$ . This language cannot be recognized by a DFA, which will be proved later.

On the other hand, if a language is not right linear or left linear, we cannot say that the language generated by this grammar is not regular. For example, the following grammar is not

Step	Method
Construct a right-linear grammar for the (different) language $L^R$	Replace each production $A \rightarrow x$ of $G$ with a production $A \rightarrow x$ , and replace each production $A \rightarrow Bx$ with a production $A \rightarrow xB$ .
Construct an NFA for $L$ from the right-linear grammar. This NFA should have just one final state.	We talked about deriving an NFA from a right-linear grammar earlier. If the NFA has more than one final state, we can make those states nonfinal, add a new final state, and put transitions from each previously final state to the new final state.
Reverse the NFA for $L^R$ to obtain an NFA for $L$	
Construct a right-linear grammar for $L$ from the NFA for $L$	This is the technique we just talked about earlier.

regular grammar.

$$\begin{aligned} S &\rightarrow Xa \\ X &\rightarrow aS \\ X &\rightarrow a \end{aligned}$$

However, the language generated by this grammar is  $a^{2n+1}$ , which is a regular language, because we can easily construct a DFA to accept this language.

## 2.8 Three Ways of Defining a Language

This section presents an example solved three different ways: grammar, automata, and regular expression. No new information is presented. The problem is defined as follows:

**Problem 6.** Define a language containing all strings over  $\Sigma = \{a, b, c\}$  where no symbol ever follows itself; that is, no string contains any of the substrings  $aa$ ,  $bb$ , or  $cc$ .

### 2.8.1 Definition by Grammar

Define the grammar  $G = (V, T, S, P)$ , where

- $V = \{S, \dots \text{some other variables} \dots\}$ .
- $T = \Sigma = \{a, b, c\}$ .

- The start symbol is  $S$ .
- $P$  is given as follows (These should be pretty obvious except for the set  $V$ , which we generally make up as we construct  $P$ .)

Since the empty string belongs to the language, we need the production

$$S \rightarrow \lambda$$

Some strings belonging to the language begin with the symbol  $a$ . The character  $a$  can be followed by any other string in the language, so long as this other string does not begin with  $a$ . So we make up a variable, call it  $N_a$ , to produce these other strings, and add the production

$$S \rightarrow aN_a$$

By similar logic, we add the variables  $N_b$  and  $N_c$  and the productions

$$\begin{aligned} S &\rightarrow bN_b \\ S &\rightarrow cN_c \end{aligned}$$

Now,  $N_a$  is either the empty string, or some string that begins with  $b$ , or some string that begins with  $c$ . If it begins with  $b$ , then it must be followed by a (possibly empty) string that does not begin with  $b$ —and we already have a variable for that case,  $N_b$ . Similarly, if  $N_a$  is some string beginning with  $c$ , the  $c$  must be followed by  $N_c$ . This gives the productions

$$\begin{aligned} N_a &\rightarrow \lambda \\ N_a &\rightarrow bN_b \\ N_a &\rightarrow cN_c \end{aligned}$$

Similar logic gives the following productions for  $N_b$  and  $N_c$ :

$$\begin{aligned} N_b &\rightarrow \lambda \\ N_b &\rightarrow aN_a \\ N_b &\rightarrow cN_c \\ \\ N_c &\rightarrow \lambda \\ N_c &\rightarrow bN_b \\ N_c &\rightarrow aN_a \end{aligned}$$

We add  $N_a$ ,  $N_b$ , and  $N_c$  to set  $V$ , and we're done.

## 2.8.2 Definition by NFA

Defining the language by an NFA follows almost exactly the same logic as defining the language by a grammar. Whenever an input symbol is read, go to a state that will accept any symbol other than the one read. To emphasize the similarity with the preceding grammar, we will name our states to correspond to variables in the grammar. See Figure 2.17 for illustration of the NFA.

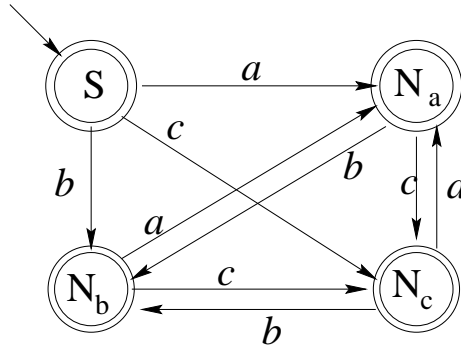


Figure 2.17. The NFA to accept the language with no repeating characters.

## 2.8.3 Definition by regular expression

As usual, it is more difficult to find a suitable regular expression to define this language, and the regular expression we do find bears little resemblance to the grammar or to the NFA.

The key insight is that strings of the language can be viewed as consisting of zero or more repetitions of the symbol  $a$ , and between them must be strings of the form  $bcbcbc\dots$  or  $cbcbc\dots$ . So we can start with

$$XaYaYaYa\dots YaZ$$

where we have to find suitable expressions for  $X, Y$ , and  $Z$ . Here  $X, Y$ , and  $Z$  do not contain character  $a$ . Moreover,  $Y$  cannot be empty. But first, let's get the above expression in a proper form, by getting rid of the " $\dots$ ". This gives

$$Xa(Ya)^*Z$$

and, since we might not have any  $a$ 's at all,

$$(Xa(Ya)^*Z) + X$$

Now  $X$  can be empty, a single  $b$ , a single  $c$ , or can consist of an alternating sequence of  $b$ 's and  $c$ 's. This gives

$$X = (\lambda + b)(cb)^*(\lambda + c)$$

Now, what about  $Z$ ? As it happens, there isn't any difference between what we need for  $Z$  and what we need for  $X$ , so we can also use the above expression for  $Z$ .

Finally, what about  $Y$ ? This is just like the others, except that  $Y$  cannot be empty. Luckily, it's easy to adjust the above expression for  $X$  and  $Z$  so that it can't be empty:

$$Y = c(bc)^* + b(cb)^*$$

Substituting  $X, Y$ , and  $Z$  into  $Xa(Ya)^*Z$ , we get

$$(\lambda + b)(cb)^*(\lambda + c)a((c(bc)^* + b(cb)^*)a)^*(\lambda + b)(cb)^*(\lambda + c)$$

## 2.9 Properties of Regular Languages

**Definition 6.** A set is closed under an operation if, whenever the operation is applied to members of the set, the result is also a member of the set.

For example, the set of integers is closed under addition, because  $x + y$  is an integer whenever  $x$  and  $y$  are integers. However, integers are not closed under division: if  $x$  and  $y$  are integers,  $x/y$  may or may not be an integer.

We have defined several operations on languages:

- $L_1 \cup L_2$ : Strings in either  $L_1$  or  $L_2$
- $L_1 \cap L_2$ : Strings in either  $L_1$  or  $L_2$
- $L_1L_2$ : Strings composed of one string from  $L_1$  followed by one string from  $L_2$
- $\overline{L_1} = \Sigma^* - L_1$  All strings (over the same alphabet) not in  $L_1$
- $L_1^*$ : Zero or more strings from  $L_1$  concatenated together
- $L_1 - L_2$ : Strings in  $L_1$  that are not in  $L_2$
- $L_1^R$ : Reversed of the strings in  $L_1$

We will show that the set of regular languages is closed under each of these operations. We will also define the operations of "homomorphism" and "right quotient" and show that the set of regular languages is also closed under these operations

### 2.9.1 Closure I: Union, Concatenation, Negation, Kleene Star, Reverse

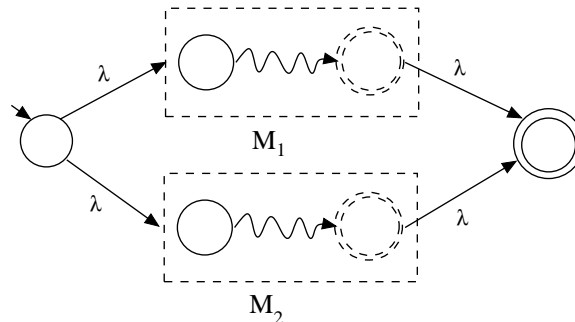
#### General Approach

- Build automata (DFAs or NFAs) for each of the languages involved.
- Show how to combine the automata to create a new automaton that recognizes the desired language.
- Since the language is represented by an NFA or DFA, conclude that the language is regular.

### Union of $L_1$ and $L_2$

Assume that automata  $M_1$  and  $M_2$  are the NFAs that accept the languages  $L_1$  and  $L_2$  respectively. We construct an NFA  $M$  to accept  $L_1 \cup L_2$  as follows. The initial and final states of the original NFAs  $M_1$  and  $M_2$  (boxed) stop being initial and final states. We add new initial and final states for the automaton  $M$  accepting the language  $L_1 \cup L_2$ . (We could construct a NFA that accepting the language  $L_1 \cup L_2$  with fewer states and fewer transitions here, but we aren't trying for the best construction; we're just trying to show that a construction is possible.) We add the following four arcs:

- an arc from the initial state of  $M$  to the initial state of  $M_1$  with label  $\lambda$ ;
- an arc from the initial state of  $M$  to the initial state of  $M_2$  with label  $\lambda$ ;
- an arc from the final state of  $M_1$  to the final state of  $M$  with label  $\lambda$ ;
- an arc from the final state of  $M_2$  to the final state of  $M$  with label  $\lambda$ .



**Figure 2.18.** The NFA for  $L_1 + L_2$ .

### Concatenation of $L_1$ and $L_2$

We construct an NFA  $M$  to accept  $L_1L_2$  as follows. The initial and final states of the original NFAs  $M_1$  and  $M_2$  (boxed) stop being initial and final states. We add new initial and final states for the automaton  $M$  accepting the language  $L_1L_2$ . We add the following arcs:

- an arc from the initial state of  $M$  to the initial state of  $M_1$  with label  $\lambda$ ;
- an arc from the final state of  $M_1$  to the initial state of  $M_2$  with label  $\lambda$ ;
- an arc from the final state of  $M_2$  to the final state of  $M$  with label  $\lambda$ .

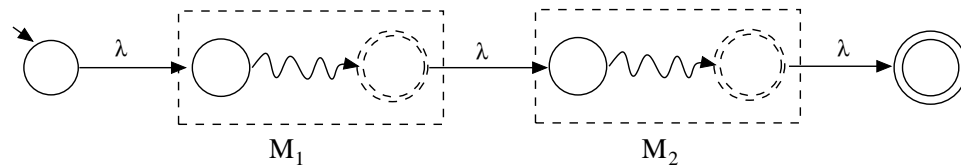


Figure 2.19. The NFA for  $L_1L_2$ .

### Negation of $L_1$

Assume that  $M_1$  is the complete DFA to accept the language  $L_1$ . Here a DFA is complete if for every state  $q$  and every possible input character  $x$ ,  $\delta(q, x)$  is defined. A simple trick to transform a non-complete DFA to complete DFA is to add a dummy state  $Q_d$ ; if for some state  $q$  and some possible input character  $x$ ,  $\delta(q, x)$  is not defined, then add  $\delta(q, x) = Q_d$ .

Start with a (complete) DFA accepting the language  $L_1$ . Make every final state non-final and every non-final state final.

### Kleene Star of $L_1$

We construct an NFA  $M$  to accept  $L_1^*$  as follows. The initial and final states of the original NFAs  $M_1$  (boxed) stop being initial and final states. We add new initial and final states for the automaton  $M$  accepting the language  $L_1^*$ . We add the following four arcs:

- an arc from the initial state of  $M$  to the initial state of  $M_1$  with label  $\lambda$ ;
- an arc from the initial state of  $M$  to the final state of  $M$  with label  $\lambda$ ;
- an arc from the final state of  $M_1$  to the final state of  $M$  with label  $\lambda$ ;
- an arc from the final state of  $M$  to the initial state of  $M$  with label  $\lambda$ .

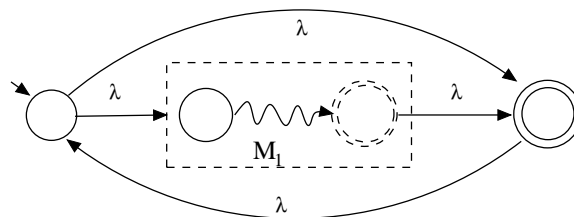


Figure 2.20. The NFA for  $L_1^*$ .

### Reverse of $L_1$

Start with an automaton with just one final state. Make the initial state final and the final state initial. Reverse the direction of every arc.

## 2.9.2 Closure II: Intersection and Set Difference

Just as with the other operations, you prove that regular languages are closed under intersection and set difference by starting with automata for the initial languages, and constructing a new automaton that represents the operation applied to the initial languages. However, the constructions are somewhat trickier.

In these constructions you form a completely new machine, whose states are each labeled with an ordered pair of state names: the first element of each pair is a state from the automaton  $M_1$  accepting  $L_1$ , and the second element of each pair is a state from the automaton  $M_2$  accepting  $L_2$ . (Usually you won't need a state for every such pair, just some of them.)

1. Begin by creating a start state whose label is (start state of  $M_1$ , start state of  $M_2$ ).
2. Repeat the following until no new arcs can be added:
  - (a) Find a state  $(A, B)$  that lacks a transition for some  $x$  in  $\Sigma$ .
  - (b) Add a transition on  $x$  from state  $(A, B)$  to state  $(\delta_1(A, x), \delta_2(B, x))$ . (If this state doesn't exist, create it.). Here  $\delta_1$  is the transition function for  $M_1$ ;  $\delta_2$  is the transition function for  $M_2$ .
  - (c) The same construction is used for both intersection and set difference. The distinction is in how the final states are selected.
    - *Intersection*: Mark a state  $(A, B)$  as final if both  $A$  is a final state in  $M_1$ , and  $B$  is a final state in  $M_2$ .
    - *Set difference*: Mark a state  $(A, B)$  as final if  $A$  is a final state in  $M_1$ , and  $B$  is not a final state in  $M_2$ .
    - *Union*: Mark a state  $(A, B)$  as final if both  $A$  is a final state in  $M_1$ , or  $B$  is a final state in  $M_2$ .

## 2.9.3 Closure III: Homomorphism

Note: "Homomorphism" is a term borrowed from group theory. What we refer to as a "homomorphism" is really a special case. Suppose  $\Sigma$  and  $\Gamma$  are alphabets (not necessarily distinct). Then a homomorphism  $h$  is a function from  $\Sigma$  to  $\Gamma^*$ .

If  $w$  is a string in  $\Sigma^*$ , then we define  $h(w)$  to be the string obtained by replacing each symbol  $x$  by the corresponding string  $h(x) \in \Gamma^*$ .

**Definition 7.** If  $L$  is a language on  $\Sigma^*$ , then its homomorphic image  $h(L)$  is a language on  $\Gamma^*$ . Formally,

$$h(L) = \{h(w) : w \in L\}$$

**Theorem 2.9.1.** If  $L$  is a regular language on  $\Sigma^*$ , then its homomorphic image  $h(L)$  is a regular language on  $\Gamma^*$ . That is, if you replaced every string  $w$  in  $L$  with  $h(w)$ , the resultant set of strings would be a regular language on  $\Gamma^*$ .

PROOF. We prove this based on construction. We first construct a DFA representing  $L$ . This is possible because  $L$  is regular.

For each arc in the DFA, replace its label  $x$  with  $h(x)$ . If an arc is labeled with a string  $w$  of length greater than one, replace the arc with a series of arcs and (new) states, so that each arc is labeled with a single element of  $\Gamma$ . The result is an NFA that recognizes exactly the language  $h(L)$ .

Since the language  $h(L)$  can be specified by an NFA, the language is regular.  $\square$

## 2.9.4 Closure IV: Right Quotient

**Definition 8.** Let  $L_1$  and  $L_2$  be languages on the same alphabet. The right quotient of  $L_1$  with  $L_2$  is

$$L_1/L_2 = \{w : wx \in L_1 \text{ and } x \in L_2\}$$

That is, the strings in  $L_1/L_2$  are strings from  $L_1$  "with their tails (from  $L_2$ ) cut off." If some string of  $L_1$  can be broken into two parts,  $w$  and  $x$ , where  $x$  is in language  $L_2$ , then  $w$  is in language  $L_1/L_2$ .

**Theorem 2.9.2.** If  $L_1$  and  $L_2$  are both regular languages, then  $L_1/L_2$  is a regular language.

PROOF. Again, the proof is by construction. We start with a DFA  $M(L_1)$  for  $L_1$ . The DFA we construct is exactly like the DFA for  $L_1$ , except that (in general) different states will be marked as final.

For each state  $Q_i$  in  $M(L_1)$ , determine if it should be final in  $M(L_1/L_2)$  as follows:

Starting in state  $Q_i$  as if it were the initial state, determine if any of the strings in language  $L_2$  are accepted by  $M(L_1)$ . If there are any, then state  $Q_i$  should be marked as final in  $M(L_1/L_2)$ . (Why?)

That's the basic algorithm. However, one of the steps in it is problematical: since language  $L_2$  may have an infinite number of strings, how do we determine whether some unknown string in the language is accepted by  $M(L_1)$  when starting at  $Q_i$ ? We cannot try all the strings, because we insist on a finite algorithm. The trick is to construct a new DFA that recognizes the intersection of two languages: (1)  $L_2$ , and (2) the language that would be accepted by DFA  $M(L_1)$  if  $Q_i$  were its initial state. We already know we can build this machine. Now, if this machine recognizes any string whatever (we can check this easily), then the two machines have a nonempty intersection, and  $Q_i$  should be a final state.

We have to go through this same process for every state  $Q_i$  in  $M(L_1)$ , so the algorithm is too lengthy to step through by hand. However, it is enough for our purposes that the algorithm exists.

Finally, since we can construct a DFA that recognizes  $L_1/L_2$ , this language is therefore regular, and we have shown that the regular languages are closed under right quotient.  $\square$

## 2.9.5 Standard Representations

A regular language is given in a standard representation if it is specified by one of:

- A finite automaton (DFA or NFA).

- A regular expression.
- A regular grammar.

The importance of these particular representations is simply that they are precise and unambiguous; thus, we can prove things about languages when they are expressed in a standard representation.

### Membership.

**Lemma 2.9.3.** *If  $L$  is a language on alphabet  $\Sigma$ ,  $L$  is in a standard representation, and  $w \in \Sigma^*$ , then there is an algorithm for determining whether  $w \in L$ .*

PROOF. Build the automation and use it to test  $w$ . □

### Finiteness.

**Lemma 2.9.4.** *If language  $L$  is specified by a standard representation, there is an algorithm to determine whether the set  $L$  is empty, finite, or infinite.*

PROOF. Build the automaton.

If there is no path from the initial state to a final state, then the language is empty (and finite). This test can be done by some simple graph algorithms.

If there is a path from the initial state to some final state and that path contains a cycle, then the language is infinite.

If no path from the initial state to a final state contains a cycle, then the language is finite. □

### Equivalence.

**Lemma 2.9.5.** *If languages  $L_1$  and  $L_2$  are each given in a standard representation, then there is an algorithm to determine whether the languages are identical.*

PROOF. Construct the automata to accept languages  $L_1$  and  $L_2$ . Then build the automaton to accept the language  $(L_1 - L_2) \cup (L_2 - L_1)$ . If this language is empty, then  $L_1 = L_2$ . □

## 2.10 The Pumping Lemma

There are pumping lemmas for different kinds of grammars. This lesson concerns the pumping lemma for regular languages. In this section, we will cover the following topics:

- The Pigeonhole Principle
- The Pumping Lemma for Regular Languages
- Applying the Pumping Lemma
- Pumping Lemma Examples

### 2.10.1 The Pigeonhole Principle

pigeonhole:

- a hole or small recess for pigeons to nest
- a small open compartment (as in a desk or cabinet) for keeping letters or documents
- a neat category which usually fails to reflect actual complexities.

**Remark 3.** *pigeonhole principle: if  $n$  objects are put into  $m$  containers, where  $n > m$ , then at least one container must hold more than one object.*

The pigeonhole principle can be used to prove that certain infinite languages are not regular. (Remember, any finite language is regular.)

As we have informally observed, DFAs "can't count." This can be shown formally by using the pigeonhole principle. As an example, we show that

$$L = \{a^n b^n : n > 0\}$$

is not regular.

**Lemma 2.10.1.** *The following language*

$$L = \{a^n b^n : n > 0\}$$

*is not regular.*

**PROOF.** The proof is by contradiction. Suppose that  $L$  is regular. Then there is a DFA  $M(L)$  that accepts  $L$ . There are an infinite number of values of  $n$  but  $M(L)$  has only a finite number of states. By the pigeonhole principle, there must be distinct values of  $i$  and  $j$  such that  $a^i$  and  $a^j$  end in the same state. From this state,

- $b^i$  must end in a final state, because  $a^i b^i$  is in  $L$ ; and
- $b^i$  must end in a non-final state, because  $a^j b^i$  ( $j \neq i$ ) is not in  $L$ .

Since the state reached cannot be both final and non-final, we have a contradiction. Thus our assumption, that  $L$  is regular, must be incorrect.  $\square$

## 2.10.2 The Pumping Lemma

Informally, what the pumping lemma says is:

*If an infinite language is regular, it can be defined by a DFA. The DFA has some finite number of states (say,  $n$ ). Since the language is infinite, some strings of the language must have length  $> n$ . For a string of length  $> n$  accepted by the DFA, the walk through the DFA must contain a cycle. Repeating the cycle an arbitrary number of times must yield another string accepted by the DFA.*

The pumping lemma for regular languages is another way of proving that a given (infinite) language is not regular.

**Remark 4.** *The pumping lemma cannot be used to prove that a given language is regular.*

The proof that a language is not regular is always by contradiction. A brief outline of the technique is as follows:

- Assume the language  $L$  is regular.
- By the pigeonhole principle, any sufficiently long string  $w$  (with length larger than  $n$ ) in  $L$  must repeat some state in the DFA. Thus, the walk (when processing the input  $w$ ) contains a cycle.
- Show that repeating the cycle some number of times ("pumping" the cycle) yields a string that is not in  $L$ .
- Conclude that  $L$  is not regular.

Why this is hard:

- We don't know the DFA (if we did, the language would be regular!). Thus, we have to do the proof for an arbitrary DFA that accepts  $L$ .
- Since we don't know the DFA, we certainly don't know the cycle.

Why we can sometimes pull it off:

- We get to choose the string (but it must be in  $L$ ).
- We get to choose the number of times to "pump."

## 2.10.3 Applying the Pumping Lemma

Here's a more formal definition of the pumping lemma:

**Lemma 2.10.2.** *If  $L$  is an infinite regular language, then there exists some positive integer  $m$  such that any string  $w \in L$  whose length is  $m$  or greater can be decomposed into three parts,  $xyz$ , where*

- $\|xy\|$  is less than or equal to  $m$ ;
- $\|y\| > 0$ , in other words,  $y$  is not empty;

- $w_i = xy^i z$  is also in  $L$  for all  $i = 0, 1, 2, 3, \dots$

Here's what it all means:

$L$  is an infinite regular language, then there exists some finite DFA accepting  $L$ . Let  $m$  be a (finite) number chosen so that strings of length  $m$  or greater must contain a cycle. Hence,  $m$  must be equal to or greater than the number of states in the DFA. Remember that we don't know the DFA, so we can't actually choose  $m$ ; we just know that such an  $m$  must exist.

Since string  $w$  has length greater than or equal to  $m$ , we can break it into two parts,  $xy$  and  $z$ , such that  $xy$  must contain a cycle. We don't know the DFA, so we don't know exactly where to make this break, but we know that  $xy$  can be less than or equal to  $m$  because any string of length  $m$  or greater must contain a cycle.

We let  $x$  be the part before the cycle,  $y$  be the cycle, and  $z$  the part after the cycle. (It is possible that  $x$  and  $z$  contain cycles, but we don't care about that.) Again, we don't know exactly where to make this break.

Since the walk relating to  $y$  is the cycle we are interested in, we must have  $\|y\| > 0$ , otherwise it isn't a cycle.

By repeating  $y$  an arbitrary number of times,  $xy^*z$ , we must get other strings in  $L$ .

If, despite all the above uncertainties, we can show that the DFA has to accept some string that we know is not in the language, then we can conclude that the language is not regular.

To use this lemma, we need to show:

1. For any choice of positive integer  $m$ ,
2. for some  $w \in L$  that we get to choose (and we will choose the string  $w$  with length at least  $m$ ),
3. for any way of decomposing  $w$  into  $xyz$ , so long as  $\|xy\| \leq m$  and  $\|y\| > 0$ ,
4. we can choose an integer  $i \geq 0$  such that  $xy^i z$  is not in  $L$ .

We can view this as a game wherein our opponent makes moves 1 and 3 (choosing  $m$  and choosing the decomposition  $xyz$ ) and we make moves 2 and 4 (choosing  $w$  and choosing  $i$ ). Our goal is to show that we can always beat our opponent. If we can show this, we have proved that  $L$  is not regular.

### 2.10.4 Pumping Lemma Examples

**Example 13.** Prove that  $L = \{a^n b^n : n > 0\}$  is not regular.

**SOLUTION.** We prove this by contradiction. Assume that language  $L$  is a regular language. Then it must satisfy the pumping lemma.

For any positive integer  $m$ , choose a string  $w = a^n b^n$  where  $n > m$ , so that any prefix of length  $m$  consists only of  $a$ 's. We don't know the decomposition of  $w$  into  $xyz$ , but since  $\|xy\| \leq m$ , then  $xy$  must consist entirely of  $a$ 's. Moreover,  $y$  cannot be empty. For any decomposition of  $w = xyz$  such that  $\|xy\| \leq m$  and  $\|y\| > 0$ , we then choose  $i = 0$ . This has the effect of dropping  $y$  (which is composed of  $a$ 's) out of the string, without affecting the number of  $b$ 's. The resultant string

$w_0 = xz$  has fewer  $a$ 's than  $b$ 's, hence does not belong to  $L$ . In other words,  $L$  does not satisfy the pumping lemma. Therefore  $L$  is not regular.  $\square$

**Example 14.** Prove that  $L = \{a^n b^k : n > k \text{ and } n > 0\}$  is not regular.

SOLUTION. We prove this by contradiction. Assume that language  $L$  is a regular language. Then it must satisfy the pumping lemma.

For any positive integer  $m$ , choose a string  $w = a^{n+1}b^n$  where  $n > m$ , so that any prefix of length  $m$  consists only of  $a$ 's. We don't know the decomposition of  $w$  into  $xyz$ , but since  $\|xy\| \leq m$ , then  $xy$  must consist entirely of  $a$ 's. Moreover,  $y$  cannot be empty. For any decomposition of  $w = xyz$  such that  $\|xy\| \leq m$  and  $\|y\| > 0$ , we then choose  $i = 0$ . This has the effect of dropping  $y$  (which is composed of  $a$ 's) out of the string, without affecting the number of  $b$ 's. The resultant string  $w_0 = xz$  has at most the same number of or fewer  $a$ 's than  $b$ 's, hence does not belong to  $L$ . In other words,  $L$  does not satisfy the pumping lemma. Therefore  $L$  is not regular.  $\square$

**Example 15.** Prove that  $L = \{a^n : n \text{ is a prime number}\}$  is not regular.

SOLUTION. We prove this by contradiction. Assume that language  $L$  is a regular language. Then it must satisfy the pumping lemma.

For any positive integer  $m$ , choose a string  $w = a^n$  where  $n$  is a prime number and  $\|xyz\| = n > m + 1$ . (This can always be done because there is infinite number of prime numbers.) Any prefix of  $w$  consists entirely of  $a$ 's.

We don't know the decomposition of  $w$  into  $xyz$ , but since  $\|xy\| \leq m$ , it follows that  $\|z\| > 1$ . As usual,  $\|zy\| > 0$ . Since  $\|z\| > 1$ , then  $\|xz\| > 1$ . Choose  $i = \|xz\|$ . Then  $\|xy^i z\| = \|xz\| + \|y\| \cdot \|xz\| = (1 + \|y\|) \|xz\|$ . Since  $1 + \|y\|$  and  $\|xz\|$  are each greater than 1, the product must be a composite number. Thus  $xy^i z$  is a string composed of a composite number of  $a$ 's. Hence  $xy^i z$  does not belong to  $L$ . In other words,  $L$  does not satisfy the pumping lemma. Therefore  $L$  is not regular.  $\square$