Chapter 2
Regular Machines

In order to answer the question that what a computer can or cannot do, we have to define precisely the notion of a computer, i.e., a device that stores information and provides answers to questions based on the information it keeps.

We are not so much interested in the engineering aspects such as the clock speed of the CPU, the actual size of the memory, etc., we are more interested in the ability, particularly, the property of the memory, finite or infinite, as well as the I/O relationship. We will present a couple of mathematical models to capture the basic characteristics of a computer in terms of these factors.

We begin with the simplest model, i.e., the finite state machine or finite automaton, for those computers with an extremely limited memory.
An example

Some computer needs very little information. For example, the controller for an automatic door, such as the one in the now closed Tilton Shaws.

An automatic door always swings to the right.

There are two pads, a *front one* to detect the person who is approaching the door, and a *rear pad* to tell the controller if the person has walked all the way through and also tell the controller that there is no one behind the door when the door is to open.

---

![Diagram of a door with a front pad and a rear pad]
What should the controller do?

The controller can be in one of two states, “OPEN” or “CLOSED”, representing the two states of the door.

There are four possible inputs for the controller: “FRONT”, ”REAR”, “BOTH” and “NEITHER”, as there might be just one person standing either in the front, or in the rear, or there are people standing on both pads, or no one is standing in either pad.

The controller thus moves between its two states according to the input it just received, according to the common sense.
For example, when the door is closed, it should not be open unless there is someone standing in the front, while nobody is standing in the rear. On the other hand, if it is open, it should remain open until it detects that no one is standing in either place.

As the controller usually works in a continuous way, we usually represent the controller as the following *state diagram*.

![State Diagram](image)

Thus, if it starts in CLOSED, and receives the following input series: FRONT, REAR, NEITHER, FRONT, BOTH, NEITHER, ..., it will then go through the following sequence of states: CLOSED, OPEN, OPEN, CLOSED, OPEN, OPEN, OPEN, CLOSED, ....
A controller as a computer

The door controller is a computer, in the sense that based on the stored information and input, it makes a logic decision.

However, it is a very simple one, as it needs only one bit of memory to keep track of its “state”. Other such controllers need more memory to keep information, e.g., elevator controller, dish washer, lexicon, etc. Nevertheless, all these controllers can be represented and designed by following the finite automata theory.

In the rest of this chapter, we will study this simple, but useful and important, class of computers, called the regular machines.
A couple of terms

Below shows a *state diagram* of a finite automaton, $M_1$.

$M_1$ has three *states*: $q_1$, $q_2$ and $q_3$, among which $q_1$ is the *start state*, as indicated by the arrow pointing at it from nowhere; and $q_2$ is the *accept state*, as indicated by a double circle.

The collection of all the arrows going from one state to another is used to define the *transition* between the states.

**Homework:** Exercise 1.1.
FA in motion

When a finite automaton, such as $M_1$, receives an input string, e.g., 1101, it processes that string and produces an output, either accept or reject.

The processing begins with the start state. While reading each symbol in the input string, one by one, from left to right, the automaton moves to another state as specified by the transition.

After reading the last symbol, the automaton sends out the output of either “accept” or “reject”, according to if it is in an accept state or not after reading this last symbol.

Based on this final output, we will say the automaton either accepts or reject the input string, accordingly.
For example, if we give the input string 1101 to $M_1$, the processing proceeds as follows:

1. start in state $q_1$;

2. read 1, thus goes to $q_2$;

3. read 1, then goes to $q_2$;

4. read 0, then goes to $q_3$;

5. read 1, the last input symbol, then goes to $q_2$;

6. $M_1$ accepts the input string, as $q_2$ is an accept state.
What does $M_1$ do?

A finite automata only accepts a specific set of strings. This is why the lexicon, a finite automaton, for any programming language can recognize those, and only those, reserved words for that language and treat all the others as user-defined variables. We will learn how to design such an automaton later.

We will first do some analysis, i.e., given an automaton, we want to find out what it does.

If we try to play with $M_1$ a bit, we will easily find out that it won’t accept any string containing only 0 and it won’t accept anything ending with an odd number of 0’s either.

Actually, $M_1$ accepts anything containing at least a ‘1’ and the last 1 is followed by an even number of ‘0’s.
Formal definition

To further discuss various properties of finite automata, we need to have a precise definition.

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

1. \(Q\) is a finite set called the *states*,
2. \(\Sigma\) is a finite set called the *alphabet*,
3. \(\delta : Q \times \Sigma \rightarrow Q\) is the *transition function*,
4. \(q_0 \in Q\) is the *start state*, and
5. \(F \subseteq Q\) is the set of *accept states*.

Among other things, the formal definition requires that, given any state and input symbol, there be only one state to which the automaton can go (?)

**Homework:** Exercise 1.3.
An example

We can describe any concrete finite automaton by specifying the five parts in the formal definition. For example, $M_1$ can be described as follows:

$$M_1 = (Q, \Sigma, \delta, q_1, F),$$

where

$Q = \{q_1, q_2, q_3\}$,

$\Sigma = \{0, 1\}$,

$F = \{q_2\}$,

and the transition function is given as follows:

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1$</td>
<td>$q_1$</td>
<td>$q_2$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$q_3$</td>
<td>$q_2$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>$q_2$</td>
<td>$q_2$</td>
</tr>
</tbody>
</table>
The name of the game

One way to look at an finite automaton is that it corresponds to a unique collection of input strings, those that it accepts. More formally, we have the following:

Recall that, by a language, we mean a set of strings.

**Definition.** If $A$ is the set of input strings that a finite automaton $M$ accepts, we say that $A$ is the *language of machine* $M$ and write $L(M) = A$.

We also say that $M$ accepts or recognizes $A$.

For example, let $A = \{ w \mid w \text{ contains at least one ‘1’ and the last ‘1’ is followed by an even number of 0’s} \}$. Then, $L(M_1) = A$, or $M_1$ accepts $A$. 
Examples of FA’s

Below is the diagram of $M_2$.

![Diagram of $M_2$]

Obviously, $M_2 = \{\{q_1, q_2\}, \{0, 1\}, \delta_2, q_1, \{q_2\}\}$, where, the transition function can be specified as follows:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1$</td>
<td>$q_1$</td>
<td>$q_2$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$q_1$</td>
<td>$q_2$</td>
</tr>
</tbody>
</table>

**Question:** What is $L(M_2)$?

**Answer:** $L(M_2) = \{w | w \text{ ends in a 1}\}$.

Let $M_3 = \{\{q_1, q_2\}, \{0, 1\}, \delta_2, q_1, \{q_1\}\}$. We have that $L(M_3) = \{w | w = \epsilon \text{ or } w \text{ ends in a 0}\}$. 

13
Below is the diagram of another finite automaton $M_4$.

![Finite Automaton Diagram]

**Question:** What is $L(M_4)$?

Any string accepted by $M_4$ is accepted by either the left or the right. Thus, $L(M_4)$ is the union of the languages accepted by the left and that by the right.

**Answer:** The language accepted by $M_4$ is the collection of the strings that either both starting and ending with $a$ or both starting and ending with $b$, or the union of these two collections.
Below is the diagram of $M_5$.

We treat RESET as a single symbol.

**Question:** What does $M_5$ do?

**Answer:** $M_5$ keeps a running sum, modulo 3, of the input symbol it reads. Whenever, it receives a RESET, it resets the sum to 0. It accepts any input string of 0, 1 and 2, such that its sum is 0, modulo 3.
Homework assignment

1. Go to the simulator site as given in the course site, download the JFLAP simulator software, and then install it in your computer. We also have put it up in Memorial 312, or any cluster for this matter.

2. Do at least half and try as much as you can for the problems in Parts \{a, c, e, f, g\} of Exercise 1.4.

3. Use the program to check out your results.

4. When you hand in this batch of homework, you have to attach JFlap program files that can be run in the JFlap simulator.
FA as computers

From the previous examples, we informally see how an finite automaton sends out an output, when given some input, i.e., computes the output. Below is a formal definition.

**Definition:** Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton and let $w = w_1w_2 \cdots w_n$ be a string over $\Sigma$. Then $M$ accepts $w$ if a sequence of states $r_0, r_1, \ldots, r_n$ exists in $Q$ such that

1. $r_0 = q_0$,
2. $\delta(r_i, w_{i+1}) = r_{i+1}$ for $i \in [0, n - 1]$ and
3. $r_n \in F$.

Finally, $L(M) = \{w | M$ accepts $w\}$.

**Definition:** A language is *regular* if some finite automaton accepts it.
Design of FA

So far, we try to figure out what does a FA do, i.e., what is the language it accepts. It is already difficult, but it will be much more difficult to design an automaton with a given language which the FA is supposed to accept.

This task is quite similar to programming: design a program to solve a given problem. It is also similar to algorithm analysis: find out a minimum machine that will accept the language. (Cf. Exercise 1.12)

The major problem is how to memorize possibly long sequence of input symbols with very little memory. Obviously, the only thing we need do is to memorize the crucial information.
Examples of design

Problem: Assume the alphabet consists of 0 and 1. Design a finite automaton that accepts all strings with an odd number of 1’s.

Solution: The only thing we need to remember is the oddity of the total number of 1’s that we have read so far, but not the number itself.

Thus, we need to have two states, $q_{\text{even}}$ and $q_{\text{odd}}$. As initially, nothing, particularly no 1, has been read, we let $q_{\text{even}}$ be the start state, as 0 is usually regarded as an even number.

As we only want to accept odd number of 1’s, we let $q_{\text{odd}}$ be the only accept state.

Finally, the transition function is easy to specify, based on our task: For example, if we are in $q_{\text{even}}$ and see another 0, we stay in $q_{\text{even}}$; otherwise, switch over to $q_{\text{odd}}$. 
**Problem:** Assume the alphabet consists of 0 and 1. Design a finite automaton that accepts all strings that contains 001 as a substring.

**Solution:** We let $Q = \{q, q_0, q_{00}, q_{001}\}$, where $q$ is the start state, $q_s$ denotes the state in which we have read in the string $s$, thus $q_{001}$ denotes that we have seen 001, i.e., the accept state.

Starting with $q$, we will skip over all 1’s in an incoming string and get serious when we see a 0 by entering $q_0$. If the next symbol is a 1, we come back to $q$; otherwise, we move to $q_{00}$ and just need to wait for another 1.

Once we have got the final 1, we slide to $q_{001}$, then we could not care less what symbols will follow.

**Question:** What to do when we see another 0 in $q_{00}$? Should we go back to $q, q_0$ or $q_{00}$? Why?
Regular Operators

Design is an important task, which is often challenging. We now present a couple of operators on languages, called *regular operations*, which will allow us to design FA’s following a divide-and-conquer approach.

**Definitions:** Let $A$ and $B$ be languages. We define the regular operations *union*, *concatenation*, and *star* as follows:

\[
A \cup B = \{x|x \in A \lor x \in B\}.
\]
\[
A \circ B = \{xy|x \in A \land x \in B\}.
\]
\[
A^* = \{x_1x_2 \cdots x_k|k \geq 0 \text{ and each } x_i \in A\}.
\]

To form a new language, the *union* operation takes all the strings in either $A$ or $B$, the *concatenation* attaches any string in $A$ to another in $B$ in all possible ways, and the unary *star* operation combines any strings in $A$ together, thus includes the empty string $\epsilon$. 

21
Closure properties(I)

When we apply addition to two natural numbers, we get a natural number back. Thus, we say that $\mathcal{N}$, the collection of all natural numbers, is closed under addition. Similarly, $\mathcal{N}$ is also under multiplication. However, $\mathcal{N}$ is not closed under either subtraction or division.

**Question:** How about $\mathbb{Z}$? Is it closed under all the four operations?

In general, a collection of objects is closed under certain operations if applying that operation to members of that collection returns an object still in that collection.
Implication on design

We will show that the collection of all the regular languages is closed under all those three regular operations.

It means that given two regular languages, if we apply either the union, the concatenation, or the star operation to them, we will get a regular language back.

Such a property is very useful when designing regular machines, since we can now follow the divide-and-conquer approach: We first decompose the language at hand to a bunch of smaller ones in terms of the three operators.

If all of them are regular, i.e., we can construct machines for all of them, then the original language is also regular, since we can combine these smaller ones into one machine for the original one.

Sounds great, but how?
The union machine

Let’s check out the union operation first.

What we have to show is that if both $A_1$ and $A_2$ are regular languages, so is $A_1 \cup A_2$. The proof is an example of proof by construction: Given the two automata for $A_1$ and $A_2$, design another one to accept $A_1 \cup A_2$.

The basic idea is, for any given string, to check if it belongs to either $A_1$ or $A_2$.

As once a symbol is read, it is gone, we can’t try to check if the string is in $A_1$ first, and if it is not, then try to check if it is in $A_2$.

Hence, we have to check its membership simultaneously, with both $A_1$ and $A_2$. 

24
How?

The crucial information we have to keep for the new machine is the state that a machine would be in if it has read this many input symbols.

One way to do this is to let any state of the new machine contain two states, each for the two machines, and let the transition function of the new machine simulate the transition functions of the two machines, respectively.

We will begin the simulation simultaneously, thus, the start state of the new machine will be the pair, consisting of the two start states of the two machines.

Whenever an accept state of either machine is reached, so should the new machine. Thus, any state of the new machine containing an accept state of either machine should be an accept state.
An example

Let $L_1$ be \{w | w contains at least a 1, and even number of 0’s after the last 1.\}. $L_1$ is regular, since $L_1 = L(M_1)$, where

$$M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$$

where

- $Q_1 = \{q_1, q_2, q_3\}$,
- $\Sigma = \{0, 1\}$,
- $F_1 = \{q_2\}$,

and $\delta_1$, the transition function, is given as follows:

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1$</td>
<td>$q_1$</td>
<td>$q_2$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$q_3$</td>
<td>$q_2$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>$q_2$</td>
<td>$q_2$</td>
</tr>
</tbody>
</table>
Another language

Let \( L_2 \) be \( \{ w \mid w \text{ contains an odd number of } 1\text{'s.}\} \). \( L_2 \) is also regular, since \( L_2 = L(M_2) \), where

\[
M_2 = (Q_2, \Sigma, \delta_2, q_e, F_2), \text{ where}
\]

\[
Q_2 = \{ q_e, q_o \},
\]

\[
\Sigma = \{ 0, 1 \},
\]

\[
F_2 = \{ q_o \},
\]

and \( \delta_2 \), the transition function, is given as follows:

<table>
<thead>
<tr>
<th>( \delta )</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_e )</td>
<td>( q_e )</td>
<td>( q_o )</td>
</tr>
<tr>
<td>( q_o )</td>
<td>( q_o )</td>
<td>( q_e )</td>
</tr>
</tbody>
</table>
The machine for the union

We construct another machine, $M$, that accepts $L_1 \cup L_2$, as follows:

$$M = (Q, \Sigma, \delta, q_s, F),$$
where

$$Q = \{(q_1, q_e), (q_1, q_o), (q_2, q_e),$$
$$ (q_2, q_o), (q_3, q_e), (q_3, q_o)\},$$

$$\Sigma = \{0, 1\},$$

$$q_s = (q_1, q_e)$$

$$F = \{(q_1, q_o), (q_2, q_e), (q_2, q_o), (q_3, q_o)\},$$

and $\delta$ is given as follows:

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(q_1, q_e)$</td>
<td>$(q_1, q_e)$</td>
<td>$(q_2, q_o)$</td>
</tr>
<tr>
<td>$(q_1, q_o)$</td>
<td>$(q_1, q_o)$</td>
<td>$(q_2, q_e)$</td>
</tr>
<tr>
<td>$(q_2, q_e)$</td>
<td>$(q_3, q_e)$</td>
<td>$(q_2, q_o)$</td>
</tr>
<tr>
<td>$(q_2, q_o)$</td>
<td>$(q_3, q_o)$</td>
<td>$(q_2, q_e)$</td>
</tr>
<tr>
<td>$(q_3, q_e)$</td>
<td>$(q_3, q_e)$</td>
<td>$(q_2, q_o)$</td>
</tr>
<tr>
<td>$(q_3, q_o)$</td>
<td>$(q_2, q_o)$</td>
<td>$(q_2, q_e)$</td>
</tr>
</tbody>
</table>
So what?

Now, given an input string, e.g., 1011, $M$ will go through the following transition to accept it.

$$(q_1, q_e) \xrightarrow{1} (q_2, q_o) \xrightarrow{0} (q_3, q_o) \xrightarrow{1} (q_2, q_e) \xrightarrow{1} (q_2, q_o).$$

When it reads in all the input symbols, $M$ terminates at $(q_2, q_o)$. Since the latter is one of the accept states of $M$, by definition, this input is accepted by $M$.

On the other hand, for the input 1010, $M$ will go through the following:

$$(q_1, q_e) \xrightarrow{1} (q_2, q_o) \xrightarrow{0} (q_3, q_o) \xrightarrow{1} (q_2, q_e) \xrightarrow{0} (q_3, q_e).$$

Since $(q_3, q_e) \notin F$, $1010 \notin L(M)$. Notice that $1010 \notin L_1 \cup L_2$, either.

In general, $L(M) = L(M_1) \cup L(M_2)$. 
Theorem 1.12: The class of regular languages is closed under the union operation.

Proof: Let $M_i = (Q_i, \Sigma, \delta_i, q_i, F_i)$, accepts $A_i, i = 1, 2$. We now construct $M = (Q, \Sigma, \delta, q_0, F)$, as follows:

1. $Q = \{(r_1, r_2) | r_i \in Q_i, i = 1, 2\}$.
2. $\delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a))$.
3. $q_0 = \{q_1, q_2\}$.
4. $F = \{(r_1, r_2) | r_1 \in F_1 \text{ or } r_2 \in F_2\}$

We have to show that for any $w \in \Sigma^*$, $w \in L(M)$ iff $w \in L(M_1) \cup L(M_2)$. 
\[ L(M) \subseteq L(M_1) \cup L(M_2) \]

Let \( w \in L(M) \). By definition, it means that for \((r_1^0, r_2^0), (r_1^1, r_2^1), \ldots, (r_1^n, r_2^n) \in Q \)

1. \((r_1^0, r_2^0) = q_0 = (q_1, q_2), \)

2. \( \delta((r_1^i, r_2^i), w_{i+1}) = (r_1^{i+1}, r_2^{i+1}), \ i \in [0, n - 1] \) and

3. \((r_1^n, r_2^n) \in F. \)

By construction of \( M \), for any \( i \), \( \delta((r_1^i, r_2^i), a) = (\delta_1(r_1^i, a), \delta_2(r_2^i, a)). \)

Hence, the above sequence is equivalent to the following two sequences: for \( r_1^0, r_1^1, \ldots, r_1^n \in Q_1 \)

1. \( r_1^0 = q_1, \)
2. \( \delta(r_1^i, w_{i+1}) = r_1^{i+1} \) for \( i \in [0, n - 1]; \)
and for $r_2^0, r_2^1, \ldots, r_2^n \in Q_2$

1. $r_2^0 = q_2$,
2. $\delta(r_2^i, w_{i+1}) = r_2^{i+1}$ for $i \in [0, n - 1]$.

By construction of $M$, $(r_1^n, r_2^n) \in F$ means either $r_1^n \in F_1$, or $r_2^n \in F_2$.

By definition, we have $w \in L(M_1)$ or $w \in L(M_2)$, namely, $w \in L(M_1) \cup L(M_2)$.

We have proved that, for any $w \in L(M)$, $w \in L(M_1) \cup L(M_2)$, namely, $L(M) \subseteq L(M_1) \cup L(M_2)$. 

32
\[ L(M_1) \cup L(M_2) \subseteq L(M) \]

Assume that \( w \in L(M_1) \cup L(M_2) \), we want to show that \( w \in L(M) \).

Without loss of generality, we assume that \( w \in L(M_1) \). By definition, there exists a sequence of states \( r_0^1, r_1^1, \ldots, r_n^1 \) in \( Q_1 \) such that

1. \( r_0^1 = q_1 \),
2. \( \delta(r_i^1, w_{i+1}) = r_{i+1}^1 \) for \( i \in [0, n - 1] \) and
3. \( r_n^1 \in F_1 \);
We now construct another sequence \((r_0^0, r_0^2), (r_1^1, r_2^1), \ldots, (r_n^1, r_n^2)\) such that \(r_2^0 = q_2\), and for all \(i \in [0, n - 1]\), \(r_i^{i+1} = \delta(r_i^2, w_{i+1})\).

Based on this definition and the existing accepting sequence of \(w\) by machine \(M_1\), we have the following sequence,

1. \((r_0^0, r_0^2) = (q_1, q_2) = q_0\),
2. \(\delta((r_i^1, r_i^2), w_{i+1}) = (\delta(r_1^i, w_{i+1}), \delta(r_2^i, w_{i+1}))\) for \(i \in [0, n - 1]\).

Finally, since \(r_n^1 \in F_1\), by the construction of \(M\), \((r_1^n, r_2^n) \in F\).

By definition, \(w \in L(M)\), namely, \(L(M_1) \cup L(M_2) \cup L(M)\).

This concludes the proof of this theorem. \(\square\)

We can also prove similar results for both concatenation and the star operations. But, we have much better ways to do it.
Nondeterminism

So far in our discussion, every step of a computation in a finite automaton follows in a unique way from the preceding step: for any given state and any given input symbol, there exists a unique successor state. Thus, such an automaton is also called a deterministic finite automaton, i.e., DFA.

Below shows, $N_1$, a more general automaton, a non-deterministic finite automaton, i.e., NFA.

Compared to DFA’s, in an NFA, any state might be associated with 0, 1, or more arrows for each alphabetic symbol. Moreover, 0, 1, or more arrows labeled with $\epsilon$ may exit from a state.
How does an NFA compute?

Assume that an NFA runs on an input string and comes to a state with multiple ways to proceed. The NFA will split into multiple copies, and follow all the possible leads in parallel.

Each copy will continue in the same style. A copy will be eliminated if no lead exists for its current state and input. On the other hand, if any copy of the NFA gets into an accept state, when the whole input string is processed, the original NFA accepts the string.

Similarly, when an \( \epsilon \) is encountered, the NFA will split into multiple copies to follow each of the \( \epsilon \) labeled arrow and one stay at the current state. Then all the copies proceed in the same style.
How to look at NFA’s?

Non-determinism may be viewed as a kind of parallel computation where several processes can be running concurrently. When an NFA splits to follow multiple leads, that corresponds to a process splitting into several children processes, each proceeds separately. If any of them accepts the input, so does the original process.

Another way to look at it is to regard it as a tree of possibilities. The root of the tree corresponds to the start of the computation.

Every branching point in the tree corresponds to a point in the computation where there exist multiple choices. The machine accepts the input, if at least one path ends up in an accept state.
An example

Below shows a computation of applying $N_1$ on 010110.

Notice that when reading in the symbol, 1, $q_1$ splits into three more machines and proceed with $q_1$ and $q_2$.

For the $\epsilon$, the machine splits into two more copies, one proceeds with $q_3$, the other stays at $q_2$. 
NFA or DFA?

Below shows an NFA that accepts all the strings that contains a 1 in the third position from the end.

Below shows a DFA that accepts the same language.
A design tool

We will show that DFA’s and NFA’s are equivalent, in the sense that they accept exactly the same class of languages, the regular languages. NFA is usually easier to understand and make, but DFA is “firmer”, thus the choice of implementation.

Thus, we can start the design by constructing an NFA, then apply the construction as shown in the following equivalence result to convert it to an equivalent DFA.
A formal definition

The key difference in the behavior of NFA from a DFA is that given a state and an input symbol, also possibly $\epsilon$, an NFA might go to either empty or a couple of states, i.e., a subset of the state set. Thus, if we let $\Sigma_\epsilon$ denote $\Sigma \cup \{\epsilon\}$ and let $\mathcal{P}(Q)$ denote the power set of $Q$, we can define an NFA as follows.

**Definition:** A non-deterministic finite automaton, NFA, is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. $Q$ is a finite set called the states,
2. $\Sigma$ is a finite set called the alphabet,
3. $\delta : Q \times \Sigma_\epsilon \rightarrow \mathcal{P}(Q)$ is the transition function,
4. $q_0 \in Q$ is the start state, and
5. $F \subseteq Q$ is the set of accept states.
An example

Similar to DFA, we can formally represent any NFA. For example, \( N_1 = (Q, \Sigma, \delta, q_1, F) \).

\[
Q = \{q_1, q_2, q_3, q_4\}, \\
\Sigma = \{0, 1\}, \\
F = \{q_4\}.
\]

Finally, \( \delta \) is given as follows:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>( \epsilon )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_1 )</td>
<td>{q_1}</td>
<td>{q_1, q_2, q_3}</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( q_2 )</td>
<td>{q_3}</td>
<td>( \emptyset )</td>
<td>{q_3}</td>
</tr>
<tr>
<td>( q_3 )</td>
<td>( \emptyset )</td>
<td>{q_4}</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( q_4 )</td>
<td>{q_4}</td>
<td>{q_4}</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>
Computation in NFA

The notion of computation in NFA is quite similar to that in DFA.

**Definition:** Let \( N = (Q, \Sigma, \delta, q_0, F) \) be an NFA and let \( w = w_1w_2 \cdots w_n \) be a string over \( \Sigma \). Then \( N \) accepts \( w \) if a sequence of states \( r_0, r_1, \ldots, r_n \) exists in \( Q \) such that

1. \( r_0 = q_0 \),
2. \( r_{i+1} \in \delta(r_i, w_{i+1}) \) for \( i \in [0, n - 1] \) and
3. \( r_n \in F \).

Finally, \( L(N) = \{ w | N \text{ accepts } w \} \).

The key difference of the above definition from that of DFA is that condition 2 only requires that \( r_{i+1} \) is one of the allowed successor states, which reflects the fact that for any given state and input symbol, an NFA can have many successor states.
Two more examples

Given the following $N_1$,

$L(N_1) = \{w | w \text{ contains either 101 or 11}\}$.

Let $M = \langle Q, \Sigma, \delta, q_1, F \rangle$ be an NFA, and $L(M) = L$, and let $L^R = \{w^R | w \in L\}$. Then, $L(M^R) = L^R$, where $M_R = \langle Q \cup \{q_R\}, \Sigma, \delta_R, q_R, \{q_1\} \rangle$.

Here,

$\delta_R(q, a) = \begin{cases} p, & p, q \in Q, \text{ and, } \delta(p, a) = q; \\ f, & q = q_R, a = \epsilon, \text{ and, } f \in F \end{cases}$

In general, this is an NFA(?).

$M_R$ is essentially the machine as requested in Exercise 1.31.
Homework: Problem 1.32.

Hint: By the hint in the book, we construct a machine to accept $B_R$, which adds up rows \textit{from left to right}, with possible carry going to the column to the right; then apply the construction obtained in 1.31, which will lead to a machine that accepts $B$.

The 8 possible inputs in $\Sigma_3$ play different roles. For example, if the current input column is any of $(0, 0, 0), (0, 1, 1), (1, 0, 1)$, the machine is self-balanced in the sense that it neither borrows a 1 from the result of columns to its \textit{left} nor generates a carry to the next column to its \textit{right}. Moreover, if the input ends the word, then it should be accepted.

If the input column is $(0, 0, 1)$, then it needs to borrow a 1 from the columns to its left; If the input column is $(1, 1, 0)$, then it generates a carry to the column to its right.
Finally, if the input column is, \((0, 1, 0), (1, 0, 0),\) or \((1, 1, 1)\), then it needs to borrow a carry, and it also generates a carry to the column to its right.

Hence, all the eight possible input symbols can be put into four groups. As a result, the machine is essentially in one of the three states: \textit{balanced}; the columns generates a \textit{carry}, thus another input is expected; or \textit{dumped}, i.e., no matter what the next input is, there is no way the input is acceptable. It surely starts with the \textit{balanced} state.

You \textit{should be} able to finish it.

**Homework:** Give the details of Problem 1.32.
NFA $\equiv$ DFA

This result is quite a surprise, as it appears that NFAs is more powerful than DFAs. On the other hand, this result is also useful, as it is easier(?) to design an NFA, which can be mechanically converted into an equivalent DFA, if it is needed.

It is quite easy to see that every DFA has an equivalent NFA.(Why?)

Answer: Let $M = (Q, \Sigma, \delta, q_0, F')$ be a DFA, we construct an NFA: $N = (Q, \Sigma, \delta', q_0, F)$ such that for all $q \in Q$ and $a \in \Sigma$

$$\delta'(q, a) = \{\delta(q, a)\}.$$
An example

Given the following DFA

\[ M_1 = (Q, \Sigma, \delta, q_1, F), \]

where

\[ Q = \{q_1, q_2, q_3\}, \Sigma = \{0, 1\}, F = \{q_2\}, \]

and the transition function is given as follows:

\[
\begin{array}{c|cc}
\delta & 0 & 1 \\
\hline
q_1 & q_1 & q_2 \\
q_2 & q_3 & q_2 \\
q_3 & q_2 & q_2 \\
\end{array}
\]

we can construct the associated NFA as follows:

\[ M'_1 = (Q, \Sigma, \delta', q_1, F') \]

and \( \delta' \) is given as follows:

\[
\begin{array}{c|cc}
\delta & 0 & 1 \\
\hline
q_1 & \{q_1\} & \{q_2\} \\
q_2 & \{q_3\} & \{q_2\} \\
q_3 & \{q_2\} & \{q_2\} \\
\end{array}
\]
\[ L(M) = L(N) \]

**Proof:** Let \( w = w_1w_2 \cdots w_n \in \Sigma^* \), by definition, \( w \in L(M) \) iff there exist \( r_0, r_1, \ldots, r_n \in Q \), such that 1) \( r_0 = q_0 \); 2) \( r_n \in F \); and 3) for \( i \in [0, n - 1] \), \( \delta(r_i, w_{i+1}) = r_{i+1} \).

Since for all \( a, a \in \{a\} \), and by the construction of \( N \), we have that

\[ r_{i+1} = \delta(r_i, w_{i+1}) \in \{\delta(r_i, w_{i+1})\} = \delta'(r_i, w_{i+1}). \]

Hence,

\[ r_{i+1} = \delta(r_i, w_{i+1}) \iff r_{i+1} \in \delta'(r_i, w_{i+1}). \]

Now, we have 1) \( r_0 = q_0 \); 2) \( r_n \in F \); and 3) for all \( i \in [0, n - 1] \), \( r_{i+1} \in \delta'(r_i, w_{i+1}) \).

Thus, by definition, \( w \in L(N) \).
**Theorem 1.19.** Every NFA has an equivalent DFA.

The proof is also based on a construction: given an NFA, build a DFA such that the latter accepts the same language as accepted by the former.

The key observation is that for any given state in the NFA and any input, the successor is a subset of states. Thus, it makes sense to use a subset of states in the NFA to represent a state in the corresponding DFA. Notice that if the NFA has $k$ states, there could be as many as $2^k$ subsets of those states. To begin with, we use $\{q_0\}$ as our start state for the DFA.

As for an NFA, if any copy leads to an accept state, so does the original one, thus, if at the end of the input, the corresponding DFA arrives at a subset that contains an original accept state, the DFA accepts this input.
Proof: Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA. We construct a DFA, $M$, that accepts the same language as $N$ does. Let $M = (Q', \Sigma, \delta', q'_0, F')$. We already know that $Q' = \mathcal{P}(Q), q'_0 = \{q_0\}$ and $F' = \{R \in Q' | R \cap F \neq \emptyset\} = \{R | \exists r \in R, r \in F\}$.

Let $R \in Q'$ and let $a \in \Sigma$, we try to define $\delta'(R, a)$. As $R \subseteq Q$, for any $q \in R$, and $a$, there might be several successor states, all of which should be taken into account. Thus,

$$\delta'(R, a) = \bigcup_{r \in R} \delta(r, a).$$

An equivalent way to express $\delta'$ is as follows.

$$\delta'(R, a) = \{q \in Q | \exists r \in R, q \in \delta(r, a)\}$$
We also have to consider the $\epsilon$’s. For every $R \in Q'$, we use $E(R)$ to denote the collection of those states that can be reached through 0 or more $\epsilon$ arrows. From DFA’s point of view, those states should be regarded the same, as no symbol is needed to go from one to another.

Thus, we replace the definition of $\delta'$ with the following one.

$$\delta'(R, a) = \bigcup_{r \in R} E(\delta(r, a)).$$

Finally, if the start state in $N$ is connected to another state $q$ with an $\epsilon$ labeled arrow, then, we also should include $q$ into our start state. Therefore, $q'_0 = E(q_0)$. $\square$
An example

Given the following $N_4$, an NFA, we show how to convert it into a DFA, $D = (Q, \{a, b\}, \delta, q_0, F)$.

To begin with, we have that $Q = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$.

As the start state of $N_4$ is 1, we have that $q_0 = E(1) = \{1, 3\}$. Also, as 1 is also the only accept state in $N_3$, we have that $F = \{\{1\}, \{1, 2\}, \{1, 3\}, \{1, 2, 3\}\}$. 
We now decide $\delta$, which is given as follows.

The above can be further simplified, since nothing is getting into the states $\{1\}$ and $\{1, 2\}$, which are thus \textit{unreachable}:

\textbf{Homework:} Give a four state NFA for the language of Exercise 1.13.
Closure properties (II)

As we have proved the equivalences between DFA’s and NFA’s, we can begin to prove the closure properties of regular languages, i.e., given two regular languages, $A_1$ and $A_2$, we want to show that $A_1 \ast A_2$ is still a regular language, where $\ast$ is either union or concatenation.

The basic strategy is as follows: As both $A_1$ and $A_2$ are regular languages, by definition, for two FA’s $M_i, i = 1, 2$, we have that $L(M_i) = A_i$. We construct an NFA, $N$ such that $L(N) = A_1 \ast A_2$. Because of the equivalence result between DFA and NFA, there must exist a DFA, $M$, such that $L(M) = L(N) = A_1 \ast A_2$. Hence, by definition, $A_1 \ast A_2$ is regular.

The strategy to show that $A^*$ is regular is the same.
**Theorem 1.22:** The class of regular languages is closed under the union operations.

Given two NFA’s, $N_1$ and $N_2$, that accept $A_1$ and $A_2$ respectively, we construct $N$, an NFA, that accept $A_1 \cup A_2$.

The key observation is that $N$ accept an input string if either $N_1$ or $N_2$ accepts it. Thus, we provide a new start state for $N$ which branches to the two start states of $N_1$ and $N_2$ via an $\epsilon$ arrow. Then, $N_i$ will take over.
Proof: Let $N_i = (Q_i, \Sigma, \delta_i, q_i, F_i)$, be an FA that accepts $A_i, i = 1, 2$. We construct another NFA, $N_\equiv (Q, \Sigma, \delta, q_0, F)$ to accept $A_1 \cup A_2$.

We let a new state, $q_0$, be the start state of $N$.

$$Q = Q_1 \cup Q_2 \cup \{q_0\},$$
$$F = F_1 \cup F_2.$$

Next, we define the transition function.

$$\delta(q, a) = \begin{cases} 
\delta_1(q, a) & q \in Q_1, \\
\delta_2(q, a) & q \in Q_2, \\
\{q_1, q_2\} & q = q_0, a = \epsilon \\
\emptyset & q = q_0, a \neq \epsilon.
\end{cases}$$

Homework: Exercise 1.8(b).
\[ L(N) = L(N_1) \cup L(N_2) \]

Let \( w \in L(N) \), i.e., there is a computation that leads from \( q_0 \) all the way to an accepting state \( q_f \in F \). Without loss of generality, let \( q_f \in F_1 \). By construction, such a computation starts with \( q_0 \xrightarrow{\epsilon} q_1 \), followed by a computation from \( q_1 \) to \( q_f \), with the latter sequence showing \( w \in L(N_1) \subseteq L(N_1) \cup L(N_2) \).

Conversely, let \( w \in L(N_1) \), i.e., there is a computation that leads from \( q_1 \) all the way to an accepting state \( q_f \in F_1 \). We now add \( q_0 \xrightarrow{\epsilon} q_1 \) to the front of the above computation, to obtain a computation from \( q_0 \) all the way through to \( q_f \in F_1 \subseteq F \). This latter computation shows that \( q \in L(N) \).

The case that \( w \in L(N_2) \) can be similarly shown.
**Theorem 1.23:** The class of regular languages are closed under the concatenation operation.

Again, we construct an NFA, $N$, to accept $A_1 \circ A_2$, if $A_i$ is accepted by $N_i, i = 1, 2$.

The key observation in this case is that $N$ accepts an input string if the first part of it is accepted by $N_1$ and the rest is accepted by $N_2$. Thus, we use the start state of $M_1$ as the start state of $N$ and for each and every accept state of $N_1$, we provide an $\epsilon$ arrow to connect this state with the start state of $N_2$. Finally, we use the accept states as those for $N$. 
Proof: Let $N_i = (Q_i, \Sigma, \delta_i, q_i, F_i)$, be an FA that accepts $A_i, i = 1, 2$. We construct an NFA, $N = (Q, \Sigma, \delta, q_1, F_2)$, to accept $A_1 \circ A_2$.

We let $Q$ be $Q_1 \cup Q_2$ and define the transition function as follows.

$$
\delta(q, a) = \begin{cases} 
\delta_1(q, a) & q \in Q_1, a \notin F_1 \\
\delta_1(q, a) & q \in F_1, a \neq \epsilon \\
\delta_1(q, a) \cup \{q_2\} & q \in F_1, a = \epsilon \\
\delta_2(q, a) & q \in Q_2.
\end{cases}
$$

Similarly to what we did for the union case, we can show that $L(N) = L(N_1) \circ L(N_2)$.

Homework: Exercise 1.9(b).
**Theorem 1.24:** The class of regular languages are closed under the star operation.

We construct an NFA, $N$, to accept $A^*$, if $A$ is accepted by an FA $M$.

The idea is that this time $N$ accepts an input string if the latter can be partitioned into a couple of parts, each of which is accepted by $M$. Thus, for each and every accept state of $M$, we add an $\epsilon$ arrow to connect it with the start state of $M$. As $\epsilon \in A^*$ for any $A$, we also have to add a new state so that the empty string will be accepted.
**Proof:** Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$, which accepts $A_1$. Construct $N = (Q, \Sigma, \delta, q_0, F)$ to accept $A_1^*$. We let a new state, $q_0$, be the start state of $N$.

$$Q = Q_1 \cup \{q_0\},$$

$$F = F_1 \cup \{q_0\}.$$

Next, we define the transition function.

$$\delta(q, a) = \begin{cases} 
\delta_1(q, a) & q \in Q_1, q \not\in F_1 \\
\delta_1(q, a) & q \in F_1, a \neq \epsilon \\
\delta_1(q, a) \cup \{q_1\} & q \in F_1, a = \epsilon \\
\{q_1\} & q = q_0, a = \epsilon \\
\emptyset & q = q_0, a \neq \epsilon.
\end{cases}$$

Similarly to what we did for the union case, we can show that $L(N) = L(N_1)^*$.  

**Homework:** Exercise 1.10(c).
Additional closure properties

**Result 1.** Let $L \subseteq \Sigma^*$ be a regular language. Then $\Sigma^* - L$ is also regular.

**Proof:** Let $\mathcal{M} = (Q, \Sigma, \delta, q_1, F)$ be a DFA that accepts $L$. Then, $\overline{\mathcal{M}} = (Q, \Sigma, \delta, q_1, Q - F)$ is a DFA that accepts $\Sigma^* - L$.

**Result 2.** Let $L_1$ and $L_2$ be regular languages, then so is $L_1 \cap L_2$.

**Proof:** Let $L_1, L_2 \in \Sigma^*$. Then, by De Morgan’s identity, we have that

$$L_1 \cap L_2 = \Sigma^* - ((\Sigma^* - L_1) \cup (\Sigma^* - L_2)).$$

By previous results, regular languages are closed under both $\cup$ and $-$. Hence, $L_1 \cap L_2$ is regular.

**Homework:** Exercise 1.14.
Result 3. $\emptyset$ is regular.

Proof: Let $\mathcal{M}_\emptyset$ be any DFA with its accept set being $\emptyset$, then $L(\mathcal{M}_\emptyset) = \emptyset$.

Result 4. Let $a \in \Sigma$, $\{a\}$ is regular.

Proof: Just let $\mathcal{N}_a = (\{q_1, q_2\}, \{a\}, \delta, q_1, \{q_2\})$, such that $\delta(q_1, a) = q_2$. Then, $L(\mathcal{N}_a) = \{a\}$.

Result 5. Let $u \in \Sigma^*$. Then, $\{u\}$ is regular.

Proof: Assume that $u = a_1 a_2 \cdots a_n$, then $\{u\} = \{a_1\} \cdot \{a_2\} \cdots \cdot \{a_n\}$. By Result 4, for each $a_i, i \in [1, n]$, there is an $\mathcal{N}_{a_i}$, such that it accepts $\{a_i\}$. Hence, by the closure property with respect to the ‘·’ operation, there exists an NFA that accepts $\{u\}$. 
Corollary 1. Every finite subset of $\Sigma^*$ is regular.

Proof: We already show that $\emptyset$ is regular. Thus, let $L = \{u_1, \ldots, u_n\}$, where for all $i, u_i \in \Sigma^*$. Then, clearly, $L = \{u_1\} \cup \cdots \cup \{u_n\}$. Then, by the previous result, for each $u_i, i \in [1, n]$, there is an NFA that accepts $\{u_i\}$. Then, by the closure property with respect to the ‘$\cup$’ operation, these exists an NFA that accepts $L = \{u_1, \ldots, u_n\}$.

All these properties can be regarded as some of the capabilities of the regular machine, as we will see later that some of the other machines do not have some of these properties.
Regular expression

In arithmetic, we can use such operators as $+$ and $\times$ to build up arithmetic expression such as $(5 + 3) \times 4$, which leads to a value 32. Similarly, we can use regular operations to build up regular expressions, e.g., $(0 \cup 1)0^*$, which leads to a language. In this case, the collection of all strings that begins with either 0 or 1, followed by any number of 0’s.

It turns out that regular expression is quite handy to represent a language. Moreover, it is equivalent to FA’s. Thus, it is quite useful in designing an FA: We represent a regular language in regular expression, then convert it into an FA.
Let $\Sigma = \{0, 1\}$. Then we can regard $\Sigma$ as a shorthand for the expression $0 \cup 1$. In general, if $\Sigma$ is any alphabet, it describes the language consisting of all strings of length 1 and $\Sigma^*$ represent the language consisting of all strings over $\Sigma$. Finally, $(0\Sigma^*) \cup (\Sigma^*1)$ represents all strings that either begin with a 0 or end with 1.

**Definition:** We say that $R$ is a regular expression, if $R$ is
1. $a$ for some $a \in \Sigma$, 
2. $\epsilon$, 
3. $\emptyset$, 
4. $(R_1 \cup R_2)$, where both $R_1$ and $R_2$ are regular expressions, 
5. $(R_1 \circ R_2)$, where both $R_1$ and $R_2$ are regular expressions, or 
6. $(R_1)^*$, where $R_1$ is a regular expression.

**Homework:** What is the regular expression for the language as defined in Exercise 1.13?
Some examples

In the above definition, the expression $\epsilon$ represents a language consisting of only one string, i.e., the empty string $\epsilon$, while $\emptyset$ represents a language consisting none.

\[
\begin{align*}
0^*10^* &= \{w \mid w \text{ contains exactly a single } 1\}. \\
\Sigma^*1\Sigma^* &= \{w \mid w \text{ has at least one } 1\}. \\
\Sigma^*001\Sigma^* &= \{w \mid w \text{ contains } 001\}. \\
(\Sigma\Sigma)^* &= \{w \mid w \text{ is a string of even length}\}. \\
(\Sigma\Sigma\Sigma)^* &= ? \\
01 \cup 10 &= ?. \\
(0 \cup \epsilon)1^* &= ? \\
(0 \cup \epsilon)(1 \cup \epsilon) &= ? \\
1^*\emptyset &= ? \\
\emptyset^* &= ? \\
R \cup \emptyset &= ? \\
R \cup \epsilon &= ? \\
R \circ \emptyset &= ? \\
R \circ \epsilon &= ?
\end{align*}
\]
Lemma 1.32: If a language is described by a regular expression, then it is regular.

Proof by induction on the length of the expression: Let $R$ be the given expression. If $|R| = 1$, then $R = a$, for some $a \in \Sigma$; or $R = \epsilon$; or $R = \emptyset$. In those three cases, we can construct three basic FAs to accept the regular expressions, respectively.

Otherwise, by definition, $R = R_1 \cup R_2$, $R = R_1 \circ R_2$, or $R = R_1^*$. In all three cases, $|R_i| < R, i = 1, 2$. Thus, by inductive assumption, the languages described by $R_i$ are regular. Therefore, the language described by $R$ is regular, as well, due to the closure properties of regular languages. \qed
An example

Below shows how to convert an expression, 
\((ab \cup a)^*\), into an NFA.

Homework: Exercise 1.19.
**Lemma 1.29:** If a language is regular, then it is described by a regular expression.

The main idea is that if a language is regular, then it must be accepted by a DFA. Thus, what we have to show is that any DFA can be converted to an equivalent regular expression.

We do this in two steps: 1. convert any DFA into a GNFA, i.e., generalized NFA, and 2. convert a GNFA to a regular expression.

For each step, we have to show that the related conversion leads to the desired equivalence.
What is a GNFA?

A GNFA is an NFA whose transition function may have regular expressions as its labels, which also has some additional properties: 1) the start state goes into any other state, but no arrow comes in from another state;

2) there is only one accept state, which is different from the start state, and for the accept state, we only allow incoming arrows from other states, but no arrow going out;

3) except for the start and the accept states, exactly one arrow goes out from one state to another state, including itself.
How to get a GNFA?

If a GNFA does not have the extra properties, we can always fix it by
1) adding a new start state, connected to the old one with an $\epsilon$ arrow;

2) adding a new accept state, to which all the old accept states are connected with $\epsilon$ arrows;

3) replacing multiple label arrows with a single one labeled with the union of all the old labels; and

4) adding an $\emptyset$ labeling arrow between any two states for which no arrow existed.
The basic strategy

If the GNFA has just two states, then by definition, there must an arrow from the start state to the accept state, its label is the regular expression we are looking for.

In general, let the number of states in the GNFA be $k$, we will show how to construct another GNFA with $k - 1$ states, and repeat this reduction process until there are only two states left, thus return to the base case.

Below shows the process of obtaining the demanded expression for a DFA with 3 states.
For a GNFA with $k > 2$ states, we select a state, $q_{\text{rip}}$, and rip it out of the machine and modify the machine so that it accepts the same language, by adding in the lost computation.

This is best described by using the following figure.

We are now ready to formalize this process and prove its correctness.
The algorithm

Let $M$ be a DFA, we make it into a GNFA, $G$, by adding a new start state and a new accept state, as well as other needed arrows. Let $k$ be the number of states of $G$. The following algorithm $\text{convert}(G)$ returns its equivalent regular expression.

1. If $k = 2$, return the label of the only arrow in $G$.

2. Otherwise, select $q_{\text{rip}} \in Q - \{q_{\text{start}}, q_{\text{accept}}\}$, and let $G'$ be $(Q', \Sigma, \delta', q_{\text{start}}, \{q_{\text{accept}}\})$, where $Q' = Q - \{q_{\text{rip}}\}$, and for any $q_i \in Q' - \{q_{\text{accept}}\}$ and any $q_j \in Q' - \{q_{\text{start}}\}$, let $\delta'(q_i, q_j)$ be the regular expression $(R_1)(R_2)^*(R_3) \cup R_4$, where $R_1 = \delta(q_i, q_{\text{rip}})$, $R_2 = \delta(q_{\text{rip}}, q_{\text{rip}})$, $R_3 = \delta(q_{\text{rip}}, q_j)$, and $R_4 = \delta(q_i, q_j)$.


76
An example

Below shows how to get an equivalent expression for a DFA.
Another example

Homework: Exercise 1.21(b).
Claim: For any GNFA $G$, $\text{convert}(G)$ is equivalent to $G$.

Proof by induction on $|G|$: If $G$ has only two states, it has only one arrow from the start state to the accept state. The regular expression label on that arrow specifies all the strings that allow $G$ to get to the accept state, thus, it is equivalent to $G$.

Assume the claim holds for any GNFA with $k - 1$ states and let $|G| = k$. If we can show that $L(G) \equiv L(G')$, where $G'$ is the GNFA constructed in $\text{convert}(G)$, then by inductive assumption, the regular expression returned by $\text{convert}(G')$ is equivalent to $G'$, thus, it is also equivalent to $G$.

The only thing left is to show that $L(G) \equiv L(G')$. 

79
$L(G) = L(G')$

To show $L(G) \subseteq L(G')$, assume that $w \in L(G)$, i.e., $G$ accepts $w$. Thus, there must be a sequence of states, $q_{\text{start}}, q_1, \ldots, q_{\text{accept}}$, accepting $w$ in $G$. If $q_{\text{rip}}$ doesn't occur in this sequence, then this sequence accepts $w$ in $G'$ as well, since each new regular expression in $G'$ contains the old one as a union part.

Otherwise, let $q_i \xrightarrow{r_1} q_{\text{rip}} \xrightarrow{r_3} q_{\text{rip}} \cdots \xrightarrow{r_3} q_{\text{rip}} \xrightarrow{r_2} q_j$ be a segment in the above sequence. Clearly, $r_1 \in L(R_1(=\delta(q_i,q_{\text{rip}})))$, $r_2 \in L(R_2(=\delta(q_{\text{rip}},q_j)))$, and $r_3 \in L(R_3(=\delta(q_{\text{rip}},q_{\text{rip}})))$.

we replace this segment with $q_i \xrightarrow{r_1 r_3 \cdots r_3 r_2} q_j$. Since $R_1(R_3)^*R_2$ is included in $\delta'(q_i,q_j)$, and $r_1 r_3 \cdots r_3 r_2 \in L(R_1(R_3)^*R_2)$, $w \in L(G')$. 

80
We now show that $L(G') \subseteq L(G)$. Assume that $G'$ accepts $w$, we construct an acceptance sequence for $w$ in $G$ as follows: Let $q_i \xrightarrow{r} q_j$ be a transition in the sequence that leads to the acceptance of $w$ in $G'$, and let $(R_1(R_3)^*R_2) \cup R_4$ be the label of $(q_i, q_j)$ in $G'$, if $r \in L(R_4)$, we leave this transition there; otherwise, if $r \in L(R_1 \circ (R_3)^n \circ R_2)$, where $R_1, R_2,$ and $R_3$ are the same as defined, we replace this transition with $q_i \xrightarrow{r_1} q_{rip} \xrightarrow{r_3} q_{rip} \cdots \xrightarrow{r_3} q_{rip} \xrightarrow{r_2} q_j$.

Once completed, the new sequence leads to the acceptance of $w$ in $G$. Hence, $L(G') \subseteq L(G')$.

Hence, we have the following result:

**Theorem 1.28:** A language is regular iff it is described by some regular expression.
Non-regular languages

We know that an FA can control an automatic door, as well as recognize all the reserved words in a programming language. Thus, it can do some useful things.

But, can it do everything? Certainly not. For example, in c and all its successors, every ‘{’ must be balanced out with a ‘}’. In an algebraic expression, every ‘(’ also must be balanced with a ‘)’. Such a requirement can be represented as a language, \( B = \{0^n1^n|n \geq 0\} \).

Can we find an FA to accept this language? It seems quite difficult for an FA to do it, as it has to remember how many 0’s it has read to match the following 1’s.

But, how can we prove that \( B \) is not regular?
A special tool

Every dog barks. Thus, if something does not bark, it can’t be a dog.

*Pumping lemma* is a classic technique to show that a language is *not* regular.

This result states that all regular languages must have certain properties.

Thus, if we can show that a language does not have at least one of these properties, it can’t be regular.
The Pumping Lemma: If $A$ is a regular language, there is a number $l$, called the critical length, where if $s \in A$ and $|s| \geq l$, then $s = xyz$, such that

1. for each $i \geq 0$, $xy^i z \in A$,
2. $|y| > 0$, and
3. $|xy| \leq l$.

Property 1 says that all the strings in the specific format must belong to $A$, as well.

Property 2 says that $y \neq \epsilon$, without which the lemma would be trivially true. (Why?)

Property 3 says that the length of $x$ and $y$ will be at most $l$. This is useful in proving some results.
Basic ideas

Let $M = (Q, \Sigma, \delta, q_1, F)$ be a DFA that accepts $A$, $|Q| = l$, and let $s \in A$ such that $|s| \geq l$, we consider the sequence of states in the computation of $M$ that accepts $s$.

Starting from $r_0$ ($= q_1$), reading $s_1$, $M$ gets into another state $r_1$; it then reads another symbol $s_2$ and gets into another state $r_2$. ..., until it reads the last one $s_n$, and get into an accept state $r_n$ $(\in F)$. 
Now what?

The key observation is that, if $|s| = n \geq l$, then the length of the above state sequence must be $n + 1 > l$.

By the pigeonhole principle, at least two states, $r_j$ and $r_k$, $j \neq i$, i.e., in different places of that sequence, must be identical.

Without loss of generality, let $q_9$ be the very first state that repeats itself. We can split $s$ into three pieces, $x, y$ and $z$, which are parts of $s$ appearing before $q_9$, in between two instances of $q_9$ and the part thereafter.

In other words, $x$ brings $M$ from $r_1 \ (= q_1)$ into $q_9$, $y$ brings $M$ from $q_9$ back to $q_9$, and $z$ brings $M$ to an accept state $r_n$. 
Why three properties?

Suppose that we run $M$ on input $xy^iz, i > 1$. As we know, $x$ takes $M$ into $q_9$, and the first occurrence of $y$ takes $M$ from $q_9$ back into $q_9$, so does the second, and any other, occurrence of $y$, and $z$ brings $M$ into an accept state. On the other hand, $xy^0z \equiv xz$, which certainly brings $M$ into the same accept state. Thus, condition 1 holds.

As the two occurrences of $q_9$ appear in the two different places of the sequence and $M$ is a DFA, $y \neq \epsilon$, thus the second condition holds as well.

As it takes $n + 1$ states to process $n$ symbols and the first $l + 1$ states must contain a repetition. If $|xy| \geq l + 1$, then, $q_9$ would not be the first state that repeats itself. Thus, the third condition also holds.
Proof of the pumping lemma: Let $M = (Q, \Sigma, \delta, q_1, F)$ be a DFA that accepts $A$, $|Q| = l$, and let $s \in A, s = s_1s_2 \cdots s_n, n \geq l$.

Let $r_0 (= q_1), \ldots, r_n (\in F)$ be the state sequence that $M$ enters while reading $s$, so $r_{i+1} = \delta(r_i, s_i), i \in [0, n)$.

By the pigeonhole principle, among the first $l+1$ elements in the above state sequence, two of them must be the same. We call the fist $r_j$, the second $r_k$. Because $r_k$ occurs within the first $l+1$ elements, $k \leq l+1$. Let $x = s_1 \cdots s_{j-1}$, $y = s_j \cdots s_{k-1}$, and $z = s_k \cdots s_n$.

As $x$ takes $M$ from $r_0$ to $r_j$, $y$ takes $M$ from $r_j$ to $r_k (= r_j)$, and $z$ takes $M$ from $r_k$ to $r_n$, an accept state, $M$ must accept $xy^iz, i \geq 0$. As $j \neq k, y \neq \epsilon$. Finally, as $|xy| = k - 1$ and $k \leq l + 1$, we have that $|xy| \leq l$. □
An example

**Result:** \( B = \{0^n1^n | n \geq 0\} \) is not regular.

**Proof:** Assume \( B \) is regular, thus, it is accepted by \( M \), a DFA and let \( l \) be the number of states of \( M \). Let \( s = 0^l1^l \). As \( s \in B \) and \( |s| > l \), the pumping lemma says that \( s = xyz \), and for all \( i \geq 0 \), \( xy^iz \in B \), as well.

1. If \( y \) contains only 0, then \( xyyz \) has more 0’s than 1’s, thus, \( xy^2z \not\in B \).
2. The case that \( y \) contains only 1 is the same as case 1.
3. If \( y \) contains both 0 and 1, then \( xyyz \) will not have the pattern as required by \( B \).

Thus, \( B \) does not have the required property, it cannot be regular. \( \square \)
More examples

Result: $C = \{ w \mid w \text{ has an equal number of 0's and 1's} \}$ is not regular.

Proof: The first part of the proof is the same as that for the previous result. The only part left is that of case 3. If $y$ contains both 0 and 1, it might be still in $C$ as $C$ doesn’t have a pattern requirement.

However, as the condition 3 of the pumping lemma says, $|xy| \leq l$. Hence, $y$ can only contains 0. \qed

Result: $E = \{ 0^i1^j \mid i > j \}$ is not regular.

Proof: Assume that $E$ is regular. Let $l$ be the critical length in the pumping lemma and let $s = 0^{l+1}1^l$. Then, $s = xyz$, satisfying all the conditions of the lemma. By the condition 2 and 3, $|y| > 0$ and $|xy| \leq l$. Thus, the number of 0’s can’t be more than that of 1’s in $xz$. \qed
What does it tell us?

The pumping lemma can be used to show that a large number of languages are not regular.

Many of them, such as B, C, and E, all seem to require “unlimited” amount of memory. However, this apparent requirement for unlimited memory does not always prohibit a language from being regular.

For example, the language that contains all the strings with an equal number of occurrences of 01 and 10 as substrings is regular.

**Homework:** Exercise 1.29.