Chapter 2
Regular Machines

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 such information.

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., but its *ability*, especially, 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 computers with finite memory.

An example

Some computer needs very little information.

For example, an automatic door comes with a controller and two sensors: a *front sensor* to detect if a person is approaching the door from the front, and a *rear one* to tell the controller if a person is approaching the door from the back.

The controller decides if the door should be open or close.

Question: What should this computer do with the information collected in the two sensors?

What should the controller do?

The controller can be in one of two *state*s, “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.*

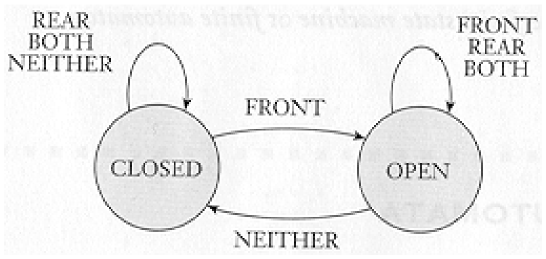
	NEITHER	FRONT	REAR	BOTH
C	C	O	C	C
O	C	O	O	O

Question: How could someone come out? ☹

What common sense?

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. And, 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 could represent the controller as the following *state diagram.*



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, 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. 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, an a *lexicon,* part of a language that decides if a string is a valid *reserved word,* such as "if", "while", etc.

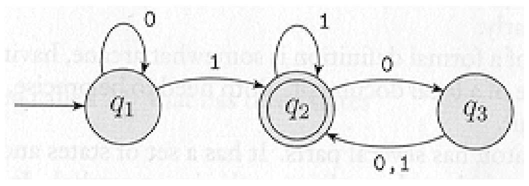
It turns out that all these controllers can be represented and designed by following the *finite automata theory.*

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

All regular machines "speak" the *regular* language, which characterizes its ability. ☺

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.

Assignment: It is a good time to start to play with *JFLAP* to appreciate its dynamics.

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,* at the end.

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, stays with $q2$;

4. read 0, then goes to q_3;

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

6. M_1 accepts the input string, as q_2 is an accept state.

JFLAP certainly makes such a process less painful. ☺

Question: What does M_1 actually do?

Design vs. analysis

A finite automata only accepts a specific set of strings, which does not need to be finite. (?) This is why a lexicon 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: *Given a set of strings, find out a machine that will accept it.*

By *analyzing an automaton,* we want to find out what set of strings *it* will accept.

If we 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. Σ is a finite set called the *alphabet,*
3. $\delta : Q \times \Sigma \mapsto Q$ 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 (?).

Thus, this notion defines a *deterministic* finite automaton.

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:

$$\begin{aligned} M_1 &= (Q, \Sigma, \delta_1, q_1, F), \text{ where} \\ Q &= \{q_1, q_2, q_3\}, \\ \Sigma &= \{0, 1\}, \\ F &= \{q_2\}, \end{aligned}$$

and the transition function is given as follows:

δ_1	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

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.

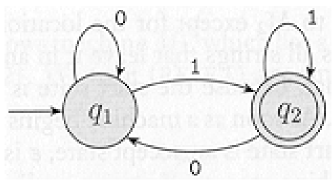
Definition. If A is the set of input strings that a finite automaton M accepts, we say that A is the *language of machine* M, and write $L(M) = A$.

We also say that M *accepts*, or *recognizes*, A.

For example, let $A = \{w : w$ 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.



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

δ_2	0	1
q_1	q_1	q_2
q_2	q_1	q_2

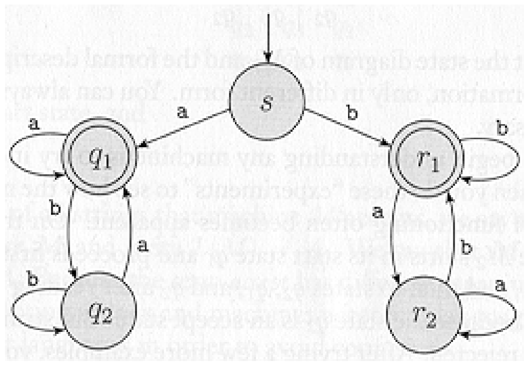
Question: What is $L(M_2)$?

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

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

Another one

Below is the diagram of another finite automaton M_4.



Question: What is $L(M_4)$?

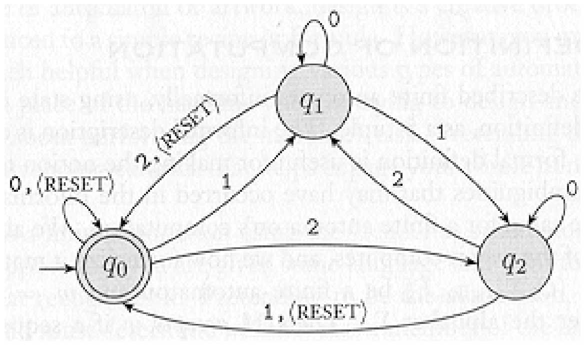
Any string accepted by M_4 is accepted by either the left or the right. Thus, $L(M_4)$ is the *union* of the languages accepted by the left and that by the right.

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.

Remember*Divide n' Conquer*?

Yet another one

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.

For example, it accepts 22120021101, as it equals 12 (= 0 mod 3).

Homework assignment

1. Download the *JFLAP* simulator software, install it in your computer, or `M` drive; and play with the *FA* part. We should have put it up in the D&M labs.

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 *JFLAP* to verify your results.

4. When you hand in this batch of homework, do attach two *screen shots* of *JFLAP* program files that you ran with the *JFLAP* simulator, showing both a positive and a negative cases.

FA as computers

From the previous examples, we informally see how a finite automaton sends out an output, when given some input, i.e., computes the output. Here comes a formal definition of this *computation process*.

Definition: Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton and let $w = w_1 w_2 \cdots w_n$ be a string over Σ. 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. $r_{i+1} = \delta(r_i, w_{i+1})$ for $i \in [0, n-1]$ and
3. $r_n \in F$.

In other words,

$$r_0\ (= q_0) \overset{w_1}{\to} r_1 \overset{w_2}{\to} r_2 \cdots \overset{w_{n-1}}{\to} r_{n-1} \overset{w_n}{\to} r_n\ (\in F).$$

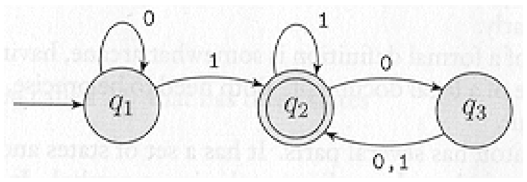
Finally, $L(M) = \{w : M \text{ accepts } w\}$.

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

An example

Question: Given M_1, why $1101 \in L(M_1)$?

Answer: With the given transition function,



we have $n = 4$, $w_1 = 1, w_2 = 1, w_3 = 0$, and $w_4 = 1$.

Starting with $r_0 = q_1$, we have found $r_1 = q_2\ (= \delta_1(r_0, 1)), r_2 = q_2\ (= \delta_1(r_1, 1)), r_3 = q_3\ (= \delta_1(r_2, 0))$, and $r_4 = q_2\ (= \delta_1(r_3, 1))$, such that

$$r_0\ (= q_1) \xrightarrow{1} r_1 \xrightarrow{1} r_2 \xrightarrow{0} r_3 \xrightarrow{1} r_4.$$

Finally, $r_4\ (= q_2) \in F$.

Thus, by the above definition, we have that $w = 1101 \in L(M_1)$.

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 much more challenging to *design* an automaton for a given language L such that this FA is supposed to accept L. ☹

We will present a few tools to help us to get it done. FA design 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)

Question: How to use a finite mechanism to accept an infinite set of strings? ☹

The major issue is how to memorize possibly long sequence of input symbols with a limited amount of memory.

Obviously, we only need 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 *that* of the number itself. ☺

Thus, we need to have two *states,* q_{even} and q_{odd}. As initially, nothing, particularly no 1, has been read, we let q_{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_{odd} be the only *accept state.*

Finally, the transition function is easy to specify, based on our task: If we are in q_{even} and see another 0, we stay in q_{even}; otherwise, switch over to q_{odd}. The case for q_{odd} is similar.

It looks like this...

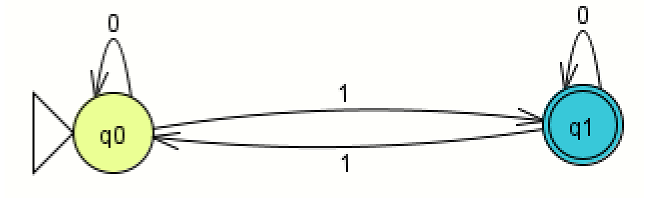
Let q_0, q_1 stand for q_{even} and q_{odd}, respectively. we have that

$$M_{\text{Odd}} = \{\{q_0, q_1\}, \{0, 1\}, \delta_{\text{Odd}}, q_0, \{q_1\}\},$$

where, the transition function, δ_{Odd}, can be specified as follows:

δ_{Odd}	0	1
q_0	q_0	q_1
q_1	q_1	q_0

Below is the diagram of this machine.



Question: Should we play with it?

Another one

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 $Q = \{q_\epsilon, q_0, q_{00}, q_{001}\}$, where q_ϵ 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.

Starting with q_ϵ, 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_0; otherwise, we move to q_{00} and just need to wait for another 1.

Once we have got the final 1, we sail through to q_{001}, then we could not care less what symbols will follow, we just stay put. ☺

Question: When seeing another 0 in q_{00}, should we go back to q_ϵ, q_0 or q_{00}? Why? ☹

Answer: Play with 0001...

It looks like this...

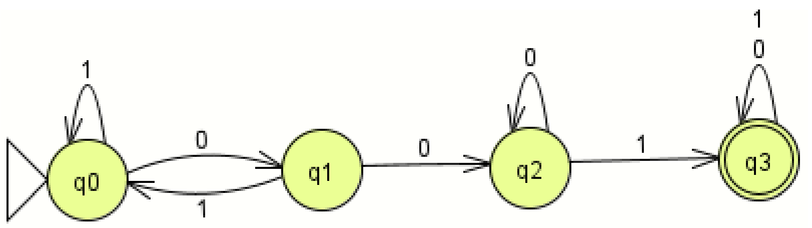
Let q_0, q_1, q_2, q_3 stand for q_ϵ, q_0, q_{00} and q_{001}, respectively. we have that

$$M_{001} = \{\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta_{001}, q_0, \{q_3\}\},$$

where, the transition function, δ_{001}, can be specified as follows:

δ_{001}	0	1
q_0	q_1	q_0
q_1	q_2	q_0
q_2	q_2	q_3
q_3	q_3	q_3

Notice that we have to let $\delta_{001}(q_2, 0) = q_2$.



Question: Will this work?

Answer: Check it out with you-know-what. ☺

Regular Operators

Design is an important task, which is often challenging. ☹ We need help... As a tool to facilitate this task, we 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:

$$\begin{aligned} A \cup B &= \{x : x \in A \vee x \in B\}. \\ A \circ B &= \{x \circ y : x \in A \wedge x \in B\}. \\ A^* &= \{x_1 \circ x_2 \circ \cdots \circ x_k : k \geq 0, x_i \in A\}. \end{aligned}$$

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.

The unary *star* operation, $*$, glues together any strings in A, of length at least 0, thus includes the empty string ϵ.

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, e.g., $2 - 3 = -1$ ($\notin \mathcal{N}$); or division, e.g., $3/2 = 1.5$ ($\notin \mathcal{N}$). ☹

Question: How about $\mathcal{Z}$, the collection of all the integers? Is it closed under all the four operations?

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

Question: Why is *it* useful in FA design?

Implication on design

We will show that the collection of all the regular languages is closed under all those three regular operations. Thus, 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: Given a language L, we have to find an FA, M, for L, such that $L(M) = L$.

We first *decompose* L to a bunch of smaller ones in terms of the three operators. *If all of them are regular,* i.e., each is associated with an FA, we can then construct an FA by *combining* these smaller machines into one for L. Thus, L is also regular. We are done. ☺

Question: *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 two automata for A_1 and A_2, come up with (design) another to accept $A_1 \cup A_2$.

Clearly, for any given string w, $w \in A_1 \cup A_2$ iff $w \in A_1$ or $w \in A_2$.

As once a symbol is read, it is gone, we can't 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.

Question: How to do that?

Some ideas first ...

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.

Hence the *state set* has to be the Cartesian product of the two state sets.

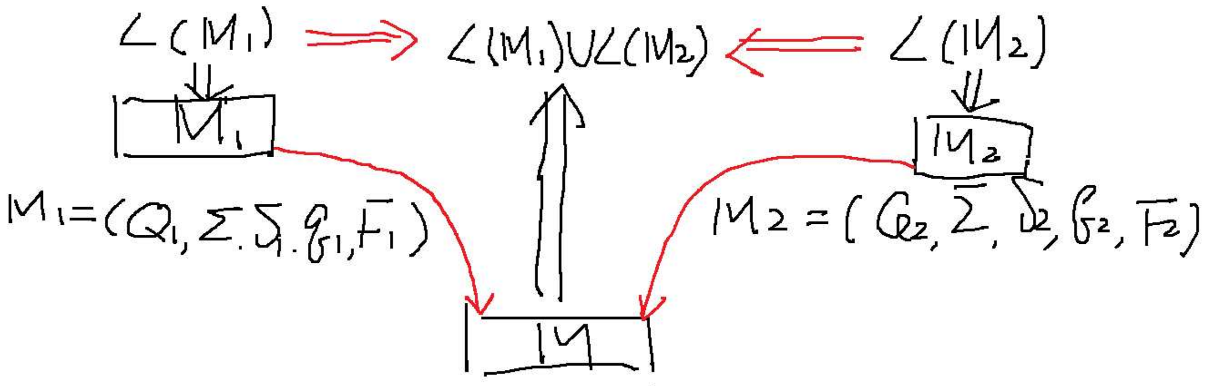
We will *start* with a pair of states, consisting of the two start states of both two machines.

The *alphabet* is certainly the same. 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.*

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 of this new machine.

I want to see...

The whole process looks like the following:



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

$M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$

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

$Q = Q_1 \times Q_2 = \{(r_1, r_2) : r_1 \in Q_1, r_2 \in Q_2\}$

$q_0 = (q_1, q_2)$, $F = \{(r_1, r_2) ; r_1 \in F_1, \text{ or } r_2 \in F_2\}$

$\delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a))$

Thus, given two machines, M_1 and M_2, we can construct another one, which accepts $L(M_1) \cup L(M_2)$.

Question: Should we trust Dr. Shen?

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

$$\begin{aligned} M_1 &= (Q_1, \Sigma, \delta_1, q_1, F_1), \text{ where} \\ Q_1 &= \{q_1, q_2, q_3\}, \\ \Sigma &= \{0, 1\}, \\ F_1 &= \{q_2\}, \end{aligned}$$

and δ_1, the transition function, is given as follows:

δ	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

This is the piece that we saw on Page 6, and defined on Page 11.

Another language

Let L_2 be $\{w : w$ contains an odd number of 1's.$\}$.

L_2 is also regular, since $L_2 = L(M_2)$, where

$$\begin{aligned} M_2 &= (Q_2, \Sigma, \delta_2, q_e, F_2), \text{ where} \\ Q_2 &= \{q_e, q_o\}, \\ \Sigma &= \{0, 1\}, \\ F_2 &= \{q_o\}, \end{aligned}$$

and δ_2, the transition function, is given as follows:

δ	0	1
q_e	q_e	q_o
q_o	q_o	q_e

This is the piece that we discussed on Page 21.

Question: How could we design a machine to accept $L_1 \cup L_2$, which would show that this latter language is also regular?

The machine for the union

As guided by the process as shown on Page 29, we can construct another machine, $M_\cup$, that accepts $L_1 \cup L_2$, as follows:

$$\begin{aligned} M_\cup &= (Q, \Sigma, \delta_\cup, q_s, F), \text{ where} \\ Q &= \{(q_1,q_e),(q_1,q_o),(q_2,q_e), \\ &\quad (q_2,q_o),(q_3,q_e),(q_3,q_o)\}, \\ \Sigma &= \{0,1\}, \\ q_s &= (q_1,q_e) \\ F &= \{(q_1,\underline{q_o}),(\underline{q_2},q_e),(\underline{q_2},\underline{q_o}),(q_3,\underline{q_o})\}, \end{aligned}$$

and $\delta_\cup$ is given as follows:

$\delta_\cup$	0	1
(q_1,q_e)	(q_1,q_e)	(q_2,q_o)
(q_1,q_o)	(q_1,q_o)	(q_2,q_e)
(q_2,q_e)	(q_3,q_e)	(q_2,q_o)
(q_2,q_o)	(q_3,q_o)	(q_2,q_e)
(q_3,q_e)	(q_3,q_e)	(q_2,q_o)
(q_3,q_o)	(q_2,q_o)	(q_2,q_e)

So what?

Now, given an input string, e.g., 1011, $M_\cup$ 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, $M_\cup$ terminates at (q_2, q_o), as it is in both.

Since the latter is one of the accept states of $M_\cup$, by definition, this input is accepted by M.

On the other hand, for the input 1010, $M_\cup$ 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 $(q_3, q_e) \not\in F$, 1010 $\not\in L(M_\cup)$, as it is in neither language. ☹

In general, $w \in L(M_\cup)$ iff it either contains odd 1's or it has even 0's after the last 1.

Theorem 1.25: 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 $M_\cup = (Q, \Sigma, \delta, q_0, F)$, 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. $F = \{(r_1, r_2) | r_1 \in F_1$ or $r_2 \in F_2\}$

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

Question: How could we do this?

Answer: Remember that $A = B$ iff $A \subseteq B$ and $B \subseteq A$. (Cf. Page 23 of the Mathy chapter)

$$L(M_\cup) \subseteq L(M_1) \cup L(M_2)$$

Let $w \in L(M_\cup)$. By definition (Cf. Page 17), 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. $(r_1^{i+1}, r_2^{i+1}) = \delta((r_1^i, r_2^i), w_{i+1})$, $i \in [0, n-1]$ and

3. $(r_1^n, r_2^n) \in F$, i.e., if either $r_1^F \in F_1$ or $r_2^F \in F_2$.

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

We now just split this sequence of "pairs" to two sequences, one for M_1, and the other for M_2.

Question: How to split?

I want to see...

Below is a visual demo of the math in the previous page. Notice, by definition, $(r_1^n, r_2^n) \in F$ iff $r_1^n \in F_1$ or $r_2^n \in F_2$.

$$\boxed{L(M) \subseteq L(M_1) \cup L(M_2)} \qquad \begin{pmatrix} p \\ q \end{pmatrix} \equiv (p, q)$$

Let $\boxed{w \in L(M)}$, $w = w_1 w_2 \cdots w_n$

$$\begin{pmatrix} q_1 \\ q_2 \end{pmatrix} = \begin{pmatrix} r_1^0 \\ r_2^0 \end{pmatrix} \xrightarrow{w_1} \begin{pmatrix} r_1^1 \\ r_2^1 \end{pmatrix} \xrightarrow{w_2} \begin{pmatrix} r_1^2 \\ r_2^2 \end{pmatrix} \xrightarrow{w_3} \cdots \xrightarrow{w_{n-1}} \begin{pmatrix} r_1^{n-1} \\ r_2^{n-1} \end{pmatrix} \xrightarrow{w_n} \boxed{\begin{pmatrix} r_1^n \\ r_2^n \end{pmatrix} \in F}$$

$$\Downarrow \qquad\qquad \Updownarrow$$

$$q_1 = r_1^0 \xrightarrow{w_1} r_1^1 \xrightarrow{w_2} r_1^2 \xrightarrow{w_3} r_1^3 \cdots \xrightarrow{w_{n-1}} r_1^{n-1} \xrightarrow{w_n} r_1^n \in F_1 \quad \text{or}$$

$$q_2 = r_2^0 \xrightarrow{w_1} r_2^1 \xrightarrow{w_2} r_2^2 \xrightarrow{w_3} r_2^3 \cdots \xrightarrow{w_{n-1}} r_2^{n-1} \xrightarrow{w_n} r_2^n \in F_2$$

$$\therefore \boxed{w \in L(M_1) \cup L(M_2)}$$

How to say *it*, mathematically?

Formally, this splitting process is this: the above sequence of "pairs" is equivalent to the following two parallel sequences: for $r_1^0, r_1^1, \ldots, r_1^n \in Q_1$

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

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

By definition, for any such a $w \in L(M_\cup)$ $(= L(M_1) \cup L(M_2))$, $w \in L(M_1)$ or $w \in L(M_2)$, namely, $w \in L(M_1) \cup L(M_2)$.

As a result, $L(M_\cup) \subseteq L(M_1) \cup L(M_2)$.

$L(M_1) \cup L(M_2) \subseteq L(M_\cup)$

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

Without loss of generality, we assume that $w \in L(M_1)$. By definition, there exists a sequence of states $r_1^0, r_1^1, \ldots, r_1^n$ in Q_1 such that

1. $r_1^0 = q_1$,
2. $r_1^{i+1} = \delta(r_1^i, w_{i+1})$, for $i \in [0, n-1]$ and
3. $r_1^n \in F_1$;

We now construct another sequence (r_1^0, r_2^0) $(r_1^1, r_2^1), \ldots, (r_1^n, r_2^n)$ such that $r_2^0 = q_2$, and for all $i \in [0, n-1], r_2^{i+1} = \delta(r_2^i, w_{i+1})$.

In other words, we just "patch" another sequence associated with M_2 to this sequence associated with M_1 to come up with a sequence of "pairs". ☺

Question: How to do this "patch"?

I want to see *it*...

The whole process can again be summarized into the following: where the green part is the patch.

$$\boxed{L(M_1)\cup L(M_2)\subseteq L(M)}$$

Let $w\in L(M_1)$, $w=w_1w_2\cdots w_n$

$$\boxed{\begin{array}{l} q_1 = r_1^0 \xrightarrow{w_1} r_1^1 \xrightarrow{w_2} r_1^2 \to \cdots \xrightarrow{w_{n-1}} r_1^{n-1} \xrightarrow{w_n} \boxed{r_1^n \in F_1} \\ q_2 = r_2^0 \xrightarrow{w_1} r_2^1 \xrightarrow{w_2} r_2^2 \to \cdots \xrightarrow{w_{n-1}} r_2^{n-1} \xrightarrow{w_n} \boxed{r_2^n \overset{?}{\in} F_2} \end{array}}$$

$$\Downarrow$$

$$\boxed{\begin{pmatrix} q_0 \\ q_1 \end{pmatrix} = \begin{pmatrix} r_1^0 \\ r_2^0 \end{pmatrix} \xrightarrow{w_1} \begin{pmatrix} r_1^1 \\ r_2^1 \end{pmatrix} \xrightarrow{w_2} \begin{pmatrix} r_1^2 \\ r_2^2 \end{pmatrix} \to \cdots \begin{pmatrix} r_1^{n-1} \\ r_2^{n-1} \end{pmatrix} \xrightarrow{w_n} \boxed{\begin{pmatrix} r_1^n \\ r_2^n \end{pmatrix} \in F}}$$

$\therefore$ $\boxed{w\in L(M)}$

Formally speaking...

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

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

Finally, since $r_1^n \in F_1$, no matter whether $r_2^n \in F_2$ or not, as indicated with ?, by the construction of M, $(r_1^n, r_2^n) \in F$.

By definition, $w \in L(M_\cup)$, namely, $L(M_1) \cup L(M_2) \subseteq L(M_\cup)$. $\square$

We can also prove similar results for both concatenation and the star operations.

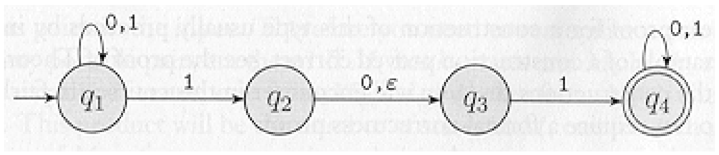
But, we have *much better* ways to do it. Let's go to Home Depot again. ☺

Non-determinism

So far in our discussion, every step of a computation in a finite automaton follows in a unique, or determined, 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 ϵ, i.e., an empty string, may exit from a state.

Question: What does N_1 do? ... Page 50.

How does an NFA compute?

Assume that an NFA runs on an input string and comes to a state with multiple ways to proceed. *This 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 ϵ is encountered, the NFA will split into multiple copies to follow each of the ϵ labeled arrow and one stay at the current state.

Then all the copies proceed in the same style by itself.

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.

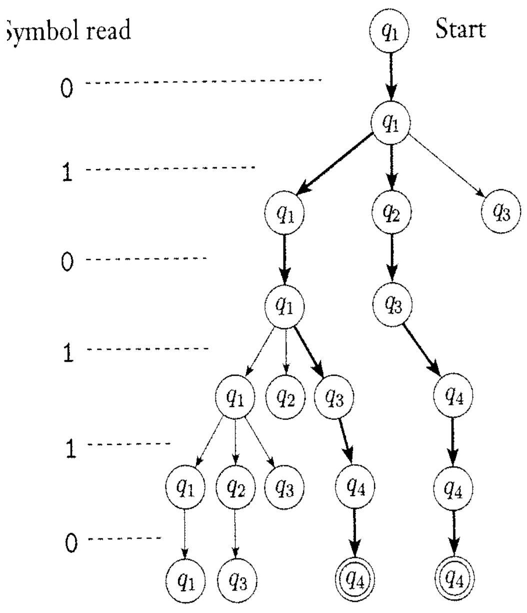
Another way to look at it is to regard it as a tree of possibilities. The root of the tree corresponds to the start of the computation.

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; ☺ *and rejects an input if none does.* ☹

An example

Below shows a computation of applying N_1 on 010110.

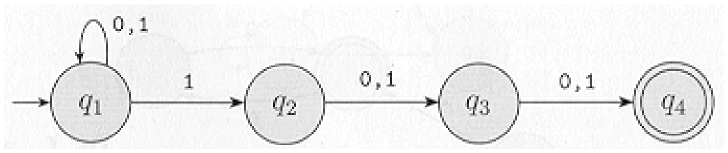


Notice that when reading in the symbol, 1, q_1 splits into three more machines and proceed with q_1 and q_2. ☹ For the ϵ, the machine splits into two more copies, one proceeds with q_3, the other stays at q_2. ☹

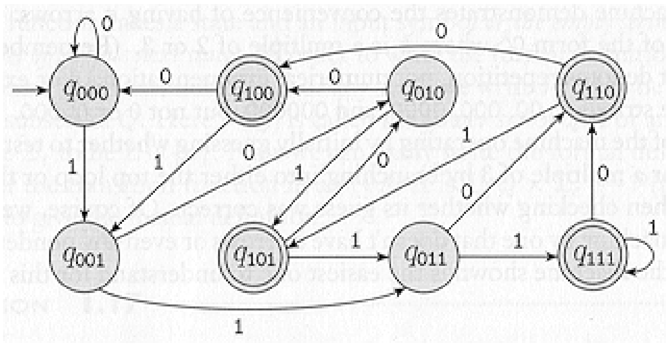
Let's play with *JFLAP* again ☺, which sends back a little different chart from the above one.

NFA or DFA?

Below shows an NFA, N_2, that accepts all the strings that contains a 1 in the third position from the end.



Here is a DFA that accepts the same language. ☹



Question: Which one do you prefer? ☺

Question: How could we get this messy DFA from this clean NFA? Just click a button... in *JFLAP*. Stay tuned for details (Page 59).

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 *for us* to understand and make, thus the choice of design; but DFA is “firmer”, thus the choice of implementation *in a computer.*

The best part is that the process of converting an NFA to a DFA is mechanical, and even a computer can do it, via, e.g., a *JFLAP* button.

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, which a computer can take over and run. ☺

Dana Scott got a *Turing award* for NFA back in 1976. Check out the article (Cf. Course page) about him on Page 27.

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 ϵ, an NFA might have no place to go, or it could to a couple of states, i.e., *a subset of the state set.* Thus, if we let Σ_ϵ denote $\Sigma \cup \{\epsilon\}$ and let $\mathcal{P}(Q)$ denote the *power set* of Q, we can define an NFA as follows.

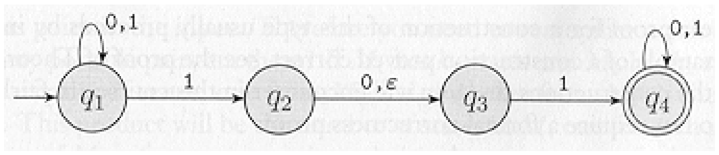
Definition: A *non-deterministic finite automaton,* NFA, is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

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

We worked with this power set stuff in Homework 0.5... and we need to do something about transition function, based on such a power set.

An example

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



$$\begin{aligned} Q &= \{q_1, q_2, q_3, q_4\}, \\ \Sigma &= \{0, 1\}, \\ F &= \{q_4\}. \end{aligned}$$

Finally, δ is given as follows:

	0	1	ϵ
q_1	$\{q_1\}$	$\{q_1, q_2, q_3\}$	$\emptyset$
q_2	$\{q_3\}$	$\emptyset$	$\{q_3\}$
q_3	$\emptyset$	$\{q_4\}$	$\emptyset$
q_4	$\{q_4\}$	$\{q_4\}$	$\emptyset$

In particular, when it sits in q_1, seeing a 1, it can stay in q_1, or go to either q_2, or even q_3 via an ϵ free ride. ☺

Computation in NFA

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

Definition: Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA and let $w = w_1 w_2 \cdots w_n$ be a string over Σ. 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. $r_n \in F$.

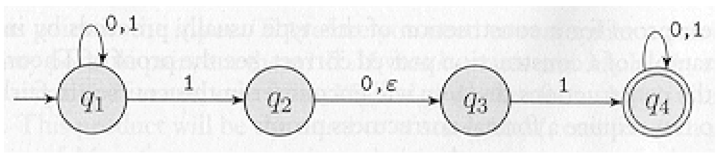
Finally, $L(N) = \{w : N \text{ accepts } w\}$.

The key difference of the above definition from that of DFA is that in condition 2, '$\in$' is used instead of '$=$' (Page 17), saying that *it only requires that* r_{i+1} *be one of the allowed successor states, but not the* only *successor state.*

This reflects the fact that, for any given state and input symbol, an NFA can have zero or many successor states.

NFA examples

Given the following N_1 again,



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

Let $M = \langle Q, \Sigma, \delta, q_1, F\rangle$ be an NFA, $L(M) = L$, and let L^R, the *reverse of* L, be $\{w^R : w \in L\}$.

Let $M_R = \langle Q \cup \{q_R\}, \Sigma, \delta_R, q_R, \{q_1\}\rangle$, we have $L(M_R) = L^R$, where

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

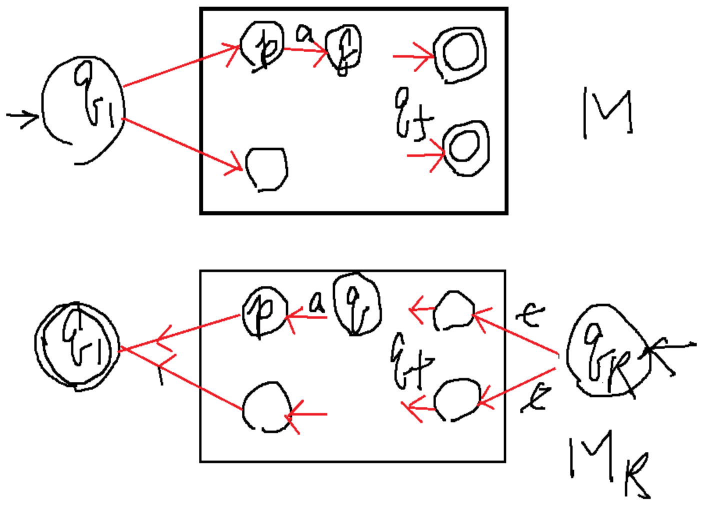
M_R is an NFA, as $\delta_R(q_R, \epsilon) = q_f \in F$.

Thus, if L is regular, so is L^R.

I just did Problem 1.31 for you. ☺

I want to see...

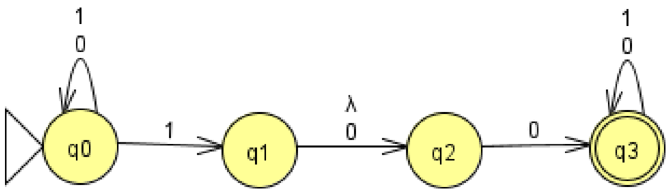
The construction of M_R from M can be summarized as follows:



Here, every transition $\delta(p, a) = q$ is flipped to $\delta(q, a) = p$. To deal with the case of multiple accepting states, we introduce a new starting state, q_R, and take a free ride to the original set of accept states.

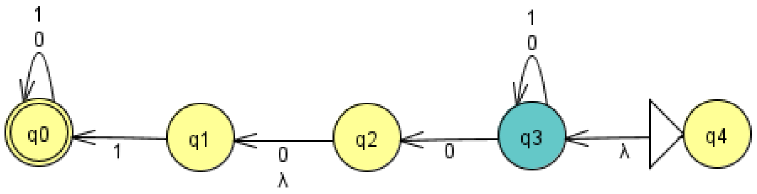
Question: Should we trust Dr. Shen?

Let's start with N_{11}...



$L(N_{11}) = \{w : w$ contains either 100 or 10$\}$.

If we apply the "reverse" construction to N_{11}, we get the following machine N_{11}^R.



$L(N_{11}^R) = \{w : w$ contains either 001 or 01$\}$.

Let's check them out with you-know-what...

Question: Why do we need L^R?

Example: add things up...

Let B be the collection of all the triplets such that the first two lines add up to the third.

For example, the following string $s \in B$

$$s = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

since

$$\begin{array}{cccccccc} & 0 & 0 & 1 & 1 & 0 & 1 \\ + & 0_0 & 0_1 & 1_1 & 1_0 & 1_0 & 0_0 \\ \hline & 0 & 1 & 1 & 0 & 1 & 1 \end{array}$$

Question: Is B regular?

Answer: It depends... on if we can come up with an FA, M_B, that accepts B.

A twist...

Because FA reads *from left to right,* what should, or could, happen is actually the following: ☹

$$\begin{array}{r@{\quad}cccccc} & 1 & 0 & 1 & 1 & 0 & 0 \\ + & 0_0 & 1_0 & 1_0 & 1_1 & 0_1 & 0_0 \\ \hline & 1 & 1 & 0 & 1 & 1 & 0 \end{array}$$

Thus, we have to design a machine M_B^R, which will accept the following string

$$s^R = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}.$$

If we could come up with M_B^R, then, $B_R = L(M_B^R)$ is regular, where B_R is a collection of all the strings that add up *from left to right.*

Then, with the reverse construction, we can build M_B, i.e., B, the collection of all the strings that add up *from right to left,* is also regular.

How to design M_B^R?

The eight possible inputs in Σ_3 play different roles: When we start in the initially *Balanced* state, 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 column to its *left* nor generates a carry to the column to its *right.* Moreover, if such an input ends the word, then it should be accepted. ☺ Hence, this *Balanced* state is both the start and the only accepted state.

Sitting in *Balanced* state, if the input column is $(0,0,1),(0,1,0),(1,0,0)$ or $(1,1,1)$, it needs to borrow a 1 from the column to its left. We have to *dump* such a string in a *Dump* state. ☹

Finally, if the input column is $(1,1,0)$, then it generates a carry to the column to its right, so we are still hopeful by moving to a *Carry* state.

Once we are in the *Dump* state, we got stuck there no matter what ☺.

If we are in the *Carry* state, i.e., we have a carry in storage, where we will go depends.... If the next input is $(0, 1, 0)$, we simply give it the carry, which gets a correct result, but generates another carry, so we stay in the *Carry* state. On the other hand, if the next input is, e.g., $(0, 0, 1)$, we give it the carry, which does not generate another carry, and it is all balanced out, going back to the *Balance state.*

If the next input is $(0, 1, 1)$, then we end up with an incorrect result, after giving it the carry. Thus, we have to dump this input by entering the *Dump* state. ☹

You *should be* able to finish M_B^R... . ☺.

Homework: This should get you started to crack Problem 1.32, with a step beyond: 1. Come up with a complete DFA to do regular addition. 2. Test *it* out with *JFlap*. (How? ☹)

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. (Dana Scott back in 1957...)

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 $a \in \Sigma$

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

Notice that for any element $x, x \in \{y\}$ iff $x = y$.

Thus, $r_{i+1} \in \delta'(q, a)$ iff $r_{i+1} \in \{\delta(q, a)\}$ iff $r_{i+1} = \delta(q, a)$.

By definition, $L(N) = L(M)$. (Cf. Pages 58 through 60 for more details.)

An example

Given the following DFA

$$\begin{aligned} M_1 &= (Q, \Sigma, \delta, q_1, F), \text{ where} \\ Q &= \{q_1, q_2, q_3\}, \Sigma = \{0, 1\}, F = \{q_2\}, \end{aligned}$$

and the transition function is given as follows:

δ	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

we can construct the associated NFA as follows:

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

and δ' is given as follows:

δ	0	1
q_1	$\{q_1\}$	$\{q_2\}$
q_2	$\{q_3\}$	$\{q_2\}$
q_3	$\{q_2\}$	$\{q_2\}$

$L(M) = L(N)$

Proof: Let $w = w_1 w_2 \cdots w_n \in \Sigma^*$, 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 $i \in [0, n-1]$, $r_{i+1} = \delta(r_i, w_{i+1})$.

Since for all $a, b, a \in \{b\}$ iff $a = b$; 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,

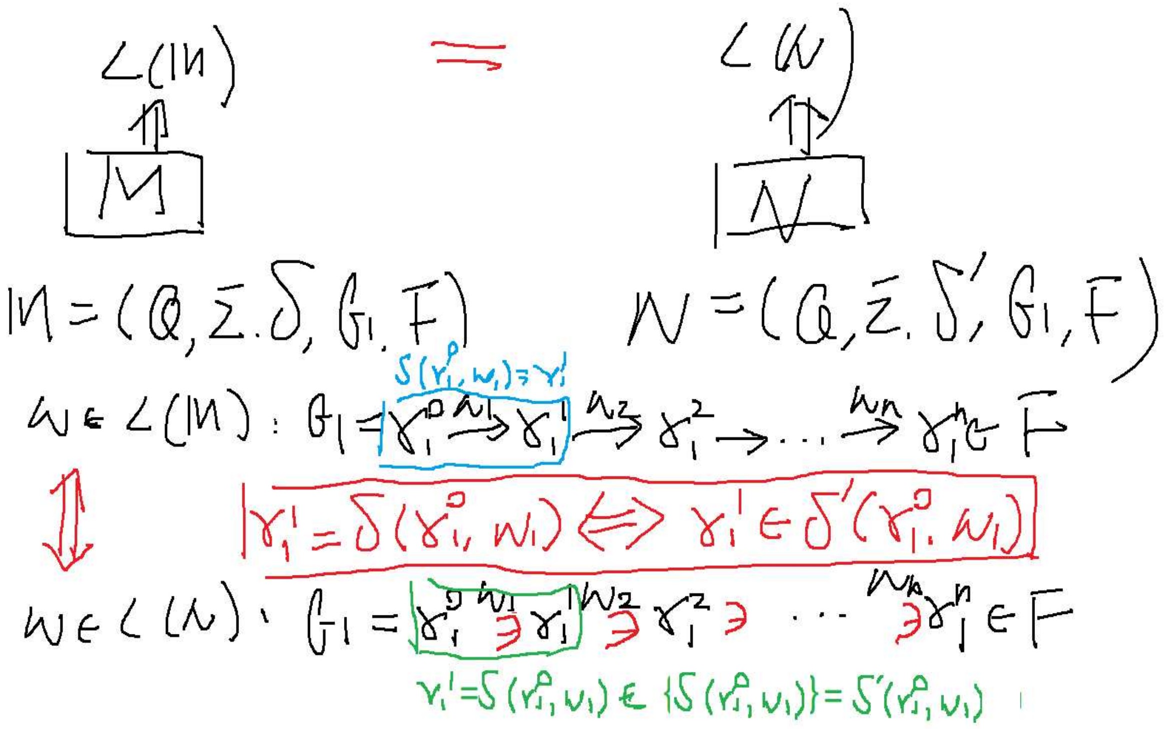
$$r_{i+1} = \delta(r_i, w_{i+1}) \text{ iff } r_{i+1} \in \delta'(r_i, w_{i+1}).$$

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, $w \in L(N)$ iff $w \in L(M)$, namely, $L(M) = L(N)$.

I want to see...

The above process can be demonstrated with the following chart:

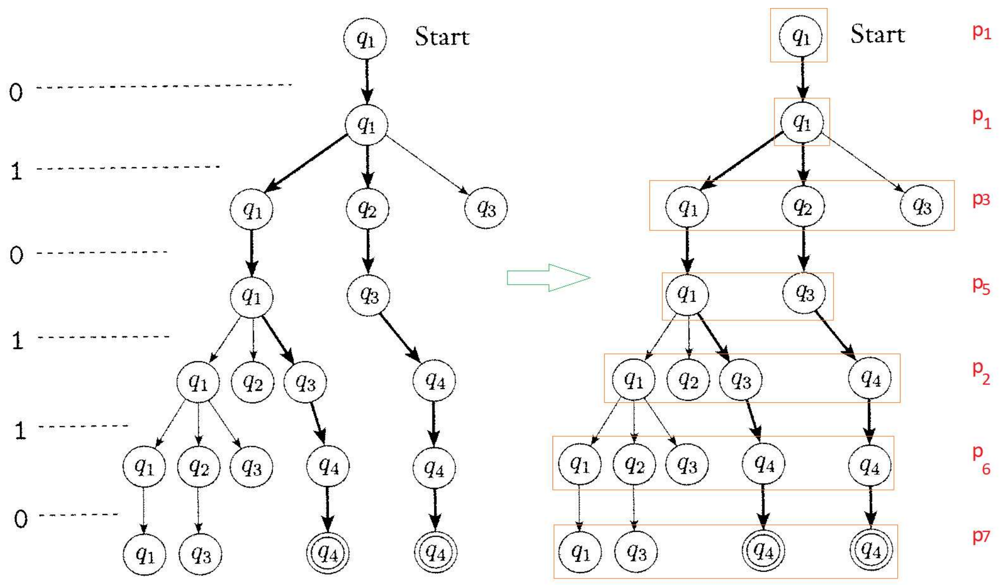

$L(M) = L(N)$
M
N
$M = (Q, \Sigma, \delta, q_1, F)$
$N = (Q, \Sigma, \delta', q_1, F)$
$\delta(r_1^0, w_1) = r_1^1$
$w \in L(M): q_1 = r_1^0 \xrightarrow{w_1} r_1^1 \xrightarrow{w_2} r_1^2 \rightarrow \cdots \xrightarrow{w_n} r_1^n \in F$
$r_1^1 = \delta(r_1^0, w_1) \Leftrightarrow r_1^1 \in \delta'(r_1^0, w_1)$
$w \in L(N): q_1 = r_1^0 \overset{w_1}{\ni} r_1^1 \overset{w_2}{\ni} r_1^2 \ni \cdots \overset{w_n}{\ni} r_1^n \in F$
$r_1^1 = \delta(r_1^0, w_1) \in \{\delta(r_1^0, w_1)\} = \delta'(r_1^0, w_1)$

Question: Given an NFA, N, how do we come up with a DFA, M, such that $L(M) = L(N)$?

This part is much more challenging, as it is "tough" to go from "≥ 0" to "$= 1$". ☹

I want to see... first

Given the NFA computation on the left, the associated DFA computation looks like the right part: given a state and a symbol, it goes uniquely to another.



We start with $p_1 = \{q_1\}$ on the DFA side.

If a DFA computation ends up in p_7, there is a way in NFA to get into q_4, an accept state of this NFA, thus, p_7 is an accept state of the DFA.

How to go from here to there?

Let δ and δ' be the transition function of the NFA and DFA, respectively.

Let $p_1 = \{q_1\}$, since

$$\delta(q_1, 0) = q_1, \text{ clearly, } \delta'(p_1, 0) = p_1.$$

Let $p_3 = \{q_1, q_2, q_3\}$,

$$\delta'(p_1, 1) = p_3.$$

Since $\delta(q_1, 1) = \{q_1, q_2, q_3\}, \text{ and } \delta(q_3, 1) = \{q_4\}$,

Let $p_5 = \{q_1, q_3\}, p_2 = \{q_1, q_2, q_3, q_4\}$,

$$\begin{aligned} \delta'(p_5, 1) &= \bigcup_{r \in p_5} \delta(r, 1) \\ &= \delta(q_1, 1) \cup \delta(q_3, 1) \\ &= \{q_1, q_2, q_3, q_4\} \\ &= p_2. \end{aligned}$$

Similarly, $\delta'(p_2, 1) = p_6$, and $\delta'(p_6, 0) = p_7$.

Let’s talk a little...

Theorem 1.39. 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, q, in the NFA and any input, a, the successor, $\delta(q, a)$, is *a subset of states* of the original NFA. (Cf. Page 27 in Dana’s article)

Thus, let Q be the state set of an NFA, we use $\mathcal{P}(Q)$, the power set of Q, as the state set of the associated DFA.

Notice that, if the NFA has k states, the state set of the associated DFA could contain up to 2^k states. ☹ (Dana called it “exponential”.)

That’s why $N_{1,1}^R$ looks much bigger than $N_{1,1}$. (Page 52)

When to start and accept?

We use $\{q_0\}$ $(\subset Q)$ as our *start state* for the DFA, *for now* ☺.

Σ, the alphabet, should stay the same.

As for an NFA, if any copy of its configuration leads to an accept state, the input is accepted by this machine. Thus, if at the end of the input, the corresponding DFA arrives at a state, a subset of the original NFA states, and it contains at least an original accept state, it would mean at least one NFA configuration accepts the input. Thus. the DFA accepts this input.

In other words, *any subset of the NFA states containing at least one accepting state in this NFA* is an *accepting state* of the DFA. ☺

We now have to specify the transition function for the DFA. ☹

A constructive 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 $Q' = \mathcal{P}(Q)$, the power set of Q, $q'_0 = \{q_0\}$ and $F' = \{R \in Q' | R \cap F \neq \emptyset\} = \{R | \exists r \in R, r \in F\}$.

The only thing left is to specify δ'. ☺

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,

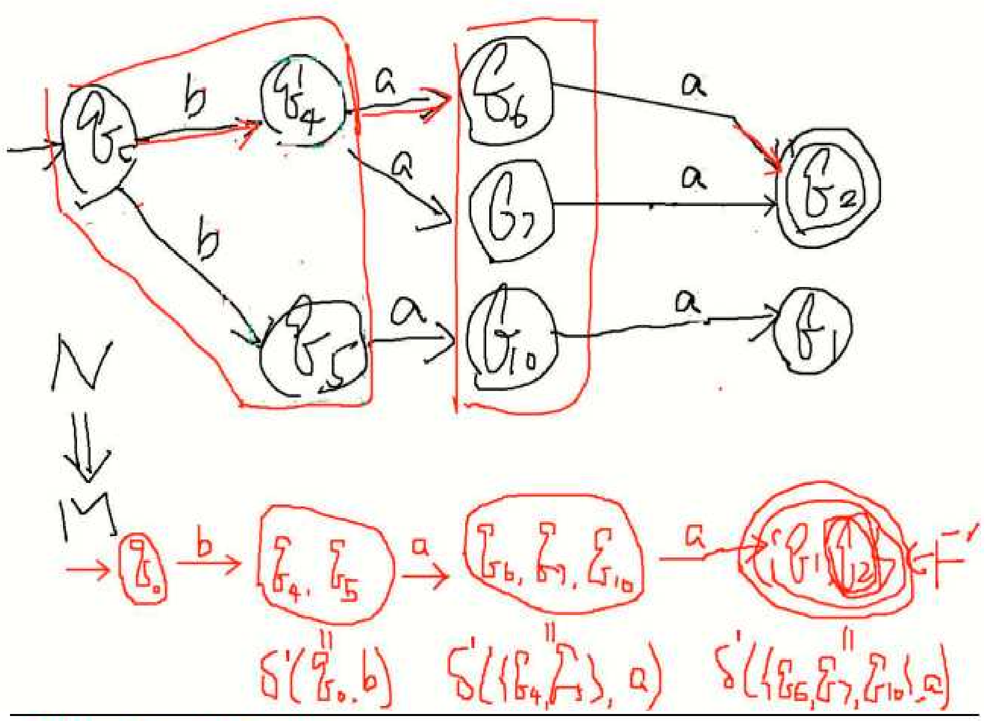
$$\delta'(R, a) = \bigcup_{r \in R} \delta(r, a).$$

An equivalent way to express δ' is as follows.

$$\delta'(R, a) = \{q \in Q | \exists r \in R, q \in \delta(r, a)\}$$

I want to see ...

The above process can be demonstrated as follows:



$\delta'(q_0, b) = \delta(q_0, b)$

$\delta'(\{q_4, q_5\}, a) = \delta(q_4, a) \cup \delta(q_5, a)$

$\delta'(\{q_6, q_7, q_{10}\}, a) = \delta(q_6, a) \cup \delta(q_7, a) \cup \delta(q_{10}, a)$

Yet another issue

We also have to consider the empty transitions, those indicated with ϵ's, part of the NFA saga.

For every $R \in Q'$, We use $E(R)$, "equivalent to R", to denote the collection of those states that can be reached through 0 or more ϵ arrows from R.

From DFA's point of view, those states should be regarded the same, as no symbol is needed to go from one sate to another.

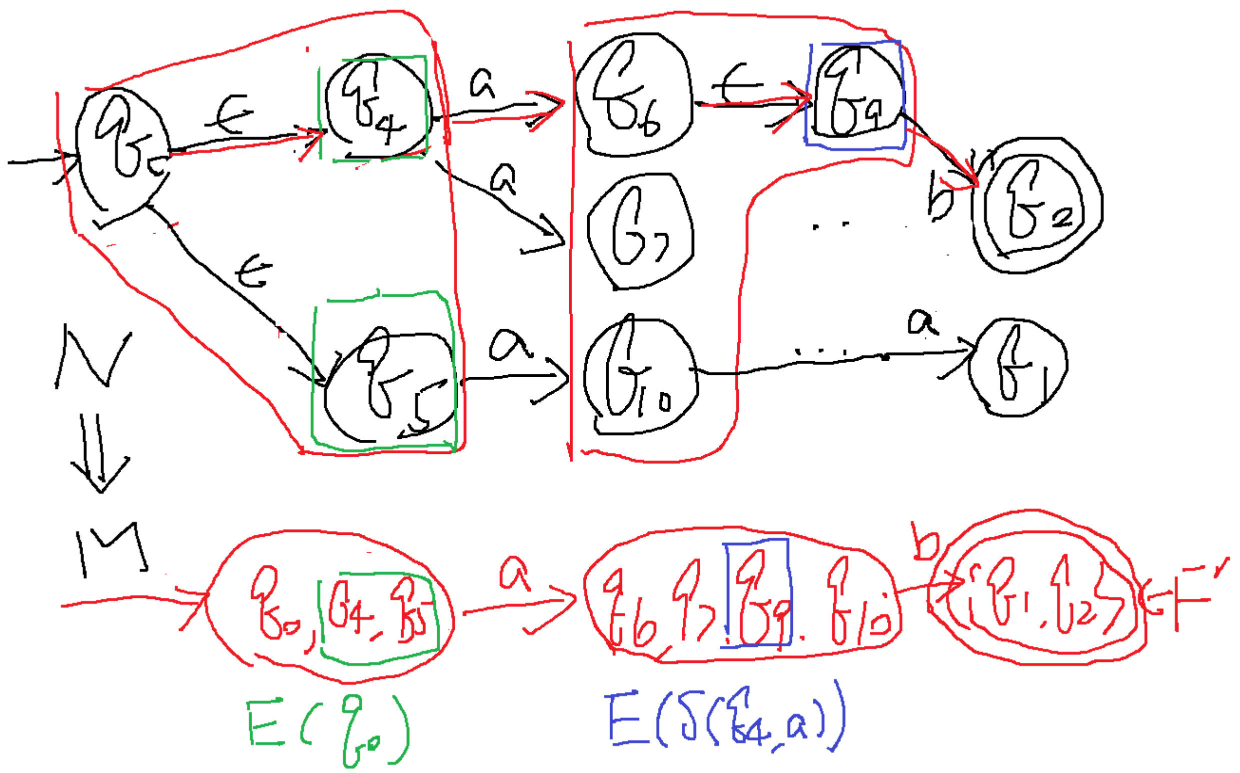
Thus, we replace the definition of δ' with the following one.

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

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

I want to see ...

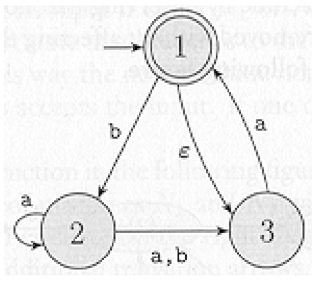
The above process can be demonstrated as follows:



The stuff in green and blue indicate this 'ϵ' effect.

How about an example?

Given the following $N_4 (= (Q, \{a, b\}, \delta, q_0, F))$, an NFA, we show how to convert it into a DFA, $N_4D = (Q', \{a, b\}, \delta', q'_0, F')$.



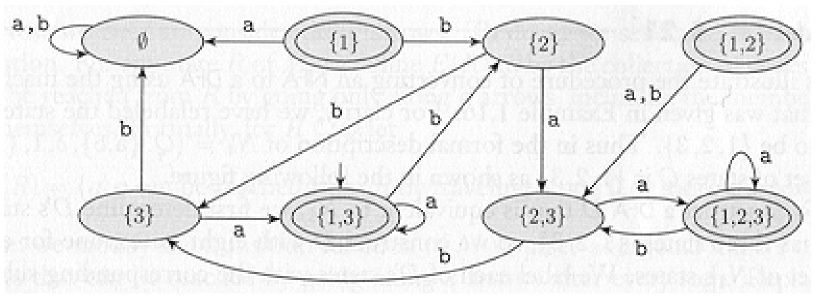
To begin with, we have that $Q' = \{\emptyset, \{1\}, \{2\}, \{3\}$ $\{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$.

As the start state of N_4 is 1, we have that $q'_0 = E(1) = \{1, 3\}$.

Since 1 is the only accept state in N_3, we have that $F' = \{\{1\}, \{1, 2\}, \{1, 3\}, \{1, 2, 3\}\}$, i.e., all the subsets containing 1.

If we end up with any of these four states, the input will be accepted. ☺

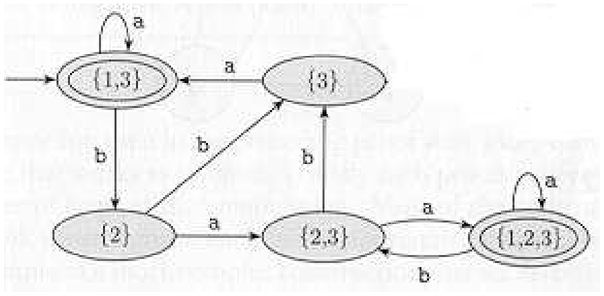
We now decide δ', which is given as follows.



For example,

$$\delta'(\{1,3\},a) = E(\delta(1,a)) \cup E(\delta(3,a)) = \{1,3\}.$$

Since nothing gets into the states $\{1\}$ and $\{1,2\}$, which are thus *unreachable;* also the state $\emptyset$ goes no where, we get the following simpler one.



Assignment: Let's play with *JFLAP* to see what it gets us. ☺

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 $A_1 \Box A_2$ is still a regular language, where $\Box$ 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 $L(M_i) = A_i$.

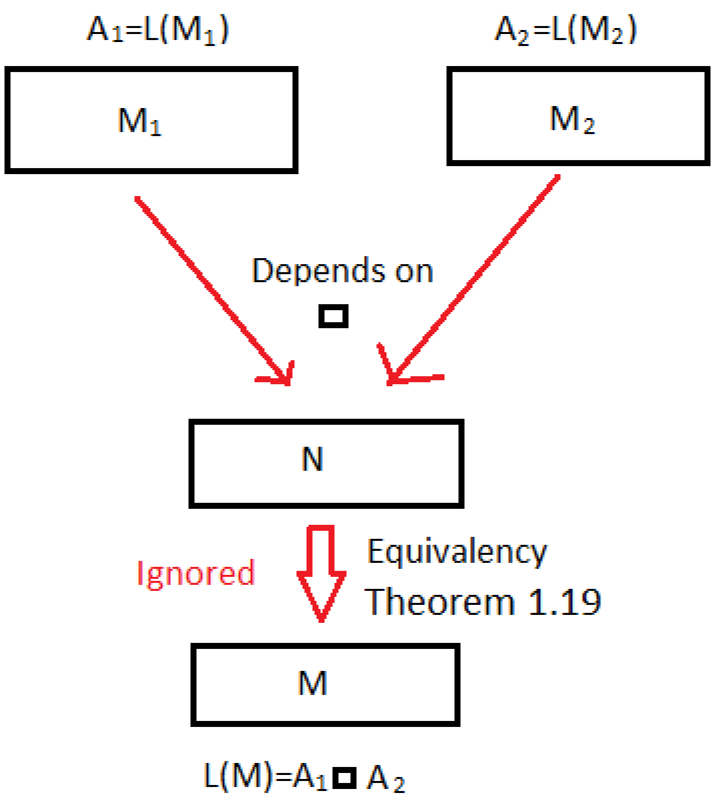
We construct an NFA, N, such that $L(N) = A_1 \Box A_2$. Because of the equivalence result between DFA and NFA, there must exist a DFA, M, such that $L(M) = L(N) = A_1 \, \Box A_2$.

Hence, by definition, $A_1 \Box A_2$ is regular. The strategy to show that A^* is regular is the same.

This makes the divide and conquer approach a reality. ☺

I want to see ...

Here is the process:



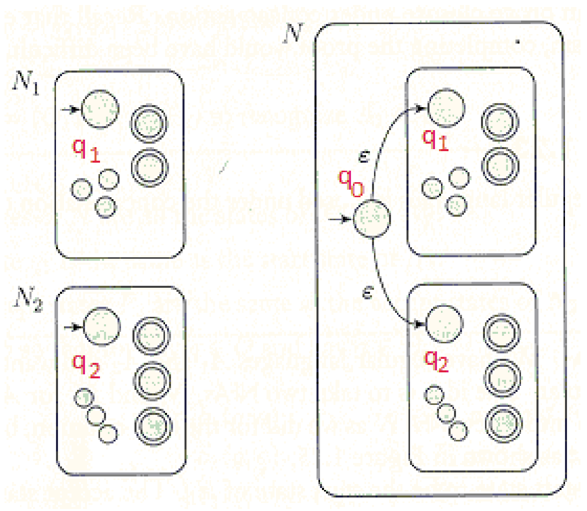
The key here is that we can ignore the technical details of coming up with this DFA M equivalent to N, as we know for sure that it exists, and how it can be constructed earlier.

Theorem 1.45: The class of regular languages is closed under the union operation.

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.

We thus provide a new start state, q_0, for N which branches to the two start states of N_1 and N_2 via an ϵ 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=$(Q, \Sigma, \delta, q_0, F)$, to accept $A_1 \cup A_2$.

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

$$\begin{aligned} Q &= Q_1 \cup Q_2 \cup \{q_0\}, \\ F &= F_1 \cup F_2. \end{aligned}$$

Next, we define the transition function.

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

The DFA construction can now follow... .

Question: Compare with the proof as given on Pages 27 through 40... . Which one is easier? ☺

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 $q_f \in F$ in this Union machine.

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 $q \in L(N)$.

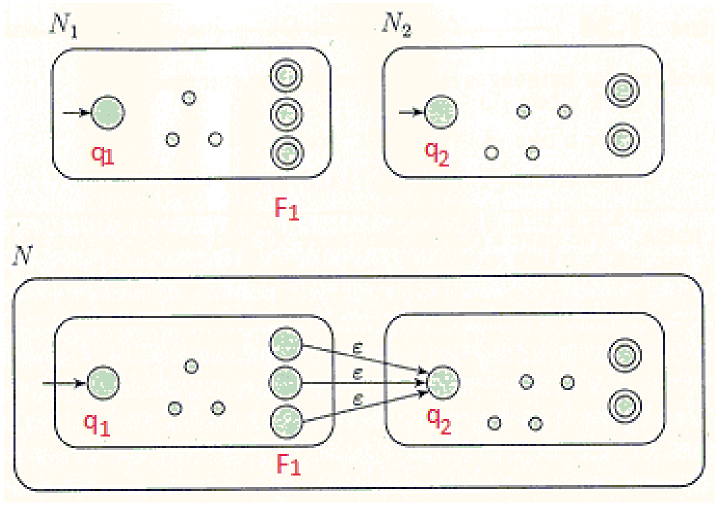
The case that $w \in L(N_2)$ can be similarly shown.

Theorem 1.47: 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 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 ϵ 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.

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

Now, the DFA construction can follow... .

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

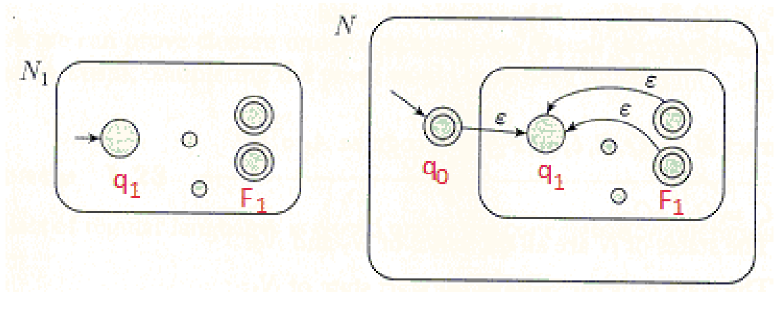
Homework: Exercise 1.9(b).

Theorem 1.49: 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 N_1.

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 N_1.

Thus, for each and every accept state of N_1, we add an ϵ arrow to connect it with the start state of N_1. As $\epsilon \in A^*$ for any A, we also have to add a new state so that the empty string will be accepted.



Question: Do we really need this new state?

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.

$$\begin{aligned} Q &= Q_1 \cup \{q_0\}, \\ F &= F_1 \cup \{q_0\}. \end{aligned}$$

Next, we define the transition function.

$$\delta(q, a) = \begin{cases} \delta_1(q, a) & q \in Q_1, q \notin F_1 \\ \delta_1(q, a) & q \in F_1, a \neq \epsilon \\ \delta_1(q, a) \cup \{q_1\} & q \in F_1, a = \epsilon \\ \{q_1\} & q = q_0, a = \epsilon \\ \emptyset & q = q_0, a \neq \epsilon. \end{cases}$$

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

Should we call Computer Science *Mathematical Engineering?*

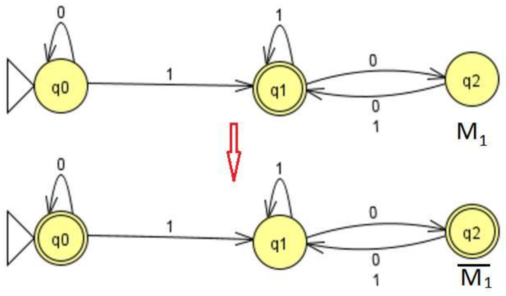
Homework: Exercise 1.10(c).

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 $\Sigma^* - L$.

For example, $L(M_1) = \{w : w$ contains at least one '1' and the last '1' is followed by an even number of 0's$\}$. (Cf. Page 12), and $L(\overline{M_1})$ is just the opposite: Any string where the last 1 is followed by an odd number of 0's.



In other words, $L(\overline{M_1}) = \Sigma^* - L(M_1)$.

I just did Exercise 14(a) for you. Let's play *it* out... .

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,*

$$\overline{L_1 \cap L_2} = \overline{L_1} \cup \overline{L_2},$$

hence

$$\begin{aligned} L_1 \cap L_2 &= \overline{\overline{L_1 \cap L_2}} = \overline{\overline{L_1} \cup \overline{L_2}} \\ &= \Sigma^* - ((\Sigma^* - L_1) \cup (\Sigma^* - L_2)). \end{aligned}$$

Since L_1 and L_2 are regular, by Result 1, both $(\Sigma^* - L_1)$ and $(\Sigma^* - L_2)$ are regular. Then, by Theorem 1.45, $(\Sigma^* - L_1) \cup (\Sigma^* - L_2)$ is also regular.

Finally, $\Sigma^* - ((\Sigma^* - L_1) \cup (\Sigma^* - L_2))$ is regular since it is closed under complementation by Result 1.

Homework: Exercise 1.14(b).

Result 3. $\emptyset$ is regular.

Proof: Let $\mathcal{M}_\emptyset$ be any DFA with its accept set being $\emptyset$, then nothing could be accepted, thus $L(\mathcal{M}_\emptyset) = \emptyset$.

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 $\{u\} = \{a_1\} \circ \{a_2\} \circ \cdots \circ \{a_n\}$. By Result 4, for each $a_i, i \in [1, n]$, there is an N_{a_i}, such that it accepts $\{a_i\}$.

Hence, by the closure property with respect to the '$\circ$' operation, Theorem 1.47, these exists an NFA that accepts $\{u\}$.

Corollary 1. Every finite subset of Σ^* is regular.

Proof: By Result 3, $\emptyset$ is regular. Let $L = \{u_1, \ldots, u_n\}$, where for all $i, u_i \in \Sigma^*$. Clearly, $L = \{u_1\} \cup \cdots \cup \{u_n\}$. By Result 5, for each $u_i, i \in [1, n]$, there is an NFA that accepts $\{u_i\}$.

By the closure property with respect to the '$\cup$' operation, Theorem 1.45, these exists an NFA that accepts $L = \{u_1, \ldots, u_n\}$.

There are just finite number of reserved words in any programming language (32 for *C* and 67 for *Java*), thus it can identify any of them via a DFA. This is one of the important applications of DFA. ☺ No programming language will work w/o *it.*

All these closure 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

This is yet another tool that could help us to design a finite automaton, besides regular operators and NFA.

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.

Regular operations can be used to construct *regular expressions,* e.g., $(0\cup 1)0^*\,(=(0\cup 1)\circ 0^*)$, which leads to a language. In this case, the collection of all strings that begins with either 0 or 1, followed zero or more 0's.

It turns out that regular expression is quite handy to represent a language. Moreover, *it is equivalent to FA's.*

When designing an FA, we represent a regular language in regular expression, then mechanically convert it into an FA, which can be further converted to a DFA, if a need arises.

Let $\Sigma = \{0, 1\}$. Then we can regard Σ as a shorthand for the expression $0 \cup 1$.

In general, if Σ is any alphabet, it describes the language consisting of all strings of length 1 and Σ^* represent the language consisting of all strings over Σ. 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. ϵ,
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: Exercise 1.13

Let's start with some examples

$$\begin{aligned}
\epsilon &= \{\epsilon\} \\
\emptyset &= \emptyset \\
0^*10^* &= \{w : w \text{ contains exactly a single 1}\} \\
\Sigma^*1\Sigma^* &= \{w : w \text{ has at least one 1}\} \\
\Sigma^*001\Sigma^* &= \{w : w \text{ contains 001}\} \\
\Sigma^* &= \{w : w \text{ is any string on } \Sigma\} \\
(\Sigma\Sigma)^* &= \{w : w \text{ is a string of even length}\}
\end{aligned}$$

Question: What languages do the following expressions represent?

$$\begin{aligned}
(\Sigma\Sigma\Sigma)^* &= ? \\
01 \cup 10 &= ?. \\
(0 \cup \epsilon)1^* &= ? \\
(0 \cup \epsilon)(1 \cup \epsilon) &= ? \\
1^*\emptyset &= ? \\
\emptyset^* &= ? \\
R \cup \emptyset &= ? \\
R \cup \epsilon &= ? \\
R \circ \emptyset &= ? \\
R \circ \epsilon &= ?
\end{aligned}$$

RA in lexical analysis

Question: What is an identifier?

$$\begin{aligned}\text{letter} &= a \cup \cdots \cup z \cup A \cup \cdots \cup Z\\ \text{digit} &= 0 \cup \cdots \cup 9\\ \text{letdig} &= \text{digit} \cup \text{letter}\\ \text{ID} &= \text{letter} \circ \text{letdig}^*\end{aligned}$$

Question: What is a signed integer?

$$\begin{aligned}\text{emptyString} &= \epsilon\\ \text{digit} &= 0 \cup 1 \cup 2 \cup \cdots \cup 9\\ \text{sign} &= +|-\\ \text{number} &= (\text{emptyString} \cup \text{digit}) \circ \text{number}\\ \text{signedInteger} &= \text{sign} \circ \text{number}\end{aligned}$$

We can then convert such a regular expression to a DFA as part of the lexical analysis process for the compiler to run through, since any such a regular expression is equivalent to an FA.

We will have a look at its application in the compilation process in the next chapter.

Regular Expression $\equiv$ FA

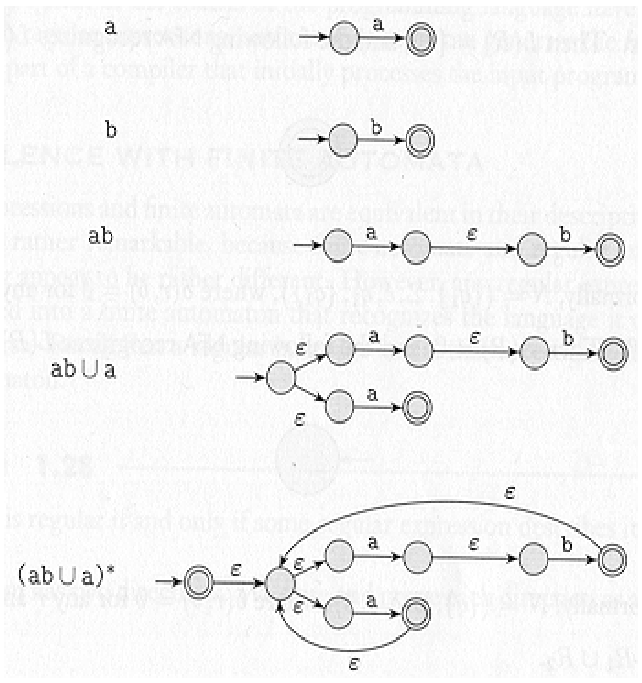
Lemma 1.55: If a language is described by a regular expression, then it is regular.

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. (Cf. Results 3 and 4) ☺

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 (Cf. Theorems 1.45 through 1.49). $\square$

An example

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



We can use , e.g., *JFlap*, to convert such an NFA mechanically into a DFA. ☺

Question: Can we use *JFlap*?

Answer: Yes, but... ☹

Homework: Exercise 1.19.

Now that we have shown any regular expression can be converted to a NFA, we will have to show the other side.

Lemma 1.60: 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 (Cf. Page 17). 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, including $\emptyset$, indicating nothing will bring *it* over, Moreover, GNFA also has some additional properties: 1) the start state goes into any other state, but no arrow comes into the start state 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 of this accept state;

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

Question: Could this be done? ☹

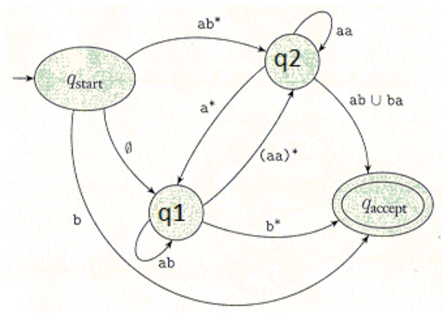
How to get a GNFA?

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 ϵ arrow;

2) adding a new accept state, to which all the old accept states are connected with ϵ 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.



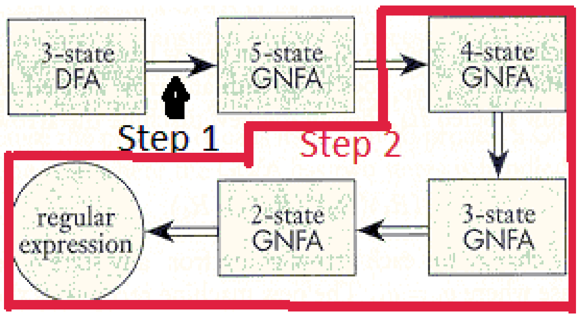
Notice that $\delta(q_{\text{start}}, q_1) = \emptyset$, when no transition exists from q_{start} to q_1.

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 *equivalent* 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.

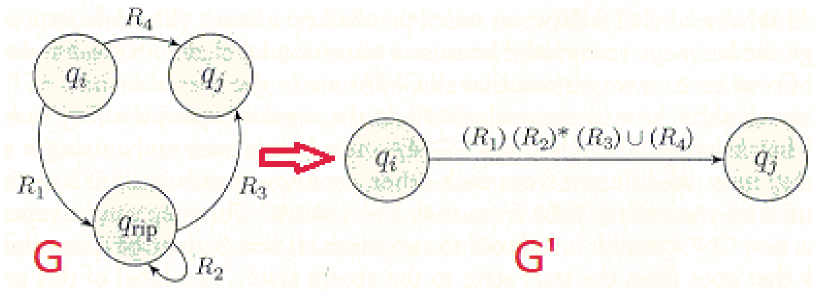


The crucial step

For a GNFA with $k > 2$ states, we select a state, q_{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.

Now, this new one has one less state, thus a right move towards the goal. ☺

This is best described by using the following figure to show how to get a three-state GNFA to a two-state GNFA.



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 $convert(G)$ returns its equivalent regular expression.

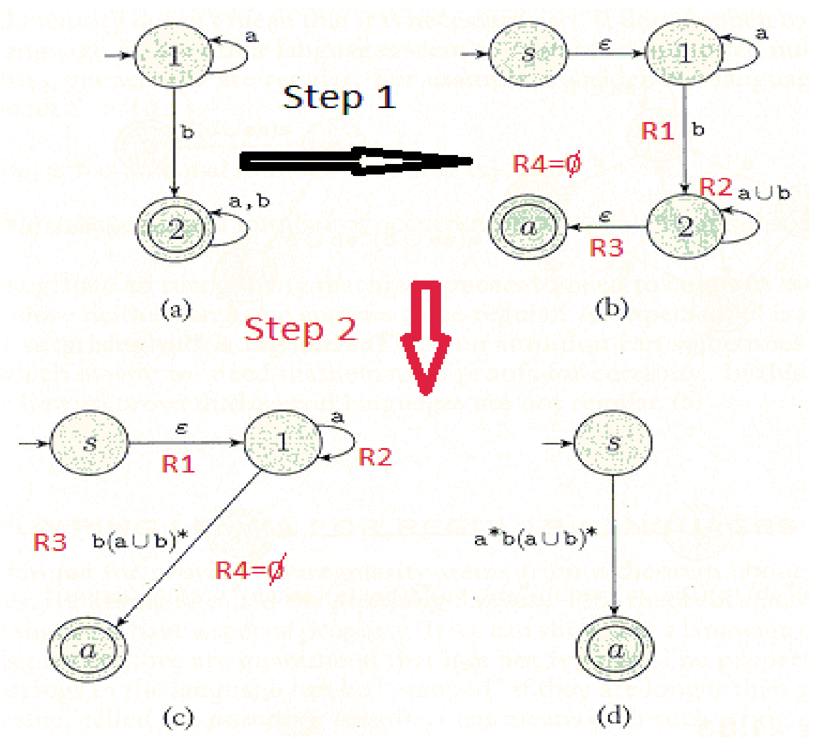
1. If $k = 2$, we are done, thus just return the label of the only arrow in G, which is the desired regular expression.

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 $\delta'(qi, qj)$ 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 $R_4 = \delta(q_i, q_j)$.

3. Return G'.

I want to see...

Below shows how to get an equivalent expression for a 2-state DFA.



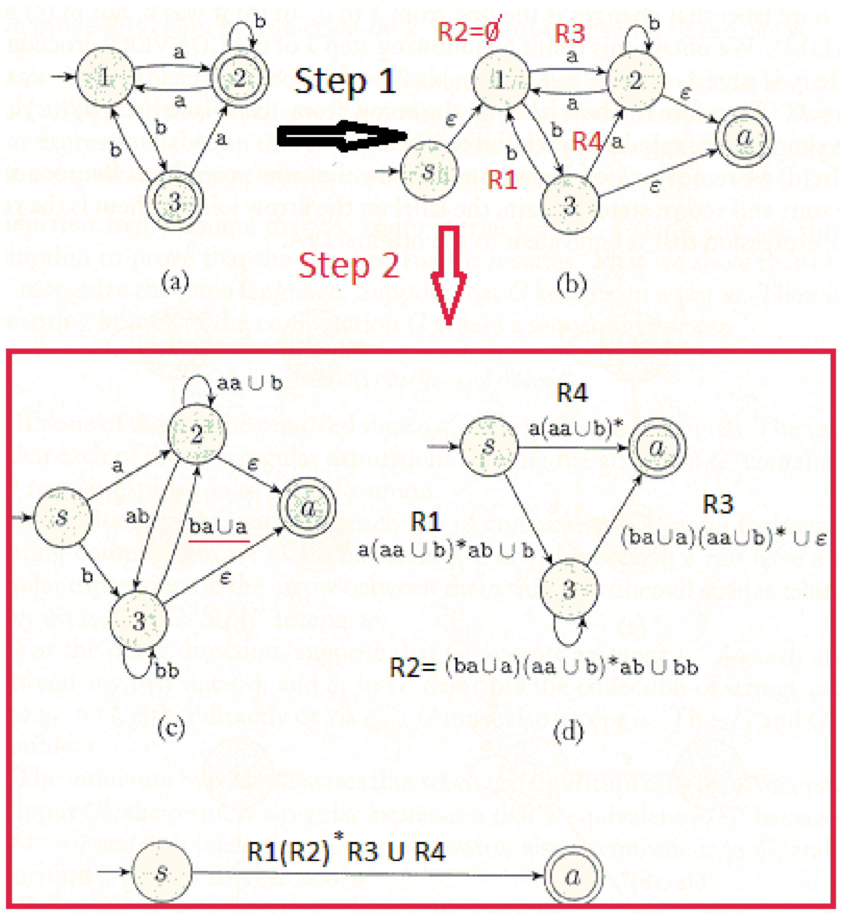
Question: Can we do it with *JFlap*?

Answer: Yes! ☺

Homework: Exercise 1.21(b).

Another example

How about a 3-state DFA, GD_2?



Question: What is *it*? ☹

Answer: $(bb)^*(a \cup ba)(a(bb)^*(a \cup ba) \cup b)^*$

Question: How did I get *it*? ☺

Theorem 1.54: A language is regular iff it is described by some regular expression.

Proof by induction on $|G|$: Let L be regular, and $L = L(M)$. Convert M to a GNFA G. If G has only two states, it has only one arrow from the start state to the accept state. The regular expression label, $RE(G)$, on that arrow specifies all the strings that allow G to get to the accept state, thus, $L(RE(G)) = L(G) = L(M) = L$.

Assume the claim holds for any GNFA with $k-1$ states and let $|G| = k$. *If we can show that* $L(G) = L(G')$, where G' is the GNFA constructed in $convert(G)$ by taking out the q_{rip} state, then by inductive assumption,

$$L(RE(convert(G'))) = L(convert(G')) = L(G)).$$

thus, it is also equivalent to $L(M)$, i.e., L.

Thus, $RE(convert(G'))$ is the regular expression that describes L. (Cf. Page 93) ☺

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

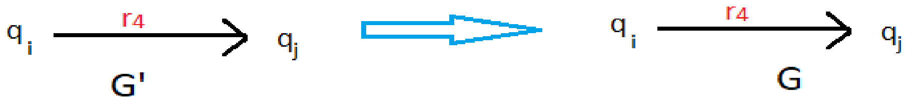
To show $L(G) \subseteq L(G')$, assume that $w \in L(G)$, i.e., G accepts w. Thus, there must be a sequence of states, $q_{\text{start}}, q_1, \ldots, q_{\text{accept}}$, accepting w in G.

If q_{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, R_4, as a union part. (Cf. Page 94)

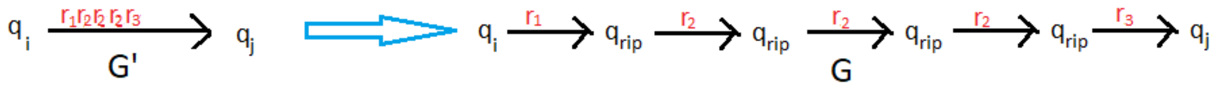
Otherwise, let $q_i \overset{r_1}{\Rightarrow} q_{\text{rip}} \overset{r_2}{\Rightarrow} q_{\text{rip}} \cdots \overset{r_2}{\Rightarrow} q_{\text{rip}} \overset{r_3}{\Rightarrow} q_j$ be a segment in the above sequence. Clearly, $r_1 \in L(R_1(= \delta(q_i, q_{\text{rip}})))$, $r_2 \in L(R_2(= \delta(q_{\text{rip}},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

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

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 $q_i \overset{r}{\Rightarrow} q_j$ be a transition in the sequence that leads to the acceptance of w in G', and let $(R_1(R_2)^*R_3) \cup R_4$ be the label of (q_i, q_j) in G', if $r \in L(R_4)$, we leave this transition there.



Otherwise, if $r \in L(R_1 \circ (R_2)^n \circ R_3)$, where R_1, R_2, and R_3 are the same as defined, we replace this transition with $q_i \overset{r_1}{\Rightarrow} q_{\text{rip}} \overset{r_2}{\Rightarrow} q_{\text{rip}} \cdots \overset{r_2}{\Rightarrow} q_{\text{rip}} \overset{r_3}{\Rightarrow} q_j$.



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

We thus have proved that a language is regular iff it is described by some regular expression.

A final example

On Pages 22 and 23, we came up with a DFA, M_{001}, that accepts all the strings containing 001 as a substring.

Let L be such a language, going through the process, we have

$$L = (1 \cup 01)^*(00)0^*1(0 \cup 1)^*$$

Let L_n be the number of such strings out all of length n, we can find out, through combinatoric analysis, that

$$L_n = 2^n - F_{n+3} + 1,$$

where F_n is the *Fibonacci* function, which grows exponentially.

Since there are 2^n words of length n, consisting of either 0 or 1, the number of such words containing no 001 as a substring is $F_{n+3} - 1$, which grows rather fast.

Indeed, only about $n/8, n \geq 3$, such strings contain 001.

Is every language regular?

By Corollary 1, we know that every finite set is regular.

From the basic fact that

$$A \rightarrow B \equiv \neg B \rightarrow \neg A,$$

we immediately know that a language that is not regular must be infinite.

But, $A \rightarrow B$ does not imply either $\neg A \rightarrow B$ or $\neg A \rightarrow \neg B$. Thus, both could be true.

We already know that an infinite language can be regular, and we have seen plenty of examples, e.g., M_1 on Page 6 and M_2 as shown on Page 13.

Question: Can an infinite language not be regular, either? If so, when?

We are going to study this question in the next few chapters.

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.

Question 1: Can an FA do everything?

For example, in *C* and all its successors, except *Python*, every '{' must be balanced out with a '}'; and, in an algebraic expression, even in *Python*, every '(' also must be balanced with a ')'. Such a requirement can be represented as a language, $B = \{a^n b^n : n \geq 0\}$ ($\overset{?}{=} (ab)^*$).

Question 2: Can we find an FA to accept B?

It seems quite difficult, as it has to remember how many 0's it has read *so far* to match the *forthcoming* 1's.

Question 3: Is B regular? If not, how can we *prove it*?

The pigeonhole principle

Recall a simple fact that, if there are ten pigeons, and nine holes, we have to put more than one pigeons into at least one hole. ☹

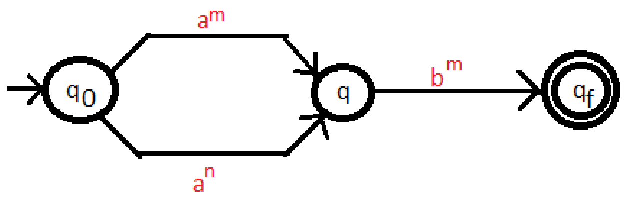
In general, *if we put n items into m boxes, and if $n > m$, then at least one box must have more than one item.* Check out Page 41 of Chapter 1 notes and course page for examples.

B is not regular...

Assume that B is regular, then for some DFA M, $L(M) = B$. Let $M = (Q, \{a, b\}, \delta, q_0, F)$, consider $\delta(q_0, a^i)$, a state where M will sit after reading i a's, $i = 1, 2, 3, \ldots$.

Since there are infinite number of such states (pigeons), but Q (holes) is finite, by the pigeonhole principle, two of these states must be the same: $\delta(q_0, a^m) = \delta(q_0, a^n) = q \in Q, m \neq n$.

Since $a^m b^m \in B$, $\delta(q, b^m) \in F$. Then
$\delta(q_0, a^n b^m) = \delta(\delta(q_0, a^n), b^m) = \delta(\delta(q_0, a^m), b^m) = \delta(q, b^m) \in F$.☹



Hence, there could not exist such a DFA M, i.e., B is not regular. ☹ Thus, *this* infinite language, B, is not regular, answering the Questions 1, 2 and 3.1, in the negative, on Page 103.

What is really going on?

We saw earlier (Cf. Page 81) that a finite language must be regular, and quite some infinite languages are regular, e.g., M_1, M_2, and $\{a^n | n \geq 0\}$. *But, it is clear that it is not the case that all the infinite languages are regular, e.g., B.*

To address Question 3.2, the key is that a finite automaton has only a finite memory, as represented with the states, which imposes a restriction on the structure of a DFA.

Intuitively, in processing a string of a regular language, the information that such an underlying DFA has to remember at any point must be limited. ☹

To accept $\{a^n b^n : n \geq 0\}$, an automaton has to distinguish all the prefixes a^m and a^n, with only a finite number of states. This cannot be done by the simple but solid reasoning under the pigeonhole principle. ☺

A bigger deal

Given a state diagram of a DFA, if it contains no cycle, its language must be finite, thus regular.

If such a state diagram contains a cycle corresponding to a *non-empty* string v, i.e., $v \neq \epsilon$ or $|v| > 0$, its language, L, must be infinite: If $w_1vw_2 \in L$, then $w_1v^nw_2 \in L, n \geq 0$.

Every infinite regular language L corresponds to a DFA with such a state diagram with a cycle.

Although we have no idea where this cycle is, but, if the DFA has m states, the cycle must be entered by the time m symbols have been read, the latest, as it needs $m + 1$ states.

If, for a language L, even for one string, any of the above rules is broken, L cannot be regular.

A special tool

Pumping lemma is a classic technique to show that a language is *not* regular. Another version of the pumping lemma for the *context-free language* will be discussed in the next chapter.

This result states that all regular languages must have certain properties. Thus, if we can show that a language, L, does not have at least one of these properties, P, it can't be regular.

Technically,

$$\forall L\ [R(L) \rightarrow P(L)] \equiv \forall L\ [\neg P(L) \rightarrow \neg R(L)].$$

Every dog barks, *and* runs. Thus, if something does not bark, it can't be a dog. ☹

Again, no matter what A and B are, the following is always true.

$$A \rightarrow B \equiv \neg B \rightarrow \neg A.$$

You cannot argue with this, no matter what language you speak. ☺

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 $i \geq 0, xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq l$.

Questions: What does it say?

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 xy $(= x \circ y)$ will be at most l. This is quite useful in proving some results.

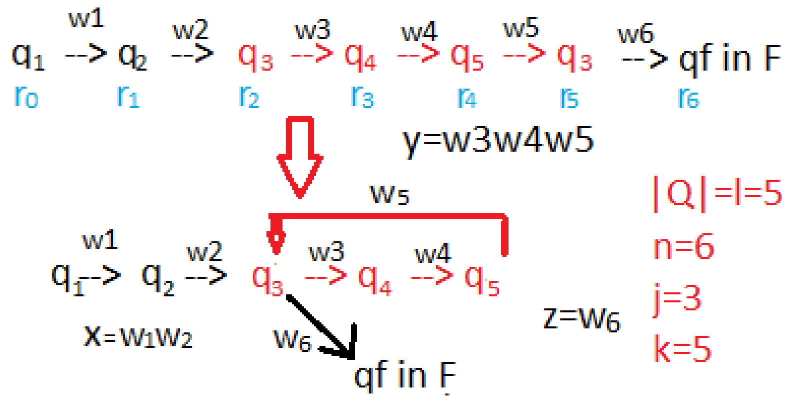
If any of these three properties is broken, A is not regular.

Question: What is this magic l?

An example

Let M be $(Q, \Sigma, \delta, q_1, \{q_f\})$, and $|Q| = 5$, and $s = w_1w_2w_3w_4w_5w_6$, and $s \in L(M)$. Then $s = xyz$, where $x = w_1w_2, y = w_3w_4w_5$ and $z = w_6$.

Here $l = |Q| = 5$. Since s contains six letters, after M scans through the first four letters, and reads the next one, w_5, it would need six states, thus *must* go back to one of the five states that it has already gone through, e.g., r_2 $(= q_3)$.


q1 w1 --> q2 w2 --> q3 w3 --> q4 w4 --> q5 w5 --> q3 w6 --> qf in F
r0 r1 r2 r3 r4 r5 r6
y=w3w4w5
w5
|Q|=l=5
n=6
j=3
k=5
q1 w1 --> q2 w2 --> q3 w3 --> q4 w4 --> q5
x=w1w2
w6
qf in F
z=w6

It is clear that $|y| = 3 > 0$, $xy^iz \in L(M)$, for all $i \geq 0$, and $|xy| = 5 \leq l$.

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.

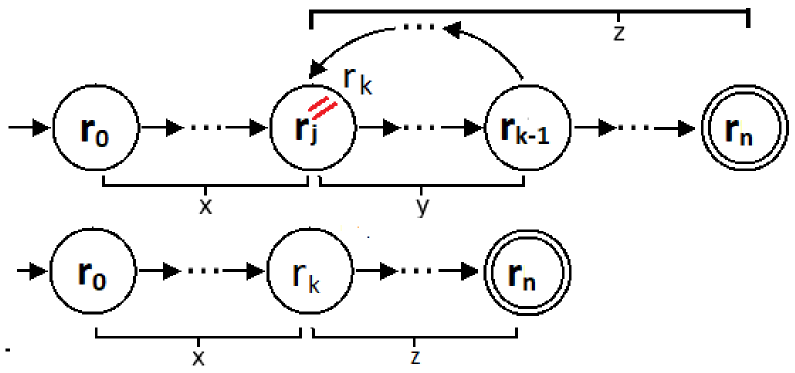
By the definition of M accepting s, as given on Page 17, as $s \in L(M) = A$, starting from r_0 $(= q_1)$, reading s_1, M gets into another state r_1; it then reads another symbol s_2 and gets into another state r_2. ..., until it reads the last one s_n, and get into an accept state r_n $(\in F)$.

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

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, $r_j = r_k$.

Now what?

Let r_j 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 r_j, in between two instances of r_j $(= r_k)$ and the part thereafter. In the example on Page 107, $j = 2$ $k = 5$ and $n = 6$; while $|Q| = l = 5$.



In other words, x brings M from r_0 $(= q_1)$ into r_j, y brings M from r_j back to r_j $(= r_k)$, and z brings M from r_k $(= r_j)$ to an accepting state r_n.

A special case is that x bring M into r_j $(= r_k)$ and then z brings it to r_n $(\in F)$, without going through the loop at all.

Why three properties?

Suppose that we run M on input $xy^iz, i \geq 0$.

As we know, x takes M into r_j, and the first occurrence of y takes M from r_j back into r_j $(= r_k)$, 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 r_j appear in the two different places of the sequence, and M is a DFA, thus $y \neq \epsilon$, the second condition holds as well.

Finally, the first $l + 1$ states in the accepting sequence must contain a repetition. If $|xy| \geq l + 1$, some state other than r_j should have already repeated itself, i.e., r_j would not be the first state that repeats itself, as assumed. Thus, the third condition also holds.

The official proof

Let $M = (Q, \Sigma, \delta, q_1, F)$ be a DFA that accepts A, $|Q| = l$, and let $s \in A, s = s_1 s_2 \cdots s_n$, $n \geq l$.

Let $r_0\ (= q_1), r_1 = \delta(r_0, s_1), \ldots, r_n\ (\in F)$ be the state sequence that M enters while reading s, so $r_{i+1} = \delta(r_i, s_{i+1}), i \in [0, n)$.

By the pigeonhole principle, among the first $l+1, [r_0, r_l]$, elements in the above state sequence, two of them must be the same. We call the fist r_j, the second r_k. Because $r_k, k \geq 0$, occurs within the first $l+1$ elements, $k \leq l$. Let $x = s_1 \cdots s_j$, $y = s_{j+1} \cdots s_k$, and $z = s_{k+1} \cdots s_n$.

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 $j \neq k, y \neq \epsilon$, i.e., $|y| > 0$. Finally, as $|xy| = k$ and $k \leq l$, we have that $|xy| \leq l$. $\square$

Check out the example on Page 107.

An example

Result: $B = \{a^n b^n : n \geq 0\}$ 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 $s = a^l b^l$. As $s \in B$ and $|s| > l$, the pumping lemma says that $s = xyz$, and for all $i \geq 0, xy^i z \in B$, as well.

1. If y contains only a, then xz has less a's than b's, thus, $xz \notin B$; and for $i \geq 2, xy^i z \notin B$.
2. The case that y contains only b is the same as case 1, which cannot actually happen (?).
3. If y contains both a and b, then $xyyz$ will not have the pattern as required by B. This case cannot happen, either (?).

Since B does not have the required property, it cannot be regular. $\square$

Question: Will *JFlap* help? ☺

Answer: Let's find it out...

More examples

Result: $C = \{w : w$ has an equal number of a's and b's$\}$ is not regular.

Proof: Let $s = a^l b^l$, the first two parts of the proof are the same as that for the previous one. For case 3, when y contains both a and b, $xyyz$ 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 a, so we come back to Case 1. $\square$

Result: $E = \{a^i b^j : i < j\}$ is not regular.

Proof: Let l be the critical length in the pumping lemma and let $s = a^l b^{l+1}$. Then, $s = xyz$, satisfying all the conditions of the lemma. By the condition 2 and 3, $|y| > 0$ and $|xy| \leq l$. Thus, y contains a, the number of a's can't be less than that of b's in xy^2z. $\square$

What does it tell us?

The pumping lemma can be used to show that a number of languages are not regular. There are more examples that you can play with *JFlap*.

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. Check out the video on the course page.

Question: Can we say anything about these languages that are not regular? ☺

Homework: Exercise 1.29.