CSE 105

Theory of Computability

Winter 2022

Taught by Professor Shachar Lovett

Table of Contents

1 Strings and Languages (Review)

Definition 1.1: Alphabet

An **alphabet** is any nonempty finite set. Generally, we use Σ and Γ to designate alphabets.

Definition 1.2: Symbols

The members of the alphabet are the **symbols** of the alphabet.

Some example of alphabets are:

$$\Sigma_1 = \{\texttt{0, 1}\}$$

$$\Sigma_2 = \{\texttt{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z}\}$$

$$\Gamma = \{\texttt{0, 1, x, y, z}\}$$

Definition 1.3: String

A **string** over an alphabet is a finite sequence of symbols from that alphabet, usually written next to one another and not separated by commas.

For example, if we use Σ_1 as our alphabet, then `01001` is a **string** over Σ_1. Likewise, if we use Σ_2 as our alphabet, then `something` is a string over Σ_2.

The set of all finite strings over Σ (any general alphabet) is denoted by Σ^*. Here, this includes:

- The empty string ϵ.
- Any **finite** combination of the symbols in this alphabet.

This does not include infinite sequences of symbols. It does have infinitely many elements.

So, for example, Σ_1^* would have strings like (and keep in mind that these are just examples):

$$\epsilon \in \Sigma_1^* \qquad \texttt{0} \in \Sigma_1^* \qquad \texttt{1} \in \Sigma_1^*$$

$$\texttt{010101} \in \Sigma_1^* \qquad \texttt{1111} \in \Sigma_1^* \qquad \texttt{0000} \in \Sigma_1^*$$

Definition 1.4: Length

If w is a string over Σ, then the **length** of w, written $|w|$, is the number of symbols that it contains.

Remarks:

- The string of length zero is called the **empty string** and is written ϵ. The empty string plays the role of 0 (like an identity) in a number system.
- If w has length n, then we can write $w = w_1 w_2 \dots w_n$ where each $w_i \in \Sigma$.

Definition 1.5: Reverse

The **reverse** of a string w, written $w^{\mathcal{R}}$, is the string obtained by writing w in the opposite order.

Remark: In other words, for a string w, we can write the reverse of w as $w^{\mathcal{R}} = w_n w_{n-1} \dots w_2 w_1$.

Definition 1.6: Substring

A string z is a **substring** of a string w if z appears consecutively within w.

For example, if we look at the string $w =$ `something`, then $z_1 =$ `some` and $z_2 =$ `thing` are both substrings of w.

Definition 1.7: Concatenation

For a string x of length m and a string y of length n, the **concatenation** of x and y, written xy, is the string obtained by appending y to the end of x, as in:

$$x_1 \dots x_m y_1 \dots y_n$$

If we want to concatenate a string x with itself many times, we use the superscript notation x^k to mean:

$$\underbrace{xx \dots x}_{k}$$

If we consider the two substrings z_1 and z_2 in the previous example, then $z_1 z_2 =$ `something`.

Definition 1.8: Prefix

A string x is a **prefix** of a string y if a string z exists where $xz = y$.

For example, if we look at the two substrings z_1 and z_2 in the previous example yet again, we say that z_1 is a prefix of w since $z_1 z_2 = w$.

Definition 1.9: Proper Prefix

A string x is a proper prefix of a string y if, in addition to x being a prefix of y, $x \neq y$.

So, `some` is a proper prefix of `something` while `something` is not a proper prefix of `something`.

Definition 1.10: Language

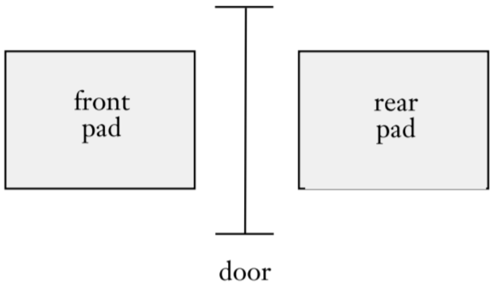
A **language** is a set of strings.

Definition 1.11: Prefix-Free

A language is **prefix-free** if no member is a proper prefix of another member.

2 Finite Automata (1.1)

Consider the controller for an automatic one-way door.



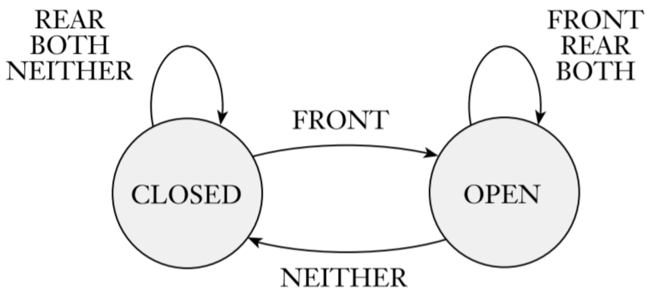
Here:

- The front pad is there is to detect the presence of a person who is about to walk through the doorway.
- The rear pad is there so that the controller can hold the door open long enough for the person to pass all the way through while also ensuring that no one behind door is hit by the door.

The controller is in either of two states: `OPEN` or `CLOSED`. This represents the condition of the door. There are also *four* possible input conditions:

- `FRONT`: A person is standing on the pad in front of the doorway (the front pad).
- `REAR`: A person is standing on the pad to the rear of the doorway (the rear pad).
- `BOTH`: People are standing on both pads.
- `NEITHER`: No one is standing on either pad.

The corresponding state diagram is:



And the corresponding transition table:

	NEITHER	FRONT	REAR	BOTH
CLOSED	CLOSED	OPEN	CLOSED	CLOSED
OPEN	CLOSED	OPEN	OPEN	OPEN

The controller moves from state to state depending on what input it receives. For example:

- When it starts off in the `CLOSED` state and receives input `NEITHER` or `REAR`, it remains in the `CLOSED` state. In the state diagram, if we start at the `CLOSED` circle (state), both `NEITHER` and `REAR` loop back to `CLOSED`.
- Again, when the controller is in the `CLOSED` state and it receives the `BOTH` input, then it stays in the `CLOSED` state because opening the door may knock someone over on the rear pad (as the door opens towards the rear side).

- If the controller is in the `OPEN` state, then receiving the inputs `FRONT`, `REAR`, or `BOTH` will result in the controller remaining `OPEN`. However, if it receives the `NEITHER` input, then it goes to a `CLOSED` state.

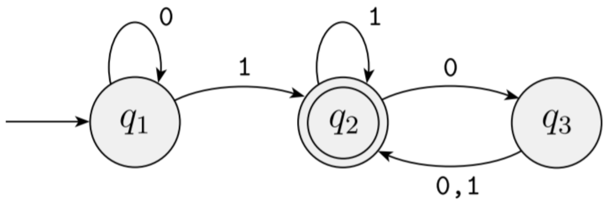
Essentially, **for the state diagram**, start at the initial state (circle) and follow the arrow depending on what input signals are received. **For the transition table**, look at the row corresponding to the initial state and the column corresponding to the input; the resulting cell will be the new state of the controller.

The figures used above (the state diagram and transition table) are both standard ways of representating a finite automaton. While this door may be very simple (due to the fact that it only really needs to store an extremely small amount of memory), in reality, we may be dealing with other devices with somewhat more memory.

Both finite automata and their probablistic counterpart **Markov chains** are useful tools when we want to attempt to recognize patterns in data.

2.1 From a Mathematical Perspective

Consider the following figure, which depicts a finite automaton called M_1:



There are a few things to note here:

- The above figure for M_1 is called the **state diagram** of M_1.
- M_1 has three **states**, labeled q_1, q_2, and q_3.
- The **start state** is the state indicated by the arrow pointing at it from nowhere. In the case of the above state diagram, this would be q_1.
- The **accept state** is the state with a double circle. In the case of the above state diagram, this would be q_2.
- The **transitions** are the arrows going from one state to another.

For a given input string, this automaton processes that string and produces an output that is either `ACCEPT` or `REJECT`. For this automaton, the processing works like so:

1. Here, the processing begins in M_1's start state.
2. Then, the automaton receives the symbols from the input string one by one from left to right.
3. After reading each symbol, M_1 moves from one state to another along the transition that has that symbol as its label.
4. When it reads the last symbol, M_1 produces its output. The output is `ACCEPT` if M_1 is now in an accept state and `REJECT` if it is not.

As an example, suppose we give M_1 the input string `1101`. Then, the processing proceeds as follows:

- Start in state q_1.
- Read `1`. Transition from q_1 to q_2.

- Read `1`. Transition from q_2 to q_2.
- Read `0`. Transition from q_2 to q_3.
- Read `1`. Transition from q_3 to q_2.
- `ACCEPT` because M_1 is in an accept state q_2 at the end of the input.

So, really, what matters is that we *end up* at the accept state.

2.2 Formal Definition of a Finite Automaton

A finite automaton has several parts.

- It has a set of states and rules for going from one state to another, depending on the input symbol.
- It has an input alphabet that indicates the allowed input symbols.
- It has a start state and a set of accept states.

We use something called a **transition function**, often denoted δ, to define the rules for moving. If the finite automaton has an arrow from a state x to a state y labeled with the input symbol `1`, that means that if the automaton is in state x when it reads a `1`, it then moves to state y. We can indicate the same thing with the transition function by saying that:

$$\delta(x, \texttt{1}) = y$$

All of this leads to the formal definition:

Definition 2.1: Finite Automaton

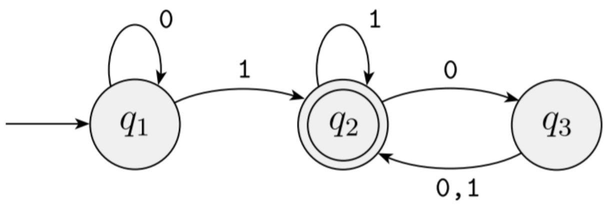
A **finite automaton** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

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**.
5. $F \subseteq Q$ is the **set of accept states** (sometimes also called *final states*).

Remark: F can be the empty set $\emptyset$, which means that there are 0 accept states.

2.3 Applying the Definition

Consider again M_1:



Using the formal definition above, we can describe M_1 formally be writing $M_1 = (Q, \Sigma, \delta, q_1, F)$, where:

- $Q = \{q_1, q_2, q_3\}$
- $\Sigma = \{\texttt{0, 1}\}$

- δ is defined as:

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

- q_1 is the start state.
- $F = \{q_2\}$.

2.4 Machine and Language

If A is the set of all strings (i.e. language) that machine M accepts, we say that A is the **language of machine** M, write $L(M) = A$, and say that M recognizes A.

A machine may accept *several strings*, but it always recognizes one language. A machine can accept no strings; in this case, it still recognizes the empty language $\emptyset$.

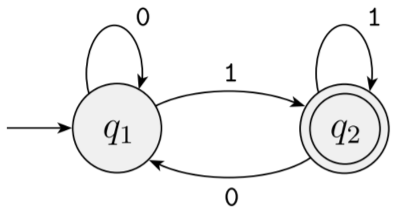
If we consider our example automaton M_1, then define:

$$A = \{w \mid w \text{ contains at least one } \texttt{1} \text{ or an even number of } \texttt{0}\text{s follow the last } \texttt{1}\}$$

Which means that $L(M_1) = A$, of equivalently, M_1 recognizes A.

2.4.1 Example 1: Simple Finite Automaton

Consider the following state diagram for the finite automaton M_2:



Here, the formal description of M_2 is as follows:

$$M_2 = (\{q_1, q_2\}, \{\texttt{0, 1}\}, \delta, q_1, \{q_2\})$$

Where δ is:

	0	1
q_1	q_1	q_2
q_2	q_1	q_2

To figure out what A is, we try a few different strings.

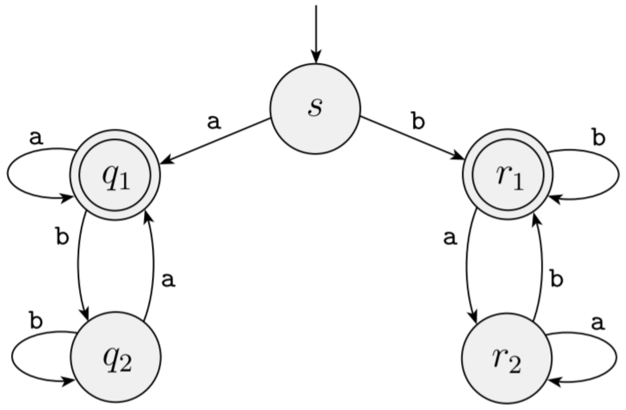
String Input	**Output**
ϵ	REJECT
1	ACCEPT
0	REJECT
01	ACCEPT
10	REJECT
11	ACCEPT

It's quite clear that A is simply the set of all strings that end with 1. So:

$$A = \{w \mid w \text{ ends with } \texttt{1}.\}$$

2.4.2 Example 2: Finite Automaton

Consider the following state diagram for the finite automaton M_3:



Here, the formal description of M_3 is as follows:

$$M_3 = (\{s, q_1, q_2, r_1, r_2\}, \{\texttt{a, b}\}, \delta, s, \{q_1, r_1\})$$

Where δ is:

	a	b
s	q_1	r_1
q_1	q_1	q_2
q_2	q_1	q_2
r_1	r_2	r_1
r_2	r_2	r_1

Here, we note that we cannot end at the start state. In other words, when we start with a, we take the left branch to q_1. In the left branch, notice how when we end with a, we will always end up at q_1, the accept state. So, it follows that a string like the one below is acceptable:

$$\texttt{a}w_2w_3 \dots w_{n-1}\texttt{a} \qquad w_i \in \{\texttt{a, b}\}$$

Likewise, if we start with b, we take the right branch to r_1. In the right branch, if our string ends with b, we will always end up at r_1. So, it follows that a string like the one below is also acceptable:

$$\texttt{b}w_2w_3 \dots w_{n-1}\texttt{b} \qquad w_i \in \{\texttt{a, b}\}$$

In other words, for this automaton, a string that starts and ends with the same symbol is accepted. That is:

$$A = \{w \mid w \text{ starts and ends with the same symbol.}\}$$

2.4.3 Example 3: Complicated Finite Automaton

Sometimes, it is hard to describe a finite automaton by state diagram. In this case, we may end up using a formal description to specify the machine. Consider the following example with the alphabet:

$$\Sigma = \{\texttt{RESET, 0, 1, 2}\}$$

Where RESET is treated as one symbol. For each $i \geq 1$, define A_i to be the language of all strings where the sum of the numbers is a multiple of i, except that the sum is reset to 0 whenever the symbol RESET appears. For each A_i, we have a finite automaton B_i which recognizes A_i. We define B_i formally like so:

$$B_i = (Q_i, \Sigma, \delta_i, q_0, \{q_0\})$$

Where $Q_i = \{q_0, q_1, q_2, \ldots, q_{i-1}\}$ and the transition function δ_i is defined so that for each j, if B_i is in q_j (i.e. B_i is in state q_j), the running sum is j modulo i. In other words, for each q_j define:

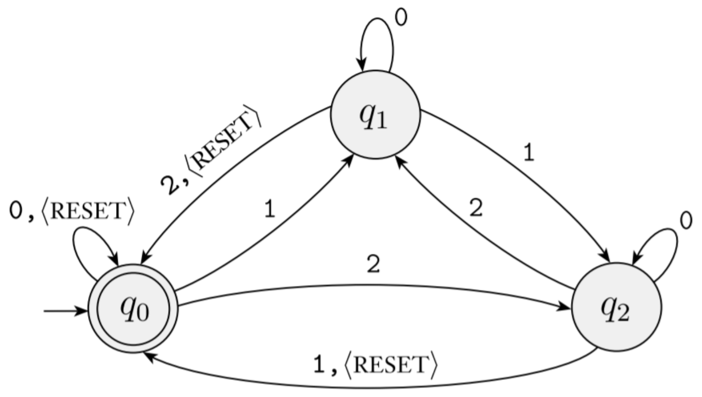
$$\delta_i(q_j, \texttt{0}) = q_j$$

$$\delta_i(q_j, \texttt{1}) = q_k \text{ where } k = j + 1 \text{ modulo } i$$

$$\delta_i(q_j, \texttt{2}) = q_k \text{ where } k = j + 2 \text{ modulo } i$$

$$\delta_i(q_j, \texttt{RESET}) = q_0$$

For example, suppose we have the following state machine B_3 which uses the same alphabet described above:



The formal description of B_3 is as follows:

$$B_3 = (\{q_0, q_1, q_2\}, \{\texttt{RESET, 0, 1, 2}\}, \delta, q_0, \{q_0\})$$

Where δ is defined by:

	RESET	0	1	2
q_0	q_0	q_0	q_1	q_2
q_1	q_0	q_1	q_2	q_0
q_2	q_0	q_2	q_0	q_1

So, as an example, let's suppose we have the string `01212`. The sum of these numbers is:

$$0 + 1 + 2 + 1 + 2 = 6 \implies 6 \equiv \boxed{0} \mod 3$$

We expect the automaton to finish at the accept state as 6 is a multiple of 3. Running through the automaton, we have:

- Input: `0`. Start at q_0, end at q_0.
- Input: `1`. Start at q_0, end at q_1. So, our automaton is at state q_1.
- Input: `2`. Start at q_1, end at q_0. So, our automaton is at state q_0.
- Input: `1`. Start at q_0, end at q_1. So, our automaton is at state q_1.
- Input: `2`. Start at q_1, end at q_0. So, our automaton is at state q_0.

Therefore, we are at an accept state as our string `01212` sums up to a multiple of 3. Of course, if there are any `RESET`s in our string, we can disregard everything up to and including the *last* `RESET` as `RESET` puts us back at the start. That is, for instance, the string `0121 RESET 21011 RESET 01212` will put the state machine in the same state as `01212`.

So, it follows that our state machine recognizes the set A_3, which consists of all strings where all digits sum up to 0 modulo 3. That is:

$$A_3 = \left\{ w \mid \sum_{\mathtt{d} \in w} d = 0 \pmod 3 \right\}$$

Note: If any `RESET`s are in the string, we can omit everything in the string *up to and including* the last `RESET`.

2.5 Formal Definition of Computation

To review, let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton and let $w = w_1 w_2 \dots w_n$ be a string where each $w_i \in \Sigma$. Then, we say that M accepts w if a sequence of states $r_0, r_1, \dots, r_n$ in Q exists with three conditions:

1. $r_0 = q_0$: The machine starts in the start state.
2. $\delta(r_i, w_{i+1}) = r_{i+1}$, for $i = 0, \dots, n-1$: The machine goes from state to state according to the transition function.
3. $r_n \in F$: The machine accepts its input if it ends up in an accept state.

In particular, we say that M recognizes language A if $A = \{w \mid M \text{ accepts } w\}$.

Definition 2.2: Regular Language

A language is called a **regular language** if some finite automaton recognizes it.

2.5.1 Example 1: State Machine

Recall, for example, our state machine B_3 in the previous lecture notes. If w was the string:

```
10 RESET 22 RESET 012
```

Then, M_5 accepts w according to the formal definition of computation because the sequence of states it enters when computing on w is:

$$q_0, q_1, q_1, q_0, q_2, q_1, q_0, q_0, q_1, q_0$$

In particular:

1. The machine starts in the start state as expected.
2. The machine goes from state to state as expected.
3. The machine ends at the accept state.

2.6 Designing Finite Automata

A helpful approach when designing various types of automata is:

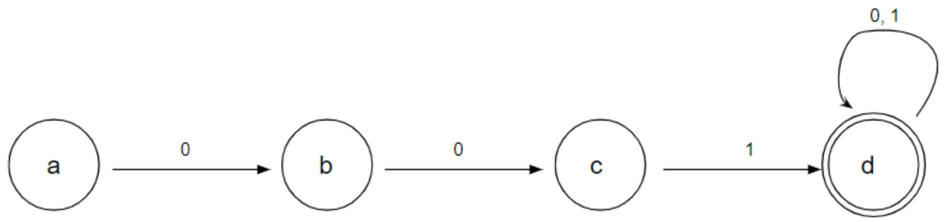
Put yourself in the place of the machine you are trying to design and then see how you would go about performing the machine's task.

Suppose you are given some language and want to design a finite automaton that recognizes it. Given some input string, your goal is to determine if it is a member of the language the automaton is supposed to recognize. However, you can only see each symbol one at a time; after each symbol, you need to decide whether the string seen is in the language. The hardest part is that you need to figure out what you need to remember about the string as you are reading it. Remember: you only have a finite number of states, which means finite memory (hence, *finite* automata).

2.6.1 Example 1: Designing a Finite Automaton

Given $\Sigma = \{0, 1\}$, suppose we need to create a finite automaton E_2 that recognizes the regular language of all strings that contain 001 as a substring. For example, 001, 1001, 0010, 111111001111101 are all in the language; however, 0000 and 11 are not.

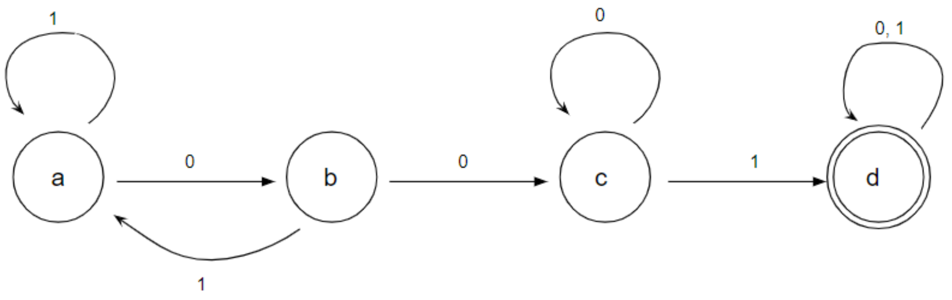
Well, the first thing we can do is create the set of states that will result in an ACCEPT state. This is as simple as:



Here, it's clear that a string like 001 will result in an ACCEPT state. Now, we need to account for any other strings. In particular, we need to account for several different possibilities:

- We haven't seen any symbols associated with the pattern (e.g. we start with 1s, or we saw a 0 and then a 1).
- We just saw 0.
- We just saw 00.
- We have seen the pattern 001.

This gives us the automaton:



2.7 The Regular Operations

In arithmetic, the basic objects are numbers and the tools are operations for manipulating them (e.g. + or ×). In the theory of computation, the objects are languages and the tools include operations designed for manipulating them. We call these **regular operations**.

Definition 2.3

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

- **Union:** $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$.
- **Concatenation:** $A \circ B = \{xy \mid x \in A \text{ and } y \in B\}$
- **Star:** $A^* = \{x_1 x_2 \ldots x_k \mid k \geq 0 \text{ and each } x_i \in A\}$

Remarks:

- The union operation simply takes all strings in both A and B and puts them together into one language.
- The concatenation operation attaches a string from A in front of a string from B in *all possible ways to get the strings in the new language.*
- The star operation attaches any number of strings in A together to get a string in the new language. Note that *any number* includes 0, so the empty string ϵ is always in A^*.

2.7.1 Example 1: String Manipulation

Suppose $\Sigma = \{\texttt{a, b, ..., z}\}$ is the standard 26 letters. Define the two languages to be:

$$A = \{\texttt{good, bad}\}$$

$$B = \{\texttt{boy, girl}\}$$

Then:

- $A \cup B = \{\texttt{good, bad, boy, girl}\}$
- $A \circ B = \{\texttt{goodboy, goodgirl, badboy, badgirl}\}$
- $A^* = \{\epsilon, \texttt{good, bad, goodgood, goodbad, badgood, badbad, goodgoodgood, ...}\}$

2.8 Justifying DFAs

To prove that the DFA that we build, M, actually recognizes the language L, we ask the following questions:

1. Is every string accepted by M in L?
2. Is every string from L accepted by M?[1]

A string is accepted by a DFA when:

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

Where δ^* is defined by:

$$\delta^*(q, w) = \begin{cases} q & w = \epsilon \\ \delta(q, c) & w = c, c \in \Sigma \\ \delta^*(\delta(q, c), w') & w \subset w', c \in \Sigma, w' \in \Sigma^* \end{cases}$$

[1]The contrapositive version is: Is every string rejected by M not in L?

2.9 Complementation of DFAs

Theorem 2.1: Complementation

If A is a regular language over $\{0,1\}^*$, then so is its complement.

Remarks:

- This is essentially the same thing as saying that the class of regular languages is closed under complementation.
- How do we apply this? Let A be a regular language. Then, there is a DFA $M = (Q, \Sigma, \delta, q_0, F)$ such that $L(M) = A$. We want to build a DFA M' whose language is $\overline{A}$. Define:

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

Proposition. *M' accepts A^c.*

Proof. Because M accepts A, we define A to be:

$$A = \{w \mid M \text{ accepts } w\} = \{w \mid \delta^*(q_0, w) \in F\}$$

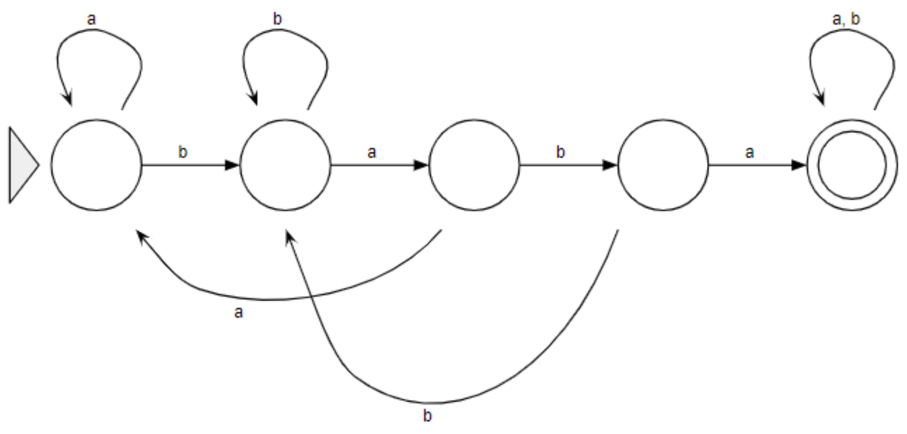
Recall that $\delta^*(q, w)$ is the state reached from q after reading the word w. Taking the complement of A, we have:

$$A^c = \{w \mid w \notin A\} = \{w \mid \delta^*(q_0, w) \notin F\} = \{w \mid \delta^*(q_0, w) \in Q \setminus F\}$$

So, M' accepts A^c. □

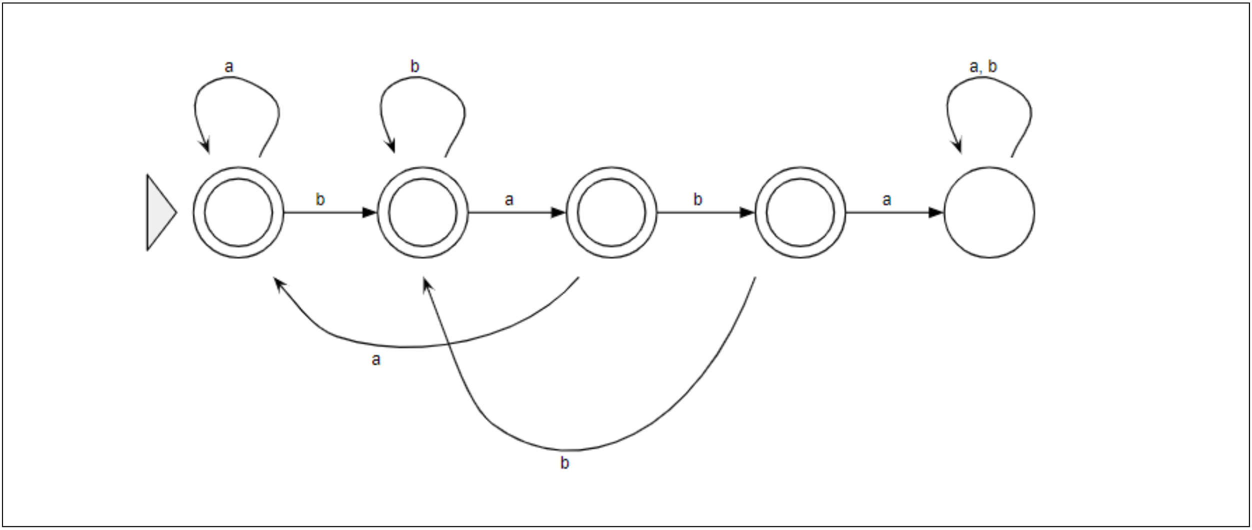
2.9.1 Example 1: Building DFA

Construct a DFA that recognizes $\{w \mid w \text{ contains the substring baba}\}$.



2.9.2 Example 2: Building DFA

Construct a DFA that recognizes $\{w \mid w \text{ doesn't contain the substring baba}\}$.



2.10 Regular Operations

In arithmetic, the basic objects are numbers and the tools are operations for manipulating them (e.g. $+$ or $\times$). In the theory of computation, the objects are languages and the tools include operations designed for manipulating them. We call these **regular operations**.

Definition 2.4

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

- **Union:** $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$.
- **Concatenation:** $A \circ B = \{xy \mid x \in A \text{ and } y \in B\}$
- **(Kleene) Star:** $A^* = \{x_1 x_2 \dots x_k \mid k \geq 0 \text{ and each } x_i \in A\}$

Remarks:

- The union operation simply takes all strings in both A and B and puts them together into one language.
- The concatenation operation attaches a string from A in front of a string from B in *all possible ways to get the strings in the new language.*
- The star operation attaches any number of strings in A together to get a string in the new language. Note that *any number* includes 0, so the empty string ϵ is always in A^*.

Note that we can prove the union operation today, but we cannot prove the concatenation or star operators until later.

2.10.1 Union

Theorem 2.2

The class of regular languages over a fixed alphabet Σ is closed under the union operator.

Remark: In other words, if A_1 and A_2 are regular language, so is $A_1 \cup A_2$.

Essentially, we want to show that if we have two regular languages A and B, then the union of them must also be regular. Thus, we want to show that if M_1 is the DFA for A and M_2 is the DFA for B, then there is a DFA that recognizes $A \cup B$:

- The goal is to build a DFA that recognizes $A \cup B$.
- The strategy is to use DFAs that recognize each of A and B.

A basic sketch of this proof is as follows:

Proof. We want to show that M accepts w if M_1 accepts w *or* M_2 accepts w. Let A and B be any two regular languages over Σ. Given:

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

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

We want to show that $A \cup B$ is regular. The idea is to run these two DFAs M_1 and M_2 in parallel. So, we define:

$$M = (Q_1 \times Q_2, \Sigma, \delta, (q_1, q_2), F)$$

Where, for $r \in Q_1$, $s \in Q_2$, and $x \in \Sigma$, we define:

$$\delta((r, s), x) = (\delta_1(r, x), \delta_2(s, x))$$

$$F = \{(r, s) \mid r \in F_1 \text{ or } s \in F_2\}$$

Note that it is not $\{(r, s) \mid r \in F_1 \text{ and } s \in F_2\}$ because this would be under intersection. Likewise, it is not $F_1 \times F_2$ because it is also intersection.

(And so on...) $\square$

2.10.2 Intersection

Proof. The proof is left for another day. $\square$

Theorem 2.3

The class of regular languages is closed under the concatenation operation.

Remark: In other words, if A_1 and A_2 are regular language, then so is $A_1 \circ A_2$.

How would you prove that the class of regular languages is closed under intersection? The idea is that:

$$A \cap B = (A^c \cup B^c)^c$$

We've already shown that the union is closed and so is its complement.

2.10.3 Payoff

Consider the set:

$$\{w \mid w \text{ contains neither the substrings aba nor baab}\}$$

Is this a regular set?

We know that:

$$A = \{w \mid w \text{ contains aba as a substring}\}$$

$$B = \{w \mid w \text{ contains baab as a substring}\}$$

From which we know:

$$\overline{A} \cap \overline{B} = \overline{A \cup B}$$

3 Nondeterministic Finite Automata (1.2)

In a deterministic finite automata, when the machine was in a given state and reads the next input symbol, we knew that the next state is; that's why it's called *determinstic*, because it was already determined. **However**, in a *nondeterministic* machine, several choices may exist for the next state at any point. In general, nondeterminism is a *generalization* of determinism; that is, every deterministic finite automaton is automatically a nondeterminism finite automaton.

3.1 The Differences Between DFA and NFA

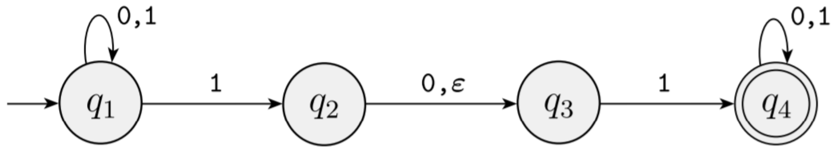


Figure: The nondeterministic finite automaton N_1.

DFA	**NFA**
• Every state of a DFA always has exactly one exiting transition arrow for each symbol in the alphabet. • There is a unique computation path for each input. • Labels on the transition arrows are symbols from the alphabet.	• Not every state in an NFA needs exactly one transition arrow for each symbol. In an NFA, a state may have zero, one, or many exiting arrows for each alphabet symbol. • We may allow several (or zero) alternative computations on the same input. • NFAs may have arrows labeled with members of the alphabet or ϵ. Zero, one, or many arrows may exit from each state with the label ϵ. For example, the above figure has one transition arrow with ϵ as a label.

3.2 NFA Computation

How does an NFA compute? Suppose that we are running an NFA on an input string and come to a state with multiple ways to proceed. Suppose, in fact, that we use the NFA above: N_1. Additionally, suppose that we are at state q_1, and the next input symbol is `1`.

- After reading this symbol, the machine **splits into multiple copies of itself** and follows *all* the possibilities in *parallel.* In other words, each copy of the machine takes one of the possible ways to proceed and continues as before.
- If there are subsequent choices, the machine splits again.
- If the next input symbol doesn't appear on any of the arrows exiting the stae occupied by a copy of the machine, that copy of the machine dies.
- If any one of these copies of the machine is in an accept state at the end of the input, the NFA *accepts* the input string.

What happens when we come across a state with an ϵ symbol on an exiting arrow? Well, without reading any input, the machine splits into *multiple* copies, one following each of the exiting ϵ-labeled arrows and one staying at the current state. The machine, then, proceeds nondeterministically as before. So, really, ϵ transitions allow the machine to **transition between states spontaneously** without consuming any input symbols.

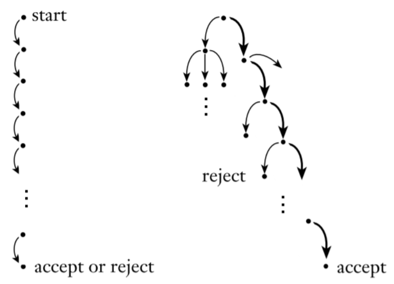


Figure: Difference between deterministic computation and nondeterministic computation.

We can see nondeterminism as some kind of parallel computation, where multiple independent "threads" or "processes" can be started concurrently. Whenever the NFA splits to follow several choices, that corresponds to a process "forking" into several children, of which each proceeds separately. Also, if one of the processes accepts, the entire computation accepts.

3.3 Formal Definition of NFA

The formal definition of a nondeterministic finitne automaton is essentially the same as the one for a deterministic finitne automaton. The major difference, though, is the transition function. In particular:

DFA	**NFA**
The transition function takes a state and an input symbol, and produces the next state.	The transition function takes a state and an input symbol *or* the empty string, and produces the *set of possible next states.*

With this in mind, we consider the definition:

Definition 3.1: Nondeterministic Finite Automaton

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

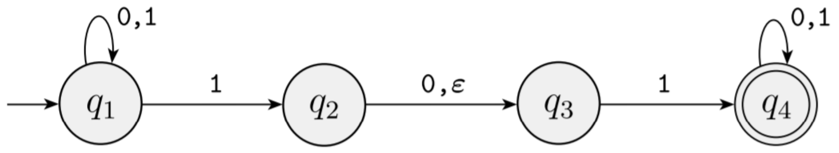
1. Q is a finite set called the **states**.
2. Σ is a finite set called the **alphabet**.
3. $\delta : Q \times \Sigma \cup \{\epsilon\} \mapsto \mathcal{P}(Q)$ is the **transition function**.
4. $q_0 \in Q$ is the **start state**.
5. $F \subseteq Q$ is the **set of accept states** (sometimes also called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

Remarks:

- $\Sigma \cup \{\epsilon\}$ is sometimes written as Σ_ϵ.
- We say that $\delta(q, c)$ returns a **set** of states; more precisely, a subset of Q. Here, $c \in \Sigma$ or ϵ and $q \in Q$.

3.3.1 Example: Starting NFA

Recall the NFA N_1:



Here, the formal description of N_1 is given by:

- $Q = \{q_1, q_2, q_3, q_4\}$
- $\Sigma = \{\texttt{0}, \texttt{1}\}$
- δ is given as:

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

- q_1 is the start state
- $F = \{q_4\}$

3.4 Acceptance in an NFA

We say that an NFA $(Q, \Sigma, \delta, q_0, F)$ accepts a string w in Σ^* if and only if we can write $w = y_1 y_2 \dots y_m$ where each $y_i \in \Sigma_\epsilon$ and there is a sequence of states $r_0, \dots, r_m \in Q$ such that:

1. $r_0 = q_0$. The machine starts in the start state.
2. $r_{i+1} \in \delta(r_i, y_{i+1})$ for each $i = 0, 1, \dots, m-1$. The state r_{i+1} is one of the allowable next states when N is in state r_i and reading y_{i+1}. Here, we note that $\delta(r_i, y_{i+1})$ is the set of allowable next states.
3. $r_m \in F$. The machine accepts its input if the last state is an accept state.

3.5 Equivalence of NFAs and DFAs

Deterministic and nondeterministic finite automata both recognize the same class of languages.

Theorem 3.1

Every nondeterministic finite automaton has an equivalent deterministic finite automaton.

Remark: Here, we say that two machines are equivalent if they recognize the same language.

The proof is as follows[2]:

Proof. Let $N = (Q, \Sigma, \delta, q_0, F)$ be the NFA that recognizes the language L. We want to show that there is a DFA $M = (Q', \Sigma, \delta', q_0', F')$ which recognizes the same L.

1. First, $Q' = \mathcal{P}(Q)$. This is because must have the states in Q' to represents the possible subset of states in Q. In an NFA, we can make multiple copies of the automaton, which may end up at different states over time. We therefore need to account for where these copies can be in our corresponding DFA.
2. The alphabet Σ is the same in both the NFA and DFA.
3. The transition function of the corresponding DFA is defined by:

$$\delta'(X, x) = \{q \in Q \mid q \in \delta(r, x) \text{ for some } r \in X \text{ or accessible via } \epsilon \text{ transitions}\}$$

Where X is a state of the DFA and $x \in \Sigma$. Because a state in an NFA can have multiple outgoing transition arrows under the same type (e.g. two outgoing arrows for `a`), we need to account for

[2]This proof was used in our submission for HW2 Problem 3 (CSE 105, WI22). The group members involved in this submission are (only initials and the last two digits of their PID are shown): CB (67), TT (96), ASRJ (73), and me.

this in the corresponding NFA. This is our first condition in our δ' function; in this sense, if we consider the possible states that we can go to in the NFA, then the corresponding state in our DFA is the union of all of those possible states. We must also consider that, for a given state in an NFA, there may be ϵ transitions. In case there are ϵ transitions, we need to consider where the ϵ transitions put a copy of the machine.

4. The start state in the corresponding DFA is the set $q_0' = \{q_0\} \cup \delta^*(q_0, \epsilon)$. First, we note that the start state in the NFA is q_0; thus, the start state in the corresponding DFA must be *at least* $\{q_0\}$. However, if there are any ϵ transitions from the start state, we must consider those as well since transitioning to another state from the state state via the ϵ transition doesn't consume any input.

5. The set of final states in the corresponding DFA is simply:

$$F' = \{X \mid X \subseteq Q \text{ and } X \cap F \neq \emptyset\}$$

Here, we're saying that if there are any sets in Q' which contain a final state in F, then said set must be a final set. This is because, in a NFA, we may have multiple copies of the machine running, and if one copy stops at a final state, then the NFA is accepted (despite the other copies not necessarily being at a final state).

The rest of the proof is omitted for now. □

3.5.1 Example: NFA to DFA

Consider the following NFA N:

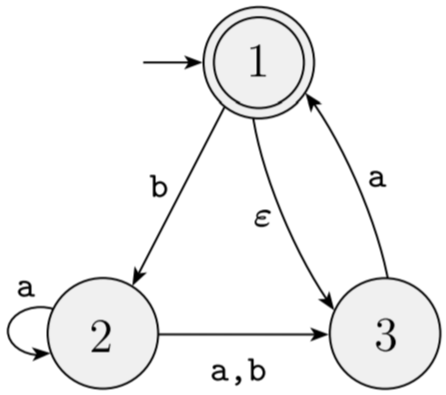


Figure: The NFA N.

We can define $N = (Q, \Sigma, \delta, q_0, F)$ like so:

- $Q = \{1, 2, 3\}$
- $\Sigma = \{a, b\}$
- δ is defined by

	a	b	ϵ
1	$\emptyset$	$\{2\}$	$\{3\}$
2	$\{2, 3\}$	$\{3\}$	$\emptyset$
3	$\{1\}$	$\emptyset$	$\emptyset$

- $q_0 = 1$
- $F = \{1\}$

We're now being asked to construct a corresponding DFA:

$$D = (Q', \Sigma', \delta', q_0', F')$$

Here, it's trivial to note that:

- $Q' = \mathcal{P}(Q) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$.
- $\Sigma = \{\mathtt{a}, \mathtt{b}\}$
- $q_0 = \{1, 3\}$. This is because we can start at both state 1 and 3 since 3 has an ϵ transition.
- $F' = \{\{1\}, \{1,2\}, \{1,3\}, \{1,2,3\}\}$. This is because we want all subsets that contain N's accept state.

The hard part is actually "wiring" the DFA up, i.e. the transition function. To do this, we need to analyze how the NFA acts and "translate" it to what the DFA would do. So, let's consider each element in Q' and see how it would relate to the NFA.

- Consider $\{1\} \in Q'$. In the NFA:
  - 1 doesn't go anywhere when a is given by itself. **However**, 1 can go to 3 since this is an ϵ transition, and 3 goes to 1 when consuming a, so it follows that $\boxed{\{1\} \xrightarrow{\mathtt{a}} \{1,3\}}$ in the corresponding DFA.
  - 1 goes to 2 when b is given, so it follows that $\boxed{\{1\} \xrightarrow{\mathtt{b}} \{2\}}$ in the corresponding DFA.
- Consider $\{2\} \in Q'$. In the NFA:
  - 2 goes to 2 *and* 3 when a is given, so it follows that $\boxed{\{2\} \xrightarrow{\mathtt{a}} \{2,3\}}$ in the corresponding DFA.
  - 2 goes to 3 when b is given, so it follows that $\boxed{\{2\} \xrightarrow{\mathtt{b}} \{3\}}$ in the corresponding DFA.
- Consider $\{3\} \in Q'$. In the NFA:
  - 3 goes to 1 when a is given, but then it can also go to 3 since there is an ϵ transition, so it follows that $\boxed{\{3\} \xrightarrow{\mathtt{a}} \{1,3\}}$.
  - 3 doesn't go anywhere when b is given, so it follows that $\boxed{\{3\} \xrightarrow{\mathtt{b}} \emptyset}$.

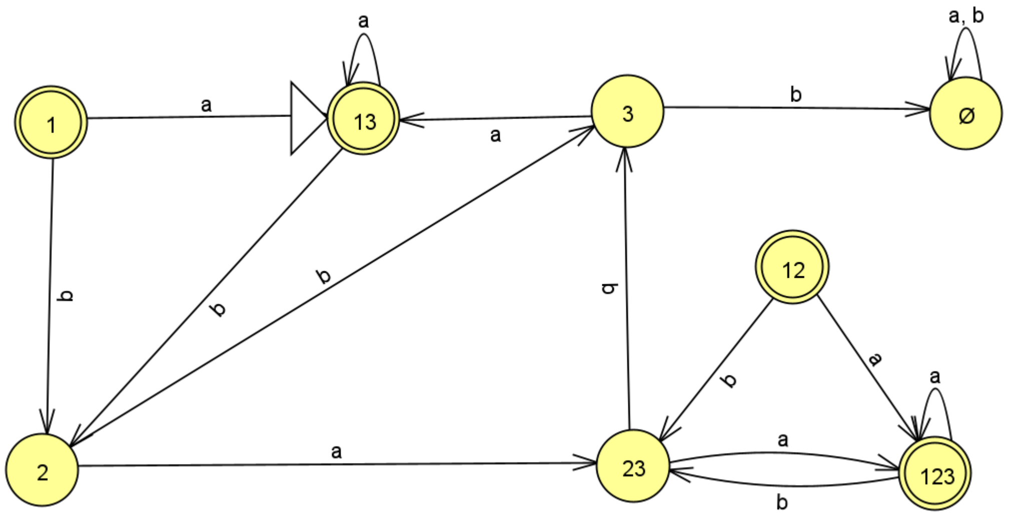
We can use the above to build cases for the remaining elements in Q'.

- Consider $\{1,2\} \in Q'$. In the corresponding NFA, this means that there's a copy at state 1 and a copy at state 2. So:
  - Suppose a is given. Then, from our previous work, we know that $\{1\} \xrightarrow{\mathtt{a}} \{1,3\}$, and $\{2\} \xrightarrow{\mathtt{a}} \{2,3\}$. Therefore, $\boxed{\{1,2\} \xrightarrow{\mathtt{a}} \{1,2,3\}}$ (recall that we take the union).
  - Suppose b is given. Then, we know that $\{1\} \xrightarrow{\mathtt{b}} \{2\}$, and $\{2\} \xrightarrow{\mathtt{b}} \{3\}$. Therefore, $\boxed{\{1,2\} \xrightarrow{\mathtt{b}} \{2,3\}}$.
- Consider $\{1,3\} \in Q'$. In the corresponding NFA, this means that there's a copy at state 1 and a copy at state 3. So:
  - Suppose a is given. Then, from our previous work, we know that $\{1\} \xrightarrow{\mathtt{a}} \{1,3\}$, and $\{3\} \xrightarrow{\mathtt{a}} \{1,3\}$. Therefore, $\boxed{\{1,3\} \xrightarrow{\mathtt{a}} \{1,3\}}$.
  - Suppose b is given. Then, we know that $\{1\} \xrightarrow{\mathtt{b}} \{2\}$, and $\{3\} \xrightarrow{\mathtt{b}} \emptyset$. Therefore, $\boxed{\{1,3\} \xrightarrow{\mathtt{b}} \{2\}}$.
- Consider $\{2,3\} \in Q'$. In the corresponding NFA, this means that there's a copy at state 2 and a copy at state 3. So:

  - Suppose `a` is given. Then, from our previous work, we know that $\{2\} \xrightarrow{\texttt{a}} \{2,3\}$, and $\{3\} \xrightarrow{\texttt{a}} \{1,3\}$. Therefore, $\boxed{\{2,3\} \xrightarrow{\texttt{a}} \{1,2,3\}}$.
  - Suppose `b` is given. Then, we know that $\{2\} \xrightarrow{\texttt{b}} \{3\}$, and $\{3\} \xrightarrow{\texttt{b}} \emptyset$. Therefore, $\boxed{\{2,3\} \xrightarrow{\texttt{b}} \{3\}}$.

- Consider $\{1,2,3\} \in Q'$. In the corresponding NFA, this means that there's a copy at state 1, 2, and 3. So:
  - Suppose `a` is given. From our previous work, we know that $\boxed{\{1,2,3\} \xrightarrow{\texttt{a}} \{1,2,3\}}$.
  - Suppose `b` is given. From our previous work, we know that $\boxed{\{1,2,3\} \xrightarrow{\texttt{b}} \{2,3\}}$.

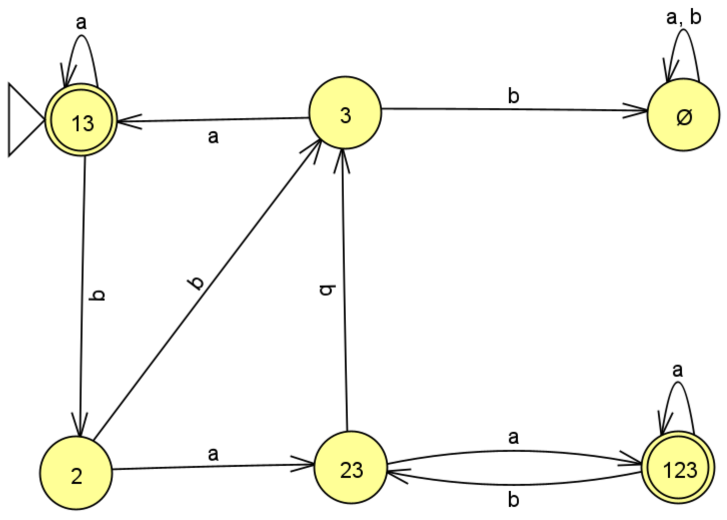
This gives us the following DFA:



However, we note a few things.

- State 1 doesn't have anything coming into it. Therefore, we can remove it.
- State 12 doesn't have anything coming into it. Therefore, we can remove it.

This gives us the simplified DFA:



3.6 Applications of Theorem

There are several applications of this theorem.

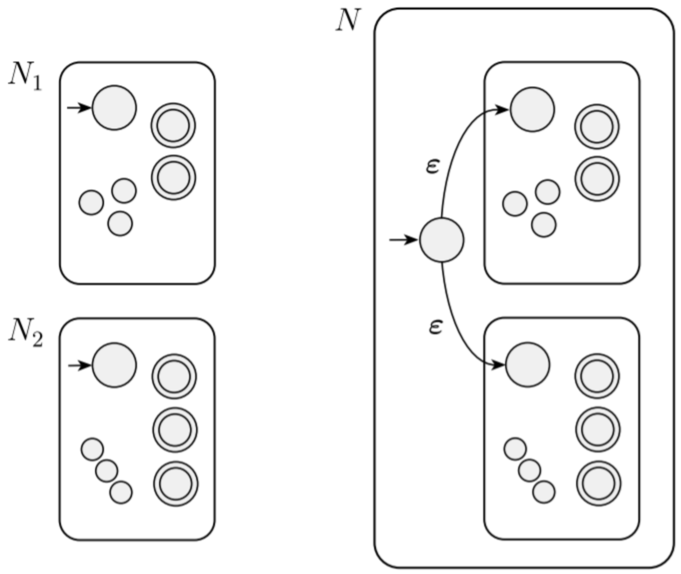
Corollary 3.1

A language is regular if and only if some nondeterministic finite automaton recognizes it.

Theorem 3.2

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

Proof. (Sketch.) Suppose A_1 and A_2 are regular languages. We want to show that $A_1 \cup A_2$ is regular. We can take two NFAs, N_1 for A_1 and N_2 for A_2, and combine them to make one new NFA N. The idea is that N must accept its input if either N_1 and N_2 accepts. So, essentially, we want to run both N_1 and N_2 in parallel. To simulate this behavior, we can create a new start state q_0 with two ϵ transitions pointing to the original start states of N_1 and N_2 (everything else about N_1 and N_2 are left unchanged). $\square$



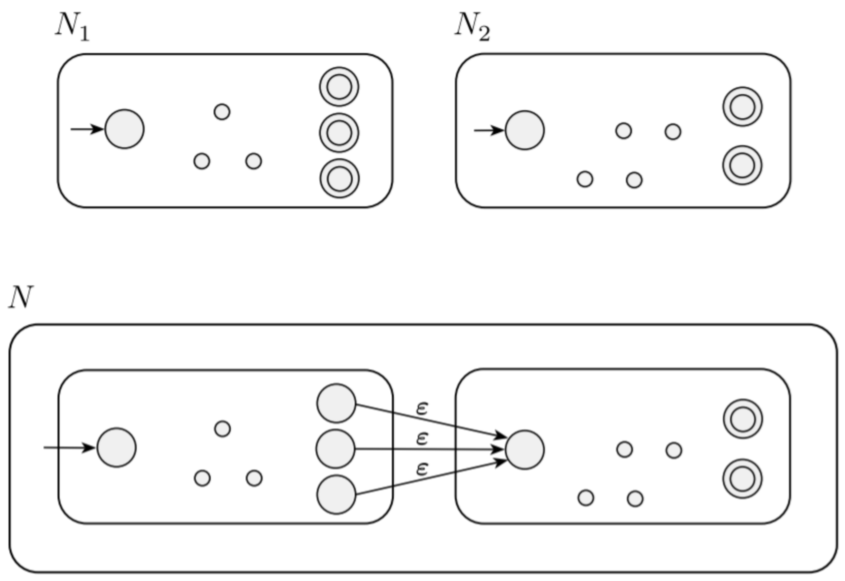
Theorem 3.3

The class of regular languages is closed under the concatenation operation.

Proof. (Sketch.) Suppose A_1 and A_2 are regular languages. We want to show that $A_1 \circ A_2$ is regular. We can take two NFAs, N_1 for A_1 and N_2 for A_2, and combine them to make one new NFA N. The idea for N is as follows:

- Start at the starting state for N_1 and remove the starting state for N_2.
- Connect each accept state in N_1 to the original start state in N_2. The accept states in N_1 will no longer be accept states.

By starting at the N_1 part of N, we guarantee that we will recognize some language A_1. Then, once we hit the original accept state in N_1, we can evaluate the rest of the string in N_2. If we hit an accept state in N_2, then we have recognized $A_1 \circ A_2$. $\square$



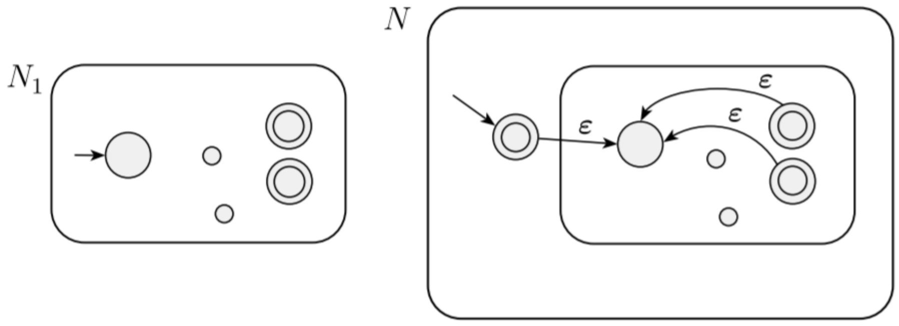
Theorem 3.4

The class of regular languages is closed under the star operation.

Proof. (Sketch.) Suppose A_1 is a regular language. We want to show that A_1^* is also regular. Consider the NFA N_1 for A_1. We want to modify N_1 so it recognizes A_1^*. Thus, our idea for the new NFA N is as follows:

- Because ϵ (the empty string) is valid under A_1^*, we must make a new start state that goes to the original start state; then, we can make the transition from the new start state to the original start state ϵ.
- We can connect the accept states in N_1 back to the original start state (not the new start state) with the labels being ϵ.
- The accept states in N_1 is the same for N.

By starting at the new start state, we can guarantee that ϵ will be accepted if it is the only thing to be read. Processing the string is as expected. However, once we reach the accept state, we need to *go back* to the original start state to process the next "word." This process keeps going until we no longer have any words to process. In this case, if we end off at any accept state with nothing left to read, then we accept. □



4 Regular Expressions (1.3)

We can use **regular expressions** (RegExp) to describe a language. An example of a regular expression is:

$$(\texttt{0} \cup \texttt{1})\texttt{0}^*$$

To give a comparison, consider the arithmetic expression:

$$(5 + 4) \times 2$$

In an arithmetic expression, the value is a number; in our case above, we would get 18. In a regular expression, the value is a **language**; in our case above, we can break the expression into multiple parts:

- $\texttt{0} \cup \texttt{1}$: This is the same thing as saying $\{\texttt{0}\} \cup \{\texttt{1}\}$, so this segment is saying that its language is $\{\texttt{0}, \texttt{1}\}$.
- $\texttt{0}^*$: This is the same thing as saying $\{\texttt{0}\}^*$, so its value is the language consisting of all strings containing any numbers of `0`s.

Putting it together, this regular expression recognizes any string which starts with `0` or `1` and ends with some number of `0`s. Just like how the multiplication sign $\times$ is often implicitly written (that is, we can write $2(5 + 4)$ instead of $2 \times (5 + 4)$), the concatenation sign $\circ$ is also implicitly written. That is, $(\texttt{0} \cup \texttt{1})\texttt{0}^*$ is the shorthand for $(\texttt{0} \cup \texttt{1}) \circ \texttt{0}^*$.

4.1 Formal Definition of a Regular Expression

Definition 4.1: Regular Expression

We say that R is a **regular expression** if R is:

1. a for some a in the alphabet Σ,
2. ϵ,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions,
6. (R_1^*), where R_1 is a regular expressions,

In items 1 and 2, the regular expressions a and ϵ represent the languages $\{a\}$ and $\{\epsilon\}$, respectively. In item 3, the regular expression $\emptyset$ represents the empty language. In items 4, 5, and 6, the expressions represent the languages obtained by taking the union or concatenation of the languages R_1 and R_2, or the star of the language R_1, respectively.

Remarks:

- Remember, ϵ and $\emptyset$ are not the same. ϵ is the same thing as $\{\epsilon\}$, i.e. the language containing only the empty string; however, $\emptyset$ represents the language that doesn't contain anything.
- In regular expressions, there is the notion of operator precedence. In our case, the star operation is done first, followed by concatenation, and finally union *unless* parentheses change the usual order.
- We may omit the $\circ$ notation for concatenation. For example, R_1R_2 is the same thing as $R_1 \circ R_2$.

Additionally, we define some more notation.

- Let R^+ be shorthand for RR^*. In other words, while R^* has all strings that are 0 or more concatenations of strings from R, the language R^+ has all strings that are **1** or more concatenations of strings from R. So, really, $R^+ \cup \epsilon = R^*$.

- We let R^k be shorthand for the concatenation of k R's with each other.

Finally, when we want to distinguish between a regular language R and the language it described, we write $L(R)$ to be the language of R.

4.1.1 Example: Regular Languages

Suppose $\epsilon = \{\texttt{0}, \texttt{1}\}$. Then, some examples of regular expressions are:

RegExp	**Examples**	**Formal Description**
$\texttt{0}^*\texttt{1}\texttt{0}^*$	1, 01, 0100	$\{w \mid w$ contains a single $\texttt{1}\}$
$\Sigma^*\texttt{1}\Sigma^*$	1, 00101101	$\{w \mid w$ has at least one $\texttt{1}\}$
$\Sigma^*\texttt{001*}$	001, 0100101	$\{w \mid w$ contains the string $\texttt{001}$ as a substring$\}$
$\texttt{1}^*(\texttt{01}^+)^*$	1010110111, 1110101	$\{w \mid$ every $\texttt{0}$ in w is followed by at least one $\texttt{1}\}$
$\underbrace{(\Sigma\Sigma\ldots\Sigma\Sigma)^*}_{n \text{ times}}$		$\{w \mid$ the length of w is a multiple of $n\}$
$\texttt{01} \cup \texttt{10}$	10, 01	$\{\texttt{01}, \texttt{10}\}$
$\texttt{0}\Sigma^*\texttt{0} \cup \texttt{1}\Sigma^*\texttt{1} \cup \texttt{0} \cup \texttt{1}$	00, 11, 10101, 0, 1	$\{w \mid w$ starts and ends with the same symbol$\}$
$(\texttt{0} \cup \epsilon)\texttt{1}^*$	11111, 01, 0111	$\texttt{01}^* \cup \texttt{1}^*$
$(\texttt{0} \cup \epsilon)(\texttt{1} \cup \epsilon)$	01, 1, 0, ϵ	$\{\epsilon, \texttt{0}, \texttt{1}, \texttt{01}\}$
$\texttt{1}^*\emptyset$		$\emptyset$
$\emptyset^*$	ϵ	$\{\epsilon\}$

Remarks:

- Concatenating the empty set to any set yields the empty set.
- The star operation on the empty set produces the set containing only the empty string.

4.2 Identities

Let R be any regular expression. The following identities hold:

1. $R \cup \emptyset = R$. Adding the empty language to any other language will not change it.
2. $R \circ \epsilon = R$. Joining the empty string to any string will not change it.

As a warning, the following do not necessarily hold:

1. $R \cup \epsilon = R$. If $R = \texttt{0}$, then $L(R) = \{\texttt{0}\}$ but $L(R \cup \epsilon) = \{\texttt{0}, \epsilon\}$
2. $R \circ \emptyset = R$. If $R = \texttt{0}$, then $L(R) = \{\texttt{0}\}$ but $L(R \circ \emptyset) = \emptyset$.

4.3 Practical Applications of RegExp

Regular expressions have practical applications. One example is in the world of compilers for programming languages. In particular, elemental objects in a programming language, called **tokens**, such as variable names and constants, can be described with regular expression. Consider the following regular expression:

$$(\texttt{+} \cup \texttt{-} \cup \epsilon)(D^+ \cup D^+\texttt{.}D^* \cup D^*\texttt{.}D^+)$$

Where $D = \{\texttt{0}, \texttt{1}, \texttt{2}, \ldots, \texttt{8}, \texttt{9}\}$. This regular expression describes a numerical constant which may include a fractional part and/or a sign. For example, the following strings are valid:

- `3.1415926`
- `+2.`
- `-.15`

After we can describe the syntax of a programming language with a regular expression in terms of its tokens, we can generate a **lexical analyzer** which processes it.

4.4 Generalized Nondeterministic Finite Automaton

We now introduce a new type of finite automaton called a **generalized nondeterministic finite automaton**, also known as a GNFA. First, we briefly introduce what a GNFA is:

- GNFAs are simply nondeterministic finite automata wherein the transition arrows may have any *regular expressions* as labels, instead of only members of the alphabet or ϵ.
- The GNFA reads *blocks of symbols* form the input, not necessarily just one symbol at a time.
- The GNFA moves along a transition arrow connecting two states by reading a block of symbols from the input, which themselves constittue a string described by the regular expression on that arrow.
- GNFAs are nondeterministic, so there may be several different ways to process the same input string.

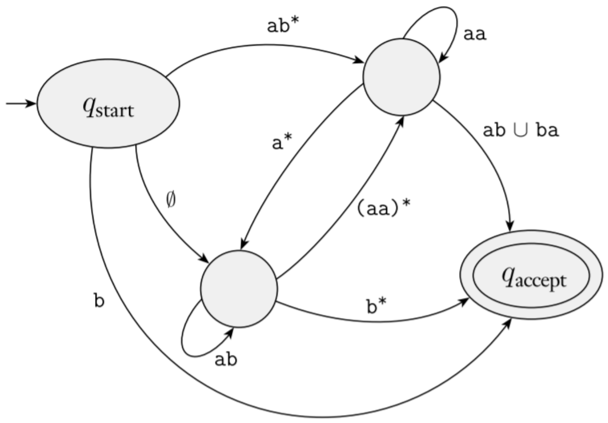


Figure: A generalized nondeterministic finite automaton.

We always require GNFAs to have a special form that meets the following conditions:

1. The start state has transition arrows going to every other state but no arrows coming in from any other state.
2. There is only a single accept state, and it has arrows coming in from every other state but no arrows going to any other state. Additionally, the start state cannot be the accept state.
3. For all other states except the start/accept states, one arrow goes from every state to every other state and also from each state to itself.

4.4.1 DFA to GNFA

To convert a DFA to a GNFA, we do the following:

- We can add a new start state with a ϵ arrow to the old start state and a new accept state with ϵ from the old accept states.
- If any arrows have multiple labels, or if there are multiple arrows going between the same two states in the same direction, replace each with a single arrow whose label is the union of the previous labels.
- Finally, add arrows labeled $\emptyset$ between states that have no arrows.

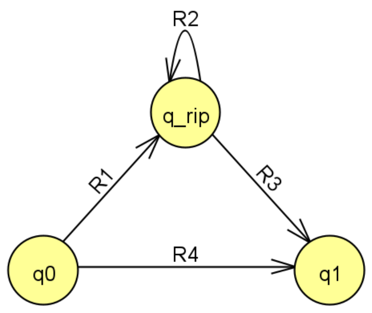
4.4.2 GNFA to Regular Expression

We now need to convert a GNFA to a regular expression. Say that a GNFA has k states. Then, because a GNFA must have a start and an accept state and they must be different from each other, we know that $k \geq 2$. If $k > 2$, we construct an equivalent GNFA with $k-1$ states. We continue to do this until the GNFA is reduced to two states. If $k = 2$, then the GNFA has a single arrow that goes from the start state to the accept state. The label of this arrow would then be the *equivalent regular expression.*

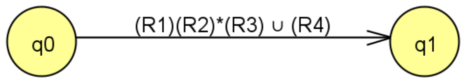
The most important step in this process is constructing an equivalent GNFA with one fewer state when $k > 2$. How can we do this? Well:

- Select a state that isn't the start or accept state, rip that state out of the machine, and then repairing what is left of the machine so the same language is still recognized. Call this state q_{rip}.
- After removing q_{rip}, we need to repair the machine by altering the regular expressions that label each of the remaining arrows. We use these new labels because they add back the lost computations (from ripping q_{rip}).

Consider the following GNFA:



If we remove q_{rip}, we get the following GNFA:



Essentially, in the old machine, if:

1. q_0 goes to q_{rip} with an arrow labeled R_1, and
2. q_{rip} goes to itself with an arrow labeled R_2, and
3. q_{rip} goes to q_1 with an arrow labeld R_3, and
4. q_0 goes to q_1 with an arrow labeled R_4

Then, in the new revised machine, the arrow from q_0 to q_1 gets the label

$$(R_1)(R_2)^*(R_3) \cup (R_4)$$

We can make this change for each arrow going from any state q_0 to any state q_1, including when $q_0 = q_1$.

4.4.3 Formal Definition

The formal definition of a GNFA is:

Definition 4.2: Generalized Nondeterministic Finite Automaton

A **generalized nondeterministic finite automaton** is a 5-tuple $(Q, \Sigma, \delta, q_{\text{start}}, q_{\text{accept}})$ where

1. Q is the finite set of tuples.
2. Σ is the input alphabet.
3. $\delta : (Q \setminus \{q_{\text{accept}}\}) \times (Q \setminus \{q_{\text{start}}\}) \mapsto \mathcal{R}$ is the transition function.
4. q_{start} is the start state.
5. q_{accept} is the accept state.

Remarks:

- Here, $\mathcal{R}$ is the collection of all regular expressions over the alphabet Σ.
- If $\delta(q_i, q_j) = R$, then the arrow from state q_i to state q_j has the regular expression R as its label.

4.4.4 Convert Algorithm

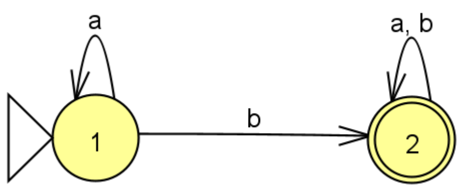
Suppose G is an GNFA. Then, the `CONVERT`(G) algorithm takes a GNFA and returns an equivalent regular expression. The algorithm works like so (Page 73):

`CONVERT`(G)

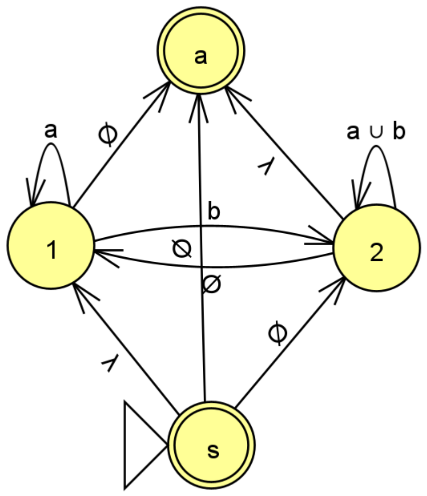
1. Let k be the number of states of G.
2. If $k = 2$, then G must consist of a start state, an accept state, and a single arrow connecting them and labeled with a regular expression R. So, return R.
3. Otherwise, $k > 2$ so we select any state $q_{\text{rip}} \in Q$ different from q_{start} and q_{accept}. Let G' be the GNFA $(Q', \Sigma, \delta', q_{\text{start}}, q_{\text{accept}})$ where $Q' = Q \setminus \{q_{\text{rip}}\}$ and, for any $q_i \in G' \setminus \{q_{\text{accept}}\}$ and $q_j \in Q' \setminus \{q_{\text{start}}\}$, let
$$\delta'(q_i, q_j) = (R_1)(R_2)^*(R_3) \cup (R_4)$$
Where $R_1 = \delta(q_i, q_{\text{rip}})$, $R_2 = \delta(q_{\text{rip}}, q_{\text{rip}})$, $R_3 = (q_{\text{rip}}, q_j)$, and $R_4 = \delta(q_i, q_j)$.
4. Compute `CONVERT`(G').

4.4.5 Example 1: DFA to Regular Expression

Suppose we wanted to convert the following DFA to a regular expression:



1. First, we need to convert this DFA to a GNFA. This would look like:



Here, we've made a few changes.

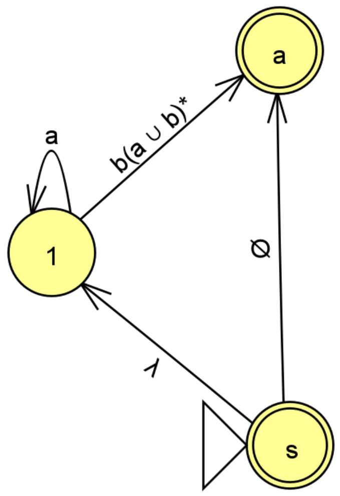
- First, we added two new states: s for the new *start* state and a for the new *accept* state. We have an arrow from s to 1 (the old start state) with ϵ as its label[3]. We also have an arrow from 2 (the old accept state) to a with ϵ as its label.
- Next, note that there was an arrow labeled a, b at state 2. We take the *union* of these two labels to get $a \cup b$. Thus, state 2 now has an arrow with $a \cup b$ instead of a, b. This is because the DFA's label represents two transitions, but a GNFA may only have a single transition going from a state to itself.
- Finally, we add several arrows with the labels being $\emptyset$:
  - 2 to 1 since every state needs to be able to transition to all non-start states.
  - 1 to a for the same reason as above.
  - s to 2 for the same reason as above.
  - s to a for the same reason as above.

2. Next, we pick one non-start/accept state as q_{rip}. We'll pick 2 for our case, so let $2 = q_{\text{rip}}$. We're going to make use of the `CONVERT` algorithm. So, we pick $q_i = 1$ and $q_j = a$. Then:

   - $\delta(q_i, q_{\text{rip}}) = R_1 = b$
   - $\delta(q_{\text{rip}}, q_{\text{rip}}) = R_2 = a \cup b$
   - $\delta(q_{\text{rip}}, q_j) = R_3 = \epsilon$
   - $\delta(q_i, q_j) = R_4 = \emptyset$

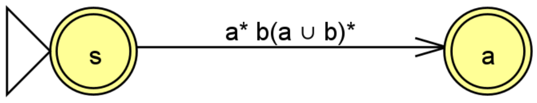
   Therefore, $\delta'(q_i, q_j) = (b)(a \cup b)^*\epsilon \cup \emptyset$. This simplifies to $\delta'(q_i, q_j) = (b)(a \cup b)^*$. So, the corresponding new state diagram is:

[3]The software used to create these state machines use λ instead of ϵ.



3. We do this process again. We pick our one non-start/accept state as $q_{\text{rip}} = 1$. By our algorithm again, let $q_i = s$ and $q_j = a$. Then:
   - $\delta(q_i, q_{\text{rip}}) = R_1 = \epsilon$
   - $\delta(q_{\text{rip}}, q_{\text{rip}}) = R_2 = a$
   - $\delta(q_{\text{rip}}, q_j) = R_3 = b(a \cup b)^*$
   - $\delta(q_i, q_j) = R_4 = \emptyset$

   Therefore, $\delta'(q_i, q_j) = (\epsilon)(a)^*b(a \cup b)^* \cup \emptyset$. This can be simplified to $\delta'(q_i, q_j) = (a)^*b(a \cup b)^*$. So, the corresponding new state diagram is:



   Thus, the regular expression corresponding to the given DFA is $\boxed{(a)^*b(a \cup b)^*}$

4.5 Regular Expressions and Regularity of Language

Theorem 4.1

A language is regular if and only if some regular expression describes it.

Proof. The proof is given by the two lemmas. □

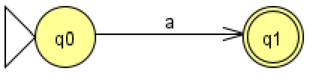
4.5.1 Regular Expression Implies Regularity

Lemma 4.1

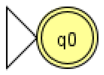
If a language is described by a regular expression, then it is regular.

Proof. Suppose we convert R into an NFA N. We then need to consider six cases as defined by the formal definition of regular expression.

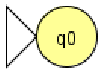
1. Let $R = a$ for some $a \in \Sigma$. Then, $L(R) = \{a\}$ and the following NFA recognizes $L(R)$:



2. Let $R = \epsilon$. Then, $L(R) = \{\epsilon\}$ and the following NFA recognizes $L(R)$:



3. Let $R = \emptyset$. Then, $L(R) = \emptyset$ and the following NFA recognizes $L(R)$:



4. $R = R_1 \cup R_2$
5. $R = R_1 \circ R_2$
6. $R = R_1^*$

Where the last three cases is given by a previous proof. □

4.5.2 Regularity Implies Regular Expression

Lemma 4.2

If a language is regular, then it is described by some regular expression.

Proof. If the language is regular, then it is accepted by a DFA. From the above, we've given a sketch of how to convert a DFA to a regular expression. □

5 Nonregular Languages (1.4)

Of course, with great power comes great responsibility. This is certainly the case with finite automata. That is, we will prove that certain languages cannot be recognized by any finite automaton. Consider the language

$$B = \{\texttt{0}^n\texttt{1}^n \mid n \geq 0\}$$

It's not possible for us to find a finite automaton that recognizes B simply because the machine needs to remember how many 0s have been seen so far as it reads the input. In other words, because the number of 0s is not limited, the machine would have to keep track of an *unlimited* number of possibilities.

Important Note 5.1

Just because the language appears to require unbounded memory doesn't mean that it is necessarily non-regular. For example, consider the two languages over $\Sigma = \{\texttt{0}, \texttt{1}\}$:

$$C = \{w \mid w \text{ has an equal number of } \texttt{0}\text{s and } \texttt{1}\text{s}\}$$

$$D = \{w \mid w \text{ has an equal number of occurrences of } \texttt{01} \text{ and } \texttt{10} \text{ as substrings}\}$$

C is not regular, but D *is* regular, despise the fact that both languages require a machine that might need to keep count.

5.1 The Pumping Lemma

We can use the concept known as the pumping lemma to prove nonregularity. In particular, this theorem states that all regular languages have a special property: the property that all strings in the language can be *pumped* if they are at least as long as a certain special value, called the **pumping length**. This means that each string contains a section that can be repeated *any number of times* with the resulting string remaining in the language.

So, if we can show that a language doesn't have this property, then it must be true that this language isn't regular.

Theorem 5.1: Pumping Lemma

If A is a regular language, then there is a number p (the *pumping length*) where if s is any string in A of length at least p, then s may be divided into three pieces, $s = xyz$, satisfying the following conditions:

1. For each $i \geq 0$, $xy^iz \in A$
2. $|y| > 0$
3. $|xy| \leq p$

General Remarks:

- The pumping lemma is used to prove that a language is not regular. It cannot be used to prove that a language is regular.

Notational Remarks:

- Recall that $|s|$ represents the length of a string s.
- y^i means that i copies of y are concatenated together.
- $y^0 = \epsilon$.
- When s is divided into xyz, either x or z may be ϵ, but $y \neq \epsilon$ by condition 2.

5.2 Using Pumping Lemma in Proofs

To prove that a language L is not regular, we use the pumping lemma like so:

1. Assume that L is regular so that the Pumping Lemma holds.
2. Let p be the pumping length for L given by the lemma.
3. Find a string $s \in L$ such that $|s| \geq p$. Your s must be parametrized by p. **Warning:** Not every string in L will work.
4. By the Pumping Lemma, there are strings x, y, z such that all three conditions hold. Pick a particular $i \geq 0$ (usually, $i = 0$ or $i = 2$ will suffice) and show that $xy^iz \notin L$, thus yielding a contradiction.

Several points to consider:

- Your proof must show that, for an arbitrary p, there is a particular string $s \in L$ (long enough) such that for any split of xyz (satisfying the conditions), there is an i such that $xy^iz \notin L$. In other words, you must:
  - Assume a general p. You **cannot** choose a particular p.
  - Find a concrete s. Your s must be parametrized by p.
  - Consider a general split x, y, z. You **cannot** choose a particular split; you must show every possible split.
  - Show a particular i for which the pumped word is not in L.
- The string s does not need to be a random, representative member of L. It may come from a *very specific* subset of L. For example, if your language is all strings with an equal number of `0`'s and `1`'s, your s might be 0^p1^p.
- Make sure your string is long enough so that the first p characters have a very limited form.
- The vast majority of proofs use $i = 0$ or $i = 2$, but there are exceptions.

5.2.1 Example 1: Pumping Lemma Application

We will show that the language B described above is not regular.

Proof. Assume to the contrary that B is regular. Then, let p be the pumping length given by the pumping lemma. Let s be the string $\texttt{0}^p\texttt{1}^p$. Because $s \in B$ and $|s| = 2p > p$, the pumping lemma guarantees that s can be split into three pieces, $s = xyz$, where for any $i \geq 0$ the string $xy^iz \in B$. We now consider three cases to show that this is impossible.

1. The string y consists of only `0`s. In this case, the string $xyyz$ has more `0`s than `1`s and so is not a member of B, violating condition 1 of the pumping lemma.
2. The string y consists of only `1`s. This also violates condition 1 of the pumping lemma.
3. The string y consists of both `0`s and `1`s. In this case, the string $xyyz$ may have the same number of `0`s and `1`s, but they will be out of order since some `1`s will come before `0`s.

Hence, a contradiction is unavoidable if we make the assumption that B is regular. Thus, B cannot be regular. □

Remark: If we applied condition 3 of the Pumping Lemma, we could have removed case 2 and 3. An alternative proof is given below.

Proof. Assume to the contrary that B is regular. Then, let p be the pumping length given by the pumping lemma. Let s be the string $\texttt{0}^p\texttt{1}^p$. Because $s \in B$ and $|s| = 2p > p$, the pumping lemma guarantees that s can be split into three pieces, $s = xyz$, where for any $i \geq 0$ the string $xy^iz \in B$. If our string looks like:

$$s = \overbrace{0000\ldots0000}^{p \text{ times}}\overbrace{1111\ldots1111}^{p \text{ times}}$$

Then, we can split the string like so:

$$s = \underbrace{000}_{x}\overbrace{0\ldots0000}^{y}\underbrace{1111\ldots1111}_{z}$$

Suppose x has length a and y has length b where $a + b \leq p$. Then, for $i = 2$, we have the string $xyyz$ where xyy has length $a + b + b > p$ while z has length p, a contradiction since we must have the same length of `0` and `1`. □