CSE 105

Theory of Computability

Winter 2022
Taught by Professor Shachar Lovett

Table of Contents

Remark: A lot of the pictures and explanations were taken from *Introduction to the Theory of Computation* (3rd Edition) by Michael Sipser and from the CSE 105 lecture slides.

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

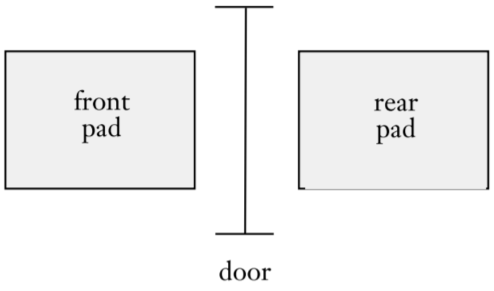
A **language** is a set of strings.

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.



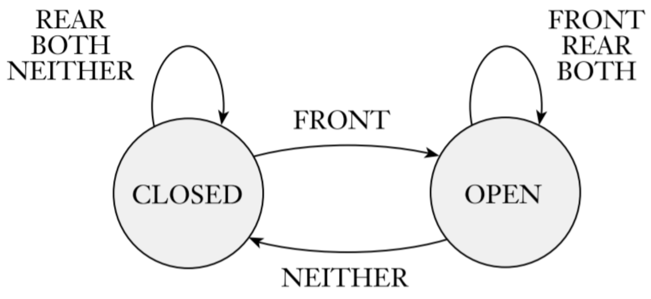
Here:

- 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.

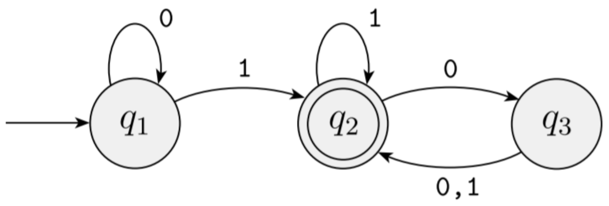
Essentially, **for the state diagram**, start at the initial state (circle) and follow the arrow depending on what input signals are received. **For the transition table**, look at the row corresponding to the initial state and the column corresponding to the input; the resulting cell will be the new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

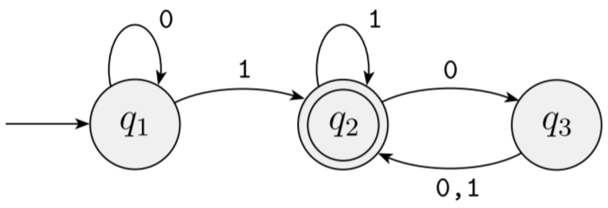
A **finite automaton** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

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 $\emptyset$.

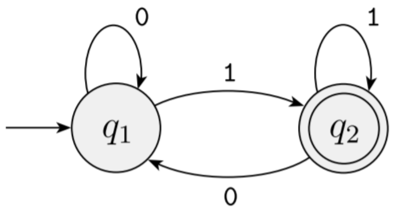
If we consider our example automaton M_1, then define:

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

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.

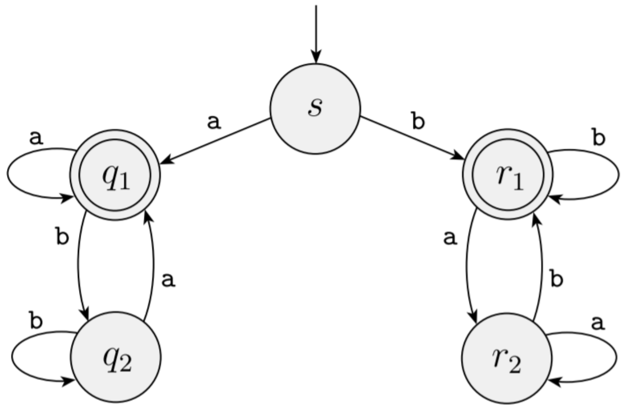
String Input	**Output**
ϵ	REJECT
1	ACCEPT
0	REJECT
01	ACCEPT
10	REJECT
11	ACCEPT

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:

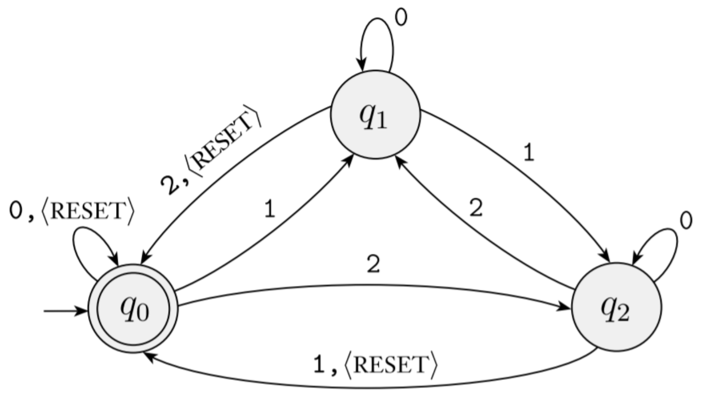
$$\delta_i(q_j, \texttt{0}) = q_j$$

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

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

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

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:

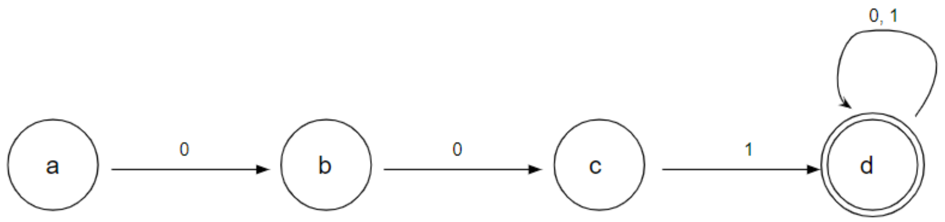
Put yourself in the place of the machine you are trying to design and then see how you would go about performing the machine's task.

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

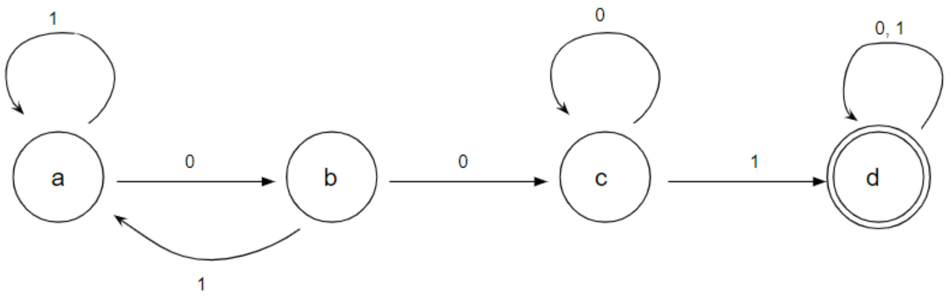
Well, the first thing we can do is create the set of states that will result in an `ACCEPT` state. This is as simple as:



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 `1`s, 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 $\times$). In the theory of computation, the objects are languages and the tools include operations designed for manipulating them. We call these **regular operations**.

Definition 2.3

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 $\Sigma = \{$`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:

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

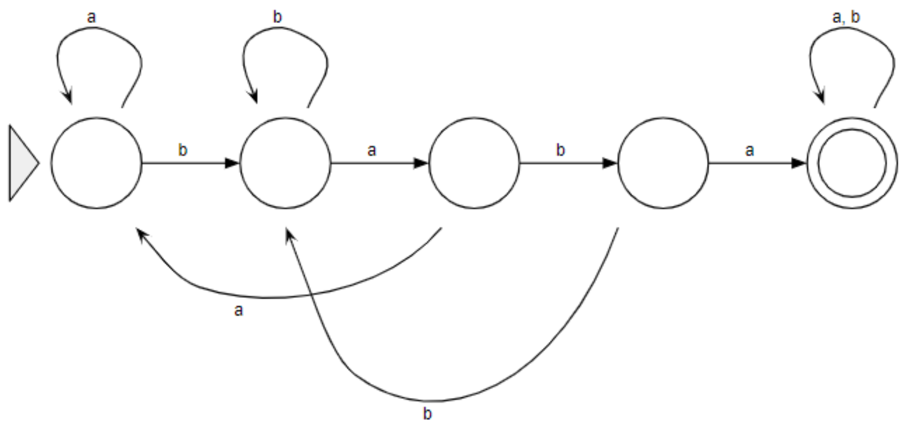
Recall that $\delta^*(q, w)$ is the state reached from q after reading the word w. Taking the complement of A, we have:

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

So, M' accepts A^c. □

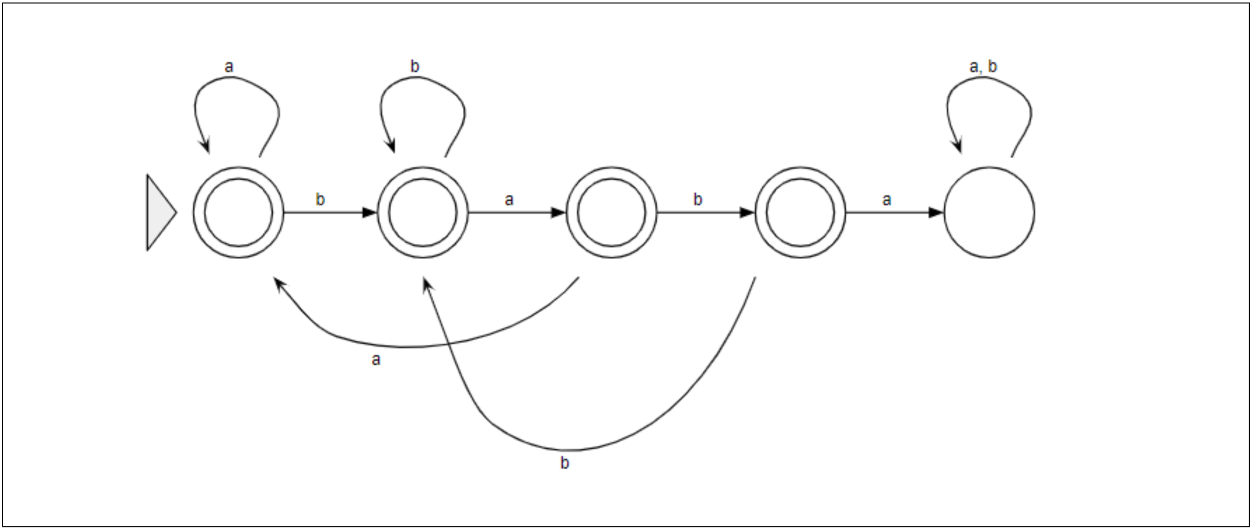
2.9.1 Example 1: Building DFA

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:** $A^* = \{x_1 x_2 \ldots x_k \mid k \geq 0 \text{ and each } x_i \in A\}$

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^*.

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

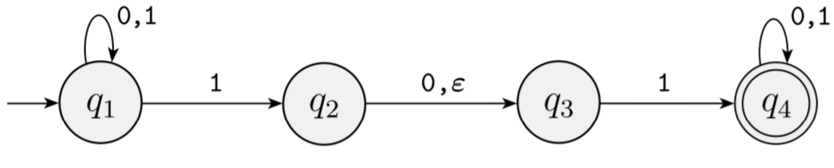


Figure: The nondeterministic finite automaton N_1.

DFA	**NFA**
• Every state of a DFA always has exactly one exiting transition arrow for each symbol in the alphabet.	• 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.
• There is a unique computation path for each input.	• We may allow several (or zero) alternative computations on the same input.
• Labels on the transition arrows are symbols from the alphabet.	• 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 $\underline{\text{end of the input}}$, 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.

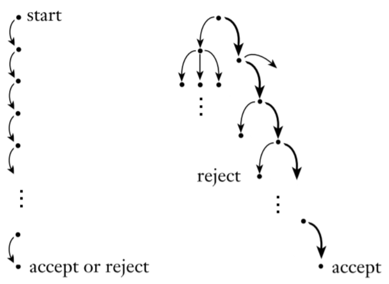


Figure: Difference between deterministic computation and nondeterministic computation.

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:

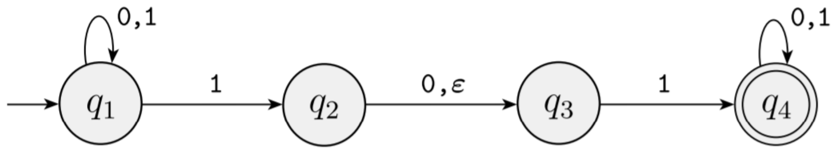
1. Q is a finite set called the **states**.
2. Σ is a finite set called the **alphabet**.
3. $\delta : Q \times \Sigma \cup \{\epsilon\} \mapsto \mathcal{P}(Q)$ is the **transition function**.
4. $q_0 \in Q$ is the **start state**.
5. $F \subseteq Q$ is the **set of accept states** (sometimes also called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

3.5.1 Example: NFA to DFA

Consider the following NFA N:

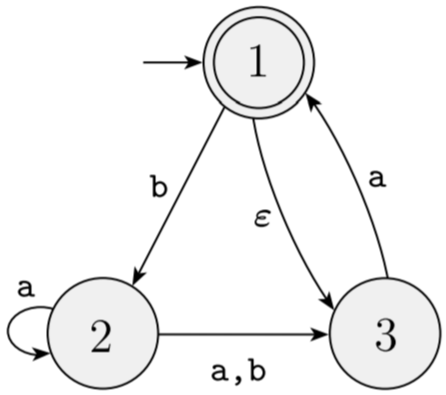


Figure: The NFA N.

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:

- $Q' = \mathcal{P}(Q) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$.
- $\Sigma = \{\texttt{a}, \texttt{b}\}$
- $q_0 = \{1,3\}$. 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 $\boxed{\{1\} \xrightarrow{\texttt{a}} \{1,3\}}$ in the corresponding DFA.
  - 1 goes to 2 when b is given, so it follows that $\boxed{\{1\} \xrightarrow{\texttt{b}} \{2\}}$ 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 $\boxed{\{2\} \xrightarrow{\texttt{a}} \{2,3\}}$ in the corresponding DFA.
  - 2 goes to 3 when b is given, so it follows that $\boxed{\{2\} \xrightarrow{\texttt{b}} \{3\}}$ 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 $\boxed{\{3\} \xrightarrow{\texttt{a}} \{1,3\}}$.
  - 3 doesn't go anywhere when b is given, so it follows that $\boxed{\{3\} \xrightarrow{\texttt{b}} \emptyset}$.

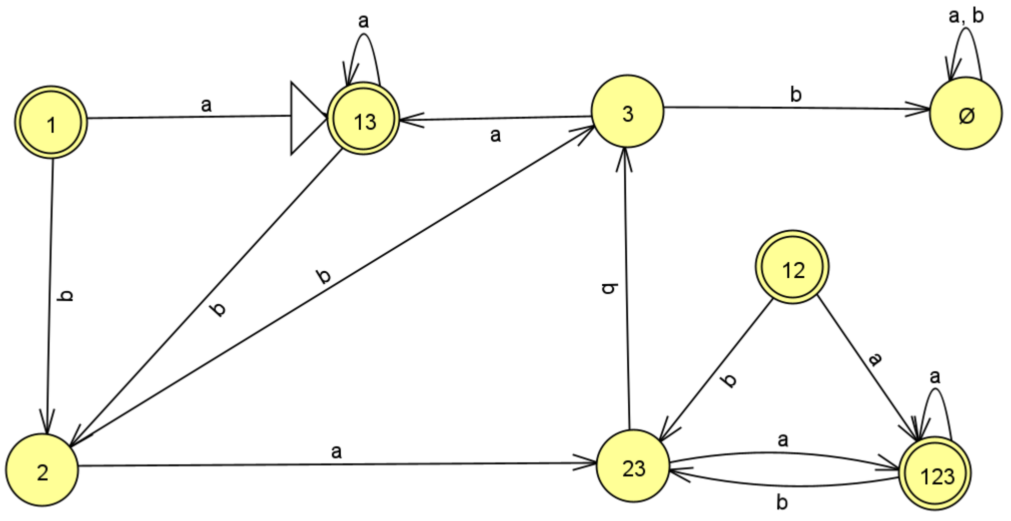
We can use the above to build cases for the remaining elements in Q'.

- 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 $\{1\} \xrightarrow{\texttt{a}} \{1,3\}$, and $\{2\} \xrightarrow{\texttt{a}} \{2,3\}$. Therefore, $\boxed{\{1,2\} \xrightarrow{\texttt{a}} \{1,2,3\}}$ (recall that we take the union).
  - Suppose b is given. Then, we know that $\{1\} \xrightarrow{\texttt{b}} \{2\}$, and $\{2\} \xrightarrow{\texttt{b}} \{3\}$. Therefore, $\boxed{\{1,2\} \xrightarrow{\texttt{b}} \{2,3\}}$.
- 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 $\{1\} \xrightarrow{\texttt{a}} \{1,3\}$, and $\{3\} \xrightarrow{\texttt{a}} \{1,3\}$. Therefore, $\boxed{\{1,3\} \xrightarrow{\texttt{a}} \{1,3\}}$.
  - Suppose b is given. Then, we know that $\{1\} \xrightarrow{\texttt{b}} \{2\}$, and $\{3\} \xrightarrow{\texttt{b}} \emptyset$. Therefore, $\boxed{\{1,3\} \xrightarrow{\texttt{b}} \{2\}}$.
- 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 $\{2\} \xrightarrow{a} \{2,3\}$, and $\{3\} \xrightarrow{a} \{1,3\}$. Therefore, $\boxed{\{2,3\} \xrightarrow{a} \{1,2,3\}}$.
  - Suppose b is given. Then, we know that $\{2\} \xrightarrow{b} \{3\}$, and $\{3\} \xrightarrow{b} \emptyset$. Therefore, $\boxed{\{2,3\} \xrightarrow{b} \{3\}}$.

- 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 $\boxed{\{1,2,3\} \xrightarrow{a} \{1,2,3\}}$.
  - Suppose b is given. From our previous work, we know that $\boxed{\{1,2,3\} \xrightarrow{b} \{2,3\}}$.

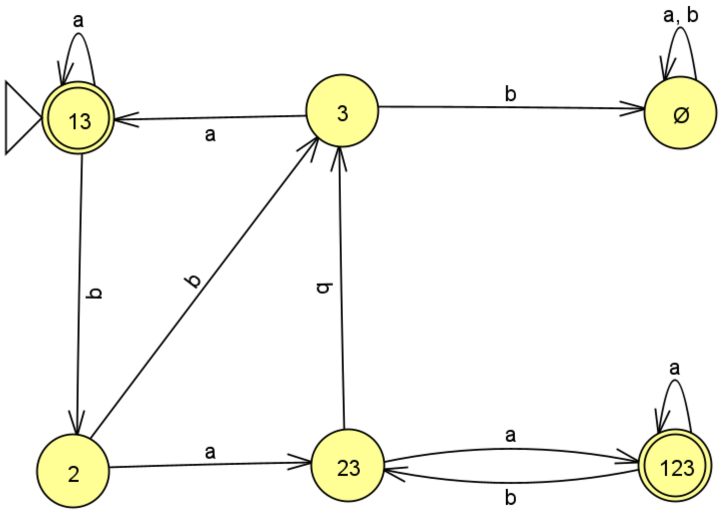
This gives us the following DFA:



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.

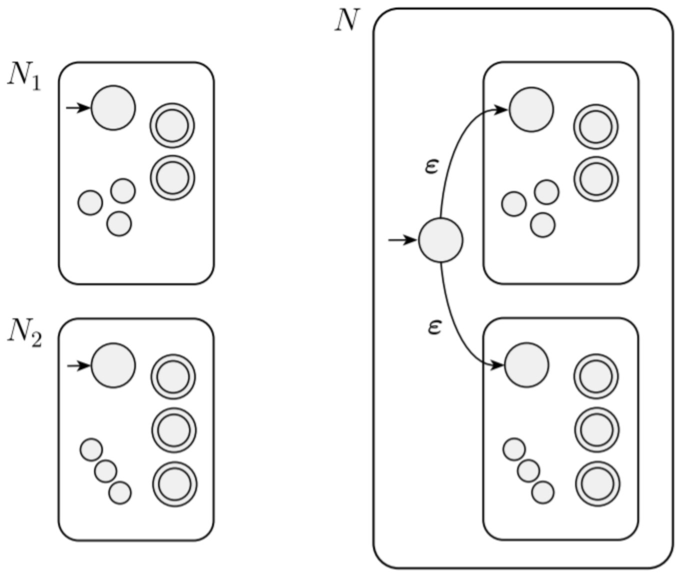
Corollary 3.1

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). $\square$



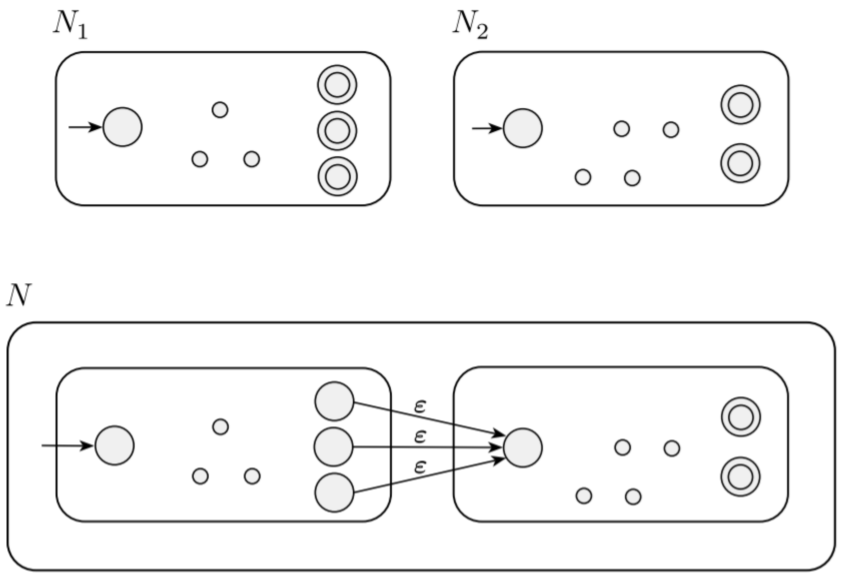
Theorem 3.3

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 $A_1 \circ A_2$. $\square$



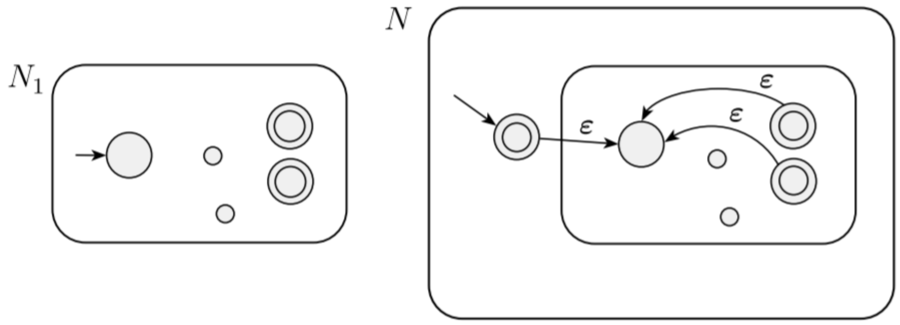
Theorem 3.4

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 001 as a substring$\}$
$\texttt{1}^*(\texttt{01}^+)^*$	1010110111, 1110101	$\{w \mid$ every 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.

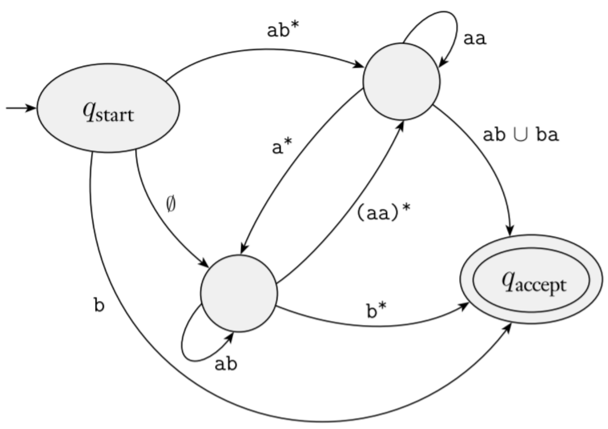


Figure: A generalized nondeterministic finite automaton.

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

1. The <u>start state</u> 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.

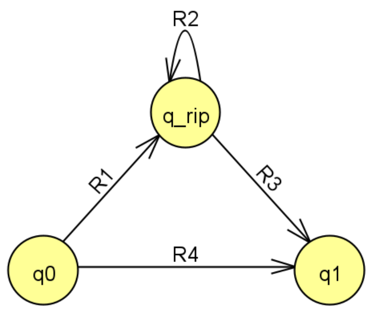
4.4.2 GNFA to Regular Expression

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.*

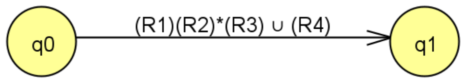
The most important step in this process is constructing an equivalent GNFA with one fewer state when $k > 2$. How can we do this? Well:

- 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

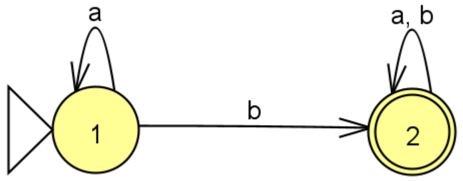
Suppose G is an GNFA. Then, the `CONVERT`(G) algorithm takes a GNFA and returns an equivalent regular expression. The algorithm works like so (Page 73):

`CONVERT`(G)

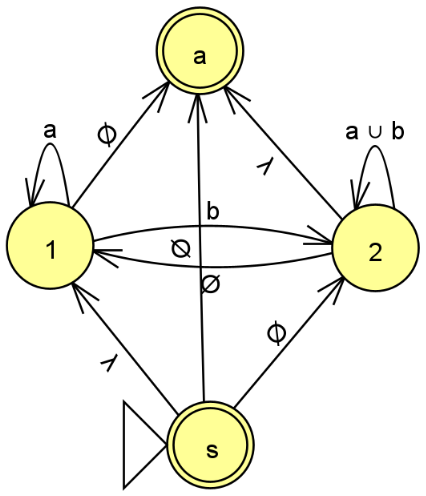
1. Let k be the number of states of G.
2. If $k = 2$, then G must consist of a start state, an accept state, and a single arrow connecting them and labeled with a regular expression R. So, return R.
3. Otherwise, $k > 2$ so we select any state $q_{\text{rip}} \in Q$ different from q_{start} and q_{accept}. 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.

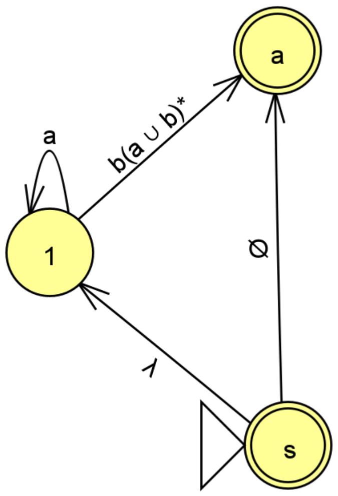
- First, we added two new states: s for the new *start* state and a for the new *accept* state. We have an arrow from s to 1 (the old start state) with ϵ as its label[3]. We also have an arrow from 2 (the old accept state) to a with ϵ as its label.
- Next, note that there was an arrow labeled a, b at 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:

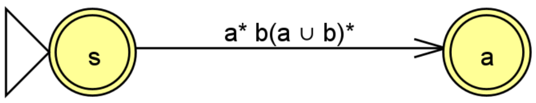
[3]The software used to create these state machines use λ instead of ϵ.



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. □

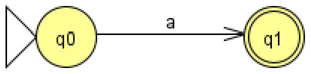
4.5.1 Regular Expression Implies Regularity

Lemma 4.1

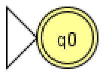
If a language is described by a regular expression, then it is regular.

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.

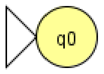
1. Let $R = a$ for some $a \in \Sigma$. Then, $L(R) = \{a\}$ and the following NFA recognizes $L(R)$:



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 `0`s have been seen so far as it reads the input. In other words, because the number of `0`s 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, despite 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`. ☐

6 Context-Free Grammars (2.1)

Consider the following context-free grammar G_1:

$$\begin{aligned} A &\mapsto \texttt{0}A\texttt{1} \\ A &\mapsto B \\ B &\mapsto \texttt{\#} \end{aligned}$$

Here, we note the following:

- A grammar consists of a collection of **substitution rules** (also called productions). Each rule appears as a line in the grammar, comprising a symbol and a string separated by an arrow.
- The symbol (e.g. A, B) is called a **variable**. They are often represented by capital letters.
- The string consists of variables and other symbols (e.g. `0`, `1`, `#`) called **terminals**. They are analogous to the input alphabet and often are represented by lowercase letters, numbers, or other special symbols. In other words, these are symbols that cannot be replaced (substituted).
- One variable, usually (but not always) the top-left one in the list of rules (e.g. A), is designated as the **start variable**.
- When we have multiple rules with the same left-hand variable, we may condense them into a single line where the right-hand sides are separated by a `|` (as an "or"). So, we can rewrite the above rules like so:

$$\begin{aligned} A &\mapsto \texttt{0}A\texttt{1} \mid B \\ B &\mapsto \texttt{\#} \end{aligned}$$

We can use a grammar to describe a language by generating each string of that language like so:

1. Write down the start variable.
2. Find a variable that is written down and a rule that starts with that variable. Then, replace the written dow nvariable with the right-hand side of that rule.
3. Repeat step 2 until no variables remain.

One **derivation** (the sequence of substitutions needed to obtain a string) is as follows:

$$\begin{aligned} A &\implies \texttt{0}A\texttt{1} && \text{By first rule.} \\ &\implies \texttt{00}A\texttt{11} && \text{By first rule.} \\ &\implies \texttt{000}A\texttt{111} && \text{By first rule.} \\ &\implies \texttt{000}B\texttt{111} && \text{By second rule.} \\ &\implies \texttt{000\#111} && \text{By third rule.} \end{aligned}$$

We say that all strings generated in this way constitute the **language of the grammar**, and write $L(G)$for the language of grammar G. So, for our introductory example above, we have that

$$L(G_1) = \{\texttt{0}^n\texttt{\#}\texttt{1}^n \mid n \geq 0\}$$

We say that any language that can be generated by some context-free grammar is called a **context-free language** (CFL).

6.1 Formal Definition

We now introduce the formal definition of a context-free grammar.

Definition 6.1: Context-Free Grammar

A **context-free grammar** is a 4-tuple (V, Σ, R, S) where

1. V is a finite set called the **variables**.
2. Σ is a finite set, disjoint from V, called the **terminals**.
3. R is a finite set of **rules**, with each rule being a variable and a string of variables and terminals.
4. $S \in V$ is the start variable.

Suppose u, v, and w are strings of variables and terminals.

Yielding	**Deriving**
Suppose $A \mapsto w$ is a rule of the grammar. Then, we say that uAv **yields** uwv, written $uAv \implies uwv$.	We say that u **derives** v, written $u \overset{*}{\implies}$, if $u = v$ or if a sequence $u_1, u_2, \ldots, u_k$ exists for $k \geq 0$ and $$u \implies u_1 \implies u_2 \implies \cdots \implies u_k \implies v$$

So, it follows that the **language of the grammar** is given by

$$\{w \in \Sigma^* \mid S \overset{*}{\implies} w\}$$

6.2 Closure Properties of CFGs

There are several closure properties which may be helpful. Suppose we have two CFGs G_1 and G_2, with start states S_1 and S_2, respectively.

- Union: Suppose we wanted to generate a CFG for the language $L(G_1) \cup L(G_2)$. This is as simple as making a new start variable:
$$S \mapsto S_1 \mid S_2$$
- Concatenation: Suppose we wanted to generate a CFG for the language $L(G_1) \circ L(G_2)$. This is as simple as making a new start variable:
$$S \mapsto S_1 S_2$$
- Kleene Star: Suppose we wanted to generate a CFG for the language $L(G_1)^*$. This is as simple as making a new start variable:
$$S \mapsto SS_1 \mid \epsilon$$

6.2.1 Example 1: Identifying Language

Consider the grammar $G_3 = (\{S\}, \{\mathtt{a}, \mathtt{b}\}, R, S)$, with R being the rule

$$S \mapsto \mathtt{a}S\mathtt{b} | SS | \epsilon$$

Describe the language of this context-free grammar.

This grammar generates strings like `abab`, `aaabbb`, and `aababb`. We can describe the language $L(G_3)$ as all strings where, for any `a`, there is a corresponding `b`. Analogously, if we let `a` be `(` and `b` be `)`, then this is the language of all properly nested parentheses.

6.3 Describing Context-Free Languages

Many context-free languages are the union of simpler context-free languages. So, if you need to construct a context-free grammar for a context-free language, break it into simpler pieces, create the corresponding grammars, and then merge them into a grammar for the original language by combining their rules and then adding the new rule

$$S \mapsto S_1 \mid S_2 \mid \cdots \mid S_k$$

where the variables S_i are the start variables for the individual grammars.

6.3.1 Example 1: Constructing Context-Free Grammar

Build a CFG to describe the language $\{\texttt{abba}\}$.

There are several ways to go about this. Consider

$$G_a = (\{S\}, \{\texttt{a}, \texttt{b}\}, \{S \mapsto \texttt{abba}\}, S)$$

which maps S to abba, exactly what we wanted.

Another example is

$$G_b = (\{S, T, V, W\}, \{\texttt{a}, \texttt{b}\}, R, S)$$

where R is defined by the rules

$$\begin{aligned} S &\mapsto \texttt{a}T \\ T &\mapsto \texttt{b}V \\ V &\mapsto \texttt{b}W \\ W &\mapsto \texttt{a} \end{aligned}$$

So, if we applied the substitution rules, we would get

$$\begin{aligned} S &\mapsto \texttt{aT} && \text{By first rule.} \\ &\mapsto \texttt{ab}V && \text{By second rule.} \\ &\mapsto \texttt{abb}W && \text{By third rule.} \\ &\mapsto \texttt{abba} && \text{By fourth rule.} \end{aligned}$$

6.3.2 Example 2: Constructing Context-Free Grammar

Build a CFG to describe the language $\{\texttt{0}^n\texttt{1}^n \mid n \geq 0\} \cup \{\texttt{1}^n\texttt{0}^n \mid n \geq 0\}$.

First, construct the grammar for $\{\texttt{0}^n\texttt{1}^n \mid n \geq 0\}$. We note that some strings in this grammar are ϵ, 01, 0011, and so on. So, the CFG would be

$$S_1 \mapsto \texttt{0}S_1\texttt{1} \mid \epsilon$$

Likewise, construct the grammar for $\{\texttt{1}^n\texttt{0}^n \mid n \geq 0\}$, which gives us

$$S_2 \mapsto \texttt{1}S_2\texttt{0} \mid \epsilon$$

So, our solution is

$$\begin{aligned} S &\mapsto S_1 \mid S_2 \\ S_1 &\mapsto \texttt{0}S_1\texttt{1} \mid \epsilon \\ S_2 &\mapsto \texttt{1}S_2\texttt{0} \mid \epsilon \end{aligned}$$

6.4 Relation Between CFGs and Regular Languages

We know that every regular language is also a context-free language (*however*, not every context-free language is a regular language). There are two approaches to show that this is the case.

1. Start with an arbitrary DFA M, then build a CFG that generates $L(M)$.
2. Build CFGs for $\{\mathtt{a}\}$, $\{\epsilon\}$, and $\emptyset$. Then, show that the class of context-free languages is closed under the regular operations (union, concatenation, Kleene star).

6.4.1 First Approach

Proposition. *Given any DFA M, there is a CFG whose language is $L(M)$.*

Proof. Given a DFA $M = (Q, \Sigma, \delta, q_0, F)$, we can define the CFG $G = (V, \Sigma, R, S)$ where

- $V = \{S_i \mid q_i \in Q\}$
- $R = \{S_i \mapsto \mathtt{a}S_j \mid \delta(q_i, \mathtt{a}) = q_j\} \cup \{S_i \mapsto \epsilon \mid q_i \in F\}$
- $S = S_0$

From this, we need to prove correctness. □

6.4.2 Second Approach

We can build CFGs for $\{\mathtt{a}\}$, $\{\epsilon\}$, and $\emptyset$. Then, show that the class of context-free languages is closed under the regular operations (union, concatenation, Kleene star).

- If $L = \{\mathtt{a}\}$, where $\mathtt{a}$ is some arbitrary character in the alphabet, then we have the CFG (V, Σ, R, S) where
  - $V = \{S\}$
  - $R = \{S \mapsto \mathtt{a}\}$
- If $L = \{\epsilon\}$, then we have the CFG (V, Σ, R, S) where
  - $V = \{S\}$
  - $R = \{S \mapsto \epsilon\}$
- If $L = \emptyset$, then we have the CFG (V, Σ, R, S) where
  - $V = \{S\}$
  - $R = \emptyset$ (or $R = \{S \mapsto S\}$)

Suppose we have $G_1 = \{V_1, \Sigma, R_1, S_1\}$ and $G_2 = \{V_2, \Sigma, R_2, S_2\}$, where G_1 describes the language L_1 and G_2 describes the language L_2, then we can describe $L_1 \cup L_2$ by combining the grammars to make the grammar $G = (V, \Sigma, R, S)$ where:

- $V = V_1 \cup V_2 \cup \{S\}$ (where we assume that $V_1 \cap V_2 = \emptyset$)
- $R = R_1 \cup R_2 \cup \{S \mapsto S_1 \mid S_2\}$

6.5 More Examples of CFG Construction

We now discuss some examples of CFG construction.

6.5.1 Example 1: Basic Construction

Build a CFG to describe the language $\{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\} \cup \{\mathtt{1}^n\mathtt{0}^n \mid n \geq 0\}$.

First, construct the grammar for $\{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$. We note that some strings in this grammar are ϵ, `01`, `0011`, and so on. So, the CFG would be

$$S_1 \mapsto \mathtt{0}S_1\mathtt{1} \mid \epsilon$$

Likewise, construct the grammar for $\{\mathtt{1}^n\mathtt{0}^n \mid n \geq 0\}$, which gives us

$$S_2 \mapsto \mathtt{1}S_2\mathtt{0} \mid \epsilon$$

So, our solution is

$$S \mapsto S_1 \mid S_2$$
$$S_1 \mapsto \mathtt{0}S_1\mathtt{1} \mid \epsilon$$
$$S_2 \mapsto \mathtt{1}S_2\mathtt{0} \mid \epsilon$$

6.5.2 Example 2: Advanced Construction

Build a CFG to describe the langauge $\{0^n1^m2^n \mid n, m \geq 0\}$.

We begin by considering some strings that are generated by this CFG. In particular, we have `02`, `1`, `012`, `0112`, `00122`, and so on. We know that there are an equal number of `0`'s and `2`'s, so our start rule must have at least

$$S \mapsto \mathtt{0}S\mathtt{2}$$

What if we have no more n though? We now need to consider the `1`s. So, we can create another rule

$$T \mapsto \mathtt{1}T$$

But, since we can have 0 `1`'s, we also need to include the empty string; therefore, our rule for T is

$$T \mapsto \mathtt{1}T \mid \epsilon$$

And, thus, our final set of rules are

$$S \mapsto \mathtt{0}S\mathtt{2}$$
$$T \mapsto \mathtt{1}T \mid \epsilon$$

To show that this works, note that we will always have the same number of `0`'s and `2`'s. After we expend all of those, then we can consider some number of `1`'s (which we may not necessarily have the same amount of as `0`'s and `2`'s). Finally, we note that S can map straight to ϵ, implying that the empty string is something that is generated from this language (and, indeed, this is true if $n = m = 0$).

6.5.3 Example 3: Wild Generations

Consider the CFG $G = (V, \Sigma, R, S)$ defined by

- $V = \{E\}$
- $\Sigma = \{\mathtt{1}, +, \times, (,)\}$
- $R = \{E \mapsto E + E \mid E \times E \mid (E) \mid \mathtt{1}\}$
- $S = E$

Which of the following strings is/are generated by this CFG?

a. E

b. `11`

c. $\mathtt{1} + \mathtt{1} \times \mathtt{1}$

d. ϵ

The answer is **C**. The reason why is because we can perform the following substitutions:

$$\begin{aligned} E &\implies E + E && \text{By the first rule.} \\ &\implies \mathtt{1} + E && \text{By the last rule.} \\ &\implies \mathtt{1} + E \times E && \text{By the second rule.} \\ &\implies \mathtt{1} + \mathtt{1} \times E && \text{By the last rule.} \\ &\implies \mathtt{1} + \mathtt{1} \times \mathtt{1} && \text{By the last rule.} \end{aligned}$$

Note that there are potentially other possible ways this string can be generated. Regardless, the answer is not A because a variable is not a valid string. The answer is not B because there is no way to expand E out twice. The answer is not D because you cannot generate the empty string with the given rules.

6.6 Ambiguity

Sometimes, a grammar can generate the same string in several different ways.

Definition 6.2: Grammar Ambiguity

A string w is derived **ambiguously** in a context-free grammar G if it has two or more leftmost derivations[a]. Grammar G is ambiguous if it generates some string ambiguously.

[a]We say that a derivation of a string w in a grammar G is a **leftmost derivation** if, at every step, the leftmost remaining variable is the one replaced.

For example, consider the last example discussed above. There are several ways to derive $\mathtt{1} + \mathtt{1} \times \mathtt{1}$. Some examples are shown below.

- Derivation 1:

$$\begin{aligned} E &\implies E + E && \text{By the first rule.} \\ &\implies \mathtt{1} + E && \text{By the last rule.} \\ &\implies \mathtt{1} + E \times E && \text{By the second rule.} \\ &\implies \mathtt{1} + \mathtt{1} \times E && \text{By the last rule.} \\ &\implies \mathtt{1} + \mathtt{1} \times \mathtt{1} && \text{By the last rule.} \end{aligned}$$

- Derivation 2:

$$\begin{aligned} E &\implies E + E && \text{By the first rule.} \\ &\implies E + E \times E && \text{By the second rule.} \\ &\implies 1 + E \times E && \text{By the last rule.} \\ &\implies 1 + 1 \times E && \text{By the last rule.} \\ &\implies 1 + 1 \times 1 && \text{By the last rule.} \end{aligned}$$

Note that some context-free languages can be generated only by ambiguous grammars; in this case, we say that these languages are **inherently ambiguous**.

6.7 Chomsky Normal Form

We can use Chomsky normal form to "simplify" a context-free grammar.

Definition 6.3: Chomsky Normal Form

A context-free grammar is in **Chomsky normal form** if every rule is of the form

$$A \mapsto BC$$
$$A \mapsto a$$

where a is any terminal and A, B, C are any variables, except that B and C may not be the start variables. In addition, we permit the rule $S \mapsto \epsilon$, where S is the start variable.

A somewhat important theorem is as follows:

Theorem 6.1

Any context-free language is generated by a context-free grammar in Chomsky normal form.

7 Pushdown Automata (2.2)

We now introduce a new type of computational model called the **pushdown automata**, which is essentially like a nondeterministic finite automata but with an extra component called a **stack**. The stack provides additional memory beyond the finite amount available in the control, additionally allowing us to recognize some nonregular languages.

Note that pushdown automata are equivalent in power to context-free grammars, giving us the ability to decide which of the two we want to use to prove that a language is context-free.

7.1 The Idea

The idea is that, at each step:

1. **Transition** to a new state based on the current state, letter read, *and* the top letter of the stack.
2. Possibly push (or pop) a letter to (or from) the top of the stack.

We *accept* a string if there is **some** sequence of states and **some** sequence of stack contents which processes the entire input string and ends in an accepting state. We assume that the stack is empty at the beginning, and don't necessarily care if the stack is empty at the end.

7.2 Formal Definition

As implied, the formal definition of a pushdown automaton is similar to that of a finite automaton, except for the stack. The stack is a device containing symbols drawn from some alphabet, and the machine may use different alphabets for its input and its stack. Therefore, we need to specify both an input alphabet Σ and a stack alphabet Γ.

Recall that $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$; the same idea applies with Γ_ϵ. Now, the domain of δ is $Q \times \Sigma_\epsilon \times \Gamma_\epsilon$. The idea behind this domain is that, given the current state $q \in Q$, next input symbol read $\sigma \in \Sigma_\epsilon$, and top symbol of the stack $\gamma \in \Gamma_\epsilon$, the machine can decide the next move. As usual, if $\sigma = \epsilon$, then the machine won't read a symbol from the input. Likewise, if $\gamma = \epsilon$, then the machine won't read a symbol from the stack.

We now consider the range of the δ function. It's possible that it can enter a new state and possibly write a symbol on top of the stack. This is indicated by some member of $Q \times \Gamma_\epsilon$.

Definition 7.1: Pushdown Automaton (PDA)

A **pushdown automaton** is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, and F are all finite sets, and

1. Q is the set of states.
2. Σ is the input alphabet.
3. Γ is the stack alphabet.
4. $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$ is the transition function.
5. $q_0 \in Q$ is the start state.
6. $F \subseteq Q$ is the set of accept states.

Remark: We write $a, b \mapsto c$ to signify that, when the machine is reading an a from the input, it may replace the symbol b on the top of the stack with a c; in other words, pop b and push c.

- If $a = \epsilon$, then the machine may make this transition without reading any symbol from the input.

- If $b = \epsilon$, then the machine may make this transition without readng any symbols from the stack, but it will push c onto the stack.
- If $c = \epsilon$, then the machine will pop b from the stack but not push anything to the stack.

Note: The formal definition of a PDA contains no explicit mechanism to allow the PDA to test for an empty stack. So, one trick that we can do is to initially push a special symbol `$` on the stack. Then, if it ever sees the `$` again, then it knows that the stack is effectively empty.

7.3 Computation of a Pushdown Automaton

A pushdown automaton $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ accepts input w if w can be written as $w = w_1 w_2 \dots w_m$, where each $w_i \in \Sigma_\epsilon$ and the sequences of starts $r_0, r_1, \dots, r_m \in Q$ and strings $s_0, s_1, \dots, s_m \in \Gamma^*$ exist that satisfy the following three conditions.

1. $r_0 = q_0$ and $s_0 = \epsilon$, which signifies that M starts out properly, in the start state, and with an empty stack.
2. For $i = 0, \dots, m-1$¡ we have $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, where $a_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Gamma_\epsilon$ and $t \in \Gamma^*$. This condition states that M moves properly according to the state, stack, and next input symbol.
3. $r_m \in F$. This condition states that the accept state occurs at the input end.

7.4 Context-Free

Theorem 7.1

A language is context free if and only if some pushdown automaton recognizes it.

7.4.1 Example 1: Designing a PDA

Consider the following language $L = \{\mathtt{0}^i \mathtt{1}^{i+1} \mid i \geq 0\}$.

1. Design a CFG that generates L.

   We can define a CFG G where the rules are

   $$S \mapsto T\mathtt{1} | \mathtt{0}S\mathtt{1}$$
   $$T \mapsto \epsilon$$

   The idea is that S can recursively map to $\mathtt{0}S\mathtt{1}$. After enough times, it can then map to $T\mathtt{1}$, which adds one more `1`.

2. Consider the notation $\epsilon, \epsilon \mapsto \mathtt{\$}$. What does this mean?

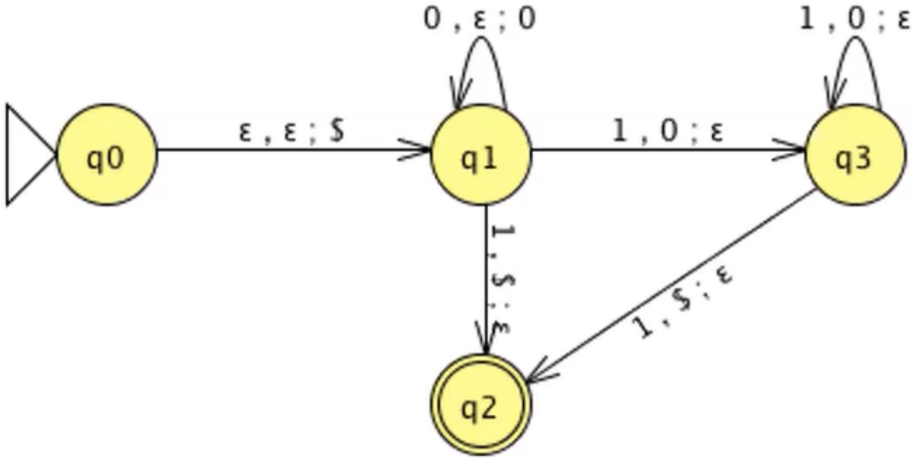
   Without reading any input, and without popping any symbols from the stack, we push `$` on top of the stack.

   Remark: This is commonly used from the initial state (at the start of computation) to record the top of the stack with a special symbol. In other words, if the stack becomes empty, then we will know.

3. Provide a rough description of how you can make a PDA that accepts upon receiving any strings that are in this language.

The idea for a PDA is that we need to read the symbols from the input. As each `0` is read, we push it onto the stack. As soon as `1`'s are seen, we pop a `0` for each `1` that is read. If the stack becomes empty and there is exactly one `1` left to read, then we read that `1` and accept the input. If the stack becomes empty and there are either zero or more than one `1`'s left to read, *or* if the `1`'s are finished while the stack still contains `0`'s, *or* if any `0`'s appear in the input following the `1`'s, then reject the input.

4. Design a PDA that does exactly what you described in the previous step.



5. Describe how your PDA works with the input `00111`.

- For our initial step, we start at q_0 and our stack can be described as `[]`.
- Looking at the only transition arrow, note that we can transition without reading any input ($\epsilon, \epsilon \mapsto \$$). Upon transitioning, we don't pop anything from the stack, but we do push the special `$` onto the stack. So, we can describe the machine like so:

```
State:  q1
Stack:  [$]     Top
Input:  00111
        ^
```

- At q_1, we have three transition arrows; one that requires a `0` (without reading and popping anything from the stack), and two that requires a `1` (one which requires the `$` to be at the top of the stack, and another that requires a `0` to be on top of the stack). As the (first and) next symbol in our input is `0`, we read that in and transition to q_1 again while also pushing `0` onto the stack.

```
State:  q1
Stack:  [$, 0]     Top
Input:  00111
         ^         Next to read
```

- We repeat the previous step since we need to read in a `0`.

```
State:  q1
Stack:  [$, 0, 0]     Top
Input:  00111
```

```
                    ^           Next to read
```

- At this point, our next input symbol is `1`. As we're at q_1 still, we now need to consider the two transition arrows that can read in a `1`. One transition arrow requires a `0` to be on top of the stack, and another requires a `$`. So, we go with the one that requires a `0` to be on top of the stack since that's what we have in the stack. So, we transition to q_3, popping a `0` from the stack.

```
State:  q2
Stack:  [$, 0]     Top
Input:  00111
           ^       Next to read
```

- We are now at state q_3. The next input symbol we need to read in is the `1` (the center one), and we still have another `0` on top of the stack. So, we take the transition arrow $\texttt{1}, \texttt{0} \mapsto \epsilon$. After transitioning back to q_3, we pop `0` from the stack.

```
State:  q2
Stack:  [$]     Top
Input:  00111
            ^   Next to read
```

- Now that we're still at q_3, we still need to read in `1` (the last one). However, the top of the stack is a `$`, implying that we're at the end of the stack. We note that there is a transition arrow $\texttt{1}, \$ \mapsto \epsilon$, we take that transition arrow. So, we transition to q_2 and pop `$` from the stack.

```
State:  q2
Stack:  []     Top
Input:  00111
             ^ Next to read
```

- There is nothing left to read, and since we're at q_2, we accept.

For a summary of what just happened, observe the below table.

Input	q	**Stack** (Top is Right)
	q_0	[]
	q_1	[$]
0	q_1	[$, 0]
0	q_1	[$, 0, 0]
1	q_3	[$, 0]
1	q_3	[$]
1	q_2	[]

7.4.2 Example 2: PDA vs. CFGs

Consider the following language $L = \{\texttt{a}^i\texttt{b}^j\texttt{c}^k \mid i = j \text{ or } i = k \text{ with } i, j, k \geq 0\}$.

1. Which of the following strings are **not** in L?

   - `b`
   - `abc`
   - `abbcc`

- `aabcc`

The answer is `abbcc`. This says that $i = 1$, $j = k = 2$, but this violates the definition that $i = j$ or $i = k$ since $i \neq j$ and $i \neq k$.

2. Create a CFG that generates L.

The idea is that

$$\begin{aligned} L &= \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i = j \text{ or } i = k \text{ with } i,j,k \geq 0\} \\ &= \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i = j \text{ with } i,j,k \geq 0\} \cup \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i = k \text{ with } i,j,k \geq 0\} \end{aligned}$$

So, we can find the CFG of each individual language and then combine them to form the CFG of interest.

- $L_1 = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i = j \text{ with } i,j,k \geq 0\}$. The CFG can be defined by

$$\begin{aligned} A &\mapsto A\mathtt{c} \mid B \\ B &\mapsto \mathtt{a}B\mathtt{b} \mid \epsilon \end{aligned}$$

- $L_2 = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i = k \text{ with } i,j,k \geq 0\}$. The CFG can be defined by

$$\begin{aligned} C &\mapsto \mathtt{a}C\mathtt{c} \mid D \\ D &\mapsto D\mathtt{b} \mid \epsilon \end{aligned}$$

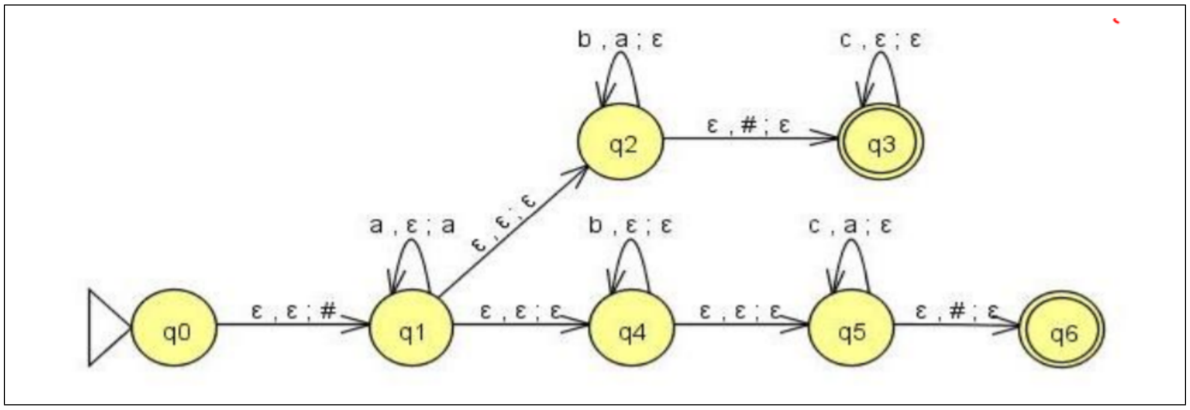
Combining these gives us

$$\begin{aligned} S &\mapsto A \mid C \\ A &\mapsto A\mathtt{c} \mid B \\ B &\mapsto \mathtt{a}B\mathtt{b} \mid \epsilon \\ C &\mapsto \mathtt{a}C\mathtt{c} \mid D \\ D &\mapsto D\mathtt{b} \mid \epsilon \end{aligned}$$

3. Give an informal description of a PDA that accepts all strings in this language. *Hint:* Consider what information you need to track, the amount of memory you need, and whether non-determinism is needed.

- The PDA pushes a `$` to indicate the top of the stack. Then, it starts reading `a`'s, pushing each one onto the top of the stack.
- The PDA then guesses when it has reached the end of the `a`'s and whether to match the number of `a`'s to the number of `b`'s or the number of `c`'s.
- If trying to match number of `b`'s with the number of `a`'s, the PDA pops off `a`'s for each `b` that is read in. If there are more `a`'s on the stack but no more `b`'s being read, then we reject. When the end of the stack `$` is reached, the number of `a`'s matches the number of `b`'s. If this is the end of the input or if any number of `c`'s is read at this point, accept. Otherwise, reject.
- If trying to match the number of `c`'s with the number of `a`'s, first read any number of `b`'s without changing the stack content. Then, nondeterministically guess when to start reading the `c`'s. For each `c` read, pop one `a` off the stack. When the end of the stack `$` is reached, the number of `a`'s and `c`'s match.

4. Design a PDA that does exactly what you described in the previous step.



7.4.3 Example 3: PDA vs. CFGs

Consider the language

$$L = \{x\texttt{\#}w \mid x, w \in \{\texttt{0}, \texttt{1}\}^* \text{ and } w^R \text{ is a substring of } x\}$$

1. Recall that a CFG is a 4-tuple (V, Σ, R, S). Here, $\Sigma = \{\texttt{0}, \texttt{1}, \texttt{\#}\}$ and $S \in V$. What is V and R?

Let's consider some example strings.

- `#`
- `0#0`
- `1011#110`
- `0100101#1010`

In particular, we note several things.

- $|x| \geq |w|$. This means we can add some number of `0`'s or `1`'s to the left side of the `#` without adding them to the right side.
- At some point, when we generate the string, we're just adding `0` (or `1`) to the ends of both sides.

So, we can define R like so:

$$\begin{aligned} S &\mapsto \texttt{0}S \mid \texttt{1}S \mid A \\ A &\mapsto \texttt{0}A\texttt{0} \mid \texttt{1}A\texttt{1} \mid B\texttt{\#} \\ B &\mapsto \texttt{0}B \mid \texttt{1}B \mid \epsilon \end{aligned}$$

Therefore, V is given by

$$V = \{S, A, B\}$$

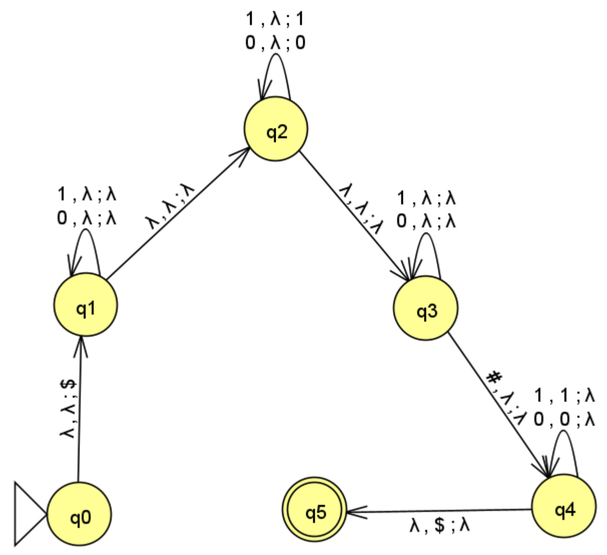
2. Give an informal description of a PDA that accepts all strings in this language.

We're going to take a non-deterministic approach to this problem. Essentially, the idea is as follows:

- The PDA pushes a `$` to indicate the top of the stack.
- Then, it starts reading `0`'s and `1`'s without adding or removing anything from the stack. This accounts for the possibility where we added `0`'s or `1`'s to the string on the left-side of the `#` without adding it to the right-side.
- Each time the PDA performs the previous step, it nondeterministically moves to the next state, where for each `1` it reads in, it pops a `1` from the stack. For each `0` it read in, it pops a `0` from the stack.

- Each time the PDA performs the previous step, it nondeterministically moves to the next state, where it starts reading `0`'s and `1`'s without adding or removing anything from the stack. This accounts for the possibility where we added `0`'s or `1`'s to the string on the left-side of the `#` without adding it to the right-side.
- When we see a `#`, we move to the next state, where whenever we read in a `0` (or `1`), that `0` (or `1`) better be on top of the stack.
- Finally, when we see the `$`, we read it in and move to the accept state.

3. Design a PDA that does exactly what you said in the previous step.



7.5 Conventions for PDAs

- We can *test for the end of the stack*, without providing details. We can always push the end-of-stack symbol `$` at the start.
- We can *test for the end of the input* without providing details. We can transform the PDA to one where accepting states are only those reachable when there are no more input symbols.
- We don't always need to provide a state transition diagram.

8 Turing Machines (3.1)

First proposed by Alan Turing in 1936, the **Turing machine** is a much more accurate model of a general purpose computer. It can do everything that a real computer can do, but even it cannot solve certain problems[4].

8.1 The Idea

- The input string is *written* on the leftmost squares of the tape. The rest of the tape is empty.
- We can read *and* write on the tape. The read/write head starts at the leftmost position on the tape.
- Computation proceeds according to the transition function. In other words, given the current state of machine, and the current symbol being read, the machine will
  - Transition to a new state.
  - Write a symbol to its current position, overwriting the existing symbol.
  - Moves the tape head L or R.
- Computation ends if and when it enters either the **accept** or the **reject** state. This means that we can have programs that can run forever.

8.2 Language of a Turing Machine

Given a Turing machine M, the language $L(M)$ is the set of all strings w such that the computation of M on w *halts* after entering the accept state. That is, $L(M) = \{w \mid w \text{ is accepted by } M\}$.

8.3 Formal Definition

As usual, the most important thing about the Turing machine is the transition function

$$\delta : Q \times \Gamma \mapsto Q \times \Gamma \times \{L, R\}$$

That is, when the machine is in a certain state q and the head is over a tape square containing the symbol a, and if $\delta(q, a) = (r, b, L)$, then the machine writes the symbol b replacing the a, and goes to state r. The third component is either L or R, and indicates whether the head moves to the left or right after writing. In this case, the L indicates that we move the tape to the left.

Definition 8.1: Turing Machine

A **Turing machine** is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ where Q, Σ, Γ are all finite sets and

1. Q is the set of states.
2. Σ is the input alphabet not containing the *blank symbol* $\sqcup$.
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$.
4. $\delta : Q \times \Gamma \mapsto Q \times \Gamma \times \{L, R\}$ is the transition function.
5. $q_0 \in Q$ is the start state.
6. $q_{\text{accept}} \in Q$ is the accept state.
7. $q_{\text{reject}} \in Q$ is the reject state, where $q_{\text{reject}} \neq q_{\text{accept}}$.

[4]In a very real sense, these problems are beyond the theoretical limits of computation.

8.4 Configuration of a Turing Machine

As a Turing machine computes, changes occur in the current state, the current tape contents, and the current head location. A setting of these three items is called a **configuration** of the Turing machine. They are often represented in a special way. For a state q and two strings u and v over the tape alphabet Γ, we write uqv for the configuration where the current state is q, the current tape contents is uv, and the current head location is the first symbol of v.

For example, $\texttt{1011}q_7\texttt{01111}$ represents the configuration when the tape is `101101111`, the current state is q_7, and the head is currently on the second `0`.

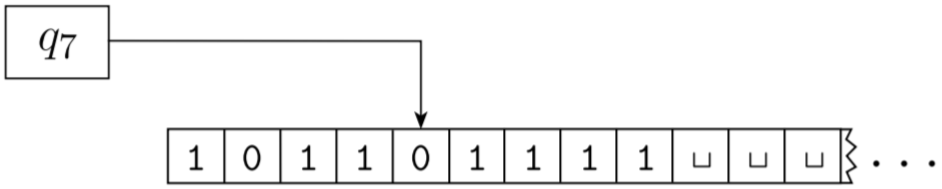


Figure: The configuration $\texttt{1011}q_7\texttt{01111}$.

8.4.1 Transitioning Between Configurations

Suppose that a configuration C_1 **yields** configuration C_2 if the Turing machine can legally go from C_1 to C_2 in a single step. We can define this notion formally as follows: suppose we have $a, b, c \in \Gamma$, as well as $u, v \in \Gamma^*$ and states $q_i, q_j \in Q$. In that case, uaq_ibv and uq_jacv are two arbitrary configurations. Say that

$$uaq_ibv \text{ yields } uq_jacv$$

if, in the transition function, $\delta(q_i, b) = (q_j, c, L)$. This handles the case where the Turing machine moves leftward. For a rightward move, say that

$$uaq_ibv \text{ yields } uacq_jv$$

if, in the transition function, $\delta(q_i, b) = (q_j, c, R)$.

8.4.2 Start, Accepting, Rejecting, and Halting Configurations

The start configuration of M on input w is the configuration q_0w, which indicates that the machine is in the start state q_0 with its head at the leftmost position on the tape.

In an accepting configuration, the state of the configuration is q_{accept}.

In a rejecting configuration, the state of the configuration is q_{reject}.

Accepting and rejecting configurations are halting configurations and do not yield further configurations.

8.5 Deciders and Recognizers

We now briefly talk about the difference between deciders and recognizers.

8.5.1 Turing-Recognizable

A language is **Turing-recognizable** if some Turing machine recognizes it, i.e. if $L = L(M)$ for some Turing machine M. When we start a Turing machine on some input, the machine can either *accept*, *reject*, or *loop*. However, sometimes we don't want the machine to loop.

8.5.2 Turing-Decider

A Turing machine M is a **decider** Turing machine (either Turing-decidable or decidable) if it halts on all inputs (i.e. never loops).

L is **Turing-decidable** if some Turing machine that is a decider recognizes it.

8.5.3 Example 1: Turing Machine

Consider the language $L = \{w\texttt{\#}w \mid w \in \{\texttt{0}, \texttt{1}\}^*\}$, which is both not regular and not context-free.

1. Give an idea for a potential Turing machine that recognizes L.

   The idea is as follows

   - We want to zig-zag across tapes to corresponding positions on either side of # to check whether these positions agree. If they do not, or there is no #, then we reject. If they do, then cross them off.
   - Once all symbols to the left of the # are crossed off, check for any symbols to the right of #. If there are any, *reject*. Otherwise, accept.

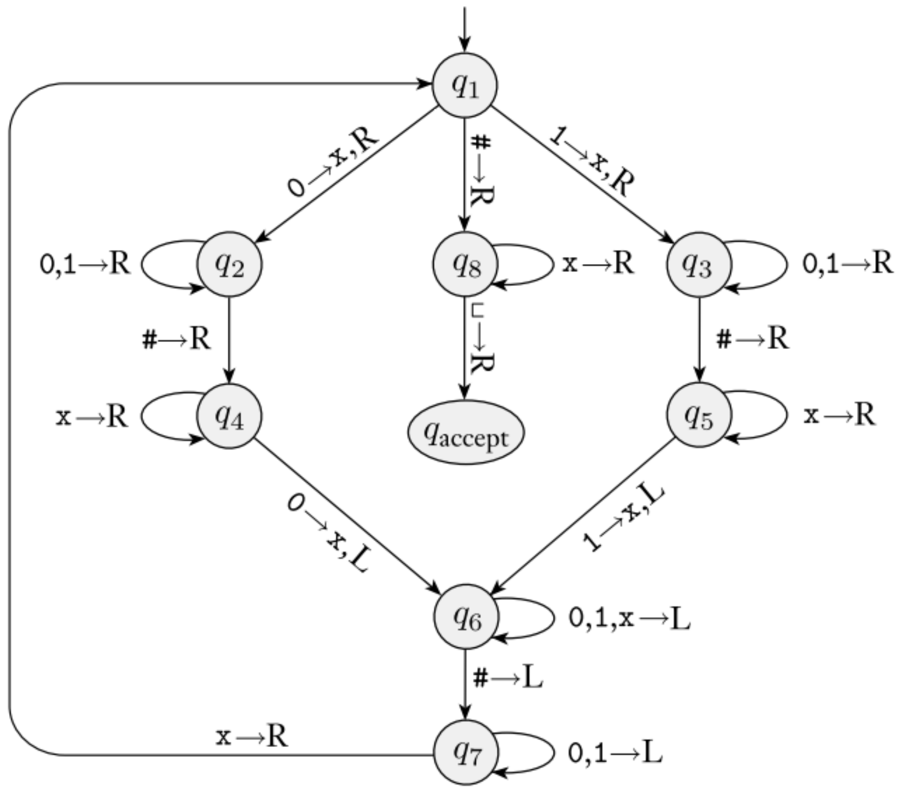
   To see what we mean, consider the following example

   ```
   0 1 # 0 1
   -> x 1 # 0 1
      ^

   -> x 1 # x 1
            ^

   -> x x # x 1
        ^

   -> x x # x x
              ^
   ```

2. Is this machine a decider?

   Yes, because it will halt (either accept or reject) no matter what the input is.

3. Draw the state diagram corresponding to the Turing machine.



To simplify the figure, we don't show the reject state or the transitions going to the reject state. Those transitions occur implicitly whenever a state lacks an outgoing transition for a particular symbol.

4. What is Q (the set of states)?

$$Q = \{q_1, \ldots, q_8, q_{\text{accept}}, q_{\text{reject}}\}$$

5. What is Σ?

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

6. What is Γ?

$$\Gamma = \Sigma \cup \{\texttt{b}, \texttt{x}\}$$

7. Given the string `01#01`, run through the Turing machine.

We begin with the initial configuration.

```
State:      q1
Tape:       0 1 # 0 1
  (Next:)   ^
Config:     q1 0 1 # 0 1
```

Here, we read in the `0` and move to q_2, replacing `0` with `x` and moving the tape to the right.

```
State:      q2
Tape:       x 1 # 0 1
  (Next:)     ^
Config:     x q2 1 # 0 1
```

At this point, we read in the `1` as well (without crossing anything out) and move the tape to the right.

```
State:      q2
Tape:       x 1 # 0 1
  (Next:)       ^
Config:     x 1 q2 # 0 1
```

For the same reason as above, we read in `#` as well, transitioning to q_4 and moving the tape to the right.

```
State:      q4
Tape:       x 1 # 0 1
  (Next:)         ^
Config:     x 1 # q4 0 1
```

Now, we read in the `0`, replacing it with a `x` and moving the tape left. We also transition to q_6.

```
State:      q6
Tape:       x 1 # x 1
  (Next:)       ^
Config:     x 1 q6 # x 1
```

We read in the `#`, moving the tape to the left and transitioning to q_7.

```
State:      q7
Tape:       x 1 # x 1
  (Next:)     ^
Config:     x q7 1 # x 1
```

We now keep looping at q_7 whenever we see a `0` or `1`. In our case, we only need to read one `1`, so we do that, while also moving the tape to the left.

```
State:      q7
Tape:       x 1 # x 1
  (Next:)   ^
Config:     q7 x 1 # x 1
```

We now read in the `x` and transition to q_1, moving the tape to the right.

```
State:      q1
Tape:       x 1 # x 1
  (Next:)     ^
Config:     x q1 1 # x 1
```

Now, we transition to q_3, reading in the `1` and replacing it with an `x` while also moving the tape to the right.

```
State:      q3
Tape:       x x # x 1
  (Next:)       ^
Config:     x x q3 # x 1
```

We now read in the `#`, moving the tape to the right and transitioning to q_5.

```
State:      q5
Tape:       x x # x 1
  (Next:)         ^
Config:     x x # q5 x 1
```

We now read in all of the `x`'s, moving the tape to the right. We only do this once as there is only one `x` to be read.

```
State:      q5
Tape:       x x # x 1
  (Next:)           ^
Config:     x x # x q5 1
```

We now read in a `1`, replacing it with a `x`, transitioning to q_6, and moving the tape to the left.

```
State:      q6
Tape:       x x # x x
  (Next:)         ^
Config:     x x # q6 x x
```

At q_6, we keep reading in the `x`'s. We only do this once, so we move the tape one to the left.

```
State:      q6
Tape:       x x # x x
  (Next:)       ^
Config:     x x q6 # x x
```

Now, we transition to q_7 since the next symbol is `#`, moving the tape to the left.

```
State:      q7
Tape:       x x # x x
  (Next:)     ^
Config:     x q7 x # x x
```

We transition to q_1, moving the tape to the right.

```
State:      q1
Tape:       x x # x x
  (Next:)       ^
Config:     x x q1 # x x
```

Since the next symbol to be read is a #, we transition to q_8, moving the tape to the right.

```
State:      q8
Tape:       x x # x x
  (Next:)         ^
Config:     x x # q8 x x
```

We now keep reading in any x's, moving the tape to the right. This is done twice.

```
State:      q8
Tape:       x x # x x
  (Next:)             ^
Config:     x x # x x q8
```

At this point, we are implicitly at a $\sqcup$. So, we move to q_{accept}. Thus, this string is accepted.

8.6 Describing Turing Machines

There are several ways we can describe Turing machines.

- **Formal Definition:** A set of states, input alphabet, tape alphabet, transition function, start state, accept state, and reject state. Essentially, drawing out the Turing machine. This is the most detailed level of description.
- **Implementation-Level Definition:** English prose to describe Turing machine head movements relative to the contents of the tape; in other words, we use English prose to describe the way that the Turing machine moves its head and the way that it stores data on its tape. Note that we do not give details of states or transition function.
- **High-Level Description:** A description of the algorithm, without the implementation details of a machine. As part of this description, we can "call" and run another Turing machine as a subroutine. Here, we do not need to mention how the machine manages its tape or head.

8.6.1 Example: Describing Turing Machines

Give an *implementation-level* description of a Turing machine which *decides* the empty set.

Let M be the Turing machine where, on some input w, we reject.

8.6.2 Example: Describing Turing Machines

Give a *high-level* description of a Turing machine which *decides* the language

$$A = \{\langle G\rangle \mid G \text{ is a connected undirected graph}\}$$

Here, we say that $\langle G\rangle$ is the encoding of the graph. For example, if G is defined by

$$G = (V, E)$$

where

$$V = \{1, 2, 3, 4\} \qquad E = \{(1, 2), (2, 3), (3, 1), (1, 4)\}$$

then the encoding is given by

$$\langle G\rangle = \texttt{(1,2,3,4)((1,2),(2,3),(3,1),(1,4))}$$

Let M be the Turing machine where, on input $\langle G\rangle$ (the encoding of the graph), we do the following:

1. Select the first node of G and mark it.
2. While we still have nodes to mark, for each node in G, mark it if it is attached by an edge to a node that is already marked.
3. Scan all the nodes of G to determine whether they are all marked. If they are, *accept*. Otherwise, *reject*.

8.7 Class of Recognizable and Decidable Languages

The class of decidable *and* recognizable languages is closed under:

- Union.
- Concatenation.
- Intersection.
- Kleene Star.

The class of *decidable* languages is also closed under complementation.

8.7.1 Class of Decidable Languages Closed Under Union

Theorem 8.1

The class of decidable languages over a fixed alphabet Σ is closed under union.

Proof. Let L_1 and l_2 be languages over Σ and suppose M_1 and M_2 are Turing machines that decide these languages, respectively. We will define a new Turing machine, M, via a high-level description. For some input w, the Turing machine M will do the following:

1. Run M_1 on input w. If M_1 accepts w, then accept. Otherwise, go to the next step.
2. Run M_2 on input w. If M_2 accepts w, then accept. Otherwise, we reject.

To show correctness, we need to show that $L(M) = L_1 \cup L_2$ and that M is a decider. For some string s, if L_1 or L_2 contains s, then clearly the machine must recognize it. We know this is the case because our machine runs M_1 and then runs M_2 if M_1 doesn't yield an *accept* state. Now, if M_1 and M_2 both reject, then clearly s must not be in any of the two languages, and will thus not be in M. To show that M is a decider, we note that M will either accept (if s is accepted by any of the two machines) or reject if both M_1 and M_2 reject. So, there's no chance that it will loop. $\square$

9 Variants of Turing Machines (3.2)

There are, of course, different variations of Turing machines. Some of these variants may have multiple tapes, or may use nondeterminism. The original Turing machine and its variants all have the same power in the sense that they recognize the same class of languages.

9.1 Multitape Turing Machine

A **multitape Turing machine** is like an ordinary Turing machine with several tapes. Each tape has its own head for reading and writing. Initially, the input appears on the first tape, and the other tapes start out blank. In order to accomodate for the fact that there are multiple tapes, we now introduce a new transition function which allows for reading, writing, and moving the heads on some or all of the tapes simultaneously:

$$\delta : Q \times \Gamma^k \mapsto Q \times \Gamma^k \times \{L, R, S\}^k$$

where k is the number of tapes. So, it follows that the expression

$$\delta(q_i, a_1, \ldots, a_k) = (q_j, b_1, \ldots, b_k, L, R, \ldots, L)$$

means that if the machine is in state q_i and heads 1 through k are reading the symbols a_1 through a_k, the machine goes to state q_j and writes symbols b_1 through b_k, and directs each head to move left or right (or stay put, S), as specified.

9.1.1 Multitape vs. Single-tape Turing Machines

Theorem 9.1

Every multitape Turing machine has an equivalent single-tape Turing machine.

Remark: This is Theorem 3.13 in the textbook. The proof can be found there.

(Idea.) To show equivalence, we need to do the following:

- Given a Turing machine, we can build a multitape Turing machine which recognizes the same language.
- Given a k-tape Turing machine, we can build a one-tape Turing machine recognizing the same language.

To do this, we can make use of a delimiter to keep tape contents separate, and then use a special symbol to indicate the location of each read/write head.

9.1.2 Turing-Recognizability

Corollary 9.1

A language is Turing-recognizable if and only if some multitape Turing machine recognizes it.

Proof. A Turing-recognizable language is recognized by an ordinary (single-tape) Turing machine, which is a special case of a multitape Turing machine. This proves one direction of the corollary. The other direction comes as part of the proof of the above theorem. □

9.2 Nondeterministic Turing Machines

A **nondeterministic Turing machine** is defined in the expected way. At any point in a computation, the machine can proceed according to several possibilities. The transition function, then, has the form

$$\delta : Q \times \Gamma \mapsto \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

Like a nondeterministic finite automaton, if a computation ends up at an accept state, then the machine as a whole accepts the input. If, after going through all possibilities, the machine doesn't reach an accept state, then the machine rejects the input.

9.2.1 Nondeterministic vs. Deterministic Turing Machines

Theorem 9.2

Every nondeterministic Turing machine has an equivalent deterministic Turing machine.

Proof. (Idea.) For the proof of equivalence, we need to show two things.

1. Given a Turing machine, we need to build a nondeterminism Turing machine recognizing the same language.
2. Given a nondeterminism Turing machine, we need to build a deterministic Turing machine recognizing the same language.

To do this, we can try all possible branches of the nondeterministic computation. We have three tapes: a "read-only" input tape, a simulation tape, and a tape tracking nondeterminism branching. □

9.2.2 Turing-Recognizability

Corollary 9.2

A language is Turing-recognizable if and only if some nondeterministic Turing machine recognizes it.

Proof. Any deterministic Turing machine is automatically a nondeterministic Turing machine, so one direction of the corollary follows immediately. For the other direction, we can consider the proof of the theorem above. □

9.2.3 Turing-Decidable

The proof of the theorem can be modifier so that if the nondeterministic Turing machine always halts on all branches of its computation, then its corresponding deterministic Turing mahcine will as well.

Corollary 9.3

A language is decidable if and only if some nondeterministic Turing machine recognizes it.

9.3 Enumerators

Loosely defined, an enumerator is a Turing machine with an attached printer. Then, the Turing machine can use that printer as an output device to print strings out. Every time the Turing machine wants to ad a string to the list, it sends that string to the printer.

An enumerator E starts with a blank tape on its work tape. Now, if the enumerator doesn't half, then it may print an infinite list of strings. The language enumerated by E is the collection of all the strings that it will eventually print out. Moreover, E may generate the strings of the langauge in *any order*, possibly *with repetition*.

For any given Σ, there is an enumerator whose language is the set of all strings over Σ. To see this, we do standard string ordering. In particular:

- Order strings first by length.
- Then, by dictionary order.

9.3.1 Turing-Recognizability

Theorem 9.3

A language is Turing-recognizable if and only if some enumerator enumerates it.

Proof. First, we want to show that if we have an enumerator E that enumerates a language A, then a Turing machine A recognizes A. To describe M, we note that on some input w, we:

1. Run E. Every time that E outputs a string, compare it with w.
2. If w ever appears in the output of E, accept it.

Clearly, M accepts those strings that appear on E's list. To show the other direction, note that if a Turing machine M recognizes a language A, then we can construct the following enumerator E for A. Say that $s_1, s_2, s_3 \dots$ is a list of all possible strings in Σ^*. Then, for some input w, we can describe E like so:

1. Ignore the input.
2. Repeat the following for $i = 1, 2, 3, \dots$:
   - Run M for i steps on each input $s_1, s_2, \dots, s_i$.
   - If any computations accept, print out the corresponding s_j.

If M accepts a particular string w, then it will eventually appear on the list generated by E. In fact, it will appear on the list infinitely many times because M runs from the beginning on each string for each repetition of step 1. This procedures gives the effect of running M in parallel on all possible input strings. □

9.4 Summary

Suppose M is a Turing machine and E is an enumerator. Suppose that L is a language and w is a string.

	M Recognizes L	M Decides L	E Enumerates L
$w \in L$	Accept	Accept.	Prints w at some point.
$w \notin L$	Reject or Loop.	Reject.	Never prints w.

10 Decidability (4.1)

In this section we give some examples of languages that are decidable by algorithms.

A computational problem is **decidable** if and only if the language encoding the problem instances is decidable.

10.1 Encoding Inputs

First, we should mention that not all inputs are strings. In past examples and problems, we've generally worked with strings in a set alphabet. But, what if our input is an array of integers?

By definition, all Turing machine inputs are *strings*. If we're given an input that isn't a string, we need to **encode** the input (represent it as a string) first.

Before we discuss further, here is some notation:

- $\langle O \rangle$ is the string that represents, or encodes, the object O (O could be, for example, an integer, an array, etc.)
- $\langle O_1, O_2, \ldots, O_n \rangle$ is the single string that represents the tuple of objects $O_1, O_2, \ldots, O_n$.

In this way, problems that we care about can be reframed as languages of strings.

10.1.1 Example: Encoding Inputs

- Recognizes whether a string is a palindrome.

  $$\{w \mid w \in \{0,1\}^* \text{ and } w = w^R\}$$

  No encoding is necessary since w is a bitstring.

- Check whether a string is accepted by a DFA.

  $$\{\langle B, w\rangle \mid B \text{ is a DFA over } \Sigma, w \in \Sigma^*, w \in L(B)\}$$

  Here, we encode the pair B and w, where

  - B is a DFA.
  - w is a string.

  Essentially, for this language, saying that it's decidable is equivalent to checking if the string is accepted by the DFA.

  As a more general example, suppose we have a JSON file describing a DFA and a string. Then, we want to create an algorithm that checks whether this string is accepted by this DFA.

- Check whether the language of a PDA is infinite.

  $$\{\langle A\rangle \mid A \text{ is a PDA and } L(A) \text{ is infinite}\}$$

  Here, A is an encoding of a PDA.

10.2 Computational Problems

We now go over some example computational problems that are decidable.

10.2.1 Example: Recognizing Input Strings

Consider, again, the following language

$$A_{\text{DFA}} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}$$

This language contains the encodings of all DFAs together with strings that the DFAs accept. Then, the problem of testing whether the DFA B accepts an input w is the same as the problem of testing whether $\langle B, w\rangle \in A_{\text{DFA}}$.

Theorem 10.1

A_{DFA} is a decidable language.

Proof. (Idea.) We simply need to present a Turing machine M that decides A_{DFA}. Let M be the Turing machine which takes in an input $\langle B, w\rangle$, where B is a DFA and w is a string, and does the following:

1. Simulate B on input w (by keeping track of states in B, the transition function of B, etc.)[a]
2. If the simulation ends in an accept state, then accept. If it ends in a nonaccepting state, then reject.

A reasonable implementation of B is simply a list of its five components, Q, Σ, δ, q_0, and F. Then, when M receives its input, M first determines whether it properly represents a DFA B and a string w. If not, M rejects. Otherwise, M carries out the simulation directly; it keeps track of B's current state and B's current position in the input w by writing this information to its tape. Initially, B's current state is q_0, and B's current input position is the leftmost symbol of w. The states and position are updated according to the specified transition function δ. When M finishes processing the last symbol of w, it checks whether B is in an accept state. If so, M accepts. Otherwise, M rejects. □

[a]It's like writing a program that simulates the DFA.

10.2.2 Example: Emptiness Test

Consider the following language

$$E_{\text{DFA}} = \{\langle A\rangle \mid A \text{ is a DFA and } L(A) = \emptyset\}$$

Here, we want to determine whether a finite automaton accepts any strings at all.

Theorem 10.2

E_{DFA} is a decidable language.

Proof. A DFA accepts some string if and only if reaching an accept state from the start state by traveling along the arrows of the DFA is possible. To test this condition, we can design a TM T that uses a marking algorithm similar to what we used in the previous sections. For an input A, where A is a DFA, we describe the TM T as follows:

1. Check whether A is a valid encoding of a DFA; if not, reject.
2. Mark the start state of A.
3. Repeat until no new states get marked:
    - Loop over states of A and mark any unmarked state that has an **incoming** edge from a marked state.

4. If no accept state is marked, *accept*. Otherwise, *reject*.

This concludes the proof. □

10.2.3 Example: Equivalent DFAs

Consider the following language

$$EQ_{\text{DFA}} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$$

Here, this language contains the encodings of all equivalent DFAs.

Theorem 10.3

EQ_{DFA} is a decidable language.

Proof. To prove this theorem, we make use of the previous theorem. We construct a DFA C from A and B, where C accepts only those strings that are accepted by either A and B but *not* by both. Thus, if A and B recognize the same language, then C will accept nothing. Recall that, for two sets X and Y:

$$X = Y \iff \left(X \cap \overline{Y}\right) \cup \left(Y \cap \overline{X}\right) = \emptyset$$

Using this as inspiration, the language of C is then given by:

$$L(C) = \left(L(A) \cap \overline{L(B)}\right) \cup \left(\overline{L(A)} \cap L(B)\right)$$

This expression is sometimes called the **symmetric difference** of $L(A)$ and $L(B)$. Then, we can build a new DFA that recognizes the symmetric difference of A and B. Then, we can check if this set is empty by using the example from the previous section. If so, then we *accept*. Otherwise, we *reject*.

The construction of this Turing machine is as follows; define a Turing machine M_3 that takes in the input $\langle A, B\rangle$ and does the following:

1. Check whether A and B are valid encodings of DFAs; if not, reject.
2. Construct a new DFA, D, from A and B using the algorithms for complementing and taking union of regular languages such that

$$L(D) = \text{Symmetric difference of } A \text{ and } B$$

3. Run the machine from the previous example on D.
4. If the machine from the previous example accepts, then accept. If it rejects, then reject.

Therefore, we are done. □

10.2.4 Example: Regular Expressions and DFAs

Consider the following language

$$L = \{\langle R, D\rangle \mid R \text{ is a regular expression, } D \text{ is a DFA, and } L(R) = L(D)\}$$

We now show that this language is decidable.

Proof. (Idea.) To show that this language is decidable, we construct a Turing machine. For input $\langle R, D\rangle$, where R is some arbitrary regular expression and D is some arbitrary DFA, the Turing machine M_4 will do the following:

1. Convert R to a DFA D_R.
2. Check if $L(D_R) = L(D)$.
   - We have reduced this problem to EQ_{DFA}.
   - Note that EQ_{DFA} is decidable as EQ_{DFA} depends on E_{DFA} and E_{DFA} is decidable.
   - Since E_{DFA} is decidable, we can check the symmetric difference of D_R and D and see if it is empty by using E_{DFA}.

Therefore, L is decidable. □

10.2.5 Example: 10 Steps in a Turing Machine

Consider the following language

$$L = \{\langle M, w\rangle \mid M \text{ is a Turing machine which runs for at least 10 steps on input } w\}$$

We now show that this langauge is decidable.

Proof. (Idea.) Run M on w for 9 steps. If M accepts or rejects after at most 9 steps, then we can reject (as the machine didn't run at least 10 steps). Otherwise, accept. □

10.3 Techniques

The techniques we have discussed are as follows:

- **Subroutines:** We can use decision procedures of decidable problems as subroutines in other algorithms.
- **Constructions:** We can use algorithms for constructions as subroutines in other algorithms.
  - Converting DFA to DFA recognizing complement or Kleene star.
  - Converting two DFAs/NFAs to one recognizing union, intersection, concatenation.
  - Converting NFA to equivalent DFA.
  - Converting regular expression to equivalent NFA.
  - Converting DFA to equivalent regular expression.

11 Undecidability (4.2)

We will now talk about several computationally unsolvable problems, along with proving that such problems are actually unsolvable.

11.1 Computational Problems

We begin by going over some examples of computational problems.

11.1.1 Simple Problems: DFAs

Recall the following examples from the previous notes.

- Check whether a string is accepted by a DFA.
$$A_{\text{DFA}} = \{\langle B, w\rangle \mid B \text{ is a DFA over } \Sigma, w \in \Sigma^*, w \in L(B)\}$$
- Check whether the language of a DFA is empty.
$$E_{\text{DFA}} = \{\langle A\rangle \mid A \text{ is a DFA and } L(A) = \emptyset\}$$
- Check whether the languages of two DFAs are equal.
$$EQ_{\text{DFA}} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$$

These are **all** decidable; we did this by constructing algorithms that could solve them.

11.1.2 More Complex Problems: Pushdown Automaton

Consider the following examples.

- Check whether a string is accepted by a PDA.
$$A_{\text{PDA}} = \{\langle B, w\rangle \mid B \text{ is a PDA over } \Sigma, w \in \Sigma^*, w \in L(B)\}$$
- Check whether the language of a PDA is empty.
$$E_{\text{PDA}} = \{\langle A\rangle \mid A \text{ is a PDA and } L(A) = \emptyset\}$$
- Check whether the languages of two PDAs are equal.
$$EQ_{\text{PDA}} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are PDAs and } L(A) = L(B)\}$$

Some of these are decidable, and some are not. We note that, because a PDA has a stack which can be infinite in size, we cannot just simulate a PDA with the infinite because there can be infinitely many options.

11.1.3 Complex Problems: Turing Machine

Consider the following examples.

- Check whether a string is accepted by a TM.
$$A_{\text{TM}} = \{\langle B, w\rangle \mid B \text{ is a TM over } \Sigma, w \in \Sigma^*, w \in L(B)\}$$
- Check whether the language of a TM is empty.
$$E_{\text{TM}} = \{\langle A\rangle \mid A \text{ is a TM and } L(A) = \emptyset\}$$
- Check whether the languages of two TMs are equal.
$$EQ_{\text{TM}} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are TMs and } L(A) = L(B)\}$$

These are all undecidable. There are no algorithms which can solve A_{TM}, for example.

11.2 A_{TM}: An Algorithm

We will now consider an algorithm for solving A_{TM}. For an input $\langle M, w\rangle$, define the Turing machine N as follows.

1. Simulate M on input w.
2. If M ever enters its accept state, then accept; if M ever enters its reject state, then reject.

Then, we can define

$$L(N) = A_{\text{TM}}$$

and say that N is a **recognizer** for A_{TM}. However, N is not a *decider* for A_{TM} because it's possible that the Turing machine M can *loop* (i.e. not accept or reject); if M loops, then so will N.

11.3 Diagonalization Method

We now talk about the diagonalization method as a way to show that A_{TM} is undecidable.

11.3.1 Proof

Proof. Assume towards a contradiction that A_{TM} is decidable. Then, call $M_{A_{\text{TM}}}$ the decider for A_{TM}. We can define $M_{A_{\text{TM}}}$ like so:

(**A**.) For input $\langle M, w\rangle$, where M is a Turing machine and w is a string, we can define the Turing machine $M_{A_{\text{TM}}}$ like so:

- If w is in $L(M)$, i.e. if M accepts w, then halt and *accept*.
- If w is not in $L(M)$, i.e. if M rejects w, then halt and *reject*.

Next, we can construct a new Turing machine D, which uses $M_{A_{\text{TM}}}$ as a subroutine:

(**B**.) For some input $\langle M\rangle$, we can define the Turing machine D as follows:

- Run $M_{A_{\text{TM}}}$ on $\langle M, \langle M\rangle\rangle$ (here, $w = \langle M\rangle$).
- If $M_{A_{\text{TM}}}$ (in the previous step) accepts, then D rejects. Likewise, if $M_{A_{\text{TM}}}$ (in the previous step) rejects, then D accepts.

Now, suppose we run D on input $\langle D\rangle$. Because D is a decider, then either the computation halts and accepts, or the computation halts and rejects. Now:

- If $D(\langle D\rangle)$ accepts, then $M_{A_{\text{TM}}}$ with $\langle D, \langle D\rangle\rangle$ rejects[a]. But, we note that $M_{A_{\text{TM}}}$ with $\langle D, \langle D\rangle\rangle$ rejects if D rejects $\langle D\rangle$[b]. So, it follows that $D(\langle D\rangle)$ rejects.
- If $D(\langle D\rangle)$ rejects, then $M_{A_{\text{TM}}}$ with $\langle D, \langle D\rangle\rangle$ accepts[c]. But, we note that $M_{A_{\text{TM}}}$ with $\langle D, \langle D\rangle\rangle$ accepts if D accepts $\langle D\rangle$[d]. So, it follows that $D(\langle D\rangle)$ accepts.

This is a contradiction since D cannot accept its own input and cannot reject its own input and cannot halt. □

[a]by the first step implying the second step of **B**
[b]by the second step of **A**
[c]by the first step implying the second step of **B**
[d]by the first step of **A**

11.3.2 Another Way of Seeing It

Recall that we can list all of the Turing machines since they are countable:

$$M_1, M_2, M_3, \ldots$$

Suppose we also have a list of *distinct* inputs, call them:

$$w_1, w_2, w_3, \ldots$$

We can create this two-dimensional table where $A_{i,j}$ is defined to be the cell $[i, j]$, which is the result of running M_i on input w_j; that is, the cell will either be 1 if $w_J \in L(M_i)$ and 0 if not.

	w_1	w_2	w_3	$\ldots$
M_1	1	0	1	
M_2	0	0	1	
M_3				
$\vdots$				

Note that we made up these entries. Suppose we take the opposite of the diagonals:

	w_1	w_2	w_3	$\ldots$
M_1	**0**	0	1	
M_2	0	**1**	1	
M_3				
$\vdots$				

We claim that there is no Turing machine M such that $M(w_j)$ accepts if and only if $A_{j,j} = 0$ for all j. So, if M_1 accepted w_1, then this Turing machine will reject w_1; if M_2 rejected w_2, then this Turing machine will accept w_2.

Proof. We know that $M \neq M_i$ because $M(w_i) \neq M_i(w_i)$. □

The proof in the previous section is simply what we have here when $w_i = \langle M_i \rangle$.

11.4 Summary of this Algorithm

So, we have shown that this language is not decidable. Note that this language is also recognizable. As a general fact, a language is decidable if and only if it and its complement are both recognizable. Additionally, the complement of A_{TM} is unrecognizable.

12 Reductions and the Halting Problem (5.1)

We now begin by talking about reductions. Before we talk about reductions, consider the following statements:

- If problem X is no harder than problem Y, and if Y is easy, then X must also be easy.

> To get the intuition behind this, suppose you have two numbers x and y such that x is no greater than y; that is, $x \leq y$. Suppose y is negative. Then, x must also be negative (it cannot be positive).
>
> The same idea holds here. If we have two problems X and Y such that X is no harder than (or at least as easy as) Y and Y is easy, then X cannot be harder than Y; X has to be *as easy as, or easier than,* Y.

- If problem X is no harder than problem Y, and if X is hard, then Y must also be hard.

> Again, to get the intuition behind this, suppose you have two numbers x and y such that x is no greater than y; that is, $x \leq y$. Suppose x is positive. Then, y must also be positive (it cannot be negative).
>
> The same idea holds here. If we have two problems X and Y such that X is no harder than (or at least as easy as) Y and X is hard, then Y cannot be easier than X; Y has to be *as hard as, or harder than,* X.

We can now translate this to *reductions* specifically, especially in terms of decidability. That is:

- If problem X is no harder than problem Y, and if Y is **decidable**, then X must also be **decidable**.
- If problem X is no harder than problem Y, and if X is **undecidable**, then Y must also be **undecidable**.

When we say that problem X is no harder than problem Y, what we really mean is: given an algorithm deciding problem Y, we can solve problem X. We will use some guaranteed solution for problem Y to solve problem X. And, so really, what we're doing is: *design an algorithm that decides problem X, but allow it to use the subroutine that decides problem Y.* We also know that, if problem X is undecidable, then such an algorithm that decides problem Y cannot exist, a *contradiction.*

12.1 Reduction in a Nutshell

So, when we say that problem X *reduces to* problem Y, we mean that:

- Problem X is no harder than problem Y, which means that
- Given an algorithm deciding problem Y, we can solve problem X, which means that
- Given a solution for problem Y, we have a solution for problem X.

We can think about this in relation to packages. Suppose you wanted to solve some problem X. You have access to a package that can solve problem Y. Then, you can download this package and use this package to help solve problem X.

12.1.1 More General Examples of Reduction

Consider the following statements.

- Σ^* reduces to A_{TM}.

This statement is *true*. The problem essentially says, if you were given an algorithm to decide A_{TM}, then you can use that algorithm to decide Σ^*. Now, of course, you don't need to use A_{TM} at all since Σ^* is decidable.

- E_{DFA} reduces to $\emptyset$.

This statement is also *true*. The problem is saying, if you are given an algorithm to decide $\emptyset$, then you can use that algorithm to decide E_{DFA}. Now, of course, you don't need to use $\emptyset$ at all since $\emptyset$ is decidable to begin with.

- A_{TM} reduces to $\{\texttt{aa}\}$.

This statement is *false*. Suppose this statement is true; then, this problem is saying that if you have a way to decide $\{\texttt{aa}\}$, then you have a way to decide A_{TM}. Well, we know that A_{TM} is undecidable to begin with, so the statement is entirely false.

So, essentially, any decidable problem can be reduced to a decidable problem. However, no undecidable problem can be reduced to a decidable problem.

12.2 The Halting Problem

Consider the **Halting problem**, defined by

$$\text{HALT}_{\text{TM}} = \{(\langle M \rangle, w) \mid M \text{ is a TM and } M \text{ halts on input } w\}$$

We can actually make use of the fact that A_{TM} is undecidable to show that HALT_{TM} is undecidable.

12.2.1 Relationship Between the Halting Problem and A_{TM}

To see the difference between the two problems, we note that:

- HALT_{TM} is the language of all Turing machines M that *halt* on input w. In other words, M only needs to accept or reject w.
- A_{TM} is a language of all Turing machines M that *accept* w.

In this sense, A_{TM} is more strict than HALT_{TM}. Therefore, A_{TM} is a subset[5] of HALT_{TM}.

12.2.2 Proving that the Halting Problem is Undecidable

For our proof, we can use the following outline.

- Suppose, by way of contradiction, that HALT_{TM} is decidable.
- Then, assume that we have a machine G which decides HALT_{TM}.
- Build an algorithm that uses G as a subroutine that decides A_{TM}.
- However, this is impossible.
- So, G cannot possibly exist and hence HALT_{TM} is undecidable.

With this in mind, our proof is as follows.

[5]Note that being a subset means nothing in terms of computational complexity/decidability.

Proof. Suppose, by way of contradiction, that HALT_{TM} is decidable. Then, assume that we have a machine G which decides HALT_{TM}. We can construct a new Turing machine $M_{A_{\text{TM}}}$ to decide A_{TM} like so:

On input $\langle M, w\rangle$, where M is a Turing machine and w is some input, define the Turing machine $M_{A_{\text{TM}}}$ like so:

1. Run G on input $\langle M, w\rangle$.
2. If G rejects, then reject[a]. Otherwise, run M on w until it halts[b].
   (a) If this computation accepts, then accept.
   (b) If this computation rejects, then reject.

[a]We have this condition because it's possible that M with w as its input can loop. So, if M with w loops, then $G(\langle M, w\rangle)$ will reject as G is a decider. Thus, it follows that $M_{A_{\text{TM}}}$ also rejects.
[b]Since, now, we've established that M cannot loop.

To show correctness, note that:

- If M with input w accepts, then $G(\langle M, w\rangle)$ accepts, so we run $M(w)$ which accepts, so $M_{A_{\text{TM}}}$ also accepts.
- If M with input w rejects, then $G(\langle M, w\rangle)$ accepts since it at least halted, so we run $M(w)$ which rejects, so $M_{A_{\text{TM}}}$ also rejects.

Clearly, if G decides HALT_{TM}, then $M_{A_{\text{TM}}}$ decides A_{TM}. But, because A_{TM} is undecidable, then it follows that HALT_{TM} is also undecidable. □

12.3 E_{TM} is Undecidable

Suppose we wanted to check whether the language of a TM is empty[6]. Recall that this problem was described like so:

$$E_{\text{TM}} = \{\langle A\rangle \mid A \text{ is a TM and } L(A) = \emptyset\}$$

We can use proof by reduction to show that this problem is undecidable. Now, to use a proof by reduction to show that E_{TM} is undecidable, we need to reduce an undecidable set *to* E_{TM}.

12.3.1 Proving that E_{TM} is Undecidable

For our proof, we can use the following outline. Our goal is to show that A_{TM} reduces to E_{TM}. To do so, we want to show that A_{TM} reduces to E_{TM}.

- Assume that we have a Turing machine, R, that decides E_{TM}.
- Build a new Turing machine, $M_{A_{\text{TM}}}$, that decides A_{TM}.
  - This new Turing machine always halts.
  - Accept an input $\langle M, w\rangle$ if and only if w is in $L(M)$.

12.3.2 Attempt 1: An Incorrect Proof

Consider the following purported proof.

Suppose by way of contradiction that E_{TM} is decidable. Then, we can build a Turing machine, R, that decides E_{TM}. We can construct a new Turing machine $M_{A_{\text{TM}}}$ that decides A_{TM} like so:

[6]In other words, we want to recognize codes of Turing machines that always reject or loop.

On input $\langle M, w\rangle$, where M is a Turing machine and w is an input, define the Turing machine $M_{A_{\text{TM}}}$ like so:

1. Run R on input $\langle M\rangle$. If this rejects, then reject.
2. If R accepts, then run M on input w. If this accepts, then accept; if this rejects, then rejects.

Clearly, if R decides E_{TM}, then $M_{A_{\text{TM}}}$ decides A_{TM}. But, because A_{TM} is undecidable, then it follows that E_{TM} is also undecidable.

This solution **fails** when:

1. $M(w)$ loops.
2. $R(\langle M\rangle)$ accepts.

Regarding the second point, the reason why is because if we run R on some input $\langle M\rangle$, and if it rejects, then this implies that the language is not empty. If it accepts, then the language is empty. So, in other words, by saying that R accepts, this implies that the language is empty. If the language is empty, then there is no change that w can be in the language. So, by the definition of $M_{A_{\text{TM}}}$ (which is that the string w must be in $L(M)$), we can reject immediately.

Now, the case that we're not sure about is when R rejects. In particular, when we have R, we're not even considering w.

12.3.3 Attempt 2: Correct Proof

The trick that we can do is to design a new Turing machine that "hard-codes" both M and w in it[7]. Thus, the correct proof is as follows:

Proof. Suppose by way of contradiction that E_{TM} is decidable. Then, we can build a Turing machine, R, that decides E_{TM}. We can construct a new Turing machine $M_{A_{\text{TM}}}$ that decides A_{TM} like so:

On input $\langle M, w\rangle$, where M is a Turing machine and w is an input, define the Turing machine $M_{A_{\text{TM}}}$ like so:

1. First, *build* the Turing machine X which, on input x, ignores x and simulates M on w. In other words, we can think of it like:

```
function X(x):
    return M(w)
```

2. Run R on $\langle X\rangle$. If it accepts, then reject. If it rejects, then accept.

To show correctness, we note that:

- If $M(w)$ accepts, then for all x, $X(x)$ accepts, so $L(X) = \Sigma^*$. In particular, this means that $L(X)$ is not empty, so $R(\langle X\rangle)$ rejects and we accept.
- If $M(w)$ rejects (or loops), then for all x, $X(x)$ rejects (or loops), so $L(X) = \emptyset$. In particular, this means that $L(X)$ is empty, so $R(\langle X\rangle)$ accepts and we reject.

Clearly, if R decides E_{TM}, then $M_{A_{\text{TM}}}$ decides A_{TM}. But, because A_{TM} is undecidable, then it follows that E_{TM} is also undecidable. □

[7]Another way of thinking about this is, if w is our input, then we're essentially just creating a new Turing machine where we hardcode w into M.

12.4 L_{FINITE} is Undecidable

Suppose we wanted to show that the following language is undecidable:

$$L_{\text{FINITE}} = \{\langle M \rangle \mid M \text{ is a Turing machine and } L(M) \text{ is finite}\}$$

We again make use of proof by reduction to show that this problem is undecidable.

12.4.1 Proving that L_{FINITE} is Undecidable

We will show that A_{TM} can be reduced to L_{FINITE}.

Proof. Suppose, for the sake of contradiction, that L_{FINITE} is decidable. Then, we can build a Turing machine M_F which decides L_{FINITE}. Now, we construct a new Turing machine $M_{A_{\text{TM}}}$ that decides A_{TM} like so:

On input $\langle M, w \rangle$, where M is an arbitrary Turing machine and w is an arbitrary input, define the Turing machine $M_{A_{\text{TM}}}$ like so:

1. Let M' be the Turing machine such that, on each input x, ignore x and simulate M on w.
2. Run M_F on $\langle M' \rangle$. If M_F rejects, then accept. Otherwise, reject.

To show correctness, note that:

- For any $\langle M, w \rangle \in A_{\text{TM}}$, M accepts w. By the construction of M', when M accepts w, M' will accept every input string x. Therefore, $L(M')$ must be infinite. Then, M_F must reject $\langle M' \rangle$ and so $M_{A_{\text{TM}}}$ must accept $\langle M, w \rangle$.
- For each $\langle M, w \rangle \notin A_{\text{TM}}$, M does not accept w. Again, by the construction of M', when M doesn't accept w, M' accepts nothing. Therefore, $L(M') = \emptyset$ and so it's finite. From there, M_F accepts $\langle M' \rangle$ and so $M_{A_{\text{TM}}}$ rejects $\langle M, w \rangle$.

Since M_F is a decider and $M_{A_{\text{TM}}}$ halts when M_F halts, it follows that $M_{A_{\text{TM}}}$ is also a decider. Thus, we have a decider that decides A_{TM}. However, as A_{TM} is undecidable, we have a contradiction. Therefore, L_{FINITE} must be undecidable. □

12.5 L_{INFINITE} is Undecidable

Suppose we wanted to show that the following language is undecidable:

$$L_{\text{INFINITE}} = \{\langle M \rangle \mid M \text{ is a Turing machine and } L(M) \text{ is infinite}\}$$

We again make use of proof by reduction to show that this problem is undecidable.

12.5.1 Proving that L_{INFINITE} is Undecidable

Suppose we wanted to show that problem L_{INFINITE} is undecidable.

Proof. Assume by way of contradiction that L_{INFINITE} is decidable. Then, we can build a Turing machine, M_I that decides L_{INFINITE}. We can construct a new Turing machine $M_{A_{\text{TM}}}$ that decides A_{TM} like so:

On input $\langle M, w \rangle$, where M is some arbitrary Turing machine and w is some arbitrary input, define the Turing machine $M_{A_{\text{TM}}}$ like so:

1. Let X be another Turing machine such that, on each input x, ignore x and simulate M on w.
2. Run M_I on $\langle X \rangle$. If M_I accepts, then accept. Else, reject.

To show correctness, note that:

- For any $\langle M, w\rangle \in A_{\text{TM}}$, M accepts w. By the construction of X, when M accepts w, then X will accept every input string x. Therefore, $L(X)$ must be infinite. Then, M_I accepts $\langle X\rangle$ and so $M_{A_{\text{TM}}}$ accepts $\langle M, w\rangle$.
- For any $\langle M, w\rangle \notin A_{\text{TM}}$, M does not accept w. By the construction of X, when M does not accept w, then X will accept nothing. So, it follows that $L(X)$ is precisely just the empty set and so it is finite. Then, M_I rejects $\langle X\rangle$ and so $M_{A_{\text{TM}}}$ rejects $\langle M, w\rangle$.

Since $M_{L_{\text{INFINITE}}}$ is a decider and $M_{A_{\text{TM}}}$ halts when $M_{L_{\text{INFINITE}}}$ halts, it follows that $M_{A_{\text{TM}}}$ is also a decider. However, as A_{TM} is undecidable, we have a contradiction. Therefore, L_{INFINITE} is undecidable. □

12.6 L_{LOOP} is Undecidable

Suppose we wanted to show that the following language is undecidable:

$$L_{\text{LOOP}} = \{\langle M\rangle \mid M \text{ is a Turing machine which loops on some string.}\}$$

We again make use of proof by reduction to show that this problem is undecidable. However, instead of reducing down to A_{TM}, we will instead reduce down to HALT_{TM}.

12.6.1 Proving that L_{LOOP} is Undecidable

Suppose we wanted to show that problem L_{LOOP} is undecidable.

Proof. Assume by way of contradiction that L_{LOOP} is decidable. Then, we can build a Turing machine, $M_{L_{\text{LOOP}}}$ that decides L_{LOOP}. We can construct a new Turing machine M_{HALT} that decides HALT_{TM} like so:

On input $\langle M, w\rangle$, where M is some arbitrary Turing machine and w is some arbitrary input, define the Turing machine M_{HALT} like so:

1. Let X be another Turing machine such that, on each input x, ignore x and simulate M on w. If M halts, then accept.
2. Run $M_{L_{\text{LOOP}}}$ on $\langle X\rangle$. If $M_{L_{\text{LOOP}}}$ rejects (i.e. $\langle X\rangle$ halted), then accept. Else, reject.

To show correctness, note that:

- For any $\langle M, w\rangle \in \text{HALT}_{\text{TM}}$, M accepts w, implying that M halted on input w. By the construction of X, when M accepts w, then for any input x, X clearly cannot loop. Then, $M_{L_{\text{LOOP}}}$ rejects $\langle X\rangle$ and so M_{HALT} accepts $\langle M, w\rangle$.
- For any $\langle M, w\rangle \notin A_{\text{TM}}$, M does not accept w, implying that M did not halt on input w. By the construction of X, when M does not accept w, then for any input x, X must have looped. So, $M_{L_{\text{LOOP}}}$ accepts $\langle X\rangle$ and so M_{HALT} rejects $\langle M, w\rangle$.

Since $M_{L_{\text{LOOP}}}$ is a decider and M_{HALT} halts when $M_{L_{\text{LOOP}}}$ halts, it follows that M_{HALT} is also a decider. However, as HALT_{TM} is undecidable, we have a contradiction. Therefore, L_{LOOP} is undecidable. □

12.7 L_2 is Undecidable

Suppose we want to show that the following language is undecidable:

$$L_2 = \{\langle M\rangle \mid M \text{ is a Turing machine and } |L(M)| = 2\}$$

To show that L_2 is undecidable, we again make use of proof by reduction.

12.7.1 Proving that L_2 is Undecidable

We will show that A_{TM} can be reduced to L_2.

Proof. Assume by way of contradiction that L_2 is decidable. Then, we can build a Turing machine, T_{L_2} that decides L_2. We can construct a new Turing machine $M_{A_{\text{TM}}}$ that decides A_{TM} like so:

On input $\langle M, w\rangle$, where M is some arbitrary Turing machine and w is some arbitrary input, define the Turing machine $M_{A_{\text{TM}}}$ like so:

1. Let X be another Turing machine such that, on each input x, ignore x and simulate M on w. If M accepts w and $x \in \{0, 1\}$, then accept. Otherwise, reject.
2. Run M_{L_2} on $\langle X\rangle$. If M_{L_2} accepts, then accept. Else, reject.

To show correctness, note that:

- For any $\langle M, w\rangle \in A_{\text{TM}}$, M accepts w. By the construction of X, when M accepts w, $L(X) = \{0, 1\}$ which shows that $|L(X)| = 2$. Then, M_{L_2} accepts $\langle X\rangle$ and so $M_{A_{\text{TM}}}$ accepts $\langle M, w\rangle$.
- For any $\langle M, w\rangle \notin A_{\text{TM}}$, M does not accept w. By the construction of X, when M does not accept w, $L(X) = \emptyset$, implying that the size of $L(X)$ is 0. Then, M_{L_2} rejects $\langle X\rangle$ and so $M_{A_{\text{TM}}}$ rejects $\langle M, w\rangle$.

Since M_{L_2} is a decider and $M_{A_{\text{TM}}}$ halts when M_{L_2} halts, it follows that $M_{A_{\text{TM}}}$ is also a decider. However, as A_{TM} is undecidable, we have a contradiction. Therefore, L_2 is undecidable. □

12.8 General Proof Template

Suppose we wanted to show that problem T is undecidable.

Proof. Assume by way of contradiction that T is decidable. Then, we can build a Turing machine, M_T that decides T. We can construct a new Turing machine $M_{A_{\text{TM}}}$ that decides A_{TM} like so:

On input $\langle M, w\rangle$, where M is some arbitrary Turing machine and w is some arbitrary input, define the Turing machine $M_{A_{\text{TM}}}$ like so:

1. Let X be another Turing machine such that, on each input x, . . . (refer to M and w and possibly x in some way).
2. Run M_T on $\langle X\rangle$. If M_T (accepts/rejects), then accept. Else, reject.

To show correctness, note that:

- For any $\langle M, w\rangle \in A_{\text{TM}}$, M accepts w. By the construction of X, when M accepts w, ... Then, M_T (accepts/rejects) $\langle X\rangle$ and so $M_{A_{\text{TM}}}$ accepts $\langle M, w\rangle$.
- For any $\langle M, w\rangle \notin A_{\text{TM}}$, M does not accept w. By the construction of X, when M does not accept w, ... Then, M_T (accepts/rejects) $\langle X\rangle$ and so $M_{A_{\text{TM}}}$ rejects $\langle M, w\rangle$.

Since M_T is a decider and $M_{A_{\text{TM}}}$ halts when M_T halts, it follows that $M_{A_{\text{TM}}}$ is also a decider. However, as A_{TM} is undecidable, we have a contradiction. Therefore, T is undecidable. □

Remarks:

- The goal is to use M_T to build a machine which will decide A_{TM}.
- The key is, in reduction proofs, we always need to build a new Turing machine X.
- You might need to consider the case when M_T halts. But, this is trivial as we assumed that M_T is a decider.