For many purposes, computation is elegantly modeled with simple mathematical objects:
Turing machines, finite automata, pushdown automata, and such.

Turing machines (TMs) make precise the otherwise vague notion of an “algorithm”: no more powerful precise account of algorithms has been found. (Church-Turing Thesis)

An easy consequence of the definition of TMs: most functions (say, from $\mathbb{N}$ to $\mathbb{N}$) are not computable (by TM-equivalent machines).

Why? There are more functions than there are TMs to compute them.

Remarkably, Turing machines also serve as a common platform for characterizing computational complexity, since the time cost, and space cost, of working with a TM is within a polynomial of the performance of conventional computers.

So, for instance, the most famous open question in computer science theory —

$$P \overset{?}{=} NP$$

— while formulated in terms of TMs, is important for practical computing.
Undergraduate study of the theory of computation typically starts with finite automata. They are simpler than TMs, and it is easier to prove main results about them. They are also intuitively appealing. And automata theory has wide practical importance, in compiler theory and implementation, for instance. (lex – lexical analysis)

The typical next step is to study grammars, which are also nicely intuitive, and already somewhat familiar to students. The theory of context-free grammars is central to compiler theory and implementation. (yacc – compiler generation)

Turing machines are only a little more complicated, but the theory of computation based on them is quite sophisticated and extensive.

One thing all these models have in common is that they formulate computation in terms of questions about *languages*.

In fact, many of the central questions are formulated in terms of **language recognition**: deciding whether a given string belongs to a given language.
“Regular” languages are relatively simple languages.

We’ll study means for “generating” regular languages and also for “recognizing” them.

All finite languages are regular.

Some infinite languages are regular.

Each regular language can be characterized by a (finite!) regular expression: which says how to generate the strings in the language.

It can also be characterized by a (finite!) finite state automaton: which provides a mechanism for recognizing the strings in the language.
11.1 Regular languages

A language over an alphabet $\Sigma$ is regular if it can be constructed from the empty language, the language $\{\Lambda\}$ and the singleton languages $\{a\}$ ($a \in \Sigma$) by a finite number of applications of union, language product and Kleene star.

Sounds like an inductively-defined set...

The set of regular languages over an alphabet $\Sigma$ is the least set of languages over $\Sigma$ s.t.
1. $\emptyset$ is a regular language,
2. $\{\Lambda\}$ is a regular language,
3. For all $a \in \Sigma$, $\{a\}$ is a regular language,
4. If $A$ is a regular language, so is $A^*$,
5. If $A$ and $B$ are regular languages, so are $A \cup B$ and $AB$.

Here, conditions 1–3 are the basis part of the inductive definition, and conditions 4 & 5 are the induction part.
The set of regular languages over an alphabet $\Sigma$ is the least set of languages over $\Sigma$ s.t.
1. $\emptyset$ is a regular language,
2. $\{\Lambda\}$ is a regular language,
3. For all $a \in \Sigma$, $\{a\}$ is a regular language,
4. If $A$ is a regular language, so is $A^*$,
5. If $A$ and $B$ are regular languages, so are $A \cup B$ and $AB$.

**Example**  \( \{00, 01, 10, 11\}^* \) is a regular language.

What is a “nice” description of this language?

Let’s check that it is regular:

$L_1 = \{0\}$ and $L_2 = \{1\}$ are regular.

Hence, $L_1L_1 = \{00\}$, $L_1L_2 = \{01\}$, $L_2L_1 = \{10\}$ and $L_2L_2 = \{11\}$ are regular.

It follows that $\{00\} \cup \{01\} = \{00, 01\}$ is regular,
as are $\{00, 01\} \cup \{10\} = \{00, 01, 10\}$ and
$\{00, 01, 10\} \cup \{11\} = \{00, 01, 10, 11\}$.

So $\{00, 01, 10, 11\}^*$ is regular.
11.1.1 Regular expressions

**Definition** The set of *regular expressions* over an alphabet $\Sigma$ is the least set of strings satisfying the following 5 conditions.

1. $\emptyset$ is a regular expression, and stands for the language $\emptyset$.
2. $\Lambda$ is a regular expression, and stands for the language $\{\Lambda\}$.
3. For all $a \in \Sigma$, $a$ is a regular expression, and stands for the language $\{a\}$.
4. If $R$ is a regular expression that stands for the language $A$, then $R^*$ is a regular expression standing for the language $A^*$.
5. If $R_1$ and $R_2$ are regular expressions that stand for the languages $A$ and $B$ respectively, then $(R_1 + R_2)$ and $(R_1 \cdot R_2)$ (or simply $(R_1 R_2)$ ) are regular expressions standing for the languages $A \cup B$ and $AB$, respectively.

For example

$((((00) + (01)) + (10)) + (11))^*$

is a regular expression that stands for the language

$\{00, 01, 10, 11\}^*$. 
We can omit many of the parentheses in regular expressions.

For instance, outermost parentheses can be safely dropped. So

\[(a + b) + c = (a + b) + c.\]

Notice that when we write “=” here, we are saying that the expressions stand for the same language (not that the two expressions are identical viewed as strings).

Union and product (concatenation) are associative, and so accordingly:

\[(a + b) + c = a + (b + c) = a + b + c\]

and

\[(ab)c = a(bc) = abc.\]
Precedence conventions help

Under the convention that the Kleene closure operator has the highest precedence, + the lowest, with language product in between, we also have

\[ ab^* + c^* d = (ab^*) + (c^* d) \]
\[ ab^* + c^* d \neq (ab)^* + c^* d \]
\[ ab^* + c^* d \neq a(b^* + c^*)d \]

So the regular expression

\[ (((00) + (01)) + (10)) + (11))^* \]

can be written

\[ (00 + 01 + 10 + 11)^* . \]

Another regular expression for this language is

\[ ((0 + 1)(0 + 1))^* . \]
Exponent notation is also convenient

We can also use exponent notation, much as before. Hence,

\[(a + b)(a + b) = (a + b)^2\]

and so forth. Of course, for every regular expression \(R\)

\[R^0 = \Lambda.\]

(Recall that the regular expression \(\Lambda\) stands for the language \(\{\Lambda\}\)!) 

As with the Kleene star, the precedence of exponentiation is higher than that of product and union, so

\[ab^2 = abb\]

and

\[a + b^2 = a + bb.\]

**Example** Find a binary string of minimum length among those *not* in the language characterized by the regular expression:

\[(0^* + 1^*)^2\]

\[1^*(01)^*0^*\]
§11.2 Recognizing a regular language

For regular languages, we address the language recognition problem roughly as follows. To decide whether a given string is in the language of interest...

- Allow a single pass over the string, left to right.
- Rather than waiting until the end of the string to make a decision, we make a tentative decision at each step. That is, for each prefix we decide whether it is in the language.

Question: How much do we need to remember at each step about what we have seen previously?

Everything? (If so we’re in trouble — our memory is finite, and strings can be arbitrarily long.)

Nothing? (Fine for $\emptyset$ and $\Sigma^*$.)

In general, we can expect there to be strings $x, y$ s.t. $x \in L$ and $y \notin L$. As we read these strings from left to right, we will need to remember enough to allow us to distinguish $x$ from $y$.

We’ll eventually give a “perfect” account of what must be remembered at each step — in terms of so-called “distinguishability”. First, let’s define finite automata and get a feeling for them...
11.2.1 Deterministic finite automata

**Definition**  A deterministic finite automaton (DFA) is a five-tuple

\[ M = (S, \Sigma, T, s_0, F) \]

where

- \( S \) is a finite set of “states”,
- \( \Sigma \) is an alphabet — the (finite) “input alphabet”,
- \( T : S \times \Sigma \rightarrow S \) is the “transition function”,
- \( s_0 \in S \) is the “initial state”,
- \( F \subseteq S \) is the set of “final” or “accepting” states.

**Example**  Before we work out all the details of this definition, let’s look at a diagram of a DFA that recognizes the language

\[ \{ x \in \{0, 1\}^* \mid x \text{ has an even number of 1's} \}. \]
A DFA $M = (S, \Sigma, T, s_0, F)$, where

- $S$ is a finite set of “states”,
- $\Sigma$ is an alphabet — the “input alphabet”,
- $T : S \times \Sigma \to S$ is the “transition function”,
- $s_0 \in S$ is the “initial state”,
- $F \subseteq S$ is the set of “final” or “accepting” states.

Intuitively, a DFA “remembers” (and “decides” about) the prefix it has read so far by changing state as it reads the input string.

Initially, (i.e. after having read the empty prefix of the input string), the DFA is in state $s_0$.

The DFA $M$ “accepts” the empty string iff

$$s_0 \in F.$$ 

If $a \in \Sigma$ is the first character in the input string, then the state of $M$ after reading that first character is given by $T(s_0, a)$.

$M$ “accepts” the string $a$ iff

$$T(s_0, a) \in F.$$
A DFA \( M = (S, \Sigma, T, s_0, F) \), where

- \( S \) is a finite set of “states”,
- \( \Sigma \) is an alphabet — the “input alphabet”,
- \( T : S \times \Sigma \rightarrow S \) is the “transition function”,
- \( s_0 \in S \) is the “initial state”,
- \( F \subseteq S \) is the set of “final” or “accepting” states.

Similarly, if \( a \in \Sigma \) is the first character in the input string and \( b \in \Sigma \) is the second, then the state of \( M \) after reading that second character is given by \( T(T(s_0, a), b) \).

\( M \) “accepts” the string \( ab \) iff

\[
T(T(s_0, a), b) \in F.
\]

This notation grows cumbersome as the input string grows longer.

We’d like a more convenient way to describe the state that DFA \( M \) is in after reading an input string \( x \). . .
Defining \( T^* : S \times \Sigma^* \rightarrow S \) recursively in terms of \( T : S \times \Sigma \rightarrow S \)

The transition function \( T \) takes a state \( s \) and an input symbol \( a \) and returns the resulting state

\[
T(s, a).
\]

Given \( T \), we recursively define the function \( T^* \) that takes a state \( s \) and an input string \( x \) and returns the resulting state

\[
T^*(s, x).
\]

**Definition**  Given a DFA \( M = (S, \Sigma, T, s_0, F) \), we define the multi-step transition function

\[
T^* : S \times \Sigma^* \rightarrow S
\]

as follows:

- **Basis:** For any \( s \in S \), \( T^*(s, \Lambda) = s \),
- **Induction:** For any \( s \in S \), \( x \in \Sigma^* \) and \( a \in \Sigma \),

\[
T^*(s, xa) = T(T^*(s, x), a).
\]

You may notice that, implicitly, we seem to have in mind a different inductive definition of \( \Sigma^* \) than we used previously...
“Backwards” inductive definition of $\Sigma^*$

It turns out to be more convenient for our purposes to use the following alternative inductive definition of $\Sigma^*$.

- **Basis**: $\Lambda \in \Sigma^*$.

- **Induction**: If $x \in \Sigma^*$ and $a \in \Sigma$, then $xa \in \Sigma^*$.

Previously we used an inductive definition of $\Sigma^*$ in which $x$ and $a$ were reversed (in the induction part).

That was in keeping with what is most convenient when working with lists. (Intuitively speaking, you build a list by adding elements to the front of the list.)

From now on, we will use this new “backwards” definition.

It seems reasonably intuitive for strings. (You write a string from left to right, right?)

It is clearly equivalent to the other inductive definition.

It is much more convenient for proving things about finite automata!
First sanity check for recursive defn of $T^*$

Recall: For DFA $M = (S, \Sigma, T, s_0, F)$, we define

$$T^* : S \times \Sigma^* \rightarrow S$$

as follows:

- **Basis**: For any $s \in S$, $T^*(s, \Lambda) = s$,
- **Induction**: For any $s \in S$, $x \in \Sigma^*$ and $a \in \Sigma$,

$$T^*(s, xa) = T(T^*(s, x), a).$$

Here's a property $T^*$ should have...

**Claim**  For any DFA $M = (S, \Sigma, T, s_0, F)$, and any $s \in S$ and $a \in \Sigma$,

$$T^*(s, a) = T(s, a).$$

This is easy to check...

$$T^*(s, a) = T^*(s, \Lambda a)$$
$$= T(T^*(s, \Lambda), a) \quad \text{(defn } T^*)$$
$$= T(s, a) \quad \text{(defn } T^*)$$
Second sanity check for recursive defn of $T^*$

$$T^* : S \times \Sigma^* \to S$$

$T^*(s, \Lambda) = s$ , for all $s \in S$

$T^*(s, xa) = T(T^*(s, x), a)$ , for all $s \in S$, $x \in \Sigma^*$, and $a \in \Sigma$

Here’s another sanity check on the definition of $T^*$: Roughly, does “reading $xy$” take you to the same state as “reading $x$ and then reading $y$”?

**Claim** For any DFA $M = (S, \Sigma, T, s_0, F)$, any $s \in S$, and $x, y \in \Sigma^*$,

$$T^*(s, xy) = T^*(T^*(s, x), y).$$

**Proof.** By structural induction on $y$ (via the “backwards” ind. defn of $\Sigma^*$).

**Basis:**

$$T^*(s, x\Lambda) = T^*(s, x)$$

$$= T(T^*(s, x), \Lambda) \quad \text{(defn $T^*$, $T^*(s, x) \in S$)}$$

**Induction:** $y \in \Sigma^*$, $a \in \Sigma$.

IH: $T^*(s, xy) = T^*(T^*(s, x), y)$.

NTS: $T^*(s, xya) = T^*(T^*(s, x), ya)$.

$$T^*(s, xy) = T(T^*(s, xy), a) \quad \text{(defn $T^*$)}$$

$$= T(T^*(T^*(s, x), y), a) \quad \text{(IH)}$$

$$= T^*(T^*(s, x), ya) \quad \text{(defn $T^*$)}$$
The language $L(M)$ recognized by an DFA $M$

Now that we have adequate notation for the state of a DFA after it reads an input string, we can define when a DFA “accepts” a string.

**Definition** For any DFA $M = (S, \Sigma, T, s_0, F)$, a string $x \in \Sigma^*$ is accepted by $M$ if

$$T^*(s_0, x) \in F.$$ 

The language recognized (or accepted) by $M$, denoted $L(M)$, is the set of strings accepted by $M$.

That is,

$$L(M) = \{ x \in \Sigma^* \mid T^*(s_0, x) \in F \}.$$ 

Notice that even if $L \subseteq L(M)$ we still do not say that $L$ is recognized by $M$ unless $L = L(M)$.

(That is, if DFA $M$ accepts language $L$, then not only does $M$ accept all strings from $L$ — it also accepts only those strings!)
Kleene’s Theorem

Recall that the regular languages over alphabet $\Sigma$ are exactly those that can be constructed (in a finite number of steps) using (binary) union, language product and Kleene closure, starting from the languages $\emptyset$, $\{\Lambda\}$, and $\{a\}$ for each $a \in \Sigma$.

Here is the remarkable theorem characterizing regular languages in terms of DFA’s:

**Kleene’s Theorem** A language is regular iff some DFA recognizes it.
DFA diagrams

It is convenient to represent a DFA as a diagram, as we have seen...

Recall: A DFA is a 5-tuple \((S, \Sigma, T, s_0, F)\).

For every state \(s \in S\) there is a corresponding node, represented by the symbol \(s\) with a circle around it:

\[
\begin{array}{c}
s\
\end{array}
\]

We indicate the initial state, \(s_0\) by an incoming arrow (with no “source”):

\[
\begin{array}{c}
s_0\
\end{array}
\]

We indicate that \(s \in F\) by adding a concentric circle:

\[
\begin{array}{c}
\bigcirc s\
\end{array}
\]

For any \(q, r \in S\) and \(a \in \Sigma\),

if \(T(q, a) = r\), there is a directed edge labeled \(a\) from \(q\) to \(r\):

\[
\begin{array}{c}
q \xrightarrow{a} r\
\end{array}
\]

Let’s draw a few DFA’s...
A DFA for $01^+$

We can write $r^+$ (where $r$ is a regular expression) to stand for $rr^*$.

(Recall: If $L$ is a language, we write $L^+$ to stand for $LL^*$.)
A DFA for \(((0 + 1)(0 + 1))\)^*
A DFA for \((0 + 1)^*10\)
A DFA for $0^*10^*(10^*10^*)^*$
Closure of Regular Languages under Operations

Recall A deterministic finite automaton is a five-tuple

\[ M = (S, \Sigma, T, s_0, F) \]

where:

- \( S \) is a finite set of “states”,
- \( \Sigma \) is an alphabet — the “input alphabet”,
- \( T : S \times \Sigma \rightarrow S \) is the “transition function”,
- \( s_0 \in S \) is the “initial state”,
- \( F \subseteq S \) is the set of “final” or “accepting” states.

And we defined the multi-step transition function

\[ T^* : S \times \Sigma^* \rightarrow S \]

as follows:

- **Basis**: For any \( s \in S \), \( T^*(s, \Lambda) = s \),
- **Induction**: For any \( s \in S \), \( x \in \Sigma^* \) and \( a \in \Sigma \),

\[ T^*(s, xa) = T(T^*(s, x), a) \].

A string \( x \in \Sigma^* \) is accepted by \( M \) if \( T^*(s_0, x) \in F \).

The language recognized by \( M \), denoted \( L(M) \), is the set of strings accepted by \( M \). That is,

\[ L(M) = \{ x \in \Sigma^* \mid T^*(s_0, x) \in F \} \].
Also recall The set of *regular languages* over an alphabet \( \Sigma \) is the least set of languages over \( \Sigma \) s.t.

1. \( \emptyset \) is a regular language,
2. \( \{\Lambda\} \) is a regular language,
3. For all \( a \in \Sigma \), \( \{a\} \) is a regular language,
4. If \( A \) is a regular language, so is \( A^* \),
5. If \( A \) and \( B \) are regular languages, so are \( A \cup B \) and \( AB \).

And the set of regular expressions is similarly defined, so that it is immediately clear that every regular expression stands for a (specific) regular language, and every regular language is represented by some regular expression.

Is every regular language represented by more than one regular expression?

For a given regular language \( L \), how many regular expressions stand for \( L \)?

How many regular languages are there (for a given alphabet \( \Sigma \))? (Related question: How many regular expressions are there?)
Complement construction for DFA’s

**Observation**  By the inductive definition of regular languages, we see that the set of regular languages is closed under (finite) union, product and Kleene closure.

**Theorem**  The regular languages are closed under set complement.

*Proof.* Consider any regular language $L$ over $\Sigma$. By Kleene’s Theorem, $L$ is accepted by some DFA

$$M = (S, \Sigma, T, s_0, F).$$

Let

$$M' = (S, \Sigma, T, s_0, S - F).$$

DFA’s $M$ and $M'$ differ only on their sets of accepting states, which are complements. In particular, both DFA’s have the same multi-step transition function $T^* : S \times \Sigma^* \rightarrow S$. Hence, for any $x \in \Sigma^*$,

$$M \text{ accepts } x \iff T^*(s_0, x) \in F$$

while

$$M' \text{ accepts } x \iff T^*(s_0, x) \notin F.$$

That is, $M$ accepts $x$ iff $M'$ doesn’t. So we see that $M'$ accepts the language $L'$ (that is, $\Sigma^* - L$). By Kleene’s Theorem we can conclude that the regular languages are closed under set complement.
Theorem  The regular languages are closed under set intersection and set difference.

Proof. We know (by the inductive defn) that the regular languages are closed under union, and we have shown (by DFA construction) that the regular languages are also closed under complement. Since

\[ A \cap B = (A' \cup B')' \]

and

\[ A - B = A \cap B', \]

we can conclude that the set of regular languages is also closed under set intersection and set difference.
Question  Given DFA’s $M_1$ and $M_2$, can we construct DFA’s to accept

- $L(M_1) \cup L(M_2)$?
- $L(M_1)L(M_2)$?
- $L(M_1)^*$?
- $L(M_1) \cap L(M_2)$?
- $L(M_1) - L(M_2)$?

The short answer is “yes” for all of these. (And the ability to do the first three is crucial to the argument showing that every regular language is accepted by some DFA!)

For product and Kleene closure we will wait until we have a “generalized” version of DFA’s that makes the constructions easier.

But the construction for union is already easy, and by slightly altering this construction, we obtain DFA’s for $L_1 \cap L_2$ and $L_1 - L_2$ also...
Example  Before looking at the general construction, let’s try an example. Given DFA’s for \(\{0\}^{*}\{1\}^{*}\) and \(\{1\}^{*}\{0\}^{*}\), construct a DFA for the intersection of the two languages (Note: \(0^{*}1^{*} \cap 1^{*}0^{*} = 0^{*} \cup 1^{*}\), not \(\Lambda\)): 
DFA construction for union, intersection, and set difference

**Cartesian Product Construction Theorem**  Given DFA’s

\[ M_1 = (S_1, \Sigma, T_1, s_1, F_1) \quad \text{and} \quad M_2 = (S_2, \Sigma, T_2, s_2, F_2), \]

let

\[ M = (S, \Sigma, T, s_0, F) \]

be the “Cartesian product” DFA, where

- \( S = S_1 \times S_2, \)
- \( s_0 = (s_1, s_2), \) and
- for all \((s, s') \in S\) and \(a \in \Sigma,\)
  \[ T(((s, s'), a) = (T_1(s, a), T_2(s', a)). \]

Then the following hold.

1. If \( F = \{(s, s') \in S \mid s \in F_1 \text{ or } s' \in F_2\}, \) then
   \[ L(M) = L(M_1) \cup L(M_2). \]

2. If \( F = \{(s, s') \in S \mid s \in F_1 \text{ and } s' \in F_2\}, \) then
   \[ L(M) = L(M_1) \cap L(M_2). \]

3. If \( F = \{(s, s') \in S \mid s \in F_1 \text{ and } s' \notin F_2\}, \) then
   \[ L(M) = L(M_1) - L(M_2). \]
Note: This construction may produce states in $M$ that are not reachable from the start state, but it is easy to show that (in any DFA) unreachable states can be dropped (without affecting the strings that are accepted).

BTW How can we use our notation to express that a state in DFA $M$ is reachable (from the start state)?
DFA for the intersection of \( \{0, 11\}^* \) and \( \{00, 1\}^* \)
\( T^* : S \times \Sigma^* \to S \)

\[
\begin{align*}
T^*(s, \Lambda) &= s, &\text{for all } s \in S \\
T^*(s, xa) &= T(T^*(s, x), a), &\text{for all } s \in S, x \in \Sigma^*, \text{ and } a \in \Sigma
\end{align*}
\]

**Lemma** For any DFA's \( M_1 = (S_1, \Sigma, T_1, s_1, F_1) \), \( M_2 = (S_2, \Sigma, T_2, s_2, F_2) \), let

\[
M = (S, \Sigma, T, s_0, F)
\]

where \( S = S_1 \times S_2 \), \( s_0 = (s_1, s_2) \), and for all \( (s, s') \in S \) and \( a \in \Sigma \),

\[
T((s, s'), a) = (T_1(s, a), T_2(s', a)).
\]

For all \( (s, s') \in S \) and \( x \in \Sigma^* \),

\[
T^*((s, s'), x) = (T_1^*(s, x), T_2^*(s', x)).
\]

**Proof.** By structural induction on \( x \).

**Basis:**

\[
T^*((s, s'), \Lambda) = (s, s') \quad (\text{defn } T^*)
\]

\[
= (T_1^*(s, \Lambda), T_2^*(s', \Lambda)) \quad (\text{defn } T_1^*, \text{ defn } T_2^*)
\]

**Induction:** \( x \in \Sigma^* \), \( a \in \Sigma \).

**IH:** \( T^*((s, s'), x) = (T_1^*(s, x), T_2^*(s', x)) \).

**NTS:** \( T^*((s, s'), xa) = (T_1^*(s, xa), T_2^*(s', xa)) \).

\[
T^*((s, s'), xa) = T(T^*((s, s'), x), a) \quad (\text{defn } T^*)
\]

\[
= T((T_1^*(s, x), T_2^*(s', x)), a) \quad (\text{IH})
\]

\[
= (T_1(T_1^*(s, x), a), T_2(T_2^*(s', x), a)) \quad (\text{defn } T)
\]

\[
= (T_1^*(s, xa), T_2^*(s', xa)) \quad (\text{defn } T_1^*, \text{ defn } T_2^*)
\]
Proof for DFA union/intersection/difference construction

Proof. For part 1, we must show that, for every \( x \in \Sigma^* \),

\[
M \text{ accepts } x \iff M_1 \text{ or } M_2 \text{ does.}
\]

By the definition of \( M \) in part 1, \( M \) accepts \( x \) iff

\[
T^* \left(\left( s_1, s_2 \right), x \right) \in \left\{ \left( s, s' \right) \in S \mid s \in F_1 \text{ or } s' \in F_2 \right\}.
\]

By the Lemma

\[
T^* \left(\left( s_1, s_2 \right), x \right) = (T_1^*(s_1, x), T_2^*(s_2, x)).
\]

So

\[
M \text{ accepts } x \iff T_1^*(s_1, x) \in F_1 \text{ or } T_2^*(s_2, x) \in F_2.
\]

Which is to say that \( M \) accepts \( x \) iff \( M_1 \) or \( M_2 \) does.

Proofs for parts 2 and 3 are similar.
Question  Given DFA’s $M_1$ and $M_2$, can we construct DFA’s to accept

- $L(M_1)'$ ?
- $L(M_1) \cup L(M_2)$ ?
- $L(M_1) \cap L(M_2)$ ?
- $L(M_1) - L(M_2)$ ?
- $L(M_1)L(M_2)$ ?
- $L(M_1)^*$ ?

We already have nice DFA constructions for complement, union, intersection, and difference.

For product and Kleene closure we want a “generalized” version of DFA’s that makes the constructions easier.

First, we’ll add “nondeterminism”…
Example  Consider the family of languages

\[ L_n = (0 + 1)^* 1 (0 + 1)^n. \]

(Shortly we will use the Distinguishability Theorem to show that any DFA for \( L_n \) must have at least \( 2^{n+1} \) states).

What about the following “nondeterministic” machine for \( L_2 = (0 + 1)^* 1 (0 + 1)^2 \)?

Is it a DFA?

But it can be understood to accept \((0 + 1)^* 1 (0 + 1)^2\), and it does this using only 4 states. (And in general, it appears that for each language \( L_n \) there should be such a machine with only \( n + 2 \) states!)
§11.2.2 NFA’s

A *nondeterministic finite automaton* (NFA) is a 5-tuple

\[(S, \Sigma, T, s_0, F)\]

where

- \(S\) is a finite set of “states”,
- \(\Sigma\) is an “input” alphabet,
- \(T : S \times \Sigma \rightarrow \text{power}(S)\) is the “transition function”,
- \(s_0 \in S\) is the “initial state”, and
- \(F \subseteq S\) is the set of “final” or “accepting” states.

So the definition of an NFA is the same as for a DFA, except for the transition function \(T\), which now takes a state and a symbol to a set of states (instead of a single state).

Did our diagram for \((0 + 1)^*1(0 + 1)^2\) fit this definition?
Definition of $T^*$ for NFA’s

As with DFA’s, we need to define a multi-step transition function. But in this case it is a little harder, since reading a string in an NFA may take us to a number of different states (i.e. a set of states).

Recall that the definition of $T^*$ for DFA’s is recursive. The base case says

$$T^*(s, \Lambda) = s$$

and the recursive equation says what to do for (nonempty) strings $xa$, in terms of what to do on $x$ and what to do on $a$.

We do the same sort of thing for NFA’s.

Definition  Given an NFA $M = (S, \Sigma, T, s_0, F)$, we define the multi-step transition function

$$T^* : S \times \Sigma^* \rightarrow \text{power}(S)$$

as follows:

- **Basis:** For any $s \in S$, $T^*(s, \Lambda) = \{s\},$
- **Induction:** For any $s \in S$, $x \in \Sigma^*$ and $a \in \Sigma,$

$$T^*(s, xa) = \bigcup_{s' \in T^*(s, x)} T(s', a).$$
The text does not use $T^*$, instead it extends the definition of $T$ to $Q \times \Sigma^*$ — i.e. it overloads $T()$.

Some properties of $T^*$ for NFA's that one might want to verify. . .

$$T^*(s, a) = T(s, a)$$

$$T^*(s, xy) = \bigcup_{s' \in T^*(s, x)} T^*(s', y)$$

If $T^*(s, x) = \emptyset$, then $T^*(s, xy) = \emptyset$. 
Acceptance for NFA’s

Roughly:

An NFA $M$ accepts $x$ if there is a sequence of moves $M$ can make on input $x$ that ends in an accepting state.

Precisely:

**Definition**  Given an NFA $M = (S, \Sigma, T, s_0, F)$ and a string $x \in \Sigma^*$, $M$ accepts $x$ if

$$T^*(s_0, x) \cap F \neq \emptyset.$$  

As before, the language *recognized* by $M$, denoted $L(M)$, is the set of strings accepted by $M$. That is,

$$L(M) = \{ x \in \Sigma^* \mid T^*(s_0, x) \cap F \neq \emptyset \}.$$
§11.3.2 Reducing NFA’s to DFA’s — The Subset Construction

**Example** Any NFA can be reduced to a DFA that recognizes the same language. Before we look at the general “subset construction”, let’s try an example. Consider again the NFA for

\[(0 + 1)^* 1 (0 + 1)^2.\]
Subset Construction Theorem  Given an NFA

\[ M = (S, \Sigma, T, s_0, F), \]

let DFA

\[ M_d = (\text{power}(S), \Sigma, T_d, \{s_0\}, F_d) \]

be s.t.

- for every \( D \in \text{power}(S) \) and \( a \in \Sigma \),
  \[ T_d(D, a) = \bigcup_{s \in D} T(s, a) \]

- \( F_d = \{D \in \text{power}(S) \mid D \cap F \neq \emptyset\} \).

Then \( L(M_d) = L(M) \).

Proof. We will first show that for all \( x \in \Sigma^* \),

\[ T_d^*(\{s_0\}, x) = T^*(s_0, x) \]

by structural induction on \( x \). Notice that \( T_d^* \) and \( T^* \) are defined differently: the first belongs to a DFA, the second to an NFA...
Given NFA $M = (S, \Sigma, T, s_0, F)$, we construct DFA $M_d = (\text{power}(S), \Sigma, T_d, \{s_0\}, F_d)$, where $F_d = \{ D \in \text{power}(S) \mid D \cap F \neq \emptyset \}$, and for every $D \in \text{power}(S)$ and $a \in \Sigma$, 

$$T_d(D, a) = \bigcup_{s \in D} T(s, a).$$

According to defn of multi-step transition function for DFA's:

1. For any $D \in \text{power}(S)$, $T_d^*(D, \Lambda) = D$.
2. For any $D \in \text{power}(S)$, $x \in \Sigma^*$ and $a \in \Sigma$, 

$$T_d^*(D, xa) = T_d(T_d^*(D, x), a).$$

NFA's defn of $T^*$:

1. For all $s \in S$, $T^*(s, \Lambda) = \{s\}$.
2. For all $s \in S$, $x \in \Sigma^*$ and $a \in \Sigma$, 

$$T^*(s, xa) = \bigcup_{s' \in T^*(s, x)} T(s', a).$$

We will show by structural induction that for all $x \in \Sigma^*$, 

$$T_d^*(\{s_0\}, x) = T^*(s_0, x).$$

**Basis:** We need to show that $T_d^*(\{s_0\}, \Lambda) = T^*(s_0, \Lambda)$.

$$T_d^*(\{s_0\}, \Lambda) = \{s_0\} \quad \text{(defn $T_d^*$ for DFA)}$$

$$T^*(s_0, \Lambda) = T^*(s_0, \Lambda) \quad \text{(defn $T^*$ for NFA)}$$
Given NFA $M = (S, \Sigma, T, s_0, F)$, we construct DFA $M_d = (\text{power}(S), \Sigma, T_d, \{s_0\}, F_d)$, where $F_d = \{D \in \text{power}(S) \mid D \cap F \neq \emptyset\}$, and for every $D \in \text{power}(S)$ and $a \in \Sigma$,

$$T_d(D, a) = \bigcup_{s \in D} T(s, a).$$

According to defn of multi-step transition function for DFA's:

1. For any $D \in \text{power}(S)$, $T^*_d(D, \Lambda) = D$.
2. For any $D \in \text{power}(S)$, $x \in \Sigma^*$ and $a \in \Sigma$, $T^*_d(D, xa) = T_d(T^*_d(D, x), a)$.

NFA’s defn of $T^*$:

1. For all $s \in S$, $T^*(s, \Lambda) = \{s\}$.
2. For all $s \in S$, $x \in \Sigma^*$ and $a \in \Sigma$, $T^*(s, xa) = \bigcup_{s' \in T^*(s, x)} T(s', a)$.

**Induction:** $x \in \Sigma^*$, $a \in \Sigma$.

IH: $T^*_d(\{s_0\}, x) = T^*(s_0, x)$.

NTS: $T^*_d(\{s_0\}, xa) = T^*(s_0, xa)$.

\[
T^*_d(\{s_0\}, xa) = T_d(T^*_d(\{s_0\}, x), a) \quad \text{(defn $T^*_d$ for DFA)} \\
= T_d(T^*(s_0, x), a) \quad \text{(IH)} \\
= \bigcup_{s \in T^*(s_0, x)} T(s, a) \quad \text{(def $T_d$)} \\
= T^*(s_0, xa) \quad \text{(defn $T^*$ for NFA)}
\]

So we conclude that for all $x \in \Sigma^*$, $T^*_d(\{s_0\}, x) = T^*(s_0, x)$. 
NFA $M = (S, \Sigma, T, s_0, F)$.  
DFA $M_d = (\text{power}(S), \Sigma, T_d, \{s_0\}, F_d)$, where  
$F_d = \{ D \in \text{power}(S) \mid D \cap F \neq \emptyset \}$, and  
for every $D \in \text{power}(S)$ and $a \in \Sigma$,  
\[
T_d(D, a) = \bigcup_{s \in D} T(s, a).  
\]

So we’ve established that for all $x \in \Sigma^*$,  
\[
T_d^*(\{s_0\}, x) = T^*(s_0, x).  
\]

It remains only to use this fact to show that, for all $x \in \Sigma^*$, $M_d$ accepts $x$ iff $M$ does.

- $M_d$ accepts $x$  
  - iff $T_d^*(\{s_0\}, x) \in F_d$ (defn acceptance for DFA $M_d$)  
  - iff $T_d^*(\{s_0\}, x) \cap F \neq \emptyset$ (defn of $F_d$ in terms of $F$)  
  - iff $T^*(s_0, x) \cap F \neq \emptyset$ ($T_d^*(\{s_0\}, x) = T^*(s_0, x)$)  
  - iff $M$ accepts $x$ (defn acceptance for NFA $M$)
Reducing an NFA for $0^+ + 1(0 + 1)^*0$ to a DFA
NFA’s still aren’t convenient enough...

It is generally easier to “capture” a regular expression in an NFA. After all, a key feature of regular expressions is nondeterminism.

But to prove that every regular language is accepted by some NFA (and so by some DFA), we need a general construction for union, product and Kleene closure. This is awkward to define directly for NFA’s (but can be done).

The usual next step is to extend NFA’s by allowing so-called “Λ transitions”. We call the resulting machines “NFA-Λ’s” (which the text calls “NFA’s”).

Λ transitions make it possible for an NFA-Λ to change state without reading a symbol, which turns out to be extremely convenient. Intuitively, such state transitions can capture the idea of making a choice when generating a string from a regular expression.

NFA-Λ’s can be reduced to NFA’s (although we won’t learn how to do this). And since NFA’s can be reduced in turn to DFA’s, we know that any language recognized by an NFA-Λ is also recognized by some DFA.

And for NFA-Λ’s, there are easy constructions for union, product and Kleene closure! So it is easy to prove that every regular language is recognized by some NFA-Λ.

We’ll look at a couple of examples of the use of NFA-Λ’s before defining them...
Product made easy (by allowing Λ transitions)

Consider \((0^* + 1^*)(0^+ + 1^+)\):
Kleene closure made easy (by allowing Λ transitions)

Consider $(0(00)^* + (11)^+)^*$:
Kleene closure made easy (by allowing Λ transitions)

Consider \((0^*1)^*\):
NFA’s with \( \Lambda \) transitions

**Definition**  A *nondeterministic finite automaton with \( \Lambda \) transitions* (NFA-\( \Lambda \)) is a 5-tuple \((S, \Sigma, T, s_0, F)\) where

- \( S \) is a finite set of “states”,
- \( \Sigma \) is the “input” alphabet,
- \( T : S \times (\Sigma \cup \{\Lambda\}) \rightarrow \text{power}(S) \) is the “transition function”,
- \( s_0 \in S \) is the “initial state”,
- \( F \subseteq S \) is the set of “final” or “accepting” states.

Once again, we would want to define a multi-step transition function. But consider the kind of difficulty we face:

For NFA’s we had the base case \( T^*(s, \Lambda) = \{s\} \).

For NFA-\( \Lambda \)’s it is clear that we still have \( s \in T^*(s, \Lambda) \), but it is also clear that there may be other states “reachable” from \( s \) on the empty string — states reachable from \( s \) by one or more \( \Lambda \)-transitions. This is handled in the text by considering the \( \lambda \)-closure of sets, described on page 733 (the set of states reachable from a given state by \( \Lambda \)-transitions is an inductively defined set).
The $\lambda$-closure of a set

**Definition** of $\lambda(A)$, the $\lambda$-closure of a set of states $A$

- **Basis:** If $s \in A$, then $s \in \lambda(A)$
- **Induction:** If $s \in \lambda(A)$ and there is a $\Lambda$ transition from $s$ to $t$ (i.e. $t \in T(s, \Lambda)$), then $t \in \lambda(A)$

**Fact:** If $A = A_1 \cup A_2 \cup \ldots \cup A_k$, then

$$\lambda(A) = \lambda(A_1) \cup \lambda(A_2) \cup \ldots \cup \lambda(A_k)$$

That is, the $\lambda$-closure of a union is the union of the $\lambda$-closures.
Reducing NFA-Λ’s to DFA’s

Any NFA-Λ can be reduced to a DFA that recognizes the same language. The proof is virtually identical to the proof that any NFA-Λ can be reduced to a DFA if we take λ-closures into account. Let \( M = (S, \Sigma, T, s_0, F) \) be an NFA-Λ, and define the corresponding DFA as follows:

\[
M_d = (\text{power}(S), \Sigma, T_d, \lambda(\{s_0\}), F_d)
\]

such that

- the start state is \( \lambda(\{s_0\}) \)
- for every \( D \in \text{power}(S) \) and \( a \in \Sigma \),
  \[
  T_d(D, a) = \bigcup_{s \in D} \lambda(T(s, a))
  \]
- \( F_d = \{D \in \text{power}(S) \mid D \cap F \neq \emptyset\} \).

Then \( L(M_d) = L(M) \).

Proof. As before, we start by showing that for all \( x \in \Sigma^* \),
\( T_d^*(\{s_0\}, x) = T^*(s_0, x) \) by structural induction on \( x \) (which is a bit messier due to λ-closures), from which the result easily follows the same as before.
Theorem  For any alphabet $\Sigma$ and any language $L \subseteq \Sigma^*$, the following are equivalent:

- There is a DFA that recognizes $L$.
- There is an NFA that recognizes $L$.
- There is an NFA-$\Lambda$ that recognizes $L$. 
Lemma Every regular language is recognized by some DFA.

Proof sketch. By the previous theorem, any language recognized by some NFA-Λ is recognized by some DFA. So it is enough to show that every regular language is recognized by some NFA-Λ (which is done “top-down” in Algorithm 11.4 on pages 710, 711 of the text in §11.2.3).

Proof is by structural induction on the inductive definition of the set of regular languages over Σ (a “bottom-up” method similar to Algorithm 11.7 on pages 729, 730 of the text in §11.3.1).

Case 1: There is an NFA-Λ that recognizes ∅.

Case 2: There is an NFA-Λ that recognizes {Λ}.

Case 3: For every a ∈ Σ, there is an NFA-Λ that recognizes {a}.

Case 4: L is a regular language over Σ.
   IH: There is an NFA-Λ that recognizes L.
   NTS: There is an NFA-Λ that recognizes L*.

Case 5: L₁, L₂ are regular languages over Σ.
   IH: There are NFA-Λ’s that recognize L₁ and L₂.
   NTS: There are NFA-Λ’s that recognize L₁ ∪ L₂ and L₁L₂.
Lemma  Every language recognized by any DFA, \( M \), is regular.

Proof sketch. Which uses the following definition that assumes the states of \( M \) are labelled with the integers \( 1, 2, \ldots, n \), i.e. \( Q = \{1, 2, \ldots, n\} \), and \( s_0 = 1 \). For \( 1 \leq i, j \leq n \) and \( 0 \leq k \leq n \), define \( L(i, j, k) \) as:

\[
L(i, j, k) = \{ w \in \Sigma^* \mid T^*(i, w) = j, \text{ using only intermediate states } 1, 2, \ldots, k \}
\]

Sub-Lemma  The sets \( L(i, j, k) \) are all regular.

Proof: By ordinary (weak) induction on \( k \).
Basis: \( k = 0 \), i.e. no intermediate states are used, which means that the only strings in \( L(i, j, 0) \) are the “singleton” labels \( a_m (\in \Sigma) \) on transition arrows from \( i \) to \( j \), so

\[
L(i, j, 0) = \bigcup_{T(i, a_m) = j} \{a_m\}
\]

which is regular as a union of the regular “singleton” languages \( \{a_m\} \), or \( L(i, j, 0) = \emptyset \), which is also regular.
Continuation of proof that $L(i, j, k)$’s are regular

**Induction:**
- IH: all $L(i, j, k)$’s are regular
- NTS: all $L(i, j, k + 1)$’s are regular

**Idea:** Any string $w \in L(i, j, k + 1)$ that takes $M$ from state $i$ to state $j$ either uses state $k + 1$ or not.
If not, then $w \in L(i, j, k)$ already.
If so, then $w$ takes $M$ from $i$ to $k + 1$, then may revisit $k + 1$ several times, then takes $M$ to $j$, so $w \in L(i, k + 1, k)L(k + 1, k + 1, k)^*L(k + 1, j, k)$ in this case. Thus:

$$L(i, j, k + 1) = L(i, j, k) \cup L(i, k + 1, k)L(k + 1, k + 1, k)^*L(k + 1, j, k)$$

which is regular since all the $L(i, j, k)$’s are regular by IH, and regular languages are closed under union, concatenation, and star (compare this with the formula near the bottom of Algorithm 11.5 on page 713 of the text).

**Sub-Corollary:** $L(M)$ is regular, since

$$L(M) = \bigcup_{q \in F} L(1, q, n)$$

is a union of regular languages, which proves the Lemma.
Theorem

- For every NFA-Λ, \( N \), there is a DFA, \( D \), such that \( L(D) = L(N) \) (equivalency of DFA’s, NFA’s, and NFA-Λ’s via the Subset Construction Theorem).

- For every DFA, \( D \), \( L(D) \) is a regular language (equivalently, there is a regular expression \( R \) with \( L(R) = L(D) \)). This is the second half of Kleene’s Theorem.

- For every regular language \( L \) (or regular expression), there is an NFA-Λ, \( N \), such that \( L(N) = L \). This is the first half of Kleene’s Theorem.

Consequently DFA’s, NFA-Λ’s (or NFA’s), and regular expressions all describe the same set of languages — the regular languages.