# 1

# Formal Languages

### *Mathematical Preliminaries*

A set is a collection of elements. Those elements can be anything, including other sets.

Sets can be described by listing their elements, such as $\{a, b, c, \ldots\}$.

The empty set is denoted $\varnothing$, or $\{\,\}$.

A set $A$ is *finite* if it has a finite number of elements, that is, if there is a natural number $n \in \mathbb{N}$ such that $A$ has $n$ elements. If no such $n$ exists, then $A$ is *infinite*.

The main relation on sets is *set membership*, written $a \in A$: $a$ is an element of set $A$.

Sets are *extensional*: two sets are equal if they have exactly the same elements.

Another relation on sets is *set inclusion*, written $A \subseteq B$: $A$ is a subset of $B$, meaning that every element of $A$ is an element of $B$).

Some properties of $\subseteq$:

> $\varnothing \subseteq A$ for every $A$
>
> $A \subseteq A$ for every $A$
>
> If $A \subseteq B$ and $B \subseteq C$, then $A \subseteq C$
>
> $A = B$ if and only if $A \subseteq B$ and $B \subseteq A$.

If $P$ is a property, then $\{x \mid P(x)\}$ is the set of all elements satisfying the

property. (Technically speaking, there are some restrictions on what makes up an acceptable property in this context — but they won't impact us.)

Common operations on sets:

$A \cup B = \{x \mid x \in A \text{ or } x \in B\}$

$A \cap B = \{x \mid x \in A \text{ and } x \in B\}$

$\overline{A} = \{x \in \mathcal{U} \mid x \notin A\}$ where $\mathcal{U}$ is a universe of elements with $A \subseteq \mathcal{U}$ — the definition of $\overline{\ }$ therefore depends on the universe under consideration, which will often be clear from context

$A \backslash B = \{x \mid x \in A \text{ but } y \notin B\}$

$A \times B = \{\langle x, y\rangle \mid x \in A, y \in B\}$, the set of all pairs of elements from $A$ and $B$. This generalizes in the obvious way to products $A_1 \times A_2 \times \cdots \times A_k$.

A function $f : A \longrightarrow B$ associates (or maps) every element of $A$ to an element of $B$. Set $A$ is the domain of the function, and $B$ is the codomain. The image of $A$ under $f$ is the subset of $B$ defined by $\{b \in B \mid f(a) = b \text{ for some } a \in A\}$.

If $f : A \longrightarrow B$ and $g : B \longrightarrow C$, then the *composition* $g \circ f : A \longrightarrow C$ defined by $(g \circ f)(x) = g(f(x))$.

A function $f : A \longrightarrow B$ is *one-to-one* if it maps distinct elements of $A$ into distinct elements of $B$ (that is, if $a \neq b$, then $f(a) \neq f(b)$). A function $f : A \longrightarrow B$ is *onto* if every element of $B$ is in the image of $A$ under $f$ (that is, if for every element $b \in B$ there is an element $a \in A$ with $f(a) = b$). A function $f : A \longrightarrow B$ is a *one-to-one correspondance* if it is both one-to-one and onto.

## *Decisions Problems*

Intuitively, a computation is a way to "implement" a mathematical function $f : A \longrightarrow B$. A big part of the course is to build an understanding of what that intuition means.

Arbitrary functions between arbitrary sets $A$ and $B$ is too broad a class of functions to work with. Historically, researchers have looked at two classes of functions to study computation:

1. **Natural number functions** of the form

$$f : \mathbb{N} \times \cdots \times \mathbb{N} \longrightarrow \mathbb{N}$$

2. **Decision problems** of the form

$$d : A \longrightarrow \{1, 0\}$$

   where 1 can be interpreted as true and 0 as false. Decision problems are predicates on the domain $A$.

For the first half of this course, we will consider study computability for decision problems. Example of decision problems include determining if a graph is planar, or determining if two natural numbers are coprime. Restricting to decision problems looks like a limitation, in that many functions of interest we may be interested in computing are not in that form, such as addition, but we can capture addition using a decision problems by considering the decision problem that takes three numbers and determines if the sum of the first two is equal to the third.

One reason why decision problems form an interesting class of problems for which to study computability is that they can be "summarized" using simple sets. You can "summarize" a decision problem

$$d : A \longrightarrow \{1, 0\}$$

using the set

$$S = \{a \in A \mid d(a) = 1\}$$

Given such a set $S$, you can *reconstruct* $d$ readily, by taking

$$d(a) = \begin{cases} 1 & \text{if } a \in S \\ 0 & \text{otherwise} \end{cases}$$

It is easy to check that you get the same decision problem back. Thus, you can study a decision problem by studying the set $S$ that summarizes it.

We will make a further assumption, that the domain $A$ for decision problems will be a set of *strings*.

Let $\Sigma$ be a non-empty finite set we will call the *alphabet*. A *string over* $\Sigma$ is a (possibly empty) finite sequence of elements of $\Sigma$, usually written $a_1 \cdots a_k$, where $a_i \in \Sigma$. For example, `aaccabc` is a string over alphabet $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. It

is also a string over alphabet $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$. We will use $u, v, w$ to range over strings.

The length of $u = a_1 \dots a_k$, written $|u|$, is $k$.

The empty string is written $\epsilon$. It has length 0.

The set of all strings over $\Sigma$ is denoted $\Sigma^*$. Note that this is an infinite set. (Why?)

If $u = a_1 \dots a_k$ and $v = b_1 \cdots b_m$ are strings over $\Sigma$, then the *concatenation* $uv$ is the string $a_1 \cdots a_k b_1 \cdots b_m$. Note that $\epsilon u = u\epsilon = u$ for every string $u$.

We define $u^0 = \epsilon$, $u^1 = u$, $u^2 = uu$, $u^3 = uuu$, etc.

We will transform the problem of of determining whether a decision problem is computable into the problem of determining whether a set of strings is computable. But of course, what we will mean w hen we say that a set of strings $A$ is computable is that the corresponding decision problem constructed from $A$ as above is computable. Why do we do this? Because sets of strings are much easier to work with: they are sets, and we can manipulate sets in many different ways.

## *Languages*

Because sets of strings are so important, let's give them a name. A *formal language* (usually called only a language) over alphabet $\Sigma$ is a set of strings over $\Sigma$.

Since languages are sets, we inherit the usual set operations $A \cup B$, $A \cap B$, $\overline{A}$ (where the universe of $A$ is taken to be $\Sigma^*$).

Because a language $A$ is a set of strings specifically, we can also define more specific operations.

> $A \cdot B = \{uv \mid u \in A \text{ and } v \in B\}$, that is, the set of all strings obtained by concatenating a string of $A$ and a string of $B$.
>
> $A^0 = \{\epsilon\}$
>
> $A^1 = A$
>
> $A^2 = A \cdot A$
>
> $A^3 = A \cdot A \cdot A$

etc...

$A^* = A^0 \cup A^1 \cup A^2 \cup A^3 \cup \cdots = \bigcup_{k \geq 0} A^k$

The $^*$ operation is called the *Kleene star*.

Some properties that are easy to verify:

$\varnothing \cdot A = A \cdot \varnothing = \varnothing$

$\{\epsilon\} \cdot A = A \cdot \{\epsilon\} = A$

**Example:** $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, and $A = \{\mathtt{aa}, \mathtt{bb}\}$. Then $A^* = \{\epsilon, \mathtt{aaaa}, \mathtt{aabb}, \mathtt{bbaa}, \mathtt{bbbb}, \mathtt{aaaaaa}, \mathtt{aaaabb}, \mathtt{aabbaa}, \mathtt{aabbbb}, \mathtt{bbaaaa}, \mathtt{bbaabb}, \mathtt{bbbbaa}, \mathtt{bbbbbb}, \ldots\}$.

This explains why I wrote $\Sigma^*$ for the set of all strings: if we consider $\Sigma$ as a set of strings, each of length 1, then $\Sigma^*$ according to the above definition indeed gives the set of all strings over alphabet $\Sigma$.

The operations $\cup$, $\cdot$, and $^*$ are called the *regular operations*.

## *Regular Languages*

A language over alphabet $\Sigma = \{a_1, \ldots, a_k\}$ is *regular* if it can be obtained from the sets $\varnothing, \{\epsilon\}, \{a_1\}, \ldots, \{a_k\}$ and finitely many applications of the regular operations.

**Example:** consider the language of all even-length strings over alphabet $\{\mathtt{a}, \mathtt{b}\}$. It can be obtained as:

$$((\{\mathtt{a}\} \cup \{\mathtt{b}\} \cup \{\mathtt{c}\}) \cdot (\{\mathtt{a}\} \cup \{\mathtt{b}\} \cup \{\mathtt{c}\}))^*$$

Therefore, that language is regular.

**Example:** consider the language of all strings over $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ that start and end with an $\mathtt{a}$:

$$(\{\mathtt{a}\} \cdot (\{\mathtt{a}\} \cup \{\mathtt{b}\} \cup \{\mathtt{c}\})^* \cdot \{\mathtt{a}\}) \cup \{\mathtt{a}\}$$

Therefore, that language is also regular.

Regular languages form a very natural class of languages, and we will see soon that they arise out of an equally natural model of computation.

Some natural ways to form regular languages:

- The empty language is regular, pretty much by definition.
- Every singleton language (language with a single string) is regular. This is easy to see: language $\{a_1 \dots a_k\}$ is obtained by taking $\{a_1\} \cup \dots \cup \{a_k\}$.
- Every finite language (i.e., language with a finite number of strings) is regular. Again, this is easy to see. Say $A = \{u_1, \dots, u_k\}$. By the previous statement, each of $\{u_1\}, \dots, \{u_k\}$ are regular. Therefore, their union is regular.
- if $A$ and $B$ are regular languages, then $A \cup B$ and $A \cdot B$ are regular languages. That follows directly from the definition.
- If $A$ is regular, then $A^*$ is regular. Again, this follows directly from the definition. This means, in particular, that $\Sigma^*$, the set of all strings over a given alphabet, is regular.
- If $A$ and $B$ are regular, then $A \cap B$ is regular. We'll show this is the case later, because we don't have enough techniques to show that just yet.
- If $A$ is regular, then $\overline{A} = \Sigma^* - A$ is regular. Again, we don't have enough techniques to show that just yet.
- If $A$ is regular, then $rev(A)$, the set of all strings from $A$ but in reverse (where the reverse of $a_1 \dots a_k$ is just $a_k \dots a_1$) is regular. We'll be able to show that next section.

You might be led to believe at this point that every language is regular. That's not true. The most natural non-regular language is probably the language of all *palindrome* strings over an alphabet $\Sigma$. A palindrome string is a string with the property that it and its reverse are equal, such as `aaa`, or `abba`. It's not obvious that this is not regular, and we don't have enough techniques to show that yet.

## *Regular Epressions*

Regular expressions are a convenient notation for regular languages.

A regular expression over alphabet $\Sigma$ is defined by the following syntax:

$$r ::= \quad 1$$

$$
\begin{array}{ll}
\mathtt{0} & \\
a & \text{for every } a \in \Sigma \\
(r_1 + r_2) & \\
(r_1 r_2) & \\
(r_1^*) &
\end{array}
$$

We usually drop parentheses, under the assumption that $r_1^*$ binds tighter than concatenation $r_1 r_2$ which binds tighter than $r_1 + r_2$. For example, $\mathtt{ab+ac}$ is a regular expression, as is $\mathtt{a(b+c)}$ and $\mathtt{a}^*\mathtt{(b+c)}^*$.

A regular expression $r$ denotes a language $[\![r]\!]$ over $\Sigma$ in the following way:

$$
\begin{aligned}
[\![\mathtt{1}]\!] &= \{\epsilon\} \\
[\![\mathtt{0}]\!] &= \varnothing \\
[\![a]\!] &= \{a\} \\
[\![r_1 + r_2]\!] &= [\![r_1]\!] \cup [\![r_2]\!] \\
[\![r_1 r_2]\!] &= [\![r_1]\!] \cdot [\![r_2]\!] \\
[\![r_1^*]\!] &= [\![r_1]\!]^*
\end{aligned}
$$

For example:

$$
\begin{aligned}
[\![\mathtt{ab+ac}]\!] &= [\![\mathtt{ab}]\!] \cup [\![\mathtt{ac}]\!] \\
&= ([\![\mathtt{a}]\!] \cdot [\![\mathtt{b}]\!]) \cup ([\![\mathtt{a}]\!] \cdot [\![\mathtt{c}]\!]) \\
&= (\{\mathtt{a}\} \cdot \{\mathtt{b}\}) \cup (\{\mathtt{a}\} \cdot \{\mathtt{c}\}) \\
&= \{\mathtt{ab}\} \cup \{\mathtt{ac}\} \\
&= \{\mathtt{ab}, \mathtt{ac}\}
\end{aligned}
$$

$$
\begin{aligned}
[\![\mathtt{a(b+c)}]\!] &= [\![\mathtt{a}]\!] \cdot [\![\mathtt{b+c}]\!] \\
&= [\![\mathtt{a}]\!] \cdot ([\![\mathtt{b}]\!] \cup [\![\mathtt{c}]\!]) \\
&= \{\mathtt{a}\} \cdot (\{\mathtt{b}\} \cup \{\mathtt{c}\}) \\
&= \{\mathtt{a}\} \cdot (\{\mathtt{b}, \mathtt{c}\}) \\
&= \{\mathtt{ab}, \mathtt{ac}\}
\end{aligned}
$$

$$
[\![\mathtt{a}^*\mathtt{(b+c)}^*]\!] = [\![\mathtt{a}^*]\!] \cdot [\![\mathtt{(b+c)}^*]\!]
$$

$$
\begin{aligned}
&= [\![\mathtt{a}]\!]^* \cdot [\![\mathtt{b}+\mathtt{c}]\!]^* \\
&= \{\mathtt{a}\}^* \cdot ([\![\mathtt{b}]\!] \cup [\![\mathtt{c}]\!])^* \\
&= \{\mathtt{a}\}^* \cdot (\{\mathtt{b}\} \cup \{\mathtt{c}\})^* \\
&= \{\mathtt{a}\}^* \cdot \{\mathtt{b}, \mathtt{c}\}^*
\end{aligned}
$$

And thinking about this last set (which is difficult to write down), it is basically the set of all strings obtained by concatenating a sequence of as (including none) to a sequence of bs and cs (in any order, including none). So aaaaaa is in this set, as is aaaab, aaaabbbb, aaaabbbcbcbc, etc.

**Theorem:** A language $A$ is regular exactly if there is a regular expression $r$ such that $[\![r]\!] = A$.

We can use regular expressions to show that if $A$ is regular, then $rev(A) = \{rev(u) \mid u \in A\}$, where $rev(u)$ is the reverse of string $u$, is regular: since $A$ is regular, there is a regular expression $r$ with $[\![r]\!] = A$. We take this regular expression and transform it into regular expression $\widehat{r}$ as follow:

$$
\begin{aligned}
\widehat{\mathtt{1}} &= \mathtt{1} \\
\widehat{\mathtt{0}} &= \mathtt{0} \\
\widehat{a} &= a \\
\widehat{r_1+r_2} &= \widehat{r_2}+\widehat{r_1} \\
\widehat{r_1 r_2} &= \widehat{r_2} \cdot \widehat{r_1} \\
\widehat{r_1^*} &= (\widehat{r_1})^*
\end{aligned}
$$

Now, if $[\![r]\!] = A$, then $[\![\widehat{r}]\!] = rev(A)$, and therefore $rev(A)$ is regular.

# 2

# Finite Automata

### *Formal Definitions*

A *finite automaton* is a structure

$$M = (Q, \Sigma, \Delta, s, F)$$

where

- $Q$ is a finite set of states
- $\Sigma$ is a finite alphabet
- $\Delta \subseteq Q \times \Sigma \times Q$ is a transition relation: $\langle p, a, q \rangle \in \Delta$ when there is a transition from state $p$ to state $q$ labeled by the symbol $a$
- $s \in Q$ is the start state
- $F \subseteq Q$ is a set of final states.

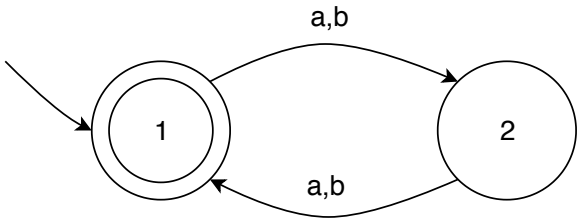
Finite automaton $M$ *accepts* string $u = a_1 \dots a_k$ if there is a path in $M$ starting with $s$ labeled by $a_1, a_2, \dots, a_k$, and ending up in a final state.

Formally: $M = (Q, \Sigma, \Delta, s, F)$ accepts $u = a_1 \dots a_k$ if there exists $q_0, q_1, \dots, q_k \in Q$ such that $q_0 = s$, $q_k \in F$, and $\langle q_{i-1}, a_i, q_i \rangle \in \Delta$ for all $1 \leq i \leq k$.

The language accepted by $M$ is

$$L(M) = \{u \mid M \text{ accepts } u\}$$

**Example:** The following finite automaton accepts exactly the strings over $\{\mathtt{a}, \mathtt{b}\}$ of even length:

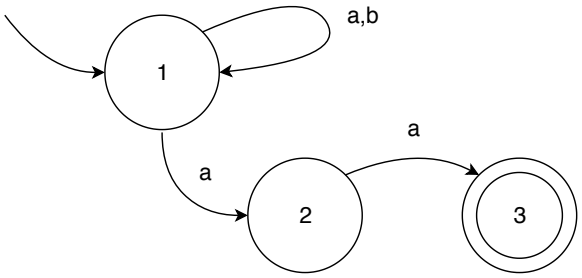



$$M_{even} = (\{1, 2\}, \{\mathtt{a}, \mathtt{b}\}, \Delta_{even}, 1, \{1\})$$

where the transitions go from state 1 to state 2 and back no matter the symbol:

$$\Delta_{even} = \{\langle 1, \mathtt{a}, 2\rangle, \langle 1, \mathtt{b}, 2\rangle, \langle 2, \mathtt{a}, 1\rangle, \langle 2, \mathtt{b}, 1\rangle\}$$

**Example:** The following finite automaton accepts exactly the strings over $\{\mathtt{a}, \mathtt{b}\}$ whose last two symbols are $\mathtt{a}$:




$$M_{last} = (\{1, 2, 3\}, \{\mathtt{a}, \mathtt{b}\}, \Delta_{last}, 1, \{3\})$$

where

$$\Delta_{last} = \{\langle 1, \mathtt{a}, 1\rangle, \langle 1, \mathtt{b}, 1\rangle, \langle 1, \mathtt{a}, 2\rangle, \langle 2, \mathtt{a}, 3\rangle\}$$

**Theorem:** A language $A$ is accepted by some finite automaton $M$ if and only if $A$ is regular.

One direction of the equivalence is pretty easy, namely showing that when $A$ is regular there is some finite automaton that accepts $A$.

Since $A$ is regular, this means that there is a regular expression $r$ with $[\![r]\!] = A$. We can define a recursive procedure that constructs a finite automaton from a regular expression and that accepts the language of the regular expression. The procedure will recurse over the structure of the regular expression.

For the bases case of the recursion, it is easy to construct finite automata for $[\![0]\!] = \varnothing$:

$$M_0 = (\{1\}, \Sigma, \varnothing, 1, \varnothing)$$

for $[\![1]\!] = \{\epsilon\}$:

$$M_1 = (\{1\}, \Sigma, \varnothing, 1, \{1\})$$

and for $[\![\mathsf{a}]\!] = \{\mathsf{a}\}$:

$$M_{\mathsf{a}} = (\{1, 2\}, \Sigma, \{\langle 1, \mathsf{a}, 2\rangle\}, 1, \{2\})$$

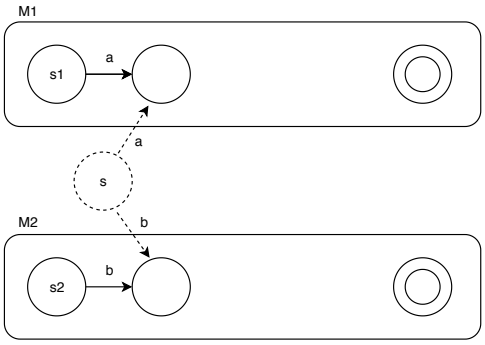
For the recursive cases, first consider $r_1 + r_2$. If $r_1$ and $r_2$ are regular expressions, by recursion we can obtain finite automata $M_1 = (Q_1, \Sigma, \Delta_1, s_1, F_1)$ and $M_2 = (Q_2, \Sigma, \Delta_2, s_2, F_2)$ that accept the languages of $r_1$ and $r_2$ respectively. Without loss of generality, we can take $Q_1$ and $Q_2$ to have no states in common—we can simply rename states if needed, adjusting the transition relation and the state and final states. We can create a finite automaton $M_{r_1+r_2}$ that accepts $[\![r_1 + r_2]\!] = [\![r_1]\!] \cup [\![r_2]\!]$ by putting together $M_1$ and $M_2$ and creating a new start state that transitions to both $M_1$ and $M_2$. (More precisely, the new start state transitions to every state that the original start states $s_1$ and $s_2$ can transition to, with the same label.) Final states are those from $M_1$ and $M_2$, with the addition that the new start start will be final if any one of $s_1$ or $s_2$ is final. Formally:

$$M_{r_1+r_2} = (Q, \Sigma, \Delta, s, F)$$

where

- $Q = Q_1 \cup Q_2 \cup \{s\}$ where $s$ is a new state not in $Q_1$ or $Q_2$
- $\Delta = \Delta_1 \cup \Delta_2 \cup \{\langle s, a, q\rangle \mid \langle s_1, a, q\rangle \in \Delta_1\} \cup \{\langle s, a, q\rangle \mid \langle s_2, a, q\rangle \in \Delta_2\}$
- $F$ is $F_1 \cup F_2 \cup \{s\}$ if $s_1 \in F_1$ or $s_2 \in F_2$, and is $F_1 \cup F_2$ otherwise.

Pictorially:



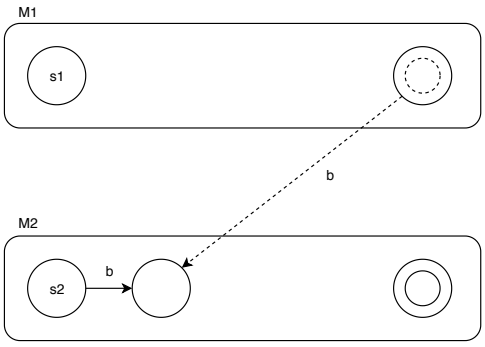

Consider $r_1 r_2$. Again, by recursion we can obtain finite automata $M_1 = (Q_1, \Sigma, \Delta_1, s_1, F_1)$ and $M_2 = (Q_2, \Sigma, \Delta_2, s_2, F_2)$ that accept the languages of $r_1$ and $r_2$ respectively, where $Q_1$ and $Q_2$ have no states in common. We can create a finite automaton $M_{r_1 r_2}$ that accepts $[\![r_1 r_2]\!] = [\![r_1]\!] \cdot [\![r_2]\!]$ by putting together $M_1$ and $M_2$ and basically sending the final states of $M_1$ into $M_2$. (More precisely, every final state of $M_1$ transitions to every state that the original start states $s_2$ transitions to, with the same label.) The start state is the original start state $s_1$ of $M_1$. Final states are those from $M_2$, along with the final states of $M_1$ if the original start state $s_2$ is final in $M_2$ (why?). Formally:

$$M_{r_1 r_2} = (Q, \Sigma, \Delta, s_1, F)$$

where

- $Q = Q_1 \cup Q_2$
- $\Delta = \Delta_1 \cup \Delta_2 \cup \{\langle f, a, q\rangle \mid f \in F_1, \langle s_2, a, q\rangle \in \Delta_2\}$
- $F$ is $F_1 \cup F_2$ if $s_2 \in F_2$, and is $F_2$ otherwise.

Pictorially:



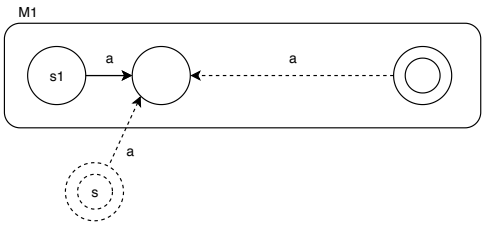

Finally, consider $r_1{}^*$. Again, by recursion we can obtain a finite automaton $M_1 = (Q_1, \Sigma, \Delta_1, s_1, F_1)$ that accepts the language of $r_1$. We can create a finite automaton $M_{r_1{}^*}$ that accepts $[\![r_1{}^*]\!] = [\![r_1]\!]^*$ by modifying $M_1$ to send the final states of $M_1$ back to the start. (More precisely, every final state of $M_1$ transitions to every state that the original start state $s_1$ transitions to, with the same label.) The one subtlety is the treatment of the empty string. Since $[\![r_1]\!]^*$ always contains the empty string, we need to make sure the resulting automaton accepts the empty string, which we cannot guarantee by simply taking the original start state of $M_1$ and making it final—if it wasn't final to start with, and there are transitions back to it, then making it final might make it possible to accept more strings than we want. (Why?) The fix is to create a new start state like we did for $M_{r_1+r_2}$, that transitions to every state that $s_1$ transitions to, and that is indeed final. Final states are those from $M_1$, along with the new start state. Formally:

$$M_{r_1{}^*} = (Q, \Sigma, \Delta, s, F_1 \cup \{s\})$$

where

- $Q = Q_1 \cup \{s\}$ where $s$ is a new state not in $Q_1$
- $\Delta = \Delta_1 \cup \{\langle f, a, q\rangle \mid f \in F_1, \langle s_1, a, q\rangle \in \Delta_1\} \cup \{\langle s, a, q\rangle \mid \langle s_1, a, q\rangle \in \Delta_1\}$.

Pictorially:



Proving the other direction of the theorem, that the language of a finite automaton is always regular, is a bit more painful. The simplest way is to create a regular expression from a description of the finite automaton, but the construction is awkward. I've added some notes about it below in the supplemental material for this lecture.

## *Deterministic Finite Automata*

In the special case when a finite automaton has, for every symbol of the alphabet, at most one transition out of every state labeled with symbol, we say the finite automaton is *deterministic*. The transition relation $\Delta$ for a deterministic finite automaton (DFA) is in fact a partial function (a function that is not defined on all elements of its domain) from $Q \times \Sigma$ to $Q$. For a DFA, it suffices to start from the start state and follow the one and only path that follows the symbols in the input string. If you don't get stuck on the way and reach a final state after consuming all symbols from the input string, you accept the input string.

(When we want to emphasize that an automaton may not be deterministic, we often use the term nondeterministic.)

**Example:** Finite automaton $M_{even}$ from earlier is a deterministic finite automaton. Finite automaton $M_{last}$ is a nondeterministic finite automaton.

Even though DFAs seem more restrictive than nondeterministic finite automata, it turns out that DFAs accept exactly the regular languages. Clearly, since every DFA is a finite automaton, if a DFA accepts a language $A$ then $A$ is regular. But if $A$ is regular, while we know it can be accepted by a finite automaton, it's not clear it can be accepted by a *deterministic* finite automaton.

Consider the following way to "execute" a nondeterministic finite automaton $M$. The idea is to look at the possible transitions that can be taken for a given symbol of the input all at once. Intuitively, we begin in the start state, and for every symbol of the input string in order, we keep track of the possible states we can reach by following a transition labeled with that symbol from any of the possible states we have already reached. Thus, at every step in this process, we maintain a set of possible states that $M$ can be in. If at the end of the string one of the possible states is final, then $M$ accepts the string.

This process, where we consider *sets* of states of $M$, in fact describes the

execution of a deterministic finite automaton, in which states are sets of states of $M$, and there is a transtion from a set $X$ of states of $M$ to a set $Y$ of states of $M$ labeled $a$ exactly when the transitions of $M$ labeled $a$ from any state in $X$ lead to the states in $Y$.

Formally, for every finite automaton $M$, there exists a deterministic finite automaton $\widehat{M}$ that accepts the same language as $M$, constructed as follows. If $M = (Q, \Sigma, \Delta, s, F)$, construct

$$\widehat{M} = (\widehat{Q}, \Sigma, \widehat{\Delta}, \widehat{s}, \widehat{F})$$

where:

- $\widehat{Q} = \{X \mid X \subseteq Q\}$
- $\widehat{\Delta} = \{\langle X, a, Y\rangle \mid X \subseteq Q, Y = \{q \in Q \mid \langle p, a, q\rangle \in \Delta \text{ for some } p \in X\}\}$
- $\widehat{s} = \{s\}$
- $\widehat{F} = \{X \subseteq Q \mid X \cap F \neq \varnothing\}$

This construction is called the *subset construction*.

We can see that $\widehat{M}$ is deterministic because any subset $X \subseteq Q$ uniquely determines a subset $Y$ to transition to in the definition of $\widehat{\Delta}$.

## *Supplementary Material*

### *The Language of a Finite Automaton is Regular*

We stated in class the following result:

**Theorem:** *A language $A$ is accepted by some finite automaton $M$ if and only if $A$ is regular.*

We showed one direction of this equivalence: when $A$ is regular, we can construct a finite automaton which accepts $A$.

To argue the other direction, that if a language is accepted by a finite automaton, then it is regular, we give a process for constructing a regular expression which denotes the same language as that accepted by any given finite automaton.

Let $M = (Q, \Sigma, \Delta, s, F)$ be a finite automaton.

We define a recursive function $R(p, q, X)$ that returns a regular expressions representing all strings that take automaton $M$ from state $p$ to state $q$ through transitions going only through states in $X$, with the possible exception of $p$ and $q$ which need not lie in $X$.

Here is the recursive function definition. It recurses on the size of $X$.

When $X$ is empty, let $a_1, \ldots, a_k$ be all the symbols in $\Sigma$ such that $\langle p, a_i, q\rangle \in \Delta$, that is, that make the automaton transition from $p$ to $q$. We have two cases: when $p \neq q$,

$$R(p, q, \varnothing) = \begin{cases} a_1 + \cdots + a_k & \text{if } k \geq 1 \\ 0 & \text{if } k = 0 \end{cases}$$

and when $p = q$,

$$R(p, p, \varnothing) = \begin{cases} a_1 + \cdots + a_k + 1 & \text{if } k \geq 0 \\ 1 & \text{if } k = 0. \end{cases}$$

For the recursive case, choose any element $v \in X$. (Different choices will yield different regular expressions at end, but all of those different regular expressions will denote the same language.) Take:

$$R(p, q, X) = R(p, q, X\backslash\{v\}) + R(p, v, X\backslash\{v\})(R(v, v, X\backslash\{v\})^* R(v, q, X\backslash\{v\})$$

(Set $A\backslash B$ is the set of all elements of $A$ minus all elements of $B$.) To understand this definition, observe that any string that takes the automaton from $p$ to $q$ with all intermediate states in $X$ either:

- never takes the automaton through state $v$, and those strings are given by this part of the regular expression:

  $$R(p, q, X\backslash\{v\})$$

- or makes the automaton reach state $v$ a first time, given by this part of the regular expression:

  $$R(p, v, X\backslash\{v\})$$

  followed by a finite number of loops (possibly zero) from $v$ back to itself without going through $v$ in between and always staying in $X$, give by this part:

  $$(R(v, v, X\backslash\{v\}))^*$$

followed by taking the automaton from $v$ to $q$ without going through $v$, given by this part:

$$R(v, q, X \backslash \{v\}).$$

Once we have such a function $R$, we can construct the regular expression whose language is the language accepted by $M$, namely all the strings taking $M$ from the start state to any of the final states $f_1, \ldots, f_k$ through any state of $M$ by:

$$R(s, f_1, Q) + \cdots + R(s, f_k, Q).$$

The result is usually a *huge* unreadable regular expression. Thankfully, it can be simplified because regular expression obey algebraic-like simplification rules. But that's beyond the scope of these notes.

### *Complementation of Regular Languages*

Using finite automata, we can show that the complement of a regular language is regular.

Let $A$ be a regular language. That means that there is a deterministic finite automaton $M = (Q, \Sigma, \Delta, s, F)$ that accepts $A$. (The determinism is important.)

The first thing we do is *complete* $M$, in such a way that for every symbol in $\Sigma$ and for every $q \in Q$, there is a $p \in Q$ such that $\langle q, a, p \rangle \in \Delta$ — in other words, there is a transition at every state for every symbol. We do so by introducing a new *deadend* state to which we transition whenever the original $M$ doesn't have a suitable transition.

Let $M' = (Q', \Sigma, \Delta', s, F)$ by taking

- $Q' = Q \cup \{s_{deadend}\}$ where $s_{deadend}$ is a new state not in $Q$
- $\Delta' = \Delta \cup \{\langle q, a, s_{deadend} \rangle \mid q \in Q, a \in \Sigma,$ and there is no $p \in Q$ with $\langle q, a, p \rangle \in \Delta\}$

It is not difficult to argue that $M'$ accepts the same language as $M$. Now, take $M'$, and construct the finite automaton $\overline{M'} = (Q', \Sigma, \Delta', s, Q' \backslash F)$. The language accepted by $\overline{M'}$ is $\overline{A}$, and therefore $\overline{A}$ is regular.

### *Non-Regular Languages*

Finite automata also let us argue that some languages are not regular.

Let $A = \{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}$. That is, $A$ is the language of all strings made up of a sequence of as followed by a sequence of the same number of bs. I claim $A$ is not regular.

Let's argue by contradiction. Let's assume $A$ *is* in fact regular, and derive a contradiction. This means that our assumption cannot be, and thus $A$ cannot be regular.

If $A$ is regular, then there exists a deterministic finite automaton $M = (Q, \{\mathtt{a}, \mathtt{b}\}, \Delta, s, F)$ that accepts $A$. Let $N = |Q|$, the number of states in $M$.

Consider the string $u = \mathtt{a}^{2N}\mathtt{b}^{2N}$, which is a string in $A$. Therefore $M$ accepts $u$. Since $M$ has only $N$ states and string $u$ has length $4N$, any sequence of transitions in $M$ that accepts $u$ must hit one state, call it $q$, at least twice. Moreover, because the first $2N$ symbols of $u$ are all $\mathtt{a}$, that state $q$ must be reached from $s$ after $k < 2N$ transitions labeled $\mathtt{a}$ and also after $k + m < 2N$ transitions labeled $\mathtt{a}$ for some $m > 0$.

In other words, there is a loop of length $m$ from $q$ to $q$ with transitions labeled $\mathtt{a}$. The string $u = \mathtt{a}^{2N}\mathtt{b}^{2N} = \mathtt{a}^k\mathtt{a}^m\mathtt{a}^{2N-k-m}\mathtt{b}^{2N}$ is accepted by the automaton by first going from $s$ to $q$ (via $\mathtt{a}^k$) and then from $q$ to $q$ (via $\mathtt{a}^m$) and then from $q$ to a final state (via $\mathtt{a}^{2N-k-m}\mathtt{b}^{2N}$).

But consider the string $v = \mathtt{a}^k\mathtt{a}^m\mathtt{a}^m\mathtt{a}^{2N-k-m}\mathtt{b}^{2N}$. It also must be accepted by $A$! Indeed, $\mathtt{a}^k$ takes $M$ from $s$ to $q$, $\mathtt{a}^m$ takes $M$ from $q$ to $q$, $\mathtt{a}^m$ takes $M$ from $q$ to $q$ again (second time around the loop!), and finally $\mathtt{a}^{2N-k-m}\mathtt{b}^{2N}$ takes $M$ from $q$ to a final state. So $M$ accepts $v$. But $v$ is of the form $\mathtt{a}^m\mathtt{a}^{2N}\mathtt{b}^{2N}$ and since $m > 0$, $v \notin A$. So $A$ cannot be the language accepted by $M$. But we chose $M$ specifically to have language $A$. That's our contradiction.

Our initial assumption cannot be: $A$ is not regular.

A similar argument shows that $Pal(\Sigma)$, the set of all palindromes over $\Sigma$, is not regular.

A general form of the argument is formalized as the *Pumping Lemma for Regular Languages*:

**Theorem** (Pumping Lemma): *If $A$ is regular language, then there is a number $p$ with the property that any string $u \in A$ of length at least $p$ can be written as $u = xyz$ for $x, y, z$ satisfying:*

*(i)* *for every $i \geq 0$, $xy^iz \in A$,*

*(ii)* *$|y| > 0$, and*

*(iii)* *$|xy| \leq p$.*