

## Week3 wednesday

So far we have that:

- If there is a DFA recognizing a language, there is a DFA recognizing its complement.
- If there are NFA recognizing two languages, there is a NFA recognizing their union.
- If there are DFA recognizing two languages, there is a DFA recognizing their union.
- If there are DFA recognizing two languages, there is a DFA recognizing their intersection.

Our goals for today are (1) prove similar results about other set operations, (2) prove that NFA and DFA are equally expressive, and therefore (3) define an important class of languages.

Suppose  $A_1, A_2$  are languages over an alphabet  $\Sigma$ . **Claim:** if there is a NFA  $N_1$  such that  $L(N_1) = A_1$  and NFA  $N_2$  such that  $L(N_2) = A_2$ , then there is another NFA, let's call it  $N$ , such that  $L(N) = A_1 \circ A_2$ .

**Proof idea:** Allow computation to move between  $N_1$  and  $N_2$  “spontaneously” when reach an accepting state of  $N_1$ , guessing that we've reached the point where the two parts of the string in the set-wise concatenation are glued together.

**Formal construction:** Let  $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$  and  $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$  and assume  $Q_1 \cap Q_2 = \emptyset$ . Construct  $N = (Q, \Sigma, \delta, q_0, F)$  where

- $Q =$
- $q_0 =$
- $F =$
- $\delta : Q \times \Sigma_\varepsilon \rightarrow \mathcal{P}(Q)$  is defined by, for  $q \in Q$  and  $a \in \Sigma_\varepsilon$ :

$$\delta((q, a)) = \begin{cases} \delta_1((q, a)) & \text{if } q \in Q_1 \text{ and } q \notin F_1 \\ \delta_1((q, a)) & \text{if } q \in F_1 \text{ and } a \in \Sigma \\ \delta_1((q, a)) \cup \{q_2\} & \text{if } q \in F_1 \text{ and } a = \varepsilon \\ \delta_2((q, a)) & \text{if } q \in Q_2 \end{cases}$$

*Proof of correctness would prove that  $L(N) = A_1 \circ A_2$  by considering an arbitrary string accepted by  $N$ , tracing an accepting computation of  $N$  on it, and using that trace to prove the string can be written as the result of concatenating two strings, the first in  $A_1$  and the second in  $A_2$ ; then, taking an arbitrary string in  $A_1 \circ A_2$  and proving that it is accepted by  $N$ . Details left for extra practice.*

Suppose  $A$  is a language over an alphabet  $\Sigma$ . **Claim:** if there is a NFA  $N$  such that  $L(N) = A$ , then there is another NFA, let's call it  $N'$ , such that  $L(N') = A^*$ .

**Proof idea:** Add a fresh start state, which is an accept state. Add spontaneous moves from each (old) accept state to the old start state.

**Formal construction:** Let  $N = (Q, \Sigma, \delta, q_1, F)$  and assume  $q_0 \notin Q$ . Construct  $N' = (Q', \Sigma, \delta', q_0, F')$  where

- $Q' = Q \cup \{q_0\}$
- $F' = F \cup \{q_0\}$
- $\delta' : Q' \times \Sigma_\varepsilon \rightarrow \mathcal{P}(Q')$  is defined by, for  $q \in Q'$  and  $a \in \Sigma_\varepsilon$ :

$$\delta'((q, a)) = \begin{cases} \delta((q, a)) & \text{if } q \in Q \text{ and } q \notin F \\ \delta((q, a)) & \text{if } q \in F \text{ and } a \in \Sigma \\ \delta((q, a)) \cup \{q_1\} & \text{if } q \in F \text{ and } a = \varepsilon \\ \{q_1\} & \text{if } q = q_0 \text{ and } a = \varepsilon \\ \emptyset & \text{if } q = q_0 \text{ and } a \in \Sigma \end{cases}$$

*Proof of correctness would prove that  $L(N') = A^*$  by considering an arbitrary string accepted by  $N'$ , tracing an accepting computation of  $N'$  on it, and using that trace to prove the string can be written as the result of concatenating some number of strings, each of which is in  $A$ ; then, taking an arbitrary string in  $A^*$  and proving that it is accepted by  $N'$ . Details left for extra practice.*

**Application:** A state diagram for a NFA over  $\Sigma = \{a, b\}$  that recognizes  $L((a^*b)^*)$ :

Suppose  $A$  is a language over an alphabet  $\Sigma$ . **Claim:** if there is a NFA  $N$  such that  $L(N) = A$  then there is a DFA  $M$  such that  $L(M) = A$ .

**Proof idea:** States in  $M$  are “macro-states” – collections of states from  $N$  – that represent the set of possible states a computation of  $N$  might be in.

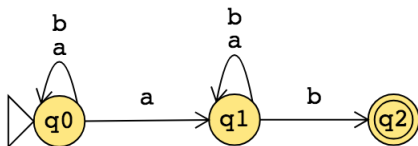
**Formal construction:** Let  $N = (Q, \Sigma, \delta, q_0, F)$ . Define

$$M = ( \mathcal{P}(Q), \Sigma, \delta', q', \{X \subseteq Q \mid X \cap F \neq \emptyset\} )$$

where  $q' = \{q \in Q \mid q = q_0 \text{ or is accessible from } q_0 \text{ by spontaneous moves in } N\}$  and

$\delta'((X, x)) = \{q \in Q \mid q \in \delta(r, x) \text{ for some } r \in X \text{ or is accessible from such an } r \text{ by spontaneous moves in } N\}$

Consider the state diagram of an NFA over  $\{a, b\}$ . Use the “macro-state” construction to find an equivalent DFA.



Consider the state diagram of an NFA over  $\{0, 1\}$ . Use the “macro-state” construction to find an equivalent DFA.



Note: We can often prune the DFAs that result from the “macro-state” constructions to get an equivalent DFA with fewer states (e.g. only the “macro-states” reachable from the start state).

## The class of regular languages

Fix an alphabet  $\Sigma$ . For each language  $L$  over  $\Sigma$ :

**There is a DFA over  $\Sigma$  that recognizes  $L$**   $\exists M$  ( $M$  is a DFA and  $L(M) = A$ )  
*if and only if*

**There is a NFA over  $\Sigma$  that recognizes  $L$**   $\exists N$  ( $N$  is a NFA and  $L(N) = A$ )  
*if and only if*

**There is a regular expression over  $\Sigma$  that describes  $L$**   $\exists R$  ( $R$  is a regular expression and  $L(R) = A$ )

A language is called **regular** when any (hence all) of the above three conditions are met.

We already proved that DFAs and NFAs are equally expressive. It remains to prove that regular expressions are too.

Part 1: Suppose  $A$  is a language over an alphabet  $\Sigma$ . If there is a regular expression  $R$  such that  $L(R) = A$ , then there is a NFA, let's call it  $N$ , such that  $L(N) = A$ .

**Structural induction:** Regular expression is built from basis regular expressions using inductive steps (union, concatenation, Kleene star symbols). Use constructions to mirror these in NFAs.

**Application:** A state diagram for a NFA over  $\{a, b\}$  that recognizes  $L(a^*(ab)^*)$ :

Part 2: Suppose  $A$  is a language over an alphabet  $\Sigma$ . If there is a DFA  $M$  such that  $L(M) = A$ , then there is a regular expression, let's call it  $R$ , such that  $L(R) = A$ .

**Proof idea:** Trace all possible paths from start state to accept state. Express labels of these paths as regular expressions, and union them all.

1. Add new start state with  $\varepsilon$  arrow to old start state.
2. Add new accept state with  $\varepsilon$  arrow from old accept states. Make old accept states non-accept.
3. Remove one (of the old) states at a time: modify regular expressions on arrows that went through removed state to restore language recognized by machine.

**Application:** Find a regular expression describing the language recognized by the DFA with state diagram



## Week2 wednesday

**Review:** Formal definition of finite automaton:  $M = (Q, \Sigma, \delta, q_0, F)$

- Finite set of states  $Q$
- Alphabet  $\Sigma$
- Transition function  $\delta$
- Start state  $q_0$
- Accept (final) states  $F$

In the state diagram of  $M$ , how many outgoing arrows are there from each state?

$M = (\{q, r, s\}, \{a, b\}, \delta, q, \{q\})$  where  $\delta$  is (rows labelled by states and columns labelled by symbols):

| $\delta$ | $a$ | $b$ |
|----------|-----|-----|
| $q$      | $r$ | $r$ |
| $r$      | $s$ | $s$ |
| $s$      | $q$ | $q$ |

The state diagram for  $M$  is

Give two examples of strings that are accepted by  $M$  and two examples of strings that are rejected by  $M$ :

$L(M) =$

A regular expression describing  $L(M)$  is

Let the alphabet be  $\Sigma_1 = \{0, 1\}$ .

A state diagram for a finite automaton that recognizes  $\{w \in \Sigma_1^* \mid w \text{ contains at most two 1's}\}$  is

A state diagram for a finite automaton that recognizes  $\{w \in \Sigma_1^* \mid w \text{ contains more than two 1's}\}$  is

**Strategy:** Add “labels” for states in the state diagram, e.g. “have not seen any of desired pattern yet” or “sink state”. Then, we can use the analysis of the roles of the states in the state diagram to work towards a description of the language recognized by the finite automaton.

A useful bit of terminology: the **iterated transition function** of a finite automaton  $M = (Q, \Sigma, \delta, q_0, F)$  is defined recursively by

$$\delta^*(q, w) = \begin{cases} q & \text{if } q \in Q, w = \varepsilon \\ \delta(q, a) & \text{if } q \in Q, w = a \in \Sigma \\ \delta(\delta^*(q, u), a) & \text{if } q \in Q, w = ua \text{ where } u \in \Sigma^* \text{ and } a \in \Sigma \end{cases}$$

Using this terminology,  $M$  accepts a string  $w$  over  $\Sigma$  if and only if  $\delta^*(q_0, w) \in F$ .

Suppose  $A$  is a language over an alphabet  $\Sigma$ . By definition, this means  $A$  is a subset of  $\Sigma^*$ . **Claim:** if there is a DFA  $M$  such that  $L(M) = A$  then there is another DFA, let's call it  $M'$ , such that  $L(M') = \overline{A}$ , the complement of  $A$ , defined as  $\{w \in \Sigma^* \mid w \notin A\}$ .

**Proof idea:**

**Proof:**

Application: Design a finite automaton that recognizes the language of all strings over  $\{a, b\}$  whose length is not a multiple of 3.

**Note:** On Friday, we'll see a new kind of finite automaton. It will be helpful to distinguish it from the machines we've been talking about so we'll use **Deterministic Finite Automaton** (DFA) to refer to the machines from Section 1.1.

## Week1 wednesday

Our motivation in studying sets of strings is that they can be used to encode problems. To calibrate how difficult a problem is to solve, we describe how complicated the set of strings that encodes it is. How do we define sets of strings?

How would you describe the language that has no elements at all?

How would you describe the language that has all strings over  $\{0, 1\}$  as its elements?

**\*\*This definition was in the pre-class reading\*\*** **Definition 1.52:** A **regular expression** over alphabet  $\Sigma$  is a syntactic expression that can describe a language over  $\Sigma$ . The collection of all regular expressions over  $\Sigma$  is defined recursively:

*Basis steps of recursive definition*

$a$  is a regular expression, for  $a \in \Sigma$

$\varepsilon$  is a regular expression

$\emptyset$  is a regular expression

*Recursive steps of recursive definition*

$(R_1 \cup R_2)$  is a regular expression when  $R_1, R_2$  are regular expressions

$(R_1 \circ R_2)$  is a regular expression when  $R_1, R_2$  are regular expressions

$(R_1^*)$  is a regular expression when  $R_1$  is a regular expression

The *semantics* (or meaning) of the syntactic regular expression is the **language described by the regular expression**. The function that assigns a language to a regular expression over  $\Sigma$  is defined recursively, using familiar set operations:

*Basis steps of recursive definition*

The language described by  $a$ , for  $a \in \Sigma$ , is  $\{a\}$  and we write  $L(a) = \{a\}$

The language described by  $\varepsilon$  is  $\{\varepsilon\}$  and we write  $L(\varepsilon) = \{\varepsilon\}$

The language described by  $\emptyset$  is  $\{\}$  and we write  $L(\emptyset) = \emptyset$ .

*Recursive steps of recursive definition*

When  $R_1, R_2$  are regular expressions, the language described by the regular expression  $(R_1 \cup R_2)$  is the union of the languages described by  $R_1$  and  $R_2$ , and we write

$$L( (R_1 \cup R_2) ) = L(R_1) \cup L(R_2) = \{w \mid w \in L(R_1) \vee w \in L(R_2)\}$$

When  $R_1, R_2$  are regular expressions, the language described by the regular expression  $(R_1 \circ R_2)$  is the concatenation of the languages described by  $R_1$  and  $R_2$ , and we write

$$L( (R_1 \circ R_2) ) = L(R_1) \circ L(R_2) = \{uv \mid u \in L(R_1) \wedge v \in L(R_2)\}$$

When  $R_1$  is a regular expression, the language described by the regular expression  $(R_1^*)$  is the **Kleene star** of the language described by  $R_1$  and we write

$$L( (R_1^*) ) = ( L(R_1) )^* = \{w_1 \cdots w_k \mid k \geq 0 \text{ and each } w_i \in L(R_1)\}$$



For the following examples assume the alphabet is  $\Sigma_1 = \{0, 1\}$ :

The language described by the regular expression 0 is  $L(0) = \{0\}$

The language described by the regular expression 1 is  $L(1) = \{1\}$

The language described by the regular expression  $\varepsilon$  is  $L(\varepsilon) = \{\varepsilon\}$

The language described by the regular expression  $\emptyset$  is  $L(\emptyset) = \emptyset$

The language described by the regular expression  $(\Sigma_1 \Sigma_1 \Sigma_1)^*$  is  $L((\Sigma_1 \Sigma_1 \Sigma_1)^*) =$

The language described by the regular expression  $1^+$  is  $L(1^+) = L(1^* \circ 1) =$

*Shorthand and conventions* (Sipser pages 63-65)

Assuming  $\Sigma$  is the alphabet, we use the following conventions

|                              |  |
|------------------------------|--|
| $\Sigma$                     | regular expression describing language consisting of all strings of length 1 over $\Sigma$     |
| $*$ then $\circ$ then $\cup$ | precedence order, unless parentheses are used to change it                                     |
| $R_1 R_2$                    | shorthand for $R_1 \circ R_2$ (concatenation symbol is implicit)                               |
| $R^+$                        | shorthand for $R^* \circ R$  |
| $R^k$                        | shorthand for $R$ concatenated with itself $k$ times, where $k$ is a (specific) natural number |

**Caution:** many programming languages that support regular expressions build in functionality that is more powerful than the “pure” definition of regular expressions given here.

Regular expressions are everywhere (once you start looking for them).

Software tools and languages often have built-in support for regular expressions to describe **patterns** that we want to match (e.g. Excel/ Sheets, grep, Perl, python, Java, Ruby).

Under the hood, the first phase of **compilers** is to transform the strings we write in code to tokens (keywords, operators, identifiers, literals). Compilers use regular expressions to describe the sets of strings that can be used for each token type.

Next time: we’ll start to see how to build machines that decide whether strings match the pattern described by a regular expression.

*Extra examples for practice:*

Which regular expression(s) below describe a language that includes the string  $a$  as an element?

$a^*b^*$

$a(ba)^*b$

$a^* \cup b^*$

$(aaa)^*$

$(\varepsilon \cup a)b$

# Week1 friday

**Review:** Determine whether each statement below about regular expressions over the alphabet  $\{a, b, c\}$  is true or false:

True or False:  $ab \in L((a \cup b)^*)$

True or False:  $ba \in L(a^*b^*)$

True or False:  $\varepsilon \in L(a \cup b \cup c)$

True or False:  $\varepsilon \in L((a \cup b)^*)$

True or False:  $\varepsilon \in L(aa^* \cup bb^*)$

**\*\*This definition was in the pre-class reading\*\*** A finite automaton (FA) is specified by  $M = (Q, \Sigma, \delta, q_0, F)$ . This 5-tuple is called the **formal definition** of the FA. The FA can also be represented by its state diagram: with nodes for the state, labelled edges specifying the transition function, and decorations on nodes denoting the start and accept states.

Finite set of states  $Q$  can be labelled by any collection of distinct names. Often we use default state labels  $q_0, q_1, \dots$

The alphabet  $\Sigma$  determines the possible inputs to the automaton. Each input to the automaton is a string over  $\Sigma$ , and the automaton “processes” the input one symbol (or character) at a time.

The transition function  $\delta$  gives the next state of the automaton based on the current state of the machine and on the next input symbol.

The start state  $q_0$  is an element of  $Q$ . Each computation of the machine starts at the start state.

The accept (final) states  $F$  form a subset of the states of the automaton,  $F \subseteq Q$ . These states are used to flag if the machine accepts or rejects an input string.

The computation of a machine on an input string is a sequence of states in the machine, starting with the start state, determined by transitions of the machine as it reads successive input symbols.

The finite automaton  $M$  accepts the given input string exactly when the computation of  $M$  on the input string ends in an accept state.  $M$  rejects the given input string exactly when the computation of  $M$  on the input string ends in a nonaccept state, that is, a state that is not in  $F$ .

The language of  $M$ ,  $L(M)$ , is defined as the set of all strings that are each accepted by the machine  $M$ . Each string that is rejected by  $M$  is not in  $L(M)$ . The language of  $M$  is also called the language recognized by  $M$ .

What is **finite** about all finite automata? (Select all that apply)

- ☐ The size of the machine (number of states, number of arrows)
- ☐ The length of each computation of the machine
- ☐ The number of strings that are accepted by the machine



The formal definition of this FA is

Classify each string  $a, aa, ab, ba, bb, \varepsilon$  as accepted by the FA or rejected by the FA.

*Why are these the only two options?*

The language recognized by this automaton is



The language recognized by this automaton is



The language recognized by this automaton is