

## Day4

**\*\*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

# Day5

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

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

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

**Note:** 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.

$M = (\{q_0, q_1, q_2\}, \{a, b\}, \delta, q_0, \{q_0\})$  where  $\delta$  is (rows labelled by states and columns labelled by symbols):

$\delta$	$a$	$b$
$q_0$	$q_1$	$q_1$
$q_1$	$q_2$	$q_2$
$q_2$	$q_0$	$q_0$

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  $M$ :

A regular expression describing  $L(M)$  is

A state diagram for a finite automaton recognizing

$$\{w \mid w \text{ is a string over } \{a, b\} \text{ whose length is not a multiple of } 3\}$$

Extra example: Let  $n$  be an arbitrary positive integer. What is a formal definition for a finite automaton recognizing

$$\{w \mid w \text{ is a string over } \{0, 1\} \text{ whose length is not a multiple of } n\}?$$

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

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

A state diagram for a finite automaton that recognizes  $\{w \mid w \text{ contains more than two 1's}\}$  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.

Or: decompose the language to a simpler one that we already know how to recognize with a DFA or NFA.

Textbook Exercise 1.14: Suppose  $A$  is a language over an alphabet  $\Sigma$ . 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:**

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  $\delta^*(q_0, w) \in F$ .

**Proof:**

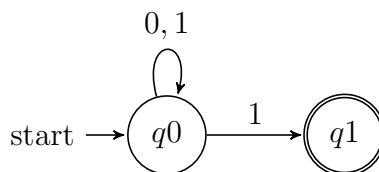
## Day6

**Nondeterministic finite automaton** (Sipser Page 53) Given as  $M = (Q, \Sigma, \delta, q_0, F)$

Finite set of states $Q$	Can be labelled by any collection of distinct names. Default: $q_0, q_1, \dots$
Alphabet $\Sigma$	Each input to the automaton is a string over $\Sigma$ .
Arrow labels $\Sigma_\epsilon$	$\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ . Arrows in the state diagram are labelled either by symbols from $\Sigma$ or by $\epsilon$
Transition function $\delta$	$\delta : Q \times \Sigma_\epsilon \rightarrow \mathcal{P}(Q)$ gives the <b>set of possible next states</b> for a transition from the current state upon reading a symbol or spontaneously moving.
Start state $q_0$	Element of $Q$ . Each computation of the machine starts at the start state.
Accept (final) states $F$	$F \subseteq Q$ .

$M$  accepts the input string  $w \in \Sigma^*$  if and only if **there is** a computation of  $M$  on  $w$  that processes the whole string and ends in an accept state.

The formal definition of the NFA over  $\{0, 1\}$  given by this state diagram is:



The language over  $\{0, 1\}$  recognized by this NFA is:

*Practice:* Change the transition function to get a different NFA which accepts the empty string (and potentially other strings too).

The state diagram of an NFA over  $\{a, b\}$  is:



The formal definition of this NFA is:



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  $L(N) = A_1 \cup A_2$ .

**Proof idea:** Use nondeterminism to choose which of  $N_1, N_2$  to run.

**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  $Q_1 \cap Q_2 = \emptyset$  and that  $q_0 \notin Q_1 \cup Q_2$ . Construct  $N = (Q, \Sigma, \delta, q_0, F_1 \cup F_2)$  where

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

*Proof of correctness would prove that  $L(N) = A_1 \cup A_2$  by considering an arbitrary string accepted by  $N$ , tracing an accepting computation of  $N$  on it, and using that trace to prove the string is in at least one of  $A_1, A_2$ ; then, taking an arbitrary string in  $A_1 \cup A_2$  and proving that it is accepted by  $N$ . Details left for extra practice.*

## Day2

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  $1^* \circ 1$  is  $L(1^* \circ 1) =$

The language described by the regular expression  $((0 \cup 1) \circ (0 \cup 1) \circ (0 \cup 1))^*$  is  $L(((0 \cup 1) \circ (0 \cup 1) \circ (0 \cup 1))^*) =$

# Day3

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

*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
$R_1R_2$	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.

Practice with the regular expressions over  $\{a, b\}$  below.

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

$$a^*b^*$$

$$a(ba)^*b$$

$$a^* \cup b^*$$

$$(aaa)^*$$

$$(\varepsilon \cup a)b$$