Week3 monday

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.

CC BY-NC-SA 2.0 Version October 28, 2024 (1)

Suppose A_1, A_2 are languages over an alphabet Σ. **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

- $Q =$
- $q_0 =$
- $F =$
- $\delta : Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$ 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.

CC BY-NC-SA 2.0 Version October 28, 2024 (2)

Suppose A is a language over an alphabet Σ. **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

- $Q' = Q \cup \{q_0\}$
- $F' = F \cup \{q_0\}$
- $\delta' : Q' \times \Sigma_\varepsilon \to \mathcal{P}(Q')$ 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 $L((a^*b)^*)$:

CC BY-NC-SA 2.0 Version October 28, 2024 (3)

Suppose A is a language over an alphabet Σ. **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.

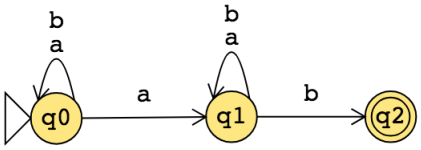
Formal construction: Let $N = (Q, \Sigma, \delta, q_0, F)$. Define

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

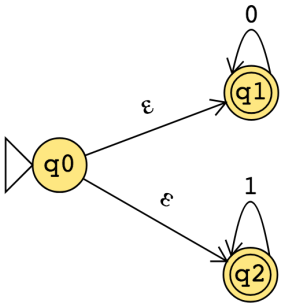
where $q' = \{q \in Q \mid q = q_0$ or is accessible from q_0 by spontaneous moves in $N\}$ and

$\delta'((X, x)) = \{q \in Q \mid q \in \delta((r, x))$ for some $r \in X$ or is accessible from such an r 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).

CC BY-NC-SA 2.0 Version October 28, 2024 (4)

The class of regular languages

Fix an alphabet Σ. For each language L over Σ:

There is a DFA over Σ that recognizes L	$\exists M$ (M is a DFA and $L(M) = A$)
if and only if	
There is a NFA over Σ that recognizes L	$\exists N$ (N is a NFA and $L(N) = A$)
if and only if	
There is a regular expression over Σ 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 Σ. 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 $L(a^*(ab)^*)$:

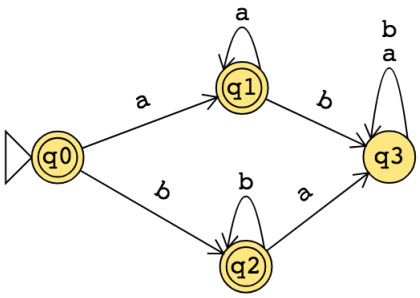
Part 2: Suppose A is a language over an alphabet Σ. If there is a DFA M such that $L(M) = A$, then there is a regular expression, let's call it R, such that $L(R) = A$.

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 ε arrow to old start state.
2. Add new accept state with ε 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.

CC BY-NC-SA 2.0 Version October 28, 2024 (5)

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



CC BY-NC-SA 2.0 Version October 28, 2024 (6)

Week2 monday

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

- Finite set of states Q
- Alphabet Σ
- Transition function δ
- 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.

$M = (\{q0, q1, q2\}, \{a, b\}, \delta, q0, \{q0\})$ where δ is (rows labelled by states and columns labelled by symbols):

δ	a	b
$q0$	$q1$	$q1$
$q1$	$q2$	$q2$
$q2$	$q0$	$q0$

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

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

CC BY-NC-SA 2.0 Version October 28, 2024 (7)

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 Σ. 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 Σ if and only if $\delta^*(\ (q_0, w)\ ) \in F$.

Proof:

CC BY-NC-SA 2.0 Version October 28, 2024 (8)

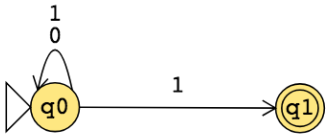
Week2 wednesday

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: $q0, q1, \ldots$
Alphabet Σ	Each input to the automaton is a string over Σ.
Arrow labels Σ_ε	$\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$. Arrows in the state diagram are labelled either by symbols from Σ or by ε
Transition function δ	$\delta : Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$ gives the **set of possible next states** 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:

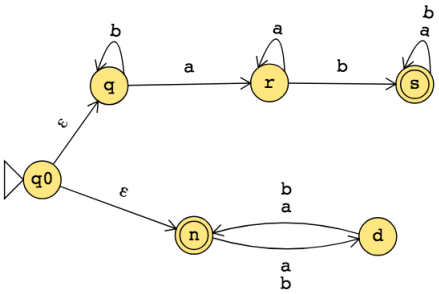


The language over $\{0, 1\}$ recognized by this NFA is:

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

CC BY-NC-SA 2.0 Version October 28, 2024 (9)

The state diagram of an NFA over $\{a, b\}$ is below. The formal definition of this NFA is:



Suppose A_1, A_2 are languages over an alphabet Σ. **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

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

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.

Week1 monday

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?

CC BY-NC-SA 2.0 Version October 28, 2024 (10)

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

Basis steps of recursive definition

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

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

CC BY-NC-SA 2.0 Version October 28, 2024 (12)

Week1 wednesday

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 Σ is the alphabet, we use the following conventions

Σ	regular expression describing language consisting of all strings of length 1 over Σ
$*$ 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.

CC BY-NC-SA 2.0 Version October 28, 2024 (13)

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^*

$a(ba)^*b$

$a^* \cup b^*$

$(aaa)^*$

$(\varepsilon \cup a)b$

CC BY-NC-SA 2.0 Version October 28, 2024 (14)

Week1 friday

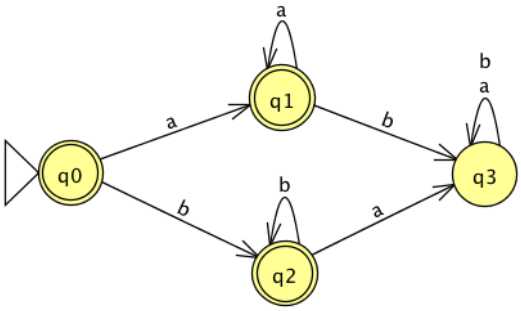
This definition was in the pre-class reading A finite automaton (FA) is specified by $M = (Q, \Sigma, \delta, q_0, F)$. This 5-tuple is called the **formal definition** of the FA. The FA can also be represented by its state diagram: with nodes for the state, labelled edges specifying the transition function, 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 $q0, q1, \ldots$
>
> The alphabet Σ determines the possible inputs to the automaton. Each input to the automaton is a string over Σ, and the automaton "processes" the input one symbol (or character) at a time.
>
> The transition function δ 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

CC BY-NC-SA 2.0 Version October 28, 2024 (15)



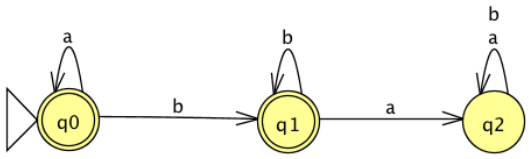
The formal definition of this FA is

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

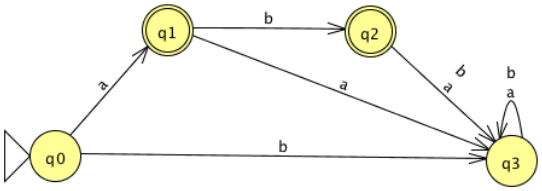
Why are these the only two options?

The language recognized by this automaton is

CC BY-NC-SA 2.0 Version October 28, 2024 (16)



The language recognized by this automaton is



The language recognized by this automaton is

CC BY-NC-SA 2.0 Version October 28, 2024 (17)