

Handout 2: NFA, Regex, and Closure Properties

Yuwen Gong, Andrew Kwok, Jeffrey Lin

October 9, 2026

1. Review

Definition: Nondeterministic finite automata

A **nondeterministic finite automaton (NFA)** is given by a five-tuple $D = (Q, \Sigma, \delta, q_0, F)$, where:

- Q is a finite set of **states**.
- Σ is the **alphabet** of input symbols.
- $\delta : Q \times (\Sigma \cup \varepsilon) \rightarrow \mathcal{P}(Q)$ is the **transition function**. It takes a state and either an input symbol or ε , and gives a set of possible next states.
- $q_0 \in Q$ is the **start state**.
- $F \subseteq Q$ is the set of **accepting states**.

Definition: Acceptance of a string by an NFA

We say that an NFA **accepts** a string $w \in \Sigma^*$ if (for some $n \geq |w|$) *there exists* a way to write w as a concatenation $w = w_1 \cdots w_n$, where each w_i is in $\Sigma \cup \{\varepsilon\}$, and *there exists* a computation path $(r_0, \dots, r_n)$ such that:

$r_0 = q_0$: starts from the start state

$r_i \in \delta(r_{i-1}, w_i)$ for $1 \leq i \leq n$: each step i takes the NFA from state r_{i-1} to some r_i that is along one of the δ transition that are labeled with the next w_i (either a symbol or ε).

$r_n \in F$: this computation path ends in an accepting state.

Unlike a DFA, for each particular string w , an NFA N may have zero, one, or more possible computations. We only care whether at least one of them ends in an accepting state, in which case the string is accepted. If all of them either fail to complete or end in a non-accept state, the string is rejected.

An NFA might seem more omnipotent than a DFA: it may have several choices on a symbol, no choice at all, or ε -arrows that move without reading input. It is natural to guess that this extra freedom lets NFAs recognize *more* languages. It does not.

1.1 NFAs are no more powerful than DFAs, and DFAs are actually a subset of NFAs

We've learned by now 3 ways of describing a language: by a DFA, by an NFA, and by a regular expression. The (drum roll) *fundamental theorem*, which the rest of the handout keeps coming back to, is that all three describe *exactly the same class of languages*: the regular languages.

Theorem:

Every nondeterministic finite automaton (NFA) has an equivalent deterministic finite automaton (DFA).

Corollary

A language is regular if and only if some nondeterministic finite automaton (NFA) recognizes it.

One direction is apparent: a DFA is just an NFA that happens to have exactly one arrow out of every state for every symbol and no ε -arrows. So every language recognized by a DFA is recognized by an NFA. Yay! The interesting part is the other direction, showing how to take a given NFA and construct an equivalent DFA. This is called the subset construction.

Proof idea: the subset construction. The construction is given in Lecture 6 (also in Sipser Theorem 1.39). but here's the intuitive idea. Suppose you're given an NFA, and think about how you would simulate its computation "by hand": you keep a finger on every state the machine could currently be in, and after each input symbol you move, add, and remove fingers (in class we sometimes referred to them as "tokens"). The only thing you ever need to remember is the *set* of states that currently have fingers on them – these are exactly the states the NFA could possibly reach via some computation path on the string so far. If the NFA has a total of k states, there are 2^k such sets of states, so a DFA with 2^k states—one state per subset—can carry out the simulation. You then accept the string if at the end you have at least one finger on some accepting state (this means that there exists a computation path for the NFA that ends in an accepting state – which is exactly the acceptance condition for NFAs).

To do this conversion for a specific NFA with k states you can either start by drawing all 2^k states of the DFA (each corresponding to a subset) and then drawing the transitions: one transition for each state and alphabet symbol. Alternatively, you can start by drawing the start state of the DFA, which corresponds to the set of states in the NFA that includes the start state and all states that can be reached from it by travelling along any number ε -transitions. You then keep adding transitions from each DFA state and each symbol in Σ until all no new state is added, and all transitions are defined. To see a worked example, consult the beginning of Lecture 7 notes by Prof. Malkin, or example 1.41 in Sipser. Note that checkup 3 also has some practice on converting an NFA to a DFA.

1.2 Describing the language of an NFA

Using the fingers/tokens analogy above, you can think of running an NFA on an input string as follows: put a token on the start state and on every state reachable from it by ε -arrows; for each symbol, move every token along the arrows labeled with that symbol, and then for every ε arrow generate a new token that moves along the ε -arrow. A token with nowhere to go, disappears. The string is accepted iff at the end *some* token is on an accept state.

Proof Template: Describing the language of an NFA**1. Strategies towards figuring out the language:**

- **Try short strings.** Run the NFA (with tokens) on ε , on each single symbol, and on a few strings of length 2–3, and note which are accepted.
- **Work backwards from the accept states.** Look at the arrows entering each accept state, and ask: when can the NFA be in the state that arrow comes from?
- **Read off the structure.** A loop means “any number of”; a loop on every symbol means “any string”; an ε -arrow switches parts without reading anything; several arrows with the same label out of one state mean the NFA *guesses* which way to go.

2. Write the answer in set notation: $L(N) = \{x \in \Sigma^* \mid \dots\}$, describing the strings themselves (do not refer to the NFA). Make sure your description is clear and non-ambiguous.**3. Check it:** try to convince yourself of *both directions*: that every string in the language you described has a way for the NFA to make guesses to reach an accept state, and for every string not in the language you describe, there’s no such way. It may help to test this on the strings you tried in the first step above, and also to test one string you claim is in $L(N)$ and one you claim is not.**1.3 Regex to Language****Definition: Regular Expressions**

We say that R is a **regular expression** if R is

1. a for some a in the alphabet Σ ,
2. ε ,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or
6. (R_1^*) , where R_1 is a regular expression.

A regular expression evaluates to a language: In arithmetic we combine numbers with $+$ and $\times$ to build expressions such as $(4 + 3) \times 6$, and the *value* of that expression is a number, 42. Similarly, we can combine languages with the regular operations (union, concatenation, and star) to build **regular expressions**, such as

$$0^*(0 \cup 1).$$

Regular expressions evaluate not to a number but a language. In this example it’s the set of all strings ending with 0 or a 1 preceded by any number of 0s (or equivalently, $\{0^n \mid n \geq 0\} \cup \{0^n 1 \mid n \geq 0\}$). For a regular expression R , we write $L(R)$ for the language (=set of strings) that it describes. We refer to $L(R)$ interchangeably as the language generated by R , the language of R , the language described by R , or the language that R evaluates to. It is defined recursively, building it from the base cases and three regular operations, as follows.

Definition: The language of a regular expression

1. $L(a) = \{a\}$: A symbol stands for the one-string language containing it. For example, $L(0) = \{0\}$.
2. $L(\varepsilon) = \{\varepsilon\}$: this is a language with one string, the empty string ε .
3. $L(\emptyset) = \{\}$ (also denoted as $\emptyset$): this is the empty language, that has no strings at all (note the difference with the previous language).
4. $L(R_1 \cup R_2) = L(R_1) \cup L(R_2)$: any string that is generated either by R_1 or by R_2 .
5. $L(R_1 \circ R_2) = L(R_1) \circ L(R_2)$: any string that can be obtained by taking a string generated by R_1 and then concatenating to it a string generated by R_2 .
6. $L(R^*) = L(R)^*$: any string that can be written as a concatenation of 0 or more strings from $L(R)$.

We sometimes omit the concatenation symbol, just as the multiplication sign is dropped in algebra ($2x$ for $2 \times x$). So $(0 \cup 1)0^*$ is shorthand for $(0 \cup 1) \circ 0^*$: a string from the first part *followed by* a string from the second.

Precedence: star binds tightest, then concatenation, then union—like exponent, then $\times$. So $10^* \cup 1$ means $(1(0^*)) \cup 1$. Parentheses override this, as usual.

A worked example. Consider $a^*(b \cup a)b$ over $\Sigma = \{a, b\}$. Dissect it into its three concatenated pieces:

$$\underbrace{a^*}_{\{\varepsilon, a, aa, aaa, \dots\}} \quad \underbrace{(b \cup a)}_{\{a, b\}} \quad \underbrace{b}_{\{b\}}$$

A string in the language is a string from the first set, followed by one from the second, followed by one from the third. In English: *any number of as, then one more symbol (either an a or a b), then a b*. As a set,

$$L(a^*(b \cup a)b) = \{a^n x b \mid n \geq 0, x \in \{a, b\}\} = \{bb, ab, abb, aab, aabb, aaab, \dots\}.$$

(another way to write this language is $\{a^n b \mid n \geq 1\} \cup \{a^n bb \mid n \geq 0\}$).

Notice what is *not* in it: ε and b are too short (the last two pieces each contribute exactly one symbol each, so every string has length at least 2), and ba, bab are also not in the language as they have a b before the final two symbols, which only a^* could have produced.

The empty string is ALWAYS in anything starred

By definition, $L^* = \bigcup_{k=0}^{\infty} L^k = \{x_1 x_2 \dots x_k \mid k \geq 0, \text{ each } x_i \in L\}$. Taking $k = 0$ pieces gives the empty string, *whatever L is*, since by definition, $L^0 = \{\varepsilon\}$ for *any* L . Thus, for any regular expression R , we have $\varepsilon \in L(R^*)$. For example, $\varepsilon \in \Sigma^*$, $\varepsilon \in 0^*$, $\varepsilon \in (ab \cup a)^*$, even $\varepsilon \in \emptyset^*$.

Another worked example. What is $L((0 \cup 1)^* 0 \cup 1(0 \cup 1)^*)$? First apply the precedence rules: star, then concatenation, then union, so this is $((0 \cup 1)^* 0) \cup (1(0 \cup 1)^*)$. Now dissect each side:

- $(0 \cup 1)^* 0$: any string at all, followed by a 0—that is, *all strings that end in 0*.
- $1(0 \cup 1)^*$: a 1, followed by any string at all—that is, *all strings that start with 1*.

Their union is the set of strings that *end in 0 or start with 1* (or both): $0, 1, 10, 100, 11, 0110, \dots$ are in the language; $\varepsilon, 01, 0011$ are not. Note that ε is *not* in the language even though $(0 \cup 1)^*$ appears twice: on each side $(0 \cup 1)^*$ is concatenated with a piece that contributes exactly one symbol.

1.4 Closure Properties

Theorem:

The class of regular languages is closed under **concatenation**, **power**, and **star** (aka Kleene star). That is, if L_1, L_2 are regular, then each of the following is also regular:

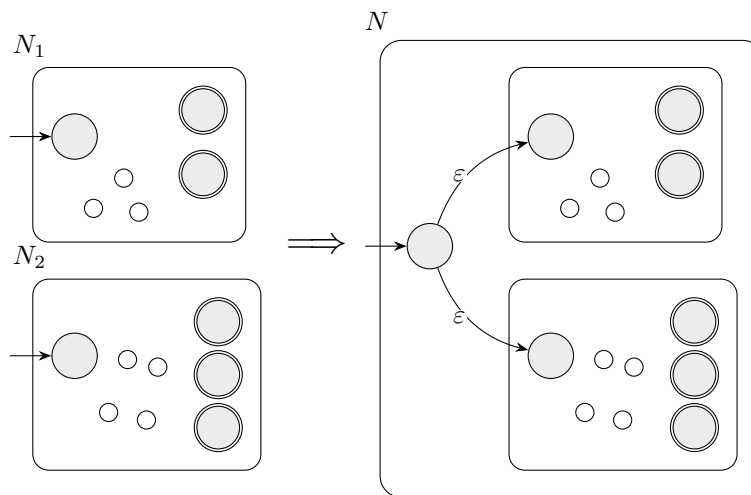
- $L_1 \circ L_2$
- L_1^k for any $k \geq 0$
- L_1^*

This is in addition to closure of regular languages under **union**, **intersection**, **complement** that we have proven before, which tell us that each of $L_1 \cup L_2$, $L_1 \cap L_2$, $\overline{L_1}$ is also regular.

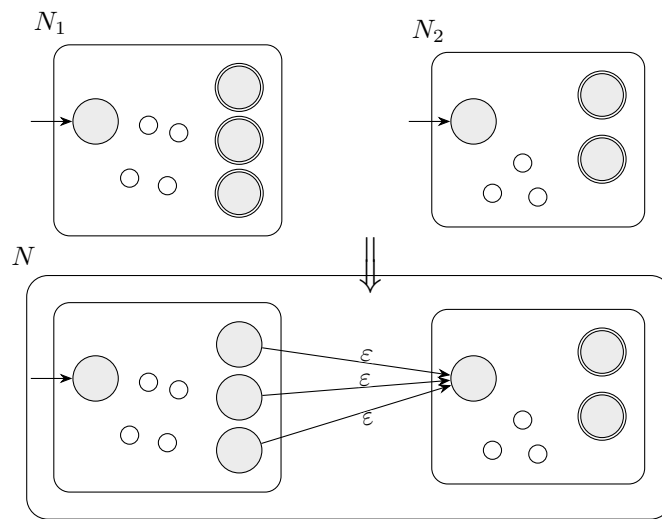
We will sketch the idea for the proof for the three regular operations. We will do so by constructing a NFA for the resulting language, which is where nondeterminism comes to shine. In each proof sketch below, the whole argument is a picture: start from NFAs N_1, N_2 recognizing L_1, L_2 respectively, and wire them together with a few ε -arrows. (For the formal construction see the lecture notes and/or the textbook).

Proof idea: Union. We have already proved closure by constructing a DFA. Here's an alternative proof using NFAs.

Add a brand-new start state and let it guess which machine to run by branching with ε -arrows to the old start states. N accepts exactly when at least one of N_1, N_2 accepts.



Proof idea: Concatenation. Run N_1 first. Whenever N_1 is in an accept state, the input seen so far is a string in L_1 , so N is allowed to *guess* that this is the right place to split the input and jump—via an ε -arrow—to the start of N_2 . The accept states of N_1 stop being accept states; only N_2 's accept states count. N accepts iff the input can be split as xy with $x \in A_1$ and $y \in A_2$.



Proof idea: Kleene Star. We add a brand-new start state, and make it accepting – this guarantees that ε is accepted (recall that ε is in the star of any language). From this new start state we add an ε transition to the old start state. We then additionally add ε transitions from any accept state to the old start state. These allow N to "guess" where to break the string into portions, take the ε transition between portions and start anew. A string is accepted if and only if it is in L^k for some $k \geq 0$. See lecture notes (or textbook) for the picture and more detail.

Proving closure under a new operation

The constructions above, as well as the previous ones we saw for DFAs, are all examples of how to prove closure under a new operation op : you can construct a DFA (as we did for intersection) or construct an NFA (as we did above), which is often easier. You can also construct a regular expression (see problem 5 below). Any of these methods can work (and depending on the question, one method may be better than another).

Proof Template: Regular languages are closed under op

1. **Start from what you are given.** Since L is regular, take a DFA (or NFA) $D = (Q, \Sigma, \delta, q_0, F)$ for L , or a regular expression R for L .
2. **Intuitive description.** Say in words how the new machine or expression works: what it guesses, which copies of D it runs, how the pieces are wired together.
3. **Formal specification.** For an automaton, give the full 5-tuple: the states, the alphabet, δ' on every state and every symbol (and ε , for an NFA), the start state, and the accept states (you can see examples in class for intersection or concatenation or in the solution to problem 5 below). For a regular expression, define the transformation recursively on all six cases of the definition: a , ε , $\emptyset$, $R_1 \cup R_2$, $R_1 \circ R_2$, R^* (you can see an example in the solutions of problem 5).
4. **Why it is correct.** In a sentence or two, explain why the new machine accepts (or the new expression describes) exactly the strings of $\text{op}(L)$ (and not any others).
5. **Conclude.** An NFA, a DFA, or a regular expression for $\text{op}(L)$ exists, so $\text{op}(L)$ is regular.

Regular Expressions Describe Regular Languages, and Have a NFA/DFA

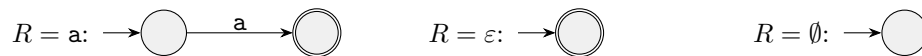
Lemma

If a language is described by a regular expression, then it is regular.

The converse: every regular language has a regular expression. Lemma 1.55 says that whatever a regular expression can describe, an NFA can recognize. The other direction is also true, and together they give *a language is regular if and only if some regular expression describes it*.

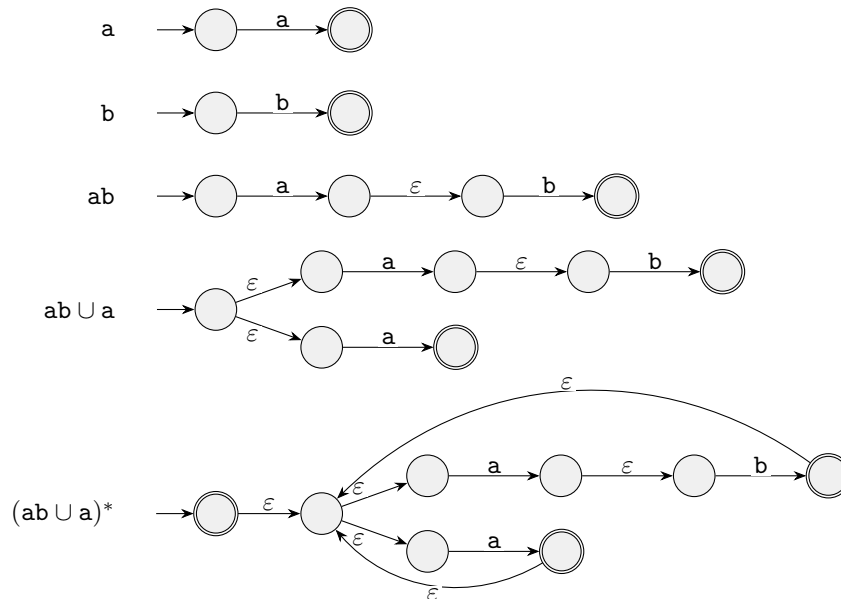
Lemma

If a language L is regular, then there is a regular expression R with $L(R) = L$.



For $R = R_1 \cup R_2$, $R = R_1 \circ R_2$, $R = R_1^*$ apply the union, concatenation, and star constructions to the NFAs for R_1 and R_2 .

Example (Sipser, Example 1.56). *Convert $(ab \cup a)^*$ to an NFA. We build up from the smallest subexpressions. (The procedure does not give the smallest NFA: this one has eight states, but a two-state NFA recognizes the same language—can you find it?)*



2. Practice Problems

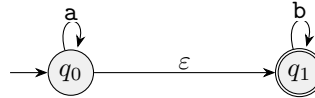
Problem 1: NFA to Language

For each NFA below, give the language it recognizes using set notation (for example, $L(N) = \{x \in \{a, b\}^* \mid x \text{ starts with } a\}$). Your description should not refer to the NFA itself. In each case, also decide whether ε is in the language or not. Take $\Sigma = \{a, b\}$ in (a)–(b) and $\Sigma = \{0, 1\}$ in (c).

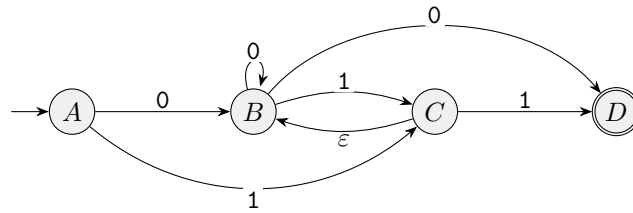
(a) (**Warm-up**) Give the language of each of these two one-state NFAs.



(b) Give the language of this NFA.



(c) (Harder:) Give the language of this NFA.



Problem 2: Regex to Language

Describe the language of each of the following regular expressions, using set notation. Take $\Sigma = \{0, 1\}$ in (a)–(c) and $\Sigma = \{a, b\}$ in (d).

- (a) $R_1 = 0^*11101^*$
- (b) $R_2 = 1(0 \cup 1)^*0$
- (c) $R_3 = ((0 \cup 1)(0 \cup 1)(0 \cup 1))^* \cup 1$
- (d) $R_4 = (a \cup \varepsilon)(\varepsilon \cup b)^*$
- (e) $R_5 = \emptyset^*$

Problem 3: Language to NFA

Let $\Sigma = \{0, 1\}$. Construct an NFA for the following languages.

- (a) $L_1 = \{1\} \cup \{w \in \Sigma^* \mid |w| = 0 \pmod 3\}$
- (b) $L_2 = \{w \in \Sigma^* \mid w \text{ does not contain } 01\}$

Problem 4: Language to Regex

Let $\Sigma = \{0, 1\}$. Give regular expressions for the following languages.

- (a) $L_1 = \{w \in \Sigma^* \mid |w| \geq 2 \text{ and } w \text{ does not end in } 01\}$
- (b) $L_2 = \{w \in \Sigma^* \mid |w| \equiv 1 \pmod{2}\}$

Problem 5: Closure properties

Let Σ be an arbitrary alphabet. For a language $L \subseteq \Sigma^*$ we define the following operations on L :

- (a) $\text{Skip}(L) = \{w \in \Sigma^* \mid aw \in L \text{ for some } a \in \Sigma\}$.
- (b) $\text{Rev}(L) = \{w \in \Sigma^* \mid w^R \in L\}$. Note that for a string w , w^R is the reverse of w (for instance, $(aab)^R = baa$).

Prove that the class of regular languages is closed under the Skip operation and under the Reverse operation.

Give an intuitive description as well as a formal specification of the construction (if you use an NFA or DFA, give the 5-tuple; if you use regular expressions, fully specify the transformation you apply to the regular expression for L), and explain in a sentence or two why it is correct.