Week4 monday

Recap so far: In DFA, the only memory available is in the states. Automata can only "remember" finitely far in the past and finitely much information, because they can have only finitely many states. If a computation path of a DFA visits the same state more than once, the machine can't tell the difference between the first time and future times it visits this state. Thus, if a DFA accepts one long string, then it must accept (infinitely) many similar strings.

Definition A positive integer p is a **pumping length** of a language L over Σ means that, for each string $s \in \Sigma^*$, if $|s| \geq p$ and $s \in L$, then there are strings x, y, z such that

$$s = xyz$$

and

$$|y| > 0, \qquad \text{for each } i \geq 0,\ xy^i z \in L, \qquad \text{and} \qquad |xy| \leq p.$$

Negation: A positive integer p is **not a pumping length** of a language L over Σ iff

$$\exists s \left(|s| \geq p \wedge s \in L \wedge \forall x \forall y \forall z \left((s = xyz \wedge |y| > 0 \wedge |xy| \leq p) \rightarrow \exists i (i \geq 0 \wedge xy^i z \notin L) \right) \right)$$

Informally:

Restating **Pumping Lemma**: If L is a regular language, then it has a pumping length.

Contrapositive: If L has no pumping length, then it is nonregular.

The Pumping Lemma *cannot* be used to prove that a language *is* regular.

The Pumping Lemma **can** be used to prove that a language *is not* regular.

Extra practice: Exercise 1.49 in the book.

Proof strategy: To prove that a language L is **not** regular,

- Consider an arbitrary positive integer p

- Prove that p is not a pumping length for L

- Conclude that L does not have *any* pumping length, and therefore it is not regular.

Example: $\Sigma = \{0, 1\}$, $L = \{0^n 1^n \mid n \geq 0\}$.

Fix p an arbitrary positive integer. List strings that are in L and have length greater than or equal to p:

Pick $s =$

Suppose $s = xyz$ with $|xy| \leq p$ and $|y| > 0$.

Then when $i =$, $xy^i z =$

Example: $\Sigma = \{0, 1\}$, $L = \{ww^{\mathcal{R}} \mid w \in \{0,1\}^*\}$. Remember that the reverse of a string w is denoted $w^{\mathcal{R}}$ and means to write w in the opposite order, if $w = w_1 \cdots w_n$ then $w^{\mathcal{R}} = w_n \cdots w_1$. Note: $\varepsilon^{\mathcal{R}} = \varepsilon$.

Fix p an arbitrary positive integer. List strings that are in L and have length greater than or equal to p:

Pick $s =$

Suppose $s = xyz$ with $|xy| \leq p$ and $|y| > 0$.

$$\boxed{}$$

Then when $i =$, $xy^i z =$

Example: $\Sigma = \{0, 1\}$, $L = \{0^j 1^k \mid j \geq k \geq 0\}$.

Fix p an arbitrary positive integer. List strings that are in L and have length greater than or equal to p:

Pick $s =$

Suppose $s = xyz$ with $|xy| \leq p$ and $|y| > 0$.

$$\boxed{}$$

Then when $i =$, $xy^i z =$

Example: $\Sigma = \{0, 1\}$, $L = \{0^n 1^m 0^n \mid m, n \geq 0\}$.

Fix p an arbitrary positive integer. List strings that are in L and have length greater than or equal to p:

Pick $s =$

Suppose $s = xyz$ with $|xy| \leq p$ and $|y| > 0$.

$$\boxed{}$$

Then when $i =$, $xy^i z =$

Extra practice:

Language	$s \in L$	$s \notin L$	Is the language regular or nonregular?
$\{a^n b^n \mid 0 \leq n \leq 5\}$			
$\{b^n a^n \mid n \geq 2\}$			
$\{a^m b^n \mid 0 \leq m \leq n\}$			
$\{a^m b^n \mid m \geq n + 3, n \geq 0\}$			
$\{b^m a^n \mid m \geq 1, n \geq 3\}$			
$\{w \in \{a, b\}^* \mid w = w^{\mathcal{R}}\}$			
$\{w w^{\mathcal{R}} \mid w \in \{a, b\}^*\}$			

Week4 wednesday

Regular sets are not the end of the story

- Many nice / simple / important sets are not regular

- Limitation of the finite-state automaton model: Can't "count", Can only remember finitely far into the past, Can't backtrack, Must make decisions in "real-time"

- We know actual computers are more powerful than this model...

The **next** model of computation. Idea: allow some memory of unbounded size. How?

- To generalize regular expressions: **context-free grammars**

- To generalize NFA: **Pushdown automata**, which is like an NFA with access to a stack: Number of states is fixed, number of entries in stack is unbounded. At each step (1) Transition to new state based on current state, letter read, and top letter of stack, then (2) (Possibly) push or pop a letter to (or from) top of stack. Accept a string iff there is some sequence of states and some sequence of stack contents which helps the PDA processes the entire input string and ends in an accepting state.

Is there a PDA that recognizes the nonregular language $\{0^n 1^n \mid n \geq 0\}$?

The PDA with state diagram above can be informally described as:

Read symbols from the input. As each 0 is read, push it onto the stack. As soon as 1s are seen, pop a 0 off the stack for each 1 read. If the stack becomes empty and we are at the end of the input string, accept the input. If the stack becomes empty and there are 1s left to read, or if 1s are finished while the stack still contains 0s, or if any 0s appear in the string following 1s, reject the input.

Trace the computation of this PDA on the input string 01.

Trace the computation of this PDA on the input string 011.

A PDA recognizing the set {   } can be informally described as:

> Read symbols from the input. As each 0 is read, push it onto the stack. As soon as 1s are seen, pop a 0 off the stack for each 1 read. If the stack becomes empty and there is exactly one 1 left to read, read that 1 and accept the input. If the stack becomes empty and there are either zero or more than one 1s left to read, or if the 1s are finished while the stack still contains 0s, or if any 0s appear in the input following 1s, reject the input.

Modify the state diagram below to get a PDA that implements this description:

Definition A **pushdown automaton** (PDA) is specified by a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where Q is the finite set of states, Σ is the input alphabet, Γ is the stack alphabet,

$$\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$$

is the transition function, $q_0 \in Q$ is the start state, $F \subseteq Q$ is the set of accept states.

Week4 friday

Draw the state diagram and give the formal definition of a PDA with $\Sigma = \Gamma$.

Draw the state diagram and give the formal definition of a PDA with $\Sigma \cap \Gamma = \emptyset$.

For the PDA state diagrams below, $\Sigma = \{0, 1\}$.

Mathematical description of language	State diagram of PDA recognizing language

$$\Gamma = \{\$, \#\}$$

$$\Gamma = \{@, 1\}$$

$$\{0^i 1^j 0^k \mid i, j, k \geq 0\}$$

Note: alternate notation is to replace ; with $\rightarrow$

Big picture: PDAs were motivated by wanting to add some memory of unbounded size to NFA. How do we accomplish a similar enhancement of regular expressions to get a syntactic model that is more expressive?

DFA, NFA, PDA: Machines process one input string at a time; the computation of a machine on its input string reads the input from left to right.

Regular expressions: Syntactic descriptions of all strings that match a particular pattern; the language described by a regular expression is built up recursively according to the expression's syntax

Context-free grammars: Rules to produce one string at a time, adding characters from the middle, beginning, or end of the final string as the derivation proceeds.

Week3 monday

Warmup: Design a DFA (deterministic finite automaton) and an NFA (nondeterministic finite automaton) that each recognize each of the following languages over $\{a, b\}$

$$\{w \mid w \text{ has an } a \text{ and ends in } b\}$$

$$\{w \mid w \text{ has an } a \text{ or ends in } b\}$$

Strategy: To design DFA or NFA for a given language, identify patterns that can be built up as we process strings and create states for intermediate stages. Or: decompose the language to a simpler one that we already know how to recognize with a DFA or NFA.

Recall (from Wednesday of last week, and in textbook Exercise 1.14): 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 $\{w \in \Sigma^* \mid w \notin A\}$.

Let's practice defining automata constructions by coming up with other ways to get new automata from old.

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.

**Example**: The language recognized by the NFA over $\{a, b\}$ with state diagram
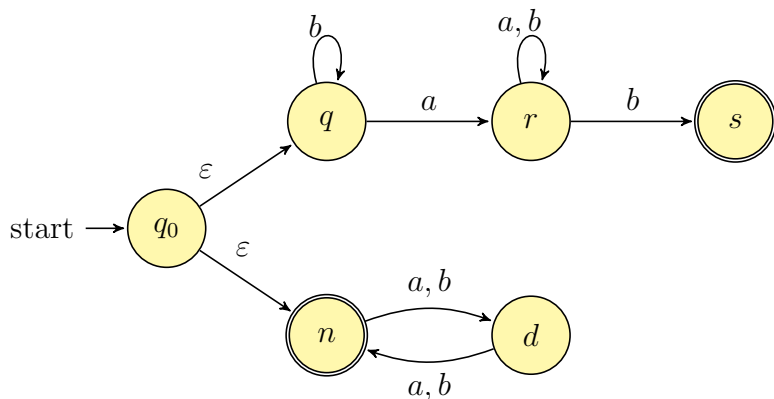
is:

Could we do the same construction with DFA?

Happily, though, an analogous claim is true!

Suppose A_1, A_2 are languages over an alphabet Σ. **Claim:** if there is a DFA M_1 such that $L(M_1) = A_1$ and DFA M_2 such that $L(M_2) = A_2$, then there is another DFA, let's call it M, such that $L(M) = A_1 \cup A_2$. *Theorem 1.25 in Sipser, page 45*

Proof idea:

Formal construction:
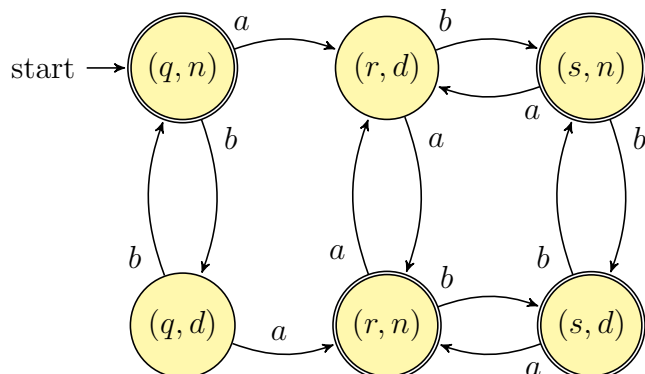
Example: When $A_1 = \{w \mid w \text{ has an } a \text{ and ends in } b\}$ and $A_2 = \{w \mid w \text{ is of even length}\}$.

Suppose A_1, A_2 are languages over an alphabet Σ. **Claim:** if there is a DFA M_1 such that $L(M_1) = A_1$ and DFA M_2 such that $L(M_2) = A_2$, then there is another DFA, let's call it M, such that $L(M) = A_1 \cap A_2$. *Sipser Theorem 1.25, page 45*

Proof idea:

Formal construction:

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 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.

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 $L((a^*b)^*)$:

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

Application: Find a regular expression describing the language recognized by the DFA with state diagram
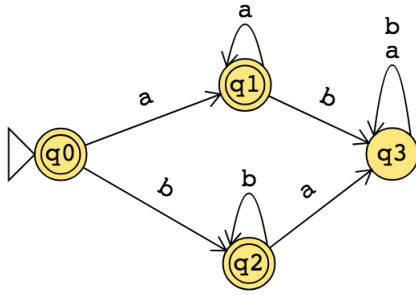
Week3 friday

Definition and Theorem: For an alphabet Σ, a language L over Σ is called **regular** exactly when L is recognized by some DFA, which happens exactly when L is recognized by some NFA, and happens exactly when L is described by some regular expression

We saw that: The class of regular languages is closed under complementation, union, intersection, set-wise concatenation, and Kleene star.

Prove or Disprove: There is some alphabet Σ for which there is some language recognized by an NFA but not by any DFA.

Prove or Disprove: There is some alphabet Σ for which there is some finite language not described by any regular expression over Σ.

Prove or Disprove: If a language is recognized by an NFA then the complement of this language is not recognized by any DFA.

Fix alphabet Σ. Is every language L over Σ regular?

Set	Cardinality
$\{0, 1\}$	
$\{0, 1\}^*$	
$\mathcal{P}(\{0, 1\})$	
The set of all languages over $\{0, 1\}$	
The set of all regular expressions over $\{0, 1\}$	
The set of all regular languages over $\{0, 1\}$	

Strategy: Find an **invariant** property that is true of all regular languages. When analyzing a given language, if the invariant is not true about it, then the language is not regular.

Pumping Lemma (Sipser Theorem 1.70): If A is a regular language, then there is a number p (a *pumping length*) where, if s is any string in A of length at least p, then s may be divided into three pieces, $s = xyz$ such that

- $|y| > 0$

- for each $i \geq 0$, $xy^i z \in A$

- $|xy| \leq p$.

Proof illustration

True or False: A pumping length for $A = \{0, 1\}^*$ is $p = 5$.

Week2 wednesday

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

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

- Start state q_0
- Accept (final) states F

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 δ is (rows labelled by states and columns labelled by symbols):

δ	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 $\{w \in \Sigma_1^* \mid w$ contains at most two 1's$\}$ is

A state diagram for a finite automaton that recognizes $\{w \in \Sigma_1^* \mid w$ 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 Σ if and only if $\delta^*(\, (q_0, w)\,) \in F$.

Suppose A is a language over an alphabet Σ. By definition, this means A is a subset of Σ^*. **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.

Week2 friday

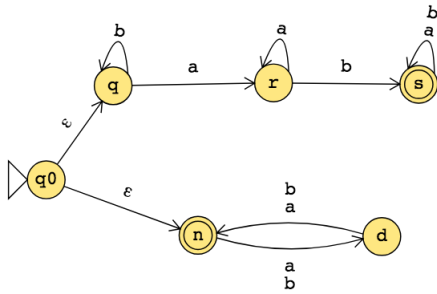The formal definition of the NFA over $\{0, 1\}$ given by this state diagram is:

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).

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

The language recognized by this NFA is:

Week1 monday

The CSE 105 vocabulary and notation build on discrete math and introduction to proofs classes. Some of the conventions may be a bit different from what you saw before so we'll draw your attention to them.

For consistency, we will use the notation from this class' textbook[1].

These definitions are on pages 3, 4, 6, 13, 14, 53.

Term	Typical symbol or **Notation**	Meaning
Alphabet	Σ, Γ	A non-empty finite set
Symbol over Σ	σ, b, x	An element of the alphabet Σ
String over Σ	u, v, w	A finite list of symbols from Σ
(The) empty string	ε	The (only) string of length 0
The set of all strings over Σ	Σ^*	The collection of all possible strings formed from symbols from Σ
(Some) language over Σ	L	(Some) set of strings over Σ
(The) empty language	$\emptyset$	The empty set, i.e. the set that has no strings (and no other elements either)
The power set of a set X	$\mathcal{P}(X)$	The set of all subsets of X
(The set of) natural numbers	$\mathcal{N}$	The set of positive integers
(Some) finite set		The empty set or a set whose distinct elements can be counted by a natural number
(Some) infinite set		A set that is not finite.
Reverse of a string w	$w^{\mathcal{R}}$	write w in the opposite order, if $w = w_1 \cdots w_n$ then $w^{\mathcal{R}} = w_n \cdots w_1$. Note: $\varepsilon^{\mathcal{R}} = \varepsilon$
Concatenating strings x and y	xy	take $x = x_1 \cdots x_m$, $y = y_1 \cdots y_n$ and form $xy = x_1 \cdots x_m y_1 \cdots y_n$
String z is a substring of string w		there are strings u, v such that $w = uzv$
String x is a prefix of string y		there is a string z such that $y = xz$
String x is a proper prefix of string y		x is a prefix of y and $x \neq y$
Shortlex order, also known as string order over alphabet Σ		Order strings over Σ first by length and then according to the dictionary order, assuming symbols in Σ have an ordering

---

[1]Page references are to the 3rd edition of Sipser's Introduction to the Theory of Computation, available through various sources for approximately $30. You may be able to opt in to purchase a digital copy through Canvas. Copies of the book are also available for those who can't access the book to borrow from the course instructor, while supplies last (minnes@ucsd.edu)

Write out in words the meaning of the symbols below:

$$\{a, b, c\}$$

$$|\{a, b, a\}| = 2$$

$$|aba| = 3$$

Circle the correct choice:

A **string** over an alphabet Σ is __an element of Σ^*__ OR __a subset of Σ^*__.

A **language** over an alphabet Σ is __an element of Σ^*__ OR __a subset of Σ^*__.

With $\Sigma_1 = \{0, 1\}$ and $\Sigma_2 = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\}$ and $\Gamma = \{0, 1, x, y, z\}$

True or **False**: $\varepsilon \in \Sigma_1$

True or **False**: ε is a string over Σ_1

True or **False**: ε is a language over Σ_1

True or **False**: ε is a prefix of some string over Σ_1

True or **False**: There is a string over Σ_1 that is a proper prefix of ε

The first five strings over Σ_1 in string order, using the ordering $0 < 1$:

The first five strings over Σ_2 in string order, using the usual alphabetical ordering for single letters:

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

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
$R_1 R_2$	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 $q0, q1, \ldots$

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

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

a

b

b
a

q0   b   q1   a   q2

The language recognized by this automaton is

b

b
a

q1   b   q2

a   b

a

a

q0   b   q3

The language recognized by this automaton is

Week10 friday

Model of Computation	Class of Languages
Deterministic finite automata: formal definition, how to design for a given language, how to describe language of a machine? **Nondeterministic finite automata**: formal definition, how to design for a given language, how to describe language of a machine? **Regular expressions**: formal definition, how to design for a given language, how to describe language of expression? *Also*: converting between different models.	**Class of regular languages**: what are the closure properties of this class? which languages are not in the class? using **pumping lemma** to prove nonregularity.
Push-down automata: formal definition, how to design for a given language, how to describe language of a machine? **Context-free grammars**: formal definition, how to design for a given language, how to describe language of a grammar?	**Class of context-free languages**: what are the closure properties of this class? which languages are not in the class?
Turing machines that always halt in polynomial time Nondeterministic Turing machines that always halt in polynomial time	P NP
Deciders (Turing machines that always halt): formal definition, how to design for a given language, how to describe language of a machine?	**Class of decidable languages**: what are the closure properties of this class? which languages are not in the class? using diagonalization and mapping reduction to show undecidability
Turing machines formal definition, how to design for a given language, how to describe language of a machine?	**Class of recognizable languages**: what are the closure properties of this class? which languages are not in the class? using closure and mapping reduction to show unrecognizability

Given a language, prove it is regular

Strategy 1: construct DFA recognizing the language and prove it works.

Strategy 2: construct NFA recognizing the language and prove it works.

Strategy 3: construct regular expression recognizing the language and prove it works.

"Prove it works" means . . .

Example: $L = \{w \in \{0,1\}^* \mid w \text{ has odd number of 1s or starts with } 0\}$

Using NFA

Using regular expressions

Example: Select all and only the options that result in a true statement: "To show a language A is not regular, we can..."

a. Show A is finite

b. Show there is a CFG generating A

c. Show A has no pumping length

d. Show A is undecidable

Example: What is the language generated by the CFG with rules

$$S \rightarrow aSb \mid bY \mid Ya$$
$$Y \rightarrow bY \mid Ya \mid \varepsilon$$

Example: Prove that the language $T = \{\langle M \rangle \mid M$ is a Turing machine and $L(M)$ is infinite$\}$ is undecidable.

Example: Prove that the class of decidable languages is closed under concatenation.