Notes on the Theory of Computation

Jonathan Cui

Ver. 20240325

1 Regular Languages

1.1 Finite Automata

Finite automata, or finite-state machines, form the simplest computational model. Our everyday computers are a prototypical example that shows just how powerful and versatile these machines can be.

Intuitively, a finite automaton is defined by a diagram like the following example.

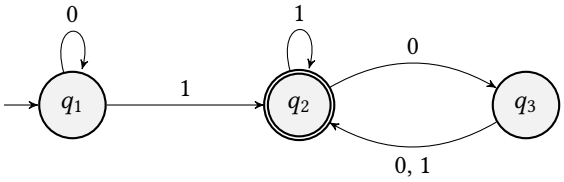


Figure 1: A finite automaton M_1

A finite automaton first defines a finite, non-empty alphabet Σ, which will be the input to the state machine at each time. Each circle, or node, represents one of finitely many states, which are q_1, q_2, q_3 in the example. Exactly one arrow points from nothing to a state, which defines the start state. Each state has $|\Sigma|$ outgoing arrows, where each character of Σ is assigned to exactly one arrow. Upon each input, the machine transitions to the next state following the arrows. At the end of the input, the machine stops at a state which may or may not be double-circled. If so, the machine is said to accept the input sequence of characters; otherwise, the machine is said to reject the input. The states that are doubly circled are called the accept states.

In the example above, $\Sigma = \{0, 1\}$. M_1 would reject 0, accept 1, reject 10, accept 1101, and so on. In fact, one can note that M_1 accepts all binary strings containing at least one 1 that have an even number of 0's after the last 1.

This idea is formalized in set-theory notation as follows.

Definition 1.1. A finite automaton M is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

- Q is the finite set of states;
- Σ is the finite, non-empty alphabet set;
- $\delta\colon Q \times \Sigma \to Q$ is the transition function;
- $q_0 \in Q$ is the initial state;
- $F \subseteq Q$ is the set of accept states.

The input is a sequence or string from the alphabet. We define this more specifically.

Definition 1.2. Suppose Σ is a finite, non-empty alphabet. A string $s = (s_1, \cdots, s_n)$ for $n \in \mathbb{Z}_{\geq 0}$ is a finite sequence of

elements in Σ, denoted simply as $s_1 \cdots s_n$, where $s_1, \cdots, s_n \in \Sigma$. The empty string is denoted as ϵ and the collection of all strings over Σ is denoted as Σ^*. We denote string concatenation as $s_1 s_2$, where $s_1, s_2 \in \Sigma^*$. A language over Σ is a subset of the strings Σ^*. The length of a string is denoted as $|\cdot| : \Sigma^* \to \mathbb{Z}_{\geq 0}$.

We now define the computation of a finite automaton using the formal definitions, which is the description of the sequence of states we obtain by passing through each character of the input string.

Definition 1.3. Suppose $M = (Q, \Sigma, \delta, q_0, F)$ is a finite automaton and $w = w_1 \cdots w_n \in \Sigma^*$ is the input string ($|w| = n$). The finite automaton carries out the following computation:

- Let $r_0 \leftarrow q_0$ be the initial state;
- For each $i = 1, \cdots, n$, let $r_i \leftarrow \delta(r_{i-1}, w_i)$.

If the final state is accepted, that is, $r_n \in F$, then M is said to accept w. Otherwise, M is said to reject w. The language of M, denoted as $L(M) \subseteq \Sigma^*$, is the collection of all strings that M accepts. M is said to recognize the language $L(M)$.

The theoretical significance lies in the class of all languages that finite automata can recognize, the regular languages. They correspond to regular expressions, which we frequently use in day-to-day programming.

Definition 1.4. A language A over a finite, non-empty alphabet Σ is said to be regular if there exists finite automaton that recognizes A. Two machines recognizing the same language are said to be equivalent.

For example, the language of binary strings that represent even integers is a regular language. To show this, we construct an explicit instance of a finite automaton that recognizes this language.

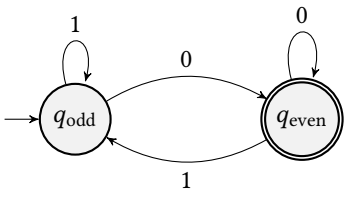


Figure 2: A finite automaton M_2 that recognizes even binary numbers.

We start at q_{odd} to avoid accepting the empty string, which is technically not a number, much less even. Regardless of the previous state, M would go to state q_{odd} if the input is 1 and go to q_{even} if the input is 0. In this way, only the parity of the final input character matters, which is exactly the math we know.

A slightly more complicated example is to recognize the language of binary strings that contain 001 as a substring. The starting point is that the states $q_\varnothing, q_0, q_{00}, q_{001}$ should keep track of the last couple characters. This motivates the following design.

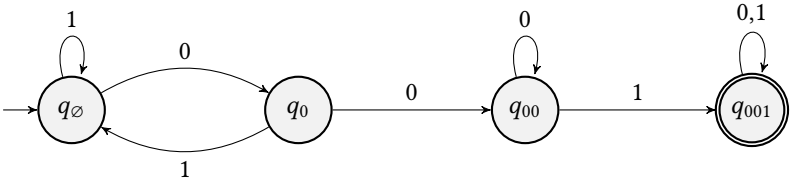


Figure 3: A finite automaton M_3 that recognizes binary strings with substring 001.

We now develop some introductory theory on regular languages. We define some sensible operations first, and show that the class of regular language is closed under these regular operations.

Definition 1.5. Suppose A and A' are languages over a finite, non-empty alphabet Σ. The regular operations are

- **Union.** The language $A \cup A'$ over Σ;

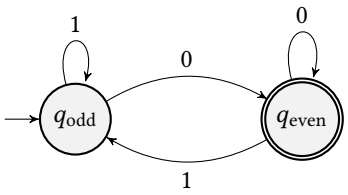


Figure 4: The finite automaton M_2 that recognizes even binary numbers.

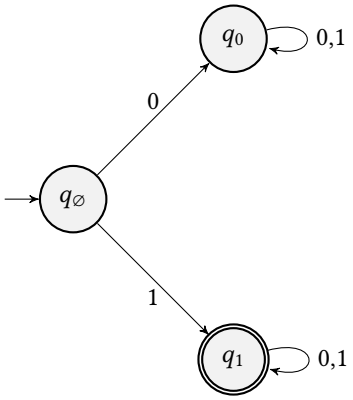


Figure 5: A finite automaton M_4 that recognizes binary string starting with a 1.

- **Concatenation.** The language $A \circ A' := \{ww' \mid w \in A, w' \in A'\}$ over Σ;
- **Kleene star.** The language $A^* = \bigcup_{n=0}^{\infty} \{w_1 \cdots w_n \mid w_1, \cdots, w_n \in A\}$ over Σ.

Note that concatenation is associative, which allows us to write expressions like $A \circ B \circ C$ without ambiguity because the order of the $\circ$'s we evaluate does not affect the result. This is also obvious for $\cup$.

Here are some examples of the regular operations.:

- **Union.** If $A = \{1, 01, 001, 0001, \cdots\}$ and $A' = \{101\}$, then $A \cup A' = \{101, \quad 1, 01, 001, 0001, \cdots\}$;
- **Concatenation.** With $A = \{1, 01, 001, 0001, \cdots\}$ and $A' = \{101, 111\}$, $A \circ A' = \{1101, 1111, 01101, 01111, 001101, 001111, \cdots\}$;
- **Kleene star.** With $A = \{0, 01\}$, $A^* = \{\epsilon, \quad 0, 01, \quad 00, 001, 010, 0101, \quad \cdots\}$.

We already have the machinery to address the closure of regular language under unions. We will assume the languages share the same alphabet, which is not as restrictive as it may first appear to be. Two languages $A_1 \in \Sigma_1^*$ and $A_2 \in \Sigma_2^*$ can be extended to share the same alphabet by considering $\Sigma := \Sigma_1 \cup \Sigma_2$ such that $A_1 \in \Sigma_1^* \subseteq \Sigma^*$ and $A_2 \in \Sigma_2^* \subseteq \Sigma^*$. Technically, we might need to create an additional "dead state" to represent whenever we receive as input from the unused part of the alphabet. Hereafter, we will assume without loss of generality that all languages in discussion are over a common alphabet.

We now show our first closure results: regular languages are closed under unions. That is, given two languages $A_1 = L(M_1)$ and $A_2 = L(M_2)$, we need to construct another M so that $L(M) = A_1 \cup A_2$. We can achieve this by running the two in parallel: since the sequences of states produced by M_1 and M_2 will have the same length (of $|w| + 1$) given a common input $w \in \Sigma^*$, we can "zip" their states together to keep track of both simultaneously. This is achieved by the Cartesian product, which we detail as follows.

Proposition 1.6. Suppose A_1 and A_2 are regular languages over a finite, non-empty alphabet Σ. Then, the union $A_1 \cup A_2$ is also a regular language.

Proof. Suppose finite automata $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognizes A_1 and $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognizes A_2. We construct another finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ that recognizes $A_1 \cup A_2$. Let $Q = Q_1 \times Q_2$, $q_0 = (q_1, q_2)$, and $F = \{(r_1, r_2) \in Q \mid r_1 \in F_1 \vee r_2 \in F_2\}$. Further, define

$$\delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a)).$$

Note that the two components of the state pair are independent or constant with respect to each other. If $w \in A_1$, then the final state $r_n = (r_1, r_2)$ must be in F since $r_1 \in F_1$. Similarly, if $w \in A_2$, then $r_n \in F$ as well since $r_2 \in F_2$. If $w \notin A_1 \cup A_2$, then

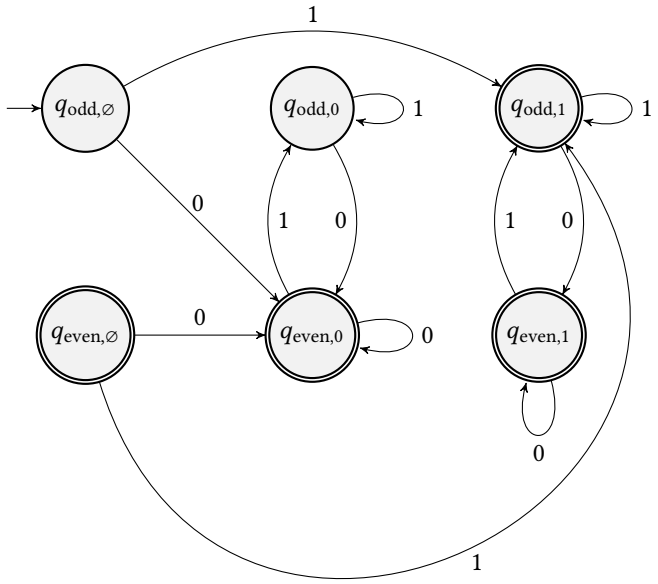


Figure 6: The finite automaton M_5 that recognizes even binary numbers as well as binary strings starting with 1.

neither $r_1 \in F_1$ nor $r_2 \in F_2$, and hence $r_n \notin F$. □

Note that we combine M_2 and M_4 to produce M_5 such that $L(M_5) = L(M_2) \cup L(M_4)$.

Now, just by changing how we construct the new accept states F, we can easily prove that regular languages are also closed under intersections. While it's not called a regular operation, this is still quite a useful fact to know.

Proposition 1.7. Suppose A_1 and A_2 are regular languages over a finite, non-empty alphabet Σ. Then, the intersection $A_1 \cap A_2$ is also a regular language.

Proof. Suppose finite automata $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognizes A_1 and $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognizes A_2. We construct another finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ that recognizes $A_1 \cap A_2$. Let $Q = Q_1 \times Q_2$, $q_0 = (q_1, q_2)$, and $F = F_1 \times F_2$. Further, define

$$\delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a)).$$

It is obvious from construction that M recognizes $A_1 \cap A_2$. □

We also have the machinery to think about complements.

Definition 1.8. Suppose A is a language over Σ. The complement of A, denoted as $\overline{A}$, is defined as the strings not in A, namely $\Sigma^* \backslash A$.

How do we achieve this? We can simply "toggle" the accept states to the complement in Q. All accept states before are now reject states and all reject states before are now accept states. Clearly, this construction "toggles" the strings we recognize.

Proposition 1.9. Suppose A is a regular language over a finite, non-empty alphabet Σ. Then, the complement $\overline{A}$ is also a regular language.

Proof. Suppose $M = (Q, \Sigma, \delta, q_0, F)$ recognizes A. Let $M' = (Q, \Sigma, \delta, q_0, Q \backslash F)$, and consider an input string $w = w_1 \cdots w_n \in \Sigma^*$, where $n = |w|$. By construction, M and M' follow the same computational sequence, yielding the same final state $r_n \in Q$.

Then,

$$\begin{aligned} M \text{ accepts } w &\iff r_n \in F \\ &\iff r_n \notin Q \backslash F \\ &\iff M' \text{ rejects } w. \end{aligned}$$

Because w is chosen arbitrarily, the above equivalence establishes that M' recognizes $\overline{A}$. □

1.2 Non-Determinism

Now, let's consider the operation of string reversal.

Definition 1.10. Let $w = w_1 \cdots w_n \in \Sigma^*$ be a string of length n. The reversal of w, denoted as w^R, is defined as $w_n \cdots w_1 \in \Sigma^*$. Given a language $A \subseteq \Sigma^*$, the reversal of A is defined as $A^R := \{w^R \mid w \in A\}$.

For example, $(00101)^R = 10100$, and $\{w \in \{0,1\}^* \mid w \text{ starts with a } 1\}^R = \{w \in \{0,1\}^* \mid w \text{ ends with a } 1\}$.

It shouldn't be too surprising that strings can be recognized backwards. How would we go about proving this? To construct a concrete DFA, we need to reverse the state sequence $r_0(= q_0), \cdots, r_n \in Q$. So we swap the initial and accept states and reverse all the arrows.

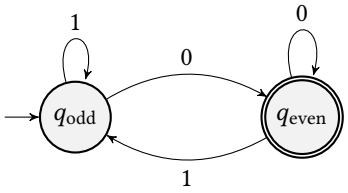


Figure 7: The finite automaton M_2 that recognizes even binary numbers.

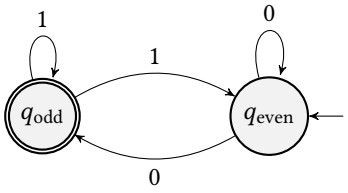


Figure 8: The "finite automaton" M_2^R that recognizes binary strings starting with 1.

In this way, we can back-trace this sequence from the new start state to a new accept state—except we can't. First, we can have multiple accept states, but the new initial state has to be unique. What's more, the reversed arrows are not even necessarily functions $Q \times \Sigma \to Q$ anymore. Consider M_2 that recognizes even integers. In M_2^R, we have two outgoing arrows labeled with 1 from q_{odd}! Even worse, some arrows are missing. What happens when we have input 0 at state q_{odd}? All these issues mean that the reversal cannot be a DFA. So what now? Well, all we need to accept a string is to have *some* path along arrow transitions. If not, we know that the original string cannot possibly be accepted by the original DFA.

For $L(M_1) \circ L(M_2)$, we could try to chain the accept states of M_1 with the initial state of M_2 by adding arrows, but this is a many-to-one relationship, and the transition should be optional and can happen without any input: we may move from an accept state in M_1 to another non-accept state or move to the next machine. The *existence* of some path along arrows of the machine should define acceptance.

We begin our introduction to nondeterminism, which is an idea much more general in the theory of computation, by defining non-deterministic finite automata (NFA) that addresses the necessities above.

We start with an elementary example.

This example showcases the advantages of an NFA. N_1 recognizes the union of $\{0\}$ and $\{w \mid w \text{ contains } 001\}$. By using ϵ arrows, we *allow* for transitions to either "actual" start states to recognize either possibilities. By having two possibilities for $q_\varnothing^2$ upon input 0, the state can choose to loop over itself arbitrarily many times as needed since we require only *some* path to reach an accept state. By not defining a next state for q_0^1 upon 0, the *existence* of any such path with the input sequence is impossible.

How, then, do we keep track of the multiple possibilities for paths? When more than one next state is specified, we "fork" the current path along the arrows into different branches that act simultaneously to subsequent input. If at some input, the

next state for a branch is undefined, then this branch dies and is not accepted. If we have moved to a state where a next state arrow exists with ϵ, then we add another branch to the current path going to that state before the next input. If any branch is accepted at the end of the input, the NFA accepts the input. Otherwise, the NFA rejects the input. If branches are led to the same next state upon some input, the branches merge to one for subsequent calculation. To keep track of the calculation, which requires considering all possibilities simultaneously, we use a set to describe all current states that can be reached along *some* path of arrows up to the current input.

Let's consider the input 0010. Note that initially, we have $\{q_0, q^1_\varnothing, q^2_\varnothing\}$ which includes q_0, not utilizing the optional ϵ arrows. Of course, this branch dies upon any input, so in effect we have two initial states $q^1_\varnothing$ and $q^2_\varnothing$.

- The start state is q_0, and we have optional ϵ arrows pointing two other states. We initially have $\{q_0, q^1_\varnothing, q^2_\varnothing\}$;
- Upon input 0, q_0 dies, $q^1_\varnothing$ moves to q^1_0, and $q^2_\varnothing$ moves to itself $q^2_\varnothing$ and q^2_0. We now have $\{q^1_0, q^2_\varnothing, q^2_0\}$;
- Upon input 0, q^1_0 dies, $q^2_\varnothing$ moves to $q^2_\varnothing$ and q^2_0, and q^2_0 moves to q^2_{00}. We now have $\{q^2_\varnothing, q^2_0, q^2_{00}\}$;
- Upon input 1, $q^2_\varnothing$ moves to $q^2_\varnothing$, q^2_0 dies, and q^2_{00} moves to q^2_{001}. We now have $\{q^2_\varnothing, q^2_{001}\}$;
- Upon input 0, $q^2_\varnothing$ moves to $q^2_\varnothing$ and q^2_0, and q^2_{001} moves to q^2_{001}. We eventually have $\{q^2_\varnothing, q^2_0, q^2_{001}\}$.

Because an accept state q^2_{001} is in the eventual set of states, we conclude that N_1 accepts 0010 as expected. Indeed, the accepted branch ends at the branch describing the strings containing 001. Note that the branching-out allows us to match 0 or more characters at the beginning and the end of a string by additional self-loops with the entire alphabet.

We now formalize NFAs using the set-theoretic notation. We first introduce a new notation related to the ϵ arrows.

Definition 1.11. Suppose Σ is a finite, non-empty alphabet (which we hereafter assume does not contain ϵ). Let $\underline{\Sigma_\epsilon}$ denote $\Sigma \cup \{\epsilon\}$, the alphabet with the empty ϵ symbol included.

The only modifications to DFA's formal definition are that we now use the alphabet Σ_ϵ and track instead *sets* of states. The absence of arrows from state $q \in Q$ upon input a is interpreted as $\delta(q, a) \coloneqq \varnothing$.

Definition 1.12. A non-deterministic finite automaton N is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

- Q is the finite set of states;
- Σ is the finite, non-empty alphabet set;
- $\delta\colon Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$ is the transition function;
- $q_0 \in Q$ is the initial state;
- $F \subseteq Q$ is the set of accept states.

Importantly, $\delta\colon Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$ maps each state (branch) and an input, possibly ϵ, to a collection of next states, possibly empty. We now allow δ to take as input any string in Σ^* with ϵ's inserted anywhere in the string to achieve the effect of ϵ arrows. This is formalized as follows.

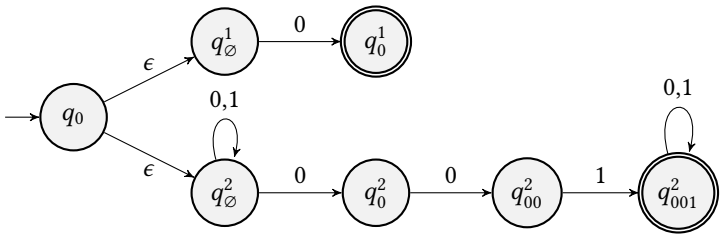


Figure 9: A non-deterministic finite automaton N_1.

Definition 1.13. Suppose $N = (Q, \Sigma, \delta, q_0, F)$ is a non-deterministic finite automaton and $w = w_1 \cdots w_n \in \Sigma^*$ is the input string. Then, N is said to accept w if w can be written as a length-m string $w = y_1 \cdots y_m$ ($m \geq n$), with $y_i \in \Sigma_\epsilon$ for $i \in \{1, \cdots, m\}$, such that there exists a sequence of states $r_0, \cdots, r_m \in Q$ satisfying

- $r_0 = q_0$;
- $r_i \in \delta(r_{i-1}, y_i)$ for each $i = 1, \cdots, m$;
- $r_m \in F$.

Otherwise, N is said to reject w. The collection of all input strings accepted by N is defined as the language of N, denoted as $L(N) \subseteq \Sigma^*$.

Note the different formulation for computing an NFA. As noted before, NFAs require only the *existence* of ***some*** sequence of states in each set of current states, instead of *computing* ***the*** sequence of states for DFAs, where the transition function allows a path of arrows through these states. If a branch meets an undefined next state by δ upon an input character, then there wouldn't be a corresponding sequence of length m because any sequence starting with this branch up to the current input will encounter an empty set for $\delta(r_{i-1}, y_i)$, making it impossible to choose any r_i. Further, the formulation means $w = y_1 \cdots y_m$ is written as an expanded string with (optional) ϵ's added wherever we need. The existence of such y_i's mean we can choose whether or not to use any ϵ arrows as necessary to reach an accept state.

A remarkable fact is that DFAs and NFAs are equivalent in the sense that they recognize the same class of languages. Clearly, a DFA is a NFA by using no ϵ arrows and making the new $\delta' : Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$ by $\delta'(r, a) = \{\delta(r, a)\}$ to force a unique path. But it is not at all obvious that NFAs, which can branch out indefinitely as needed, are no more powerful than DFAs in the sense that it could recognize more languages.

Notably, the 5-tuple looks formally the same for DFAs and NFAs, and indeed this is how we will tackle the proof of equivalence. Importantly, the power set of a finite set is also finite, so a larger DFA can keep track of all $2^{|Q|}$ possible paths, whether or not used, simultaneously. The states traversed by an NFA form a tree where all leaves share the same depth. Because we always work at the same level, as before, we can track all states at the same time with a collection of states with inputs up to the current one. We'll also consider all possible next states including those further directed by ϵ arrows.

To tackle ϵ arrows, we define a notation for all next states including those with further ϵ arrows.

Definition 1.14. Suppose $N = (Q, \Sigma, \delta, q_0, F)$ be a non-deterministic finite automaton. Given a collection of states $R \subseteq Q$, define $E(R) \subseteq Q$ as the set of states that can be reached from R by traveling along $n = 0$ or more ϵ arrows; that is,

$$E(R) = \{q \in Q \mid \text{there exist } n \geq 0 \text{ and } r_0, \cdots, r_n \in Q \text{ such that } r_0 \in R,\ r_i \in \delta(r_{i-1}, \epsilon), \text{ and } r_n = q\}.$$

Note that $R \subseteq E(R)$, where the equality holds if and only if N has no ϵ arrows from any states in R.

Alternatively, let $\vdash_\epsilon \subseteq Q \times Q$ be the ϵ step relation where $q_1 \vdash_\epsilon q_2$ iff $q_2 \in \delta(q_1, \epsilon)$, that is, some ϵ arrow goes from q_1 to q_2. Then, the reflexive and transitive closure[1] $\vdash_\epsilon^*$ induces $E \colon R \to \mathcal{P}(Q)$ by $E(R) \coloneqq \{q \in Q \mid \exists r \in R, r \vdash_\epsilon^* q\}$.

Theorem 1.15. *The DFAs and the NFAs are equivalent in the sense that they recognize the same languages.*

Proof. It is obvious that any language recognized by a DFA is also recognized by an NFA. To show the converse, suppose a language A is recognized by an NFA $N = (Q, \Sigma, \delta, q_0, F)$. Define $Q' \coloneqq \mathcal{P}(Q)$, $q_0' \coloneqq E(\{q_0\})$, and $F' \coloneqq \{R \in \mathcal{P}(Q) \mid F \cap R \neq \varnothing\}$. For all $R \in Q'$ and $a \in \Sigma$, define

$$\delta'(R, a) = \bigcup_{r \in R} E(\delta(r, a)).$$

Then, for $j = 1, \cdots, n$,

$$R_j = \bigcup_{r \in R_{j-1}} E(\delta(r, w_j)).$$

[1] That is, the smallest reflexive and transitive relation containing $\vdash_\epsilon$.

Suppose a length-n input $w = w_1 \cdots w_n$ has been rewritten as $y_1 \cdots y_m$ with $y_1, \cdots, y_m \in \Sigma_\epsilon$ with N processing the sequence of states $r_0, \cdots, r_m \in Q$. Let $r_{i_0}, \cdots, r_{i_n}$ be the subsequence of states where each either has a non-ϵ next input or is the final state. Let $R_0, \cdots, R_n \subseteq Q$ be the states of the constructed DFA. We show by induction that for $j = 0, \cdots, n$, $r_{i_j} \in R_j$.

Base case. Because $R_0 = q_0' = E(\{q_0\})$, $r_0, \cdots, r_{i_0}$ follows a path of ϵ arrows reaching r_{i_0} and hence $r_{i_0} \in R_0$.

Inductive case. Now suppose $r_{i_{j-1}} \in R_{j-1}$. Because $R_j = \bigcup_{r \in R_{j-1}} E(\delta(r, w_j))$, we have $E(\delta(r_{i_{j-1}}, w_j)) \subseteq R_j$. Now, $w_j = y_{i_{j-1}+1}$ implies $E(\{r_{i_{j-1}+1}\}) \subseteq E(\delta(r_{i_{j-1}}, y_{i_{j-1}+1})) \subseteq R_j$. By construction, $r_{i_{j-1}+1} \vdash_\epsilon^* r_{i_j}$. Then, by definition, $r_{i_j} \in E(\{r_{i_{j-1}+1}\}) \subseteq R_j$. The induction is complete.

In particular, we have shown $r_{i_n} = r_m \in R_n$. If N accepts $w = y_1 \cdots y_m$, that is, $r_m \in F$, then M will also accept w.

Now suppose instead that M accepts w with states $R_0, \cdots, R_n \subseteq Q$. Fix a particular final accept state $r^* \in R_n \cap F \neq \varnothing$. Each state transition recursively defines r_{i_j}, and we set intermediate states to those explored on the path from the definition of $E(\,\cdot\,)$ to find a possible sequence $r_0, \cdots, r_m \in Q$ of states ending in an accept state. Thus, N accepts w as well. □

This equivalence gives rise to the following equivalent condition to regular languages.

Corollary 1.16. A language $A \subseteq \Sigma^*$ is regular if and only if some non-deterministic finite automaton N recognizes A.

The NFA is much more suited to showing that regular languages are closed under regular operations. The fact that a branch can die allows us to handle mixed alphabet, where a "wrong" character for a branch means acceptance is impossible for this branch. The use of ϵ arrows allows us to account for all possibilities simultaneously with a straightforward construction.

We first show the closure under unions, now with NFAs.

Theorem 1.17. *The class of regular languages is closed under unions.*

Proof. Suppose A_1 and A_2 are languages over Σ recognized by $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ respectively. We construct a new NFA $N = (Q, \Sigma, \delta, q_0, F)$ that recognizes $A_1 \cup A_2$ as follows.

Without loss of generality, suppose Q_1 and Q_2 are disjoint, which is always possible be renaming states. Create a new symbol q_0 for an artificial initial state. Let $Q = \{q_0\} \cup Q_1 \cup Q_2$ be the set of states, q_0 the initial state, and $F = F_1 \cup F_2$. Define $\delta\colon Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$ by

$$\delta(q, a) = \begin{cases} \{q_1, q_2\}, & \text{if } q = q_0 \text{ and } a = \epsilon, \\ \varnothing, & \text{if } q = q_0 \text{ and } a \neq \epsilon, \\ \delta_1(q, a), & \text{if } q \in Q_1, \\ \delta_2(q, a), & \text{if } q \in Q_2. \end{cases}$$

It is obvious from construction that N recognizes $A_1 \cup A_2$. □

The argument above is barely more elegant than before, but it uses significantly less states (addition vs. multiplication).

As promised before, we now tackle string reversal. Not that for the input language, we can use either a DFA or an NFA, whichever is easier. DFAs are easier to handle, so we'll go with that.

Proposition 1.18. Let A be a regular language over Σ. Then, A^R is also a regular language.

Proof. Suppose $M = (Q, \Sigma, \delta, q_0, F)$ is a DFA recognizing A. Define an NFA $N = (\{q_{\text{start}}\} \cup Q, \Sigma, \delta', q_{\text{start}}, \{q_0\})$ where

$$\delta'(q, a) \coloneqq \begin{cases} F, & \text{if } q = q_{\text{start}},\ a = \epsilon, \\ \varnothing, & \text{if } q = q_{\text{start}},\ a \neq \epsilon, \\ \{q_{\text{prev}} \in Q \mid \delta(q_{\text{prev}}, a) = q\} & \text{if } q \in Q. \end{cases}$$

Suppose M accepts $w = w_1 \cdots w_n \in \Sigma^*$ (where $|w| = n$), which yields states $r_0, \cdots, r_n \in Q$. Then, $r'_0, \cdots, r'_{n+1} \in \{q_{\text{start}}\} \cup Q$ is a valid sequence of states for N where $r'_0 = q_{\text{start}}$ and $r'_i = r_{n-i+1}$ for all $i \in \{1, \cdots, n+1\}$, upon input ϵw^R. Specifically, when $i = 1$, $r'_i = r_n \in \delta'(r'_{i-1}, \epsilon) = F$. When $i > 1$, $r'_i = r_{n-i+1} \in \delta'(r'_{i-1}, w_{n-i+2}) = \{q_{\text{prev}} \in Q \mid \delta(q_{\text{prev}}, w_{n-i+2}) = r_{n-i+2}\}$ because $\delta(r_{n-i+1}, w_{n-i+2}) = r_{n-i+2}$. Because $r_0 = q_0 \in \{q_0\}$, N accepts w.

Suppose instead that N accepts w with states $r'_0, \cdots, r'_{n+1} \in \{q_{\text{start}}\} \cup Q$, which by construction of N must be of the form ϵw^R. Then, by similar reasoning, $r'_{n+1}, \cdots, r'_1 \in Q$ must the (unique) sequence of states produced by M on input w. Because $r'_1 \in F$, M accepts w. Hence, $L(N) = A^R$, and thus A^R is a regular language. □

The power of NFAs is also seen through the following argument that justifies the closure of regular languages under concatenation. Since more than one substring including the start may be recognized by the first language, it becomes non-trivial to determine when to move to the next NFA. We use the machinery of ϵ arrows to achieve this.

Theorem 1.19. *The class of regular languages is closed under concatenation.*

Proof. Suppose $A_1 \in \Sigma^*$ and $A_2 \in \Sigma^*$ are regular languages recognized by $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ respectively. Without loss of generality, assume Q_1 and Q_2 are disjoint.[2] We construct a new NFA $N = (Q, \Sigma, \delta, q_1, F_2)$ that recognizes $A_1 \circ A_2$.

Let $Q = Q_1 \cup Q_2$ and define $\delta\colon Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$ by

$$\delta(q,a) = \begin{cases} \delta_1(q,a), & \text{if } q \in Q_1 \backslash F_1, \\ \delta_1(q,\epsilon) \cup \{q_2\}, & \text{if } q \in F_1 \text{ and } a = \epsilon, \\ \delta_1(q,a), & \text{if } q \in F_1 \text{ and } a \neq \epsilon, \\ \delta_2(q,a), & \text{if } q \in Q_2. \end{cases}$$

If $w = xy \in A_1 \circ A_2$ where $x \in A_1$ and $y \in A_2$ with $x = \tilde{x}_1 \cdots \tilde{x}_m$ and $y = \tilde{y}_1 \cdots \tilde{y}_n$ ($\tilde{x}_1, \cdots \tilde{x}_m, \tilde{y}_1 \cdots \tilde{y}_n \in \Sigma_\epsilon$), then we will have moved to a state in F_1 upon input $\tilde{x}_1, \cdots, \tilde{x}_m$, and we may by the ϵ arrow move to q_2. Taking $\tilde{y}_1, \cdots, \tilde{y}_n$ then leads to a state in F_2, and N accepts w.

If N accepts $w = w_1 \cdots w_\ell$ ($|w| = \ell$) then there must be exactly one time when the ϵ arrow is used, which is only possible when N arrives at a state in F_1, matching A_1. Because N accepts w, N arrives at a state in F_2, so the string following the matching of A_1 must match A_2. Therefore, $w \in A_1 \circ A_2$, and $L(N) = A_1 \circ A_2$. □

Our last regular operation of interest is the Kleene star, which comprises 0 or more strings from a language concatenated. We will make use of the ϵ arrows to allow optional return from any accept state to the initial state for another string from the language.

Proposition 1.20. Suppose A is a regular language over Σ. Then, A^* is also a regular language.

Proof. Suppose $N = (Q, \Sigma, \delta, q_0, F)$ is an NFA recognizing A. Then, $N' = (Q, \Sigma, \delta', q_0, F \cup \{q_0\})$ is an NFA recognizing A^*, where

$$\delta'(q,a) \coloneqq \begin{cases} \delta(q,a) \cup \{q_0\}, & \text{if } q \in F,\ a = e, \\ \delta(q,a), & \text{if } q \in F,\ a \neq e, \\ \delta(q,a), & \text{if } q \in Q \backslash F. \end{cases}$$

If $w = w_1 \cdots w_k$, where $w_1, \cdots, w_k \in A$ for some $k \geq 0$, then there is a path $(q_0 \to \cdots \to f_1 \in F) \xrightarrow{\epsilon} \cdots \xrightarrow{\epsilon} (q_0 \to \cdots \to f_n \in F)$ that N will accept. Similarly, if N accept w using ϵ arrows $(k-1)$ times, then it must have been that $w = w_1 \epsilon w_2 \epsilon \cdots \epsilon w_n$, where each w_i for $i = 1, \cdots, n$ is accepted by M and thus $w_i \in A$. Therefore, $w \in A^*$, and $L(N) = A^*$. □

[2] Otherwise, replace $Q_1 \leftarrow Q_1 \times \{1\}$ and $Q_2 \leftarrow Q_2 \times \{0\}$, adjusting $\delta_{1,2}$, $q_{1,2}$, and $F_{1,2}$

1.3 Regular Expressions

Regular expressions, also known as regexes, are expressions that are used in everyday programming to match a certain type of strings. For example, when you write an app and you need an email format checker before a user tries to login, you can write something like

$$\Sigma^+(.\Sigma^+)^*@\Sigma^+(.\Sigma^+)^+$$

to match email addresses with the alphabet $\Sigma \cup \{@,\ .\}$. We define regular expressions as follows.

Definition 1.21. Regular expressions over an alphabet Σ, denoted as $\mathcal{R}[\Sigma]$ or simply $\mathcal{R}$, are strings with alphabet $\Sigma \cup \{\text{“(”, “)”, “}\cup\text{”, “}^*\text{”}\}$ generated by finitely many applications of the following rules:

- a is a regular expression for all $a \in \Sigma$;
- ϵ is a regular expression;[3]
- $\varnothing$ is a regular expression;
- If R_1 and R_2 are regular expressions, then $(R_1 \cup R_2)$ is also a regular expression;
- If R_1 and R_2 are regular expressions, then (R_1R_2) is also a regular expression;
- If R is a regular expressions, then (R_1^*) is also a regular expression.

The precedence of the operators are third over second over first, which together with the associativity of unions and concatenation allows the omission of certain pairs of parentheses. It is often useful to have shorthands “Σ” $\coloneqq (\sigma_1 \cup \cdots \cup \sigma_n)$ where $\Sigma = \{\sigma_i\}_{i=1}^n$, $R^k \coloneqq \underbrace{R \cdots R}_{k \text{ copies}}$, $R^+ \coloneqq (RR^*)$, and $R? \coloneqq (R \cup \epsilon)$.

A regular expression represents a language, which we define as follows.

Definition 1.22. Each a regular expression R over Σ defines a language $L(R) \subseteq \Sigma$ by the applying following rules in the manner the regular expression is constructed:

- If $R = a$ for some $a \in \Sigma$, then $L(R) = \{a\}$;
- If $R = \epsilon$, then $L(R) = \{\epsilon\}$;
- If $R = \varnothing$, then $L(R) = \varnothing$;
- If $R = R_1 \cup R_2$ where R_1 and R_2 are regular expressions, then $L(R) = L(R_1) \cup L(R_2)$;
- If $R = R_1R_2$ where R_1 and R_2 are regular expressions, then $L(R) = L(R_1) \circ L(R_2)$;
- If $R = R_1^*$ where R_1 is a regular expression, then $L(R) = L(R_1)^*$.

Here are some examples that show just how versatile they are:

- $L(0^*10^*) = \{w \mid w \text{ contains exactly one } 1\}$;
- $L(\Sigma^*1\Sigma^*) = \{w \mid w \text{ contains at least one } 1\}$;
- $L((\Sigma^2)^* \cup (\Sigma^3)^*) = \{w \mid 2 \mid \text{the length of } w \text{ is divisible by 2 or 3}\}$;
- $L(0^*(10^*1)^*) = \{w \mid w \text{ contains an even number of 1's}\}$;
- $L(1^*\varnothing) = \varnothing$;
- $L(\varnothing^*) = \{\epsilon\}$.

[3] This is redundant since ϵ is equivalent to $\varnothing^*$.

It's quite obvious that we can build an NFA for every regular expression just by piecing together the definition. Clearly, every character as a language can be recognized by an NFA, and so is the empty string and the empty language. The closure of regular expressions mean that the language of any regular expressions can be recognized by some NFA.

Conversely, can we build a regular expression equivalent to any given NFA? The answer is yes, which we justify with a new piece of machinery called generalized non-deterministic finite automata (GNFA).

Whereas in an NFA an arrow must be with a single input character from Σ_e, a GNFA has a regular expression over Σ attached to each arrow, allowing the machine to read chunks of the input string at each state as needed. But we want to avoid setting δ's domain to $Q \times \mathcal{R}[\Sigma]$ because a GN**F**A is **finite**, while there are infinitely many regular expressions in $\mathcal{R}[\Sigma]$. Instead, we make the following restrictions, so that now $\delta\colon (Q\backslash\{q_{\text{accept}}\}) \times (Q\backslash\{q_{\text{start}}\}) \to \mathcal{R}[\Sigma]$ can still be described in a finite manner.

- A GNFA has (exactly) one start state that has an arrow to every other state and has no incoming arrows;
- A GNFA has exactly one accept state that has an arrow from every other state and has no outgoing arrows;
- A GNFA has arrows from every non-accept state to every non-start state, including loops.

For non-existent arrows, we assigning ends of the arrow with $\varnothing$ for δ. For arrows with the same ends, we take the union of the characters to form the expression associated with the new, unique arrow.

Definition 1.23. A generalized non-deterministic finite automaton G is a 5-tuple $(Q, \Sigma, \delta, q_{\text{start}}, q_{\text{accept}})$, where

- Q is a finite set of states;
- Σ is a finite, non-empty set of the alphabet;
- $\delta\colon (Q\backslash\{q_{\text{accept}}\}) \times (Q\backslash\{q_{\text{start}}\}) \to \mathcal{R}[\Sigma]$;
- $q_{\text{start}} \in Q$ is the start state;
- $q_{\text{accept}} \in Q$, distinct from q_{start}, is the accept state.

We define the language of a GNFA in a similar way as before.

Definition 1.24. Let $G = (Q, \Sigma, \delta, q_{\text{start}}, q_{\text{accept}})$ be a generalized non-deterministic finite automaton and $w \in \Sigma^*$ be an input string. G is said to accept w if **(i)** there exists some $k > 0$ with $w = w_1 \cdots w_k$ where $w_1, \cdots, w_k \in \Sigma^*$ and **(ii)** there exist states $q_0, \cdots, q_k \in Q$ such that

- $q_0 = q_{\text{start}}$;
- $w_i \in L(\delta(q_{i-1}, q_i))$ for all $i = 1, \cdots, k$;
- $q_k = q_{\text{accept}}$.

Otherwise, G is said to reject w.

In other words, we accept a string w if we can put it into k chunks (where a chunk can be ϵ) so that we can transition along the arrows one chunk at a time to an accept state.

We first demonstrate how to convert a DFA into a GNFA.

Proposition 1.25. Suppose $M = (Q, \Sigma, \delta, q_0, F)$ is a DFA. Then, there exists a GNFA $G = (Q', \Sigma, \delta', q_{\text{start}}, q_{\text{accept}})$ such that $L(G) = L(M)$.

Proof. First, we add q_{start} and q_{accept}. Let $Q' = Q \cup \{q_{\text{start}}, q_{\text{accept}}\}$. Let $\delta'\colon (\{q_{\text{start}}\} \cup Q) \times (Q \cup \{q_{\text{accept}}\}) \to \mathcal{R}[\Sigma]$ be defined as

- If $q_i = q_{\text{start}}$:
  - If $q_j = q_0$, return ϵ;
  - Otherwise, return $\varnothing$;
- If $q_i \in Q$:
  - If $q_j \in Q$, return $\bigcup\{a \in \Sigma \mid q_j = \delta(q_i, a)\}$;
  - If $q_j = q_{\text{accept}}$,
    - If $q_i \in F$, return ϵ;
    - If $q_i \in Q \backslash F$, return $\varnothing$.

The added q_{start} and q_{accept} clearly do not change anything. The union construction for multiple arrows is as intended. Therefore, $L(G) = L(M)$. □

Now, to convert a GNFA to a regular expression, we will rip out one state at a time until there's only the start and the accept states. At this point, there's exactly one arrow from the start to accept by definition, and that must be the regular expression equivalent to the GNFA.

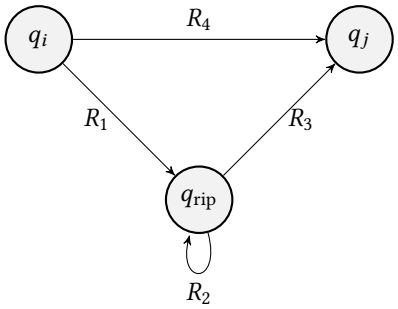


Figure 10: Two nodes q_i, q_j in a GNFA through q_{rip}.

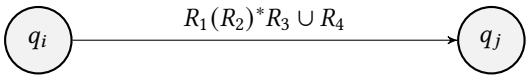


Figure 11: The transition from q_i to q_j after ripping out q_{rip}.

The basic idea is as follows. Any states $(q_i, q_j) \in (Q\backslash\{q_{\text{accept}}\}) \times (Q\backslash\{q_{\text{start}}\})$ will pass through q_{rip} (with possibly $\varnothing$ arrows), and we can create an equivalent regex to go from q_i to q_j without q_{rip}. After every pair of states is accounted for, we can safely remove q_{rip}. We'll repeat this procedure until we have only two states left, the start and the accept states.

Lemma 1.26. Suppose $G = (Q, \Sigma, \delta, q_{\text{start}}, q_{\text{accept}})$ is a GNFA with $|Q| > 2$ states. Then, there exists another GNFA $G' = (Q', \Sigma, \delta', q_{\text{start}}, q_{\text{accept}})$ with $|Q|' = |Q| - 1$ states such that $L(G') = L(G)$.

Proof. Let $k = |Q|$ be the number of states of G. Because $k > 2$, there must exist some state $q_{\text{rip}} \in Q\backslash\{q_{\text{start}}, q_{\text{accept}}\}$ which we will remove. Let $Q' \coloneqq Q\backslash\{q_{\text{rip}}\}$, and define for each $q_i \in Q\backslash\{q_{\text{accept}}\}$ and $q_j \in Q\backslash\{q_{\text{start}}\}$ the transition

$$\delta'(q_i, q_j) \coloneqq (R_1)(R_2)^*(R_3) \cup (R_4),$$

where $R_1 = \delta(q_i, q_{\text{rip}})$, $R_2 = \delta(q_{\text{rip}}, q_{\text{rip}})$, $R_3 = \delta(q_{\text{rip}}, q_j)$, and $R_4 = \delta(q_i, q_j)$ are regular expressions defined as in the Figure above.

Suppose G accepts $w \in \Sigma^*$ with a sequence of states $q_{\text{start}}, q_1, \cdots, q_k, q_{\text{accept}}$. If $q_{\text{rip}} \notin \{q_1, \cdots, q_k\}$, then clearly G' also accepts w with the same sequence of states. Otherwise, remove all occurrences of q_{rip} and consider all pairs of states (q_i, q_j) with $i < j$ surrounding the one or more removed q_{rip}'s. Because the new transition described all possible transitions from q_i to q_j with arbitrarily many intermediate repetitions of q_{rip}'s, G' will accept w with the purged sequence of states.

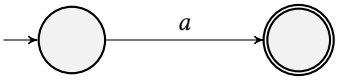
Now, suppose instead that G' accepts $w \in \Sigma^*$ with a sequence of states $q_0, \cdots, q_n$. Consider each pair of consecutive states (q_i, q_j) where $0 \leq i < n$ and $j = i + 1$. The substring w_i that allows this transition must also be matched G from q_i to q_j,

following arbitrarily many (and optional) repetitions of q_{rip}. Then, since each transition is possible, we conclude that G also accepts w. This concludes the proof of the equivalence of G and G'. □

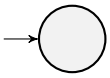
Now, we can piece together everything to show that regular expressions describe exactly the regular languages.

Proposition 1.27. The class of languages recognized by regular expressions is exactly the regular languages.

Proof. We first show that the language of every regular expression can be recognized by an NFA. The singleton language $A = \{\text{a}\}$ for $a \in \Sigma$ is recognized by the NFA



The empty language $B = \{\varnothing\}$ is recognized by the NFA



Because the regular languages are the smallest family of languages containing the above and closed under the regular operations, the regular expressions must be expressible by NFAs by the same closure properties.

We now show that every regular language is the language of some regular expression. Let M be a finite automaton recognizing a regular language, which has an equivalent GNFA G by Proposition 1.25. Applying Lemma 1.26 repeated $(n-2)$ times, where n is the number of states of G, we have an equivalent GNFA G' with exactly 2 states.

Then the range of the transition function δ' of G' contains exactly one element, which is $R \coloneqq \delta(q_{\text{start}}, q_{\text{accept}}) \in \mathcal{R}[\Sigma]$. We claim that $L(R) = L(G')$. Indeed, an input string $w \in \Sigma^*$ for G' has exactly one rewriting, so G' accepts w if and only if w is matched by the only arrow R to the accept state. Then, $L(R) = L(G') = L(G) = L(M)$, and the proof is complete. □

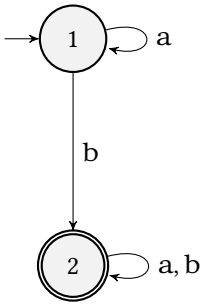


Figure 12: An example DFA.

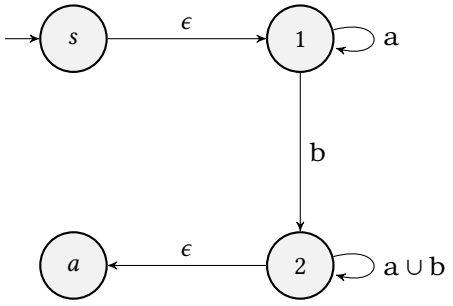


Figure 13: Conversion to a GNFA with 4 states.

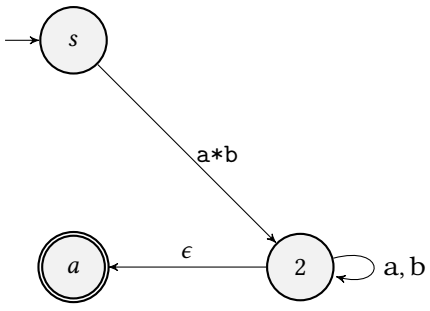


Figure 14: Removing state 1.

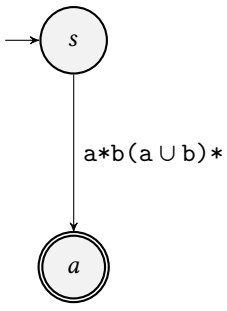


Figure 15: Removing state 2.

Here's an example of the actual reduction. We first added new start and accept states to isolate the self-loops. Then, we remove state 1 by considering $\epsilon\texttt{a*b}\cup\varnothing$. Subsequently, state 2 is also removed by considering $\texttt{a*b(a}\cup\texttt{b)*}\epsilon\cup\varnothing$. This gives us the final equivalent regex, which is $\texttt{a*b(a}\cup\texttt{b)*}$. This result will conclude our discussions on regular expressions.

1.4 Non-Regular Languages

While finite automata are powerful—our everyday computers and phone are finite automata, they are also quite limited theoretically. This is due to the finitude of their design: they cannot count, because counting requires separate states to distinguish the count so far but a count could be infinite.

Let's first take the language $A = \{0^n1^n \mid n \geq 0\}$. To design a DFA for A, it seems that we'll have to keep track of how many 0's we've seen so far. But there could be potentially infinitely many, and we cannot have that many states.

But this is not a formal argument. Just because one common way we try to find DFAs failed doesn't mean there's no way. Consider $B = \{w \in \{0,1\}^* \mid w \text{ has an equal number of 0's and 1's}\}$ and $C = \{w \in \{0,1\}^* \mid w \text{ contains an equal number of substrings 01 and 10}\}$. While B is not regular, C surprisingly is! A possible regular expression is

$$L(\epsilon|\texttt{(0+(1+0+)+|1+(0+1+)+)} = C.$$

What tool do we have, then, to show rigorously that a language is not regular? We have the following result known as the pumping lemma for regular languages, that describes the essence of the "counting" argument above.

Lemma 1.28 (Pumping Lemma for Regular Languages)**.** Suppose A is a regular language over Σ. Then, there exists a pumping length $p \in \mathbb{Z}_{>0}$ such that for all $w \in A$ with $|w| \geq p$, we have $x, y, z \in \Sigma^*$ such that

- $xy^iz \in A$ for all $i \geq 0$;
- $|y| > 0$;
- $|xy| \leq p$.

Of course, we will resort to the simplest DFAs here. The key is to exploit the fact that there can be more states/inputs than there are states. If the input is long enough with $|w| = n \geq p$, then in the sequence of $n+1 > p$ states, we have by the pigeonhole principle at least $\lceil (n+1)/p \rceil \geq 2$ states in the sequence repeated. Because a DFA doesn't remember anything about the past except the current state, we can repeat this loop of inputs indefinitely many times and end up in the same place.

Proof. Let $M = (Q, \Sigma, \delta, q_1, F)$ be a DFA recognizing A and define $p = |Q|$ to be the pumping length. Let $w = w_1 \cdots w_n$ be a string in A with $|w| = n \geq p$. Suppose the computation of w by M yields states $r_0, \cdots, r_n \in Q$. By the pigeonhole principle, at least $\lceil (n+1)/n \rceil = 2$ states will be repeated in the first n states. Denote the first two repeated states $r_s = r_t$ where $s < t$. Let $x = w_1 \cdots w_s$, $y = w_{s+1} \cdots w_t$, and $z = w_t \cdots w_n$.

We now show that these definition satisfy the stated requirements. Let $i \geq 0$ be arbitrary and consider the input xy^iz. After input x, M will transition to r_s. Because M goes from r_s to r_s upon inputs y, we conclude that M will stay at r_s after any i copies of y. Then, z will take M to r_n at which point M accepts xy^iz. Next, $|y| = t - (s+1) + 1 = t - s > 0$ because $s < t$ by construction. Lastly, $|xy| = t \leq p$ by construction as well. □

The pumping lemma is extremely useful when we want to show that a language is not regular. By the contrapositive, if we find an accept string that cannot be pumped, then the language cannot be regular.

Let's see this in practice with our previous example $A = \{0^n1^n \mid n \geq 0\}$.

Proof. Suppose for the sake of contradiction that $A \subseteq \Sigma^*$ is regular, where $\Sigma = \{0,1\}$. By the pumping lemma (Lemma 1.28), fix the pumping length $p > 0$. Let $w = 0^p1^p \in A$. Then, fix $x, y, z \in \Sigma^*$ according to the lemma. Because $|xy| \leq p$, we

conclude that x and y consist of only 0's. Then, $xy^0z = xz$ has fewer 0's than 1's, which cannot be recognized by A. This is a contradiction, which implies that A cannot be regular. □

Another interesting example is $D = \{1^{n^2} \mid n \geq 0\} \subseteq \{1\}^*$ over a alphabet consisting of only 1.

Proof. Suppose on the contrary that D is regular. Let $p > 0$ be the pumping length and consider the string $w = 1^{p^2}$. Fix $x, y, z \in \{1\}^*$ according to the pumping lemma. Since we have a unary language, it is easier to work with $a = |x|$, $b = |y|$, and $c = |z|$. By the pumping lemma, $xy^2z = 1^{p^2+b} \in D$. Let $p^2 + b = q^2$, where $q \geq 0$. Then,

$$p^2 < q^2 = p^2 + b \leq p^2 + p < p^2 + 2p + 1 = (p+1)^2.$$

Because q^2 is strictly contained between a pair of consecutive perfect squares p^2 and $(p+1)^2$, q^2 cannot be a perfect square, a contradiction. Therefore, D cannot be regular. □

The regular languages are quite nuanced; we can seem to have a problem that can be tackled by them but an essentially equal one that cannot. First, we'll consider string reversal in a different context.

Consider the alphabet

$$\Sigma_2 = \{0,1\}^2 = \left\{\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}0\\1\end{bmatrix}, \begin{bmatrix}1\\0\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}\right\},$$

a construction that allows us to read two strings (of the same length) simultaneously by reading a pair of characters each time. We claim that the language E over Σ defined by

$$E = \{w \in \Sigma_2^* \mid \text{the bottom row of } w \text{ is the reversal of the top row}\}$$

is not regular, even though regular languages are closed under reversal.

Proof. Suppose instead that E is regular, whereby we fix the pumping length $p > 0$. Let $w = \begin{bmatrix}0\\1\end{bmatrix}^p \begin{bmatrix}1\\0\end{bmatrix}^p$ which E recognizes by construction. Fix $x, y, z \in \Sigma_2^*$ according to the pumping lemma. Because $|xy| \leq p$, we conclude that x and y both consist of $\begin{bmatrix}0\\1\end{bmatrix}$ only. Then, $xy^2z \in E$ cannot be regular, which has more 0's than 1's in the top row but more 1's than 0's in the bottom row. This is a contradiction, which shows that E cannot be regular. □

Another interesting example is addition. The language

$$\{w \in \Sigma_3^* \mid \text{the sum of the two top rows of } w \text{ equals the third row when viewed as binary numbers}\}$$

over $\Sigma_3^* = \{0,1\}^3$ is regular, but the language

$$\{\text{“}z = x + y\text{”} \mid x, y, z \in \{0,1\}^* \text{ and } z = x + y\}$$

over $\{0, 1, =, +\}$ is not regular.

Our final result in this chapter is the Myhill-Nerode theorem, an "if and only if" condition about when a language is regular. We define an equivalence relation $\sim_L$ for strings indistinguishable by a language.

Definition 1.29. Let $L \subseteq \Sigma^*$ be a language. Two strings $x, y \in L$ are said to be indistinguishable by L, denoted as $x \sim_L y$, if $xz \in L \iff yz \in L$ for all suffixes $z \in \Sigma^*$. This defines an equivalence relation $\sim_L$ on Σ^*, whose collection of equivalence classes we denote with K. The index of L is then defined as $|K|$, the number of equivalence classes of $\sim_L$, which may be finite or infinite. Note that the equivalence class of an element is denoted as $[\cdot] \colon \Sigma^* \to \mathcal{P}(\Sigma^*)$.

Immediately, if one string is in L and the other isn't, then they are distinguishable by L considering the trivial extension $z = \sigma$. The converse isn't true: if $L = \{0, 1, 01\}$ with $x = 0$ and $y = 1$, then even though both $x, y \in L$, we have $xz = 01 \in L$ but $yz = 11 \notin L$. A language usually distinguishes strings better than simply recognition by the language.

There are some notions related to minimal DFAs which we detail.

Definition 1.30. Suppose M is a DFA over Σ. For strings $x, y \in \Sigma^*$, define $x \sim_M y$ if the final state of M coincides whether the input string is x or y. This defines an equivalence relation on Σ^*.

This other equivalence relation looks at the state of the DFA after an input. In general, $\sim_M \neq \sim_L$, for example, when M has two separate trap states where one is redundant. However, we will show below that this is the case when a DFA is minimal:

Definition 1.31. A DFA M is said to be minimal if every DFA equivalent to M has at least as many states as M.

Clearly, there is a minimal DFA for every regular language.

The following theorem says that a language is regular if and only if its index is finite. The big idea is that the equivalence classes contain correspond to the actually "needed" states. [TK]

Theorem 1.32 (Myhill-Nerode)**.** *Suppose $L \subseteq \Sigma^*$ is a language. Then,*

- *L is regular if and only if the index of L is finite;*
- *If L is regular, then every minimal DFA recognizing L is isomorphic[4] to the DFA $\tilde{M} = (K, \Sigma, \tilde{\delta}, [\epsilon], \{[x] \mid x \in L\})$ where*

$$\tilde{\delta}([x], a) \coloneqq [xa].$$

Before we start, it is noteworthy that the definition of δ above is consistent. Suppose $x, y \in \Sigma^*$ are indistinguishable. Then, for any suffix $z \in \Sigma^*$, $xaz \in L \iff yaz \in L$ because $az \in \Sigma^*$ is also a suffix. Then, $[xa] = [ya]$.

Proof. Suppose L is regular. Fix a *minimal* DFA $M = (Q, \Sigma, \delta, q_0, F)$ recognizing L. If $x \sim_M y$, then M goes to the same final state upon inputs x or y. By definition, any subsequent inputs will lead to the same state, and hence accept or reject both simultaneously. That is, $xz \in L \iff yz \in L$ for any subsequent inputs $z \in \Sigma^*$, so $x \sim_L y$. Any set of strings that is pairwise $\sim_L$ different must then be pairwise $\sim_M$-different. Then, $\sim_L$ can have no more equivalence classes than $\sim_M$, which equals $|Q|$.[5] Thus, the index of L is at most the number of states of a minimal DFA, which is finite.

Conversely, suppose the index of L is finite. Then, $\tilde{M}$ is a DFA, which we will show is equivalent to M. Let $w = w_1 \cdots w_n \in \Sigma^*$ be an input string ($n = |w|$) with final state $r_n \in K$ from $\tilde{M}$.

We claim that $r_n = [w]$ by proving $r_i = [w_1 \cdots w_i]$ for all $i \in \{0, \cdots, n\}$ with induction. For the base case, $r_0 = [\epsilon]$ by construction. For the inductive case with $i \in \{1, \cdots, n\}$ assuming $r_{i-1} = [w_1 \cdots w_{i-1}]$, we deduce $r_i = [(w_1 \cdots w_{i-1})(w_i)] = [w_1 \cdots w_i]$. This argument justifies the claim.

Now, if $\tilde{M}$ accepts w, then $[w] = r_n = [w']$ for some $w' \in L$. Then, $w \sim_L w'$. With the empty suffix in particular, $w \in L \iff w' \in L$. Because $w' \in L$, $w \in L$ also. Conversely, if $w \in L$, then $r_n = [w]$ already, so $\tilde{M}$ accepts w. This concludes the proof of the first item.

Now suppose L is known to be regular. We have already shown that $x \sim_M y$ implies $x \sim_L y$, so the equivalence classes of $\sim_M$ are contained among the equivalence classes of $\sim_L$. Observe that $|Q| \leq |K|$ because M is minimal, and $|K| \leq |Q|$ from the first item. Then, $|Q| = |K|$; that is, $\tilde{M}$ is a minimal DFA. This further implies that $\sim_M$ and $\sim_L$ have the same equivalence classes and hence are equal. Note that each equivalence class of $\sim_M$ corresponds uniquely to a state in Q. Because $\sim_M = \sim_L$, we naturally associate to each state of M an equivalence class of $\sim_L$. This is a bijection from the states of M to the states of $\tilde{M}$. Suppose M has transitioned to state $q \in Q$ with input $w_1 \cdots w_{i-1}$ so far ($w_1, \cdots \in \Sigma$). The state of $\tilde{M}$ associated with q is the equivalence class of the input so far, namely $[w_1 \cdots w_{i-1}]$. Now, $[w_1 \cdots w_i] = \tilde{\delta}([w_1 \cdots w_{i-1}], w_i)$, which is the state of $\tilde{M}$ associated with $\delta(q, w_i)$. By construction, the initial state $[\epsilon]$ of $\tilde{M}$ corresponds to q_0, the state of M upon no input. The accept states of $\tilde{M}$ are precisely the final states from computing the string accepted by L. The proof is now complete. □

[4] That is, equal up to renaming states.

[5] More specifically, the number of equivalence classes of M is the number of different final states that any string can reach. In a *minimal* DFA, all states must be reachable: otherwise, the unreachable states could have been removed to create a smaller equivalent DFA, a contradiction. Then, this number must be equal to $|Q|$.

The Myhill–Nerode theorem is stronger than the pumping lemma. While there are non-regular languages that still satisfy the pumping lemma, this theorem is an "if and only if." This means we can construct infinitely many distinguishable strings to show that a language can't be regular. At the same time, we can only prove how many states a DFA must have in order to recognize a regular language. Let's see these applications in practice.

We will show that any DFA recognizing the singleton language $L = \{1\}$ over $\Sigma = \{1\}$ must have at least 3 states. This is justified by considering the strings $\epsilon, 1, 11$, which we claim are pairwise distinguishable. There are three pairs to consider:

- ϵ and 1: Because $\epsilon \circ 1 \in L$ but $1 \circ 1 \notin L$ for the suffix 1, $\epsilon \not\sim_L 1$;
- ϵ and 11: Because $\epsilon \circ 1 \in L$ but $11 \circ 1 \notin L$ for the suffix 1, $\epsilon \not\sim_L 11$;
- 1 and 11: Because $1 \circ \epsilon \in L$ but $11 \circ \epsilon \notin L$ for the suffix ϵ, $1 \not\sim_L 11$.

This means that $\sim_L$ has at least 3 equivalence classes. Since the number of equivalence classes of $\sim_L$ equals the number of states in the minimal DFA, we have shown that any DFA recognizing L must have at least 3 states. Note that we could restrict $\Sigma = \{1\}$, and the argument remains valid.

Let's also try to use this theorem for showing that $L = \{0^n 1^n \mid n \geq 0\}$ over $\Sigma = \{0, 1\}$ is not regular. To this end, we will construct infinitely many strings that are pairwise different. We claim that the strings $\{0, 00, 000, \cdots\}$ are pairwise distinguishable. Let 0^n and 0^m be arbitrary strings from that set, where we assume $0 < n < m$. Then, $0^n \not\sim_L 0^m$ because $0^n 1^n \in L$ but $0^m 1^n \notin L$ considering the suffix 1^n. Then, each of the infinitely many strings will be in a separate equivalence classes, so there must be infinitely many such equivalence classes. As a result, L cannot be regular.

2 Context-Free Languages

2.1 Context-Free Grammars

We've seen the theoretical foundation of regular expressions, which turn out to be equivalent to DFAs and NFAs. However, as we've also seen, certain languages as simple and ubiquitous as $L = \{0^n 1^n \mid n \geq 0\} \subseteq \{0, 1\}^*$ are not regular. What would be a reasonable extension to regular languages? We answer this question with the introduction of context-free languages (CFGs), a radically different formalism that defines a language.

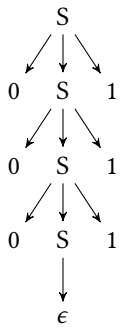
Let's first see an example that describes the L from above:

$$S \to 0S1$$
$$S \to \epsilon.$$

What do these two lines of equations do? They're called substitution rules, with which we can *derive* a string from a *start symbol* S. For instance, to get to 000111, we repeated apply the rules to replace any occurrence of S with the right hand side:

$$\begin{aligned} S &\Rightarrow 0\mathrm{S}1 && \text{(Rule 1)} \\ &\Rightarrow 00S11 && \text{(Rule 1)} \\ &\Rightarrow 000S111 && \text{(Rule 1)} \\ &\Rightarrow 000111. && \text{(Rule 2)} \end{aligned}$$

If you take a moment to think about all possible strings that can be generated in this way, it shouldn't be surprising that they form the language L. What we just showed is called the *derivation* of 000111, which can also be represented as a tree like this:



This neatly represents how the string is derived from the start. Now, we'll introduce the formal definition.

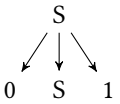
Definition 2.1. A context-free grammar is a 4-tuple (V, Σ, R, S), where

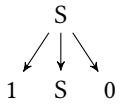
- V is a finite set of variables or nonterminals;
- Σ is a finite, non-empty set of the alphabet, called the terminals;
- $R \subseteq V \times (V \cup \Sigma)^*$;
- $S \in V$ is the start variable.

Formally, we have defined R as a collection of tuples, $(\cdot_1, \cdot_2)$, which we choose to write as $\cdot_1 \longrightarrow \cdot_2$. Now, how does a CFG define a language? We first define relations that describe how a string is generated.

Definition 2.2. Suppose $G = (V, \Sigma, R, S)$ is a context-free grammar. If $u, v, w \in (V \cup \Sigma)^*$ and "$A \longrightarrow w$" $\in R$, then we say that uAv yields uwv, denoted as $uAv \Rightarrow uwv$. The reflexive and transitive closure of $\Rightarrow$ is defined as the derivation relation $\Rightarrow^*$. The notation $u \Rightarrow^* v$ is read as "u derives v." Finally, the language of G is defined as $L(G) \coloneqq \{w \in \Sigma^* \mid S \Rightarrow^* w\}$. A language generated by a context-free grammar is said to be a context-free language.

We will refrain from formally definition a parse tree. Note that a parse tree is more than a graph-theoretic tree: it also encodes the ordering of the characters (from left to right) at each level. That is,



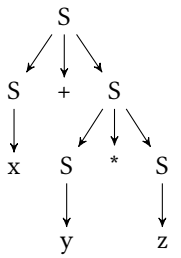


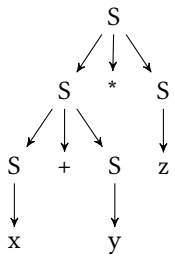
represent the same tree but not the same parse tree.

The reason why parse trees are important is because they show the structure of operations. This can be important in, e.g., parsing arithmetic expressions for a compiler, where we care about not just whether a string is a valid expression, but also what the expression is. Let's take a look at the following grammar:

$$S \longrightarrow S + S \mid S * S$$
$$S \longrightarrow x \mid y \mid z$$

Note that the bars are used to condense two or more rules with the same left hand side. Now, this grammar recognizes the string $x + y * z$, but it has two different parse trees:





This is a noteworthy issue because these two ways of parsing the string means the expression can be understood mathematically as either $x + (y * z)$ or $(x + y) * z$, which are clearly not the same. A compiler's CFG should be designed so that the first is always preferred over the second. This can be achieved by the following grammar:

$$S \to S + T \mid T$$
$$T \to T * U \mid U$$
$$U \to (S) \mid x \mid y \mid z.$$

These undesirable situations are so practically important that they deserve further investigation, which necessitates some definitions.

Definition 2.3. Suppose G is a context-free grammar. A string $w \in L(G)$ is said to be ambiguously derived in G if w admits two different parse trees in G. The grammar G is said to be ambiguous if such a string w exists. A context-free language L is said to be inherently ambiguous if every context-free grammar generating L is ambiguous.

To work with context-free grammars in general, it's usually convenient theoretically when we condense it to a specialized form, the Chomsky normal form.

Definition 2.4. A context-free grammar is said to be in Chomsky normal form if every rule is of the form

$$A \to BC \qquad \text{or} \qquad A \to a \qquad \text{or} \qquad S \to \epsilon,$$

where nonterminal $A \in V$, nonterminals $B, C \in V \backslash \{S\}$, and terminal $a \in \Sigma$.

It is not entirely obvious that every CFG can be put into Chomsky normal form. We will show that this is the case by constructing an equivalent grammar in the normal form.

Theorem 2.5. *Any context-free language is generated by a context-free grammar in Chomsky normal form.*

Proof. Suppose $G = (V, \Sigma, R, S)$ is a context-free grammar recognizing a given context-free language.

First, we add a new start variable S_0 and the rule $S_0 \to S$ which makes sure the right hand side never contains the start variable.

Then, for the ϵ-rules, we remove any rule $A \to \epsilon$ where $A \in V \backslash \{S\}$. For each occurrence of A on the right hand side of a rule $R \to uAv$, we add another rule $R \to uv$. If $R \to A$ is a rule, then we will add $R \to \epsilon$ unless this is a previously removed rule. This process is repeated until no rules of the form $A \to \epsilon$ exist.

Subsequently, we tackle the unit rules. If $A \to B$ is a rule with nonterminals $A, B \in V$, then this rule is removed. Whenever a rule $B \to u$ exists, where $u \in (V \cup \Sigma)^*$, we add another rule $A \to u$, unless it has been removed previously. This is repeated until no unit rules exist.

Finally, for each rule $A \to u_1 \cdots u_k$, where $k \geq 3$ and $u_1, \cdots, u_k \in (V \cup \Sigma)^*$, we convert them to

$$A \to u_1 A_1, \quad A_1 \to u_2 A_2, \quad \cdots, \quad A_{k-2} \to u_{k-1} u_k.$$

If any u_i is a terminal, we replace it with a new nonterminal variable U_i and an additional rule $U_i \to u_i$. The resulting grammar is clearly in Chomsky normal form and is equivalent to G. □

Let's walk through a concrete example. Consider the following CFG G:

$$S \to ASA \mid aB$$
$$A \to B \mid S$$
$$B \to b \mid \epsilon.$$

We first add the new start symbol:

$$S_0 \to S$$
$$S \to ASA \mid aB$$
$$A \to B \mid S$$
$$B \to b \mid \epsilon.$$

We now tackle ϵ rule:

$$S_0 \to S$$
$$S \to ASA \mid aB \mid a$$
$$A \to B \mid S \mid \epsilon$$
$$B \to b.$$

We also remove $A \to \epsilon$:

$$S_0 \to S$$
$$S \to ASA \mid aB \mid a \mid SA \mid AS \mid S$$
$$A \to B \mid S$$
$$B \to b.$$

We now move to the next stage, removing the unit rule $S_0 \to S$:

Don't add this since this was removed!

$$S_0 \to ASA \mid aB \mid a \mid SA \mid AS \quad \cancel{\mid S}$$
$$S \to ASA \mid aB \mid a \mid SA \mid AS \mid S$$
$$A \to B \mid S$$
$$B \to b.$$

We also remove the unit rule $S \to S$. Note that nothing needs to be added since the new rules we should add already exist.

$$S_0 \to ASA \mid aB \mid a \mid SA \mid AS$$
$$S \to ASA \mid aB \mid a \mid SA \mid AS$$
$$A \to B \mid S$$
$$B \to b.$$

Now, we remove $A \to B$:

$$S_0 \to ASA \mid aB \mid a \mid SA \mid AS$$
$$S \to ASA \mid aB \mid a \mid SA \mid AS$$
$$A \to b \mid S$$
$$B \to b.$$

And we'll remove $A \to S$ as well.

$$
\begin{aligned}
S_0 &\to ASA \mid aB \mid a \mid SA \mid AS \\
S &\to ASA \mid aB \mid a \mid SA \mid AS \\
A &\to b \mid ASA \mid aB \mid a \mid SA \mid AS \\
B &\to b.
\end{aligned}
$$

Now, there are no more unit rules. We'll address the ASA by breaking it down.

$$
\begin{aligned}
S_0 &\to AA_1 \mid aB \mid a \mid SA \mid AS \\
S &\to AA_1 \mid aB \mid a \mid SA \mid AS \\
A &\to b \mid AA_1 \mid aB \mid a \mid SA \mid AS \\
A_1 &\to SA \\
B &\to b.
\end{aligned}
$$

Lastly, we create a special symbol for the single a, so that

$$
\begin{aligned}
S_0 &\to AA_1 \mid A_2B \mid a \mid SA \mid AS \\
S &\to AA_1 \mid A_2B \mid a \mid SA \mid AS \\
A &\to b \mid AA_1 \mid A_2B \mid a \mid SA \mid AS \\
A_1 &\to SA \\
A_2 &\to a \\
B &\to b.
\end{aligned}
$$

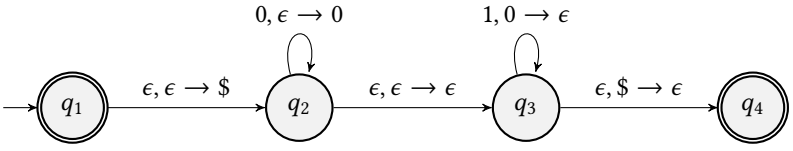
2.2 Pushdown Automata

Our brief investigation of regular languages reveals that this class of languages is captured precisely by finite automata. Then, for context-free languages, it is natural to ask whether any formal apparatus—a machine like DFAs/NFAs—is equivalent to context-free grammars. We answer this question with a new type of computational model called (non-deterministic) pushdown automata, or PDAs.

Pushdown automata look similar to NFAs with a notable distinction: PDAs are equipped with an additional structure, a stack, that can be used as "scratch paper" to store information. Our first example is the context-free language $L = \{0^n 1^n \mid n \geq 0\}$, recognized by the grammar

$$S \to 0S1 \mid \epsilon.$$

A possible PDA is given as follows.



The arrows of a PDA are labeled with three parts. Consider the arrow $\epsilon, \epsilon \to \$$ from q_1 to q_2. This means that upon no input (the first ϵ), whatever the stack is (the second ϵ), we are allowed to move to the next state along this arrow, popping nothing (the second ϵ) and pushing the symbol \$ to the stack. Similarly, the loop arrow $1, 0 \to \epsilon$ means that upon input 1 and when the top of the stack is a 0, we can move through this arrow, popping 0 and pushing nothing (ϵ) along the way.

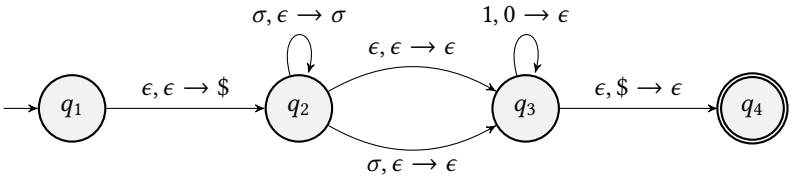
So for the input 0011, we first follow the ϵ arrow from q_1 to q_2, pushing a \$ to the stack. Then, the two 0's will be read, causing the PDA to loop to itself twice, pushing two 0's consecutively. Then, we follow the ϵ arrow to q_3, and then read the

two 1's to pop the two 0's. At this point, we have a \$ at the top of the stack, so we move to q_4 and accept the string. Note that q_1 need not be accepting, and the PDA would still accept the empty string.

As a prototypical example, we've seen that this language is not regular. Since every NFA can be easily converted to an equivalent PDA by changing arrows a to arrows $a, \epsilon \to \epsilon$, we know that PDAs are more powerful than NFAs. But PDAs are still finite! After all, the figure contains finitely many things, so why should we expect PDAs to be more powerful?

Crucially, the stack has unbounded size, which empowers it to recognize languages impossible for NFAs. At any point, the information that an NFA maintains is from which state(s) it is currently in, of which there is a finite number (namely, $2^{|Q|}$). In contrast, the configuration and behavior of PDA is influenced by not only its current state, but its current stack content. Therefore, while the specification of a PDA remains finite, its possible configurations are not.

Another example is a PDA that recognizes palindromes.



Here, σ can be any character in Σ. For example, when $\Sigma = \{a, b, c\}$, the loop $\sigma, \epsilon \to \sigma$ is actually three arrows:

$$a, \epsilon \to a,$$
$$b, \epsilon \to b,$$
$$c, \epsilon \to c.$$

When the input is, say, *abcba*, the first two characters will be pushed on the stack resulting in *ba*\$ (where left to right is top to bottom of the stack). The character c in the middle is read without modifying the stack, and the last two characters *ba* are read and popped from the stack simultaneously. Ending with a single \$ in the stack, we move to q_4 and accept the string.

We provide a formal definition of PDAs that eliminate possible confusion and ambiguity in the informal diagrammatic description above.

Definition 2.6. A pushdown automaton (PDA) is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where

- Q is a finite set of states;
- Σ is the finite alphabet over which the language of interest is defined;
- Γ is the finite alphabet of stack symbols;
- $\delta \colon Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to \mathcal{P}(Q \times \Gamma_\epsilon)$ is the transition function;
- $q_0 \in Q$ is the start state;
- $F \subseteq Q$ is the collection of accept states.

The computation of a PDA is slightly more technical than before, though it is quite understandable using the intuition from the diagrams.

Definition 2.7. A pushdown automaton $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ is said to accept a string $w \in \Sigma^*$ if $w = y_1 \cdots y_m$ for some $y_1, \cdots, y_m \in \Sigma_\epsilon$ such that there exist a sequence of states $r_0, \cdots, r_m \in Q$ and a sequence of stack content $s_0, \cdots, s_m \in \Gamma^*$ that satisfies the following: for all $i \in \{1, \cdots, m\}$, there exist $a, b \in \Gamma_\epsilon$ such that

- $r_0 = q_0$ and $s_0 = \epsilon$;

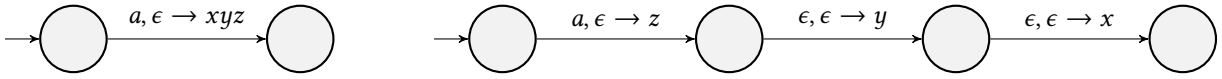
- $(r_i, b) \in \delta(r_{i-1}, y_i, a)$;
- $s_{i-1} = at$ and $s_i = bt$ for some $t \in \Gamma^*$;
- $r_m \in F$.

Otherwise, P is said to reject w.

To establish the formal equivalence between PDAs and context-free grammars, we need to show inclusion in both direction. We start with the easier construction converting a context-free grammar to a pushdown automaton.

Proposition 2.8. The language of any context-free grammar is recognized by some PDA.

Before we proceed with the proof, we introduce a notational shorthand that simplifies our argument. We extend the definition of arrows to allow pushing multiple symbols upon one input character. For instance, we'll allow arrows labeled $a, \epsilon \to xyz$, implemented by adding intermediate states.



Proof. The construction can be summarized as follows:

- Place $S\$$ on the stack;
- Consider all following cases:
  - If the top of the stack is a nonterminal variable, non-deterministically select one possible rule for A and replace the variable with the produced string according to the rule;
  - If the top of the stack is a terminal, then read and pop the terminal;
  - If the top of the stack is \$, then move to an accept state popping \$.

Formally, let $P = (Q, \Sigma, \Gamma, \delta, q_{\text{start}}, \{q_{\text{accept}}\})$, where $Q = \{q_{\text{start}}, q_{\text{loop}}, q_{\text{accept}}\}$ along with possible intermediate states for the shorthands used, Σ is the set of terminals from the given context-free grammar $G = (V, \Sigma, R, S)$, and $\Gamma = \Sigma \cup V \cup \{\$\}$.

We define the transition function δ as follows, where unspecified inputs are understood as mapped to $\varnothing$.

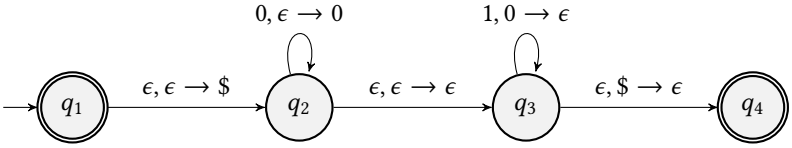
- Let $\delta(q_{\text{start}}, \epsilon, \epsilon) = \{(q_{\text{loop}}, S\$)\}$;
- For each nonterminal variable $A \in V$, let $\delta(q_{\text{loop}}, \epsilon, A) = \{(q_{\text{loop}}, w) \mid \text{“}A \to w\text{”} \in R\}$;
- For each terminal $a \in \Sigma$, let $\delta(q_{\text{loop}}, a, a) = \{(q_{\text{loop}}, \epsilon)\}$;
- Let $\delta(q_{\text{loop}}, \epsilon, \$) = \{(q_{\text{accept}}, \epsilon)\}$.

It is clear that the constructed pushdown automaton P recognizes $L(G)$. For every string w accepted by P written as $y_1, \cdots, y_m$ with states $r_0, \cdots, r_m \in Q$ (where we omit intermediate states from the shorthand used) and stack content $s_0, \cdots, s_m \in \Gamma^*$, there is a unique choice of successive yields that derive the string w. Similarly, to every derivation of a string w accepted by G is naturally associated a unique computation of P that leads to an accept state. □

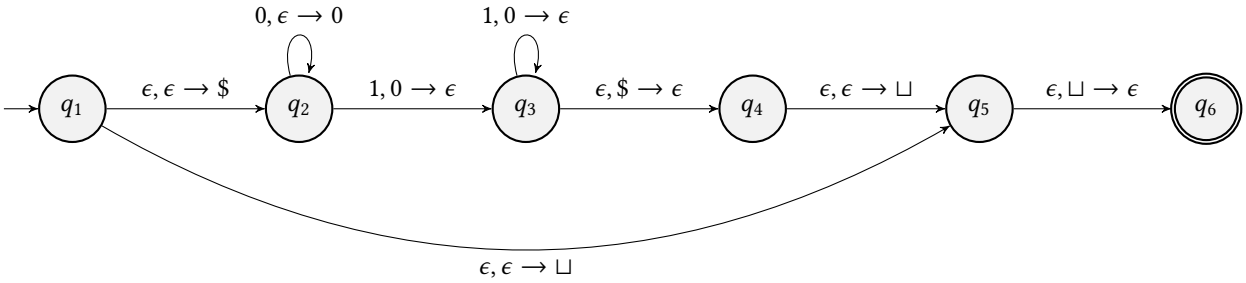
The other direction is more sophisticated. We first present the construction of a context-free grammar from a PDA before justifying its equivalence to the PDA.

Informally, we're looking for a grammar with variables A_{pq} for each pair of states $(p, q) \in Q \times Q$, whose associated rule derives all strings that can take P from p to q with an empty stack *before and after*. These strings would also take P from p to q even on a non-empty stack, leaving it unchanged, but the converse is not true: a string taking P from p to q in general could pop stuff from the stack before p, so it wouldn't always work if the stack were empty.

We'll assume that P has a single accept state q_{accept} and always empties its stack before accepting. The former can be achieved by adding $\epsilon, \epsilon \to \epsilon$ arrows to a new, unique accept state and the latter by adding a loop at the accept state dumping all stack content. Further, we stipulate that transitions would either push a symbol to or pop a symbol from the stack, but not neither or both. States can be added in an obvious manner to achieve this result. We present the conversion for the PDA recognizing $\{0^n 1^n \mid n \geq 0\}$. The PDA



is converted to



To design the production rule for such an A_{pq}, we need to take a look at how it works internally. Suppose P takes inputs $x \in \Sigma^*$ from p to q. Note that the first transition of P on x must push some symbol to the stack, since nothing can be popped from an empty stack. Similarly, the final transition of P on x must pop some symbol from the stack to finish with an empty stack.

There are two possibilities: either the first symbol pushed at state p is popped somewhere in the middle or it isn't popped until the very end.

- In the former case, the stack will be empty in the middle of the computation when the first symbol pushed is popped. We use the rule $A_{pq} \to A_{pr}A_{rq}$, where r is the state when the stack first becomes empty during the computation. Note that we account for the possibility where $p = q = r$; we could have $A_{qq} \to A_{qq}A_{qq}$;
- In the latter case, we use the rule $A_{pq} \to aA_{rs}b$, where $a, b \in \Sigma_\epsilon$ are the inputs read at the first and last transitions respectively, r is the next state of P following p, and s is the previous state of P immediately before q. Note that either a or b could be ϵ, in which case we can discard it from the right hand side of the production rule.

Finally, don't forget that we always have $A_{pp} \to \epsilon$. Formally, we have the following.

Definition 2.9. Let $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a PDA, where we assume without loss of generality that:

- Every transition in δ either pushes or pops a symbol from the stack, but not either or both; that is,

$$\forall p, q \in Q, \quad \forall a \in \Sigma_\epsilon, \quad \forall s, t \in \Gamma_\epsilon, \qquad (q, t) \in \delta(p, a, s) \quad \Longrightarrow \quad (s = \epsilon \iff t = \epsilon);$$

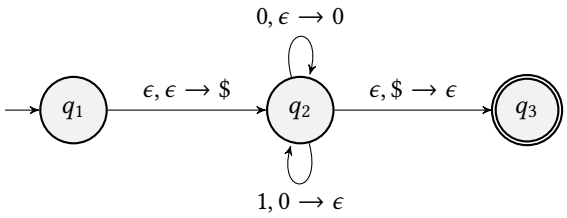
- The stack is empty whenever an input string is accepted; that is, given any string $w \in \Sigma^*$ accepted by P written as $w = y_1 \cdots y_m$, we have $s_m = \epsilon$;
- The accept state is unique, which we denote with q_{accept}; that is, $|F| = 1$ and $q_{\text{accept}} \in F$.

To each such P is associated a context-free grammar $G(P) := (V, \Sigma, R, S)$, where

- $V = \{A_{pq} \mid p, q \in Q\}$ is the set of nonterminal variables;

- R consists of the following rules:
  - For each $p, q, r, s \in Q$, $u \in \Gamma$, and $a, b \in \Sigma_\epsilon$, if $\delta(p, a, \epsilon)$ contains (r, u) and $\delta(s, b, u)$ contains (q, ϵ), include the rule $A_{pq} \to aA_{rs}b$;
  - For each $p, q, r \in Q$, include $A_{pq} \to A_{pr}A_{rq}$;
  - For each $p \in Q$, include $A_{pp} \to \epsilon$;
- $S = A_{q_0 q_{\text{accept}}}$.

Let's consider a PDA recognizing the set of balanced parentheses where the left parenthesis is denoted as 0 and the right parenthesis as 1.



We'll consider all pairs of states $p, q \in Q$, of which there are 9.

- $(p, q) = (q_1, q_1)$: $A_{q_1 q_1} \to \epsilon$;
- $(p, q) = (q_1, q_2)$: No associated rule;
- $(p, q) = (q_1, q_3)$: $A_{q_1 q_3} \to A_{q_2 q_2}$;
- $(p, q) = (q_2, q_1)$: No associated rule;
- $(p, q) = (q_2, q_2)$: $A_{q_2 q_2} \to 0A_{q_2 q_2}1 \mid A_{q_2 q_2}A_{q_2 q_2} \mid \epsilon$;
- $(p, q) = (q_2, q_3)$: No associated rule;
- $(p, q) = (q_3, q_1)$: No associated rule;
- $(p, q) = (q_3, q_2)$: No associated rule;
- $(p, q) = (q_3, q_3)$: $A_{p_3 p_3} \to \epsilon$.

The starting nonterminal variable is $A_{q_1 q_3}$, which can be replaced by $A_{q_2 q_2}$. In other words, this is the grammar $S \to 0S1 \mid SS \mid \epsilon$, which we know recognizes balanced parentheses.

To demonstrate the validity of this construction, we first validate the intended purpose of A_{pq}.

Proposition 2.10. Let $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a PDA and suppose $p, q \in Q$. Then, if A_{pq} derives $x \in \Sigma^*$ in the grammar $G(P)$, then x can bring P from p with an empty stack to q with an empty stack.

Proof. We perform induction on the number of derivation steps from A_{pq} to x.

Base case. Suppose the derivation has 1 step. By construction, the only possibility is the rule $A_{pp} \to \epsilon$ when $p = q$. It is obvious that ϵ trivially satisfies the desired property.

Inductive case. Suppose the derivation has $(k + 1)$ steps and assume by induction that the Proposition holds for all derivations from A_{pq} to x of at most k steps, where $k \geq 1$. There are two cases for the first step:

(i) The first step uses the rule $A_{pq} \to aA_{rs}b$ with the symbol $u \in \Gamma$, where $r, s \in Q$ and $a, b \in \Sigma_\epsilon$. Then $x = a\tilde{x}b$ for some $\tilde{x} \in \Sigma^*$. Since A_{rs} can be derived in at most k steps, $\tilde{x}$ will take P from r with an empty stack to s with an empty stack. Then, when the stack contains the sole symbol u, $a\tilde{x}b$ will first push u to the stack, derive $\tilde{x}$, and pop u finally. Therefore, x takes P from p with an empty stack to q with an empty stack.

(ii) The first step uses the rule $A_{pq} \to A_{pr}A_{rq}$ for some $r \in Q$. Suppose $x = wz$, where $w, z \in \Sigma^*$ are derived from A_{pr} and A_{rq} respectively. Since each of A_{pr} and A_{rq} can be derived in at most k steps, w will take P from p with an empty stack to r with an empty stack. Subsequently, z will take P from r with an empty stack to q with an empty stack. Therefore, $x = wz$ will take P from p with an empty stack to q with an empty stack.

Note that the case where $p = q$ and $A_{pp} \to \epsilon$ is impossible for the inductive case, since this requires exactly 1 step but $k + 1 \geq 2$. The proof is now complete. □

We also need to justify the other direction, using a similar induction argument. But first, we present a simple lemma, which states that all computations of the "restricted" PDA has an even number of transitions.

Lemma 2.11. Suppose P is a PDA of the special form in Definition 2.9 and $w \in L(P)$ is accepted by P with $k \geq 0$ transitions. Then, k is even.

Proof. Let $h_0, \cdots, h_k \in \mathbb{Z}_{\geq 0}$ be the stack heights between transitions. Because each transition either pushes or pops a stack symbol, we have either $h_{i-1} - 1 = h_i$ or $h_{i-1} + 1 = h_i$ for $i \in \{1, \cdots, k\}$. Therefore, each transition toggles the parity of h_i. Suppose on the contrary that k is odd. Then, because $h_0 = 0$, h_k is odd after odd toggles of parity. However, $h_k = 0$ is even, a contradiction. Hence, k is even. □

Proposition 2.12. Let $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a PDA and suppose $p, q \in Q$. Then, if $x \in \Sigma^*$ can bring P from p with an empty stack to q with an empty stack in the grammar $G(P)$, then A_{pq} can derive x.

Proof. We perform induction on the number of transitions in the computation of P from p to q with an empty stack before and after the input x.

Base case. Suppose the computation has 0 transitions from p to p, where $p \in Q$. Then, the string ϵ is accepted without any stack operations. Indeed, the construction includes the rule $A_{pp} \to \epsilon$.

Inductive case. Suppose now that the computation has $(k+1)$ transitions from p to q, where $p, q \in Q$ and $k \geq 0$. Assume by induction that the Proposition holds for all x whose associated computation has at most k transitions. Because $(k+1) \geq 1$ is even by Lemma 2.11, $(k + 1) \geq 2$. Let $u, v \in \Gamma$ be the first symbol pushed and the last symbol popped respectively. There are two possibilities:

(i) The stack is never empty except at the beginning and the end of the computation. That is, the symbol u at the bottom of the stack isn't popped until the last transition. We split x into awb, where $a, b \in \Sigma\epsilon$ are the inputs read at the first and last transitions and $w \in \Sigma^*$. Suppose during the computation that a takes P from p to $r \in Q$ in the beginning and b takes P from $s \in Q$ to q. Then, since the stack is never emptied during the transitions from r to s, the string w would also take P from r with an empty stack to s with an empty stack. Hence, by the induction hypothesis, A_{rs} can derive w. Indeed, the construction includes the rule $A_{pq} \to aA_{rs}b$, so A_{pq} can derive $awb = x$.

(ii) The stack is emptied during some intermediate transition. Let $r \in Q$ be the state immediately after the first transition that empties the stack, where $r \neq q$. Split x into wz, where $w \in \Sigma^*$ is read for the transitions from p to r and $z \in \Sigma^*$ from r to q. Then, w takes P from p with an empty stack to r with an empty stack, and z takes P from r with an empty stack to q with an empty stack. By the induction hypothesis, then, A_{pr} can derive w and A_{rq} can derive z. Indeed, the construction includes the rule $A_{pq} \to A_{pr}A_{rq}$, so A_{pq} can derive $wz = x$.

This concludes the proof by induction. □

Note that the lemma above is a bit stronger than necessary for the purpose of this proof. One can note that such PDAs must push at the first transition and pop at the last; the transitions must therefore be distinct.

Corollary 2.13. The language of any PDA is recognized by some context-free grammar.

Proof. Let P be a PDA. From Propositions 2.10 and 2.12, we conclude that the grammar $G(P)$ with the start symbol $A_{p_0 p_{\text{accept}}}$ accepts exactly the strings that take P from p_0 to p_{accept} from an empty stack to an empty stack, which by assumption covers all possible accepted strings. Therefore, $L(G(P)) = L(P)$. □

Corollary 2.14. Every regular language is context-free.

Proof. Observe that any DFA can be converted to an equivalent PDA with no use of the stack, and any PDA has a corresponding equivalent context-free grammar. Therefore, every regular language has an equivalent context-free grammar. □

2.3 Non-Context-Free Languages

As powerful as context-free grammars are, there are still languages that seem quite straightforward and simple but that cannot be recognized by any context-free grammars. Similar to regular languages, context-free languages also admit a pumping lemma that can be used to show certain languages cannot be context-free.

Lemma 2.15 (Pumping Lemma for Context-Free Languages)**.** Suppose A is a context-free language over Σ. Then, there exists some $p > 0$ such that for all $w \in L(A)$ with $|w| \geq p$, there exists $u, v, x, y, z \in \Sigma^*$ with $w = uvxyz$ such that

- For all $i \geq 0$, $uv^i xy^i z \in L(A)$;
- $|vy| > 0$;
- $|vxy| \leq p$.

Proof. Let b be the maximum number of nonterminal and terminal symbols on the right-hand side of any rule. If $b = 0$, then the language of the grammar is finite. The pumping lemma is trivially satisfied by choosing a p longer than any string recognized by G. We hereafter suppose $b \geq 2$. No node has more than b children in any parse tree of the grammar, so any parse tree of height h yields a string of length at most b^h. Conversely, any string of length at least $b^h + 1$ has parse tree of height at least $h + 1$.

Let $p = b^{|V|+1}$. For any $w \in \Sigma^*$ where $|w| \geq p$, any parse tree for w has height at least $|V| + 1$. We fix an arbitrary parse tree with the fewest number of nodes. The longest path from the root S to a terminal leaf is at least $|V| + 1$ long and contains at least $|V| + 1$ nonterminal variables, some of which must repeat. Let R be a variable repeated in the lowest $|V| + 1$ variables along the path. Let $w = uvxyz$ such that vxy is the substring derived from the upper occurrence of R in the repetition and x from the lower. Because both vxy and x are derived by R, we may replace the sub-parse tree for the lower occurrence for the upper occurrence, resulting in a valid parse tree for the grammar that yields uxz. Similarly, we may replace the upper occurrence for the lower, which yields uv^2xy^2z—and this process can be done inductively to yield any $uv^i xy^i z$ whenever $i \geq 2$. This concludes the first item.

To rule out the possibility $v = y = \epsilon$, note that this would imply the substitution of the subtree rooted at the lower R for the upper results yields the same string w with fewer nodes. This is impossible as we have chosen a parse tree with the fewest number of nodes.

Lastly, the upper subtree generating vxy has height at most $|V| + 1$. Because no node has more than b children, the string vxy has length no more than $b^{|V|+1} = p$. □

We still use the pumping lemma for context-free languages in similar to tackling regular languages. The proof becomes more technical as there are more potential cases to consider.

A prototypical example is $L = \{a^n b^n c^n \mid n \geq 0\}$, which is not context-free. Applying the lemma above, suppose the contrary and fix the pumping length $p > 0$. Consider $w = a^p b^p c^p \in L$ which has length $|w| = 3p \geq p$. Hence, $w = uvxyz$ where $uxz \in L$, $|vy| > 0$, and $|vxy| \leq p$. Now, consider a sliding window of length at most p, which will be the location of the substring vxy in the break-up. There are 5 possibilities:

- **The window contains only a's.** Because $|vy| > 0$, the string x has fewer a's than vxy, so the string $uxz \in L$ has fewer a's than b's and c's, which is impossible.
- **The window contains a's and b's.** Similarly, pumping down reduces the numbers of a's, b's, or both. The number of c's will be greater than that of either a's or b's, a contradiction.
- **The window contains only b's.** The same as case I.
- **The window contains b's and c's.** The same as case II.
- **The window contains only c's.** The same as case I.

3 The Church–Turing Thesis

3.1 Turing Machines

As we have last seen, languages as simple as $\{a^n b^n c^n \mid n \geq 0\}$ cannot be recognized by context-free grammars (much less regular languages)! What additions do we need exactly to recover the computational power of everyday computers with which we have become so familiar?

The construction of a Turing machine, introduced by Alan M. Turing, achieves precisely this. Similar to PDAs, a TM also has an additional storage device, but it is more powerful: rather than a stack which can only be modified at one fixed location and which the PDA can only interact with upon each input character, a TM has an unbounded *tape* with a *head* that it can freely move around. But differently, the moves are decoupled from input characters. Instead, the tape indexed by $\mathbb{Z}_{>0}$ from left to right starts with the input string flushed left and the head on the first input character. The TM *transitions* based on the current tape symbol and its current state to the next state, a symbol to overwrite the current one at the head position, and an instruction to move the head either left or right.

Definition 3.1. A Turing machine (TM) is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ where

- Q is a finite set of states;
- Σ is a finite set of alphabet which does not contain the reserved empty symbol $\sqcup$;
- Γ is a finite set of tape symbols such that $\{\sqcup\} \cup \Sigma \subseteq \Gamma$;
- $\delta \colon Q \times \Gamma \to Q \times \Gamma \to \{\mathrm{L}, \mathrm{R}\}$;
- $q_0, q_{\text{accept}}, q_{\text{reject}} \in Q$ are the start, accept, and reject states where $q_{\text{accept}} \neq q_{\text{reject}}$.

Now, we formalize the computation of TMs as well. Note first that at any intermediate step of the computation of a Turing machine, the tape contains at most finitely many non-empty tape symbols. This allows us to write the complete state of a Turing machine as follows.

Definition 3.2. A configuration of a Turing machine $T = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ is a string of the form $s_1 \cdots s_m q r_1 \cdots r_n$, where $s_1, \cdots, s_m, r_1, \cdots, r_{n-1} \in \Gamma$, $r_m \in \Gamma \backslash \{\sqcup\}$, and $q \in Q$. The configuration denotes the complete state of a Turing machine with tape content $(s_1, \cdots, s_m, r_1, \cdots, r_n, \sqcup, \sqcup, \cdots)$, the current state q, and the head at r_1 immediately following q in the string.

We now define a "yields" relation that describes each computational step of a TM.

Definition 3.3. Suppose $T = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ is a Turing machine.

- The configuration uaq_ibv is said to yield uq_jacv, where $u, v \in \Gamma^*$, $a, b, c \in \Gamma$, and $q_i, q_j \in Q$, if $\delta(q_i, b) = (q_j, c, \mathrm{L})$;
- The configuration uaq_ibv is said to yield $uacq_jv$, where $u, v \in \Gamma^*$, $a, b, c \in \Gamma$, and $q_i, q_j \in Q$, if $\delta(q_i, b) = (q_j, c, \mathrm{R})$;
- The configuration q_iav is said to yield q_jbv, where $v \in \Gamma^*$, $a, b \in \Gamma$, and $q_i, q_j \in Q$, if $\delta(q_i, a) = (q_j, b, \mathrm{L})$;
- The configuration q_iav is said to yield bq_jv, where $v \in \Gamma^*$, $a, b \in \Gamma$, and $q_i, q_j \in Q$, if $\delta(q_i, a) = (q_j, b, \mathrm{R})$.

We now state some common configurations.

Definition 3.4. Suppose $T = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ is a Turing machine. The start configuration of T on an input string $w \in \Sigma^*$ is the configuration q_0w. Any configuration with state q_{accept} is said to be an accepting configuration. Any configuration with state q_{reject} is said to be a rejecting configuration.

This provides adequate machinery for defining the computation of a Turing machine.

Definition 3.5. Suppose $T = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ is a Turing machine. T is said to accept $w \in \Sigma^*$ if there exists a sequence of configurations $C_1, \cdots, C_k$ where

- C_1 is the start configuration of T on w;
- For each $i = 1, \cdots, k-1$, C_i yields C_{i+1};
- C_k is an accepting configuration.

The collection of all string accepted by T is defined as the language recognized by T, denoted as $L(T)$.

Note that there are two ways a Turing machine fails to accept a string: either it enters into a rejecting configuration and halts, or it somehow "loops" forever. We quantify the latter behavior more specifically.

Definition 3.6. A Turing machine T is said to halt on on input $w \in \Sigma^*$ if there exists a sequence of configurations $C_1, \cdots, C_k$ where

- C_1 is the start configuration of T on w;
- For each $i = 1, \cdots, k-1$, C_i yields C_{i+1};
- C_k is an accepting or rejecting configuration;

in other words, T *decides* whether $w \in L(T)$ in a finite number of steps.

Definition 3.7. A Turing machine T is said to be a decider it halts on all possible inputs $w \in \Sigma^*$.

Definition 3.8. A language is said to be recognizable, or recursively enumerable, if a Turing machine recognizes it. A language is said to be decidable, or recursive, if a Turing machine decides it.

It turns out no conceivable and reasonable model of computation is more powerful than Turing machines; that is, Turing machines are the most powerful models of computers theoretically possible. The Theory of Computation, then, concerns the fundamental possibilities and limitations of such theoretical computers. The question, "Are there any problems computers (TMs) can't solve?" gave rise to the field of computability, and the question "How long does a computer (TM) need to solve a problem?" birthed complexity theory.

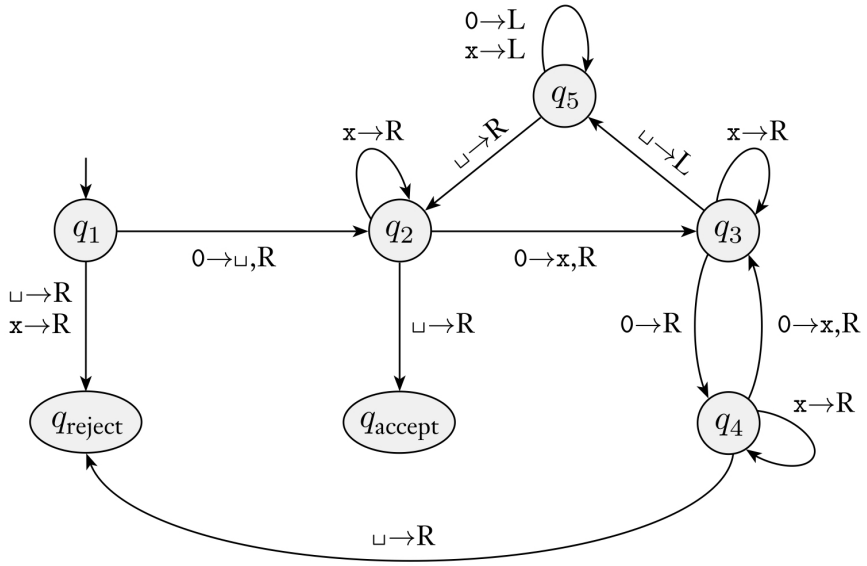
Here, we give an explicit example of a Turing machine. Consider the language $A = \{0^{2^n} \mid n \geq 0\}$ over the singleton alphabet $\{0\}$. We construct a TM T that works as follows. For an input string w,

1. Replace the current (first) symbol with $\dot{0}$;
2. Cross off every other 0 symbol by replacing $\dot{0}$ with $\not{\emptyset}$ and 0 with $\not{\emptyset}$. If there is only exactly one 0, accept; if after sweeping through the tape we find an odd number of 0's, reject;

3. Return to the dotted symbol, which is the beginning of the tape;

4. Go to step 2 and repeat.

A formal description would be extremely verbose, reproduced as follows from Figure 3.8 of [1].



We provide another Turing machine does some arithmetic by deciding the language $\{a^i b^j c^k \mid a, b, c \geq 1 \text{ and } ab = c\}$. For an input string w,

1. Mark the first character with a dot; that is, replacing a with $\dot{a}$, b with $\dot{b}$, and c with $\dot{c}$;

2. Scan through w and check if w matches $a^+b^+c^+$;

3. Return to the leftmost cell of the tape as marked earlier;

4. Cross off an a (whether or not marked) by replacing a with $\not{a}$ and $\dot{a}$ with $\not{\dot{a}}$ and move right until the first b occurs;

5. Cross off the current b, and move right to the first c not yet crossed off;

6. Cross off the current c, and move left to the first b not yet crossed off;

7. Go to stage **5** if no b's remain. Then, restore all b's crossed off. Move to leftmost a's not marked off and go to stage 4. If no such a's exist, then accept if no c's remain and reject otherwise. Reject also if any operation at any stage above is impossible to perform.

3.2 Variants of Turing Machines

Some parts of the definition of a Turing machine seems arbitrary. Can a TM choose not to move Left or Right but just stay put? Why is one end unbounded but the other bounded in the tape? Why can't we have more tapes, finitely many, or (countably) infinitely many? What about non-determinism? It turns out that all these questions can be answered affirmatively: any such addition does not allow Turing machines to recognize more languages. This is why we call the definition of computability, or recognizability, *robust*: reasonable changes or extensions to some parts of a definition do not modify the scope of the definition.

A Turing machine that stays put at a certain step can be simulated by adding an intermediate state to simulate moving left and then moving right. A Turing machine with an unbounded tape can be done by considering a particular "computable" bijection from $\mathbb{Z}_{>0}$ to $\mathbb{Z}$ such as $\phi\colon n \mapsto (-1)^n \lfloor n/2 \rfloor$. Such a function ϕ can easily be computed with a Turing machine, so

we can always convert a two-way-unbounded Turing machine with an ordinary one by thinking about which cell to use with the bijection ϕ.

We investigate the idea of multi-tape Turing machines a little more closely. A multi-tape Turing machine has all the components of an ordinary Turing machine: Q, Σ, Γ, δ, q_0, q_{accept}, and q_{reject}. The core distinction is that given $k \geq 1$ tape(s), we now have $\delta\colon Q \times \Sigma^k \to Q \times \Sigma^k \times \{L, R, S\}^k$. Specifically,

- The input to the machine is copied onto the first tape flushed left;
- Each tape is attached to a head, all of which work independently.
- The machine may transition between states by reading the current characters on all tapes, replacing them respectively on each tape, and moving to the Left or the Right, or Staying put on each tape separately.

Are they anyhow more powerful than ordinary Turing machines? It turns out not.

Proposition 3.9. Suppose $k \in \mathbb{Z}_{>0}$. Then, every k-tape Turing machine has an equivalent ordinary Turing machine.

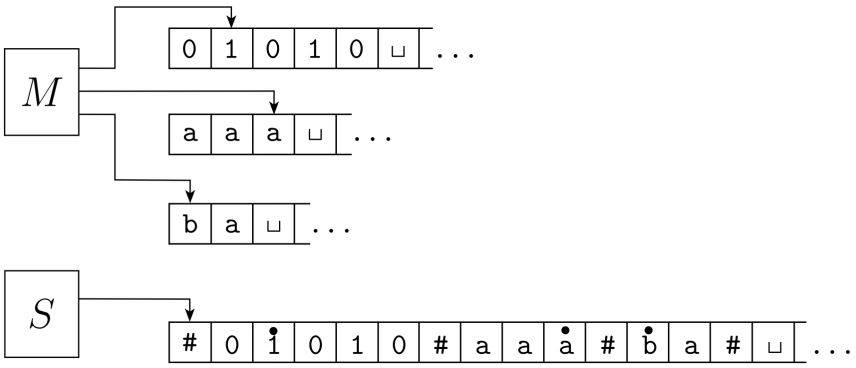


Figure 3.14 from [1].

The proof idea is as follows. Let K be an arbitrary k-tape Turing machine. We construct a Turing machine T whose input $w = w_1 \cdots w_n$, where $w_1, \cdots, w_n \in \Sigma$, is stored on the tape initially as

$$\text{“}\# \dot{w}_1 \cdots w_n \# \dot{\sqcup} \# \cdots \# \sqcup \#\text{''},$$

where there are k copies of $\sqcup$'s separated by #.

The dotted positions in T will represent the current head positions of K. For each transition of K, we first scan through all tape content stored on T and copy them after the final #. Then, transition to the corresponding state according to the scratch cells for the tape symbols at the current heads of K. Finally, go over the tape again and modify the current tape symbols before moving the dotted positions.

If at any point T moves right to a #, then we pause further operations and first move all subsequent content, up to and including the current #, to the right by one cell, which allocates an empty space dynamically for T at the current tape. The pending operations now proceed as normal.

This shows that multi-tape Turing machines are no more powerful than ordinary Turing machines.

Corollary 3.10. Multi-tape Turing machines recognize the same class of languages as Turing machines.

Similar to the distinction between DFAs and NFAs, one can construct non-deterministic Turing machines by insisting

$$\delta\colon Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \to \{\mathrm{L}, \mathrm{R}\}),$$

where the machine accepts an input if any branch accepts and rejects otherwise.

Interestingly, while deterministic PDAs demonstrably recognize "fewer" languages than PDAs, non-determinism does not affect the class of languages that Turing machines recognize. Clearly, every Turing machine is trivially a non-deterministic Turing machine; it suffices to show the other direction of inclusion.

Theorem 3.11. *Every non-deterministic Turing machine has an equivalent Turing machine.*

Proof. We will create a 3-tape Turing machine to perform BFS on the tree of all possible branches. Let $b \coloneqq \max\{|\delta(q,a)| : q \in Q \text{ and } a \in \Gamma\}$ be the largest number of choices for transitioning at any given step. Then, every particular node of the computation tree can be assigned a unique "address" as a string over $\{1, \cdots, b\}$. Note that the natural shortlex ordering gives a BFS of the tree, though it may cover some extraneous, invalid "addresses."

The input tape will be kept read-only. A second tape will keep track of simulating a TM for the computation from the root q_0 up to current node, and a third tape for keeping track of the current "address" in BFS. Clearly, such a construction yields a 3-tape Turing machine equivalent to the given NTM. Because 3-tape TMs are equivalent to TMs, there exists a TM equivalent to the given NTM. □

From this point on, we shall content ourselves with describing TMs as though we are writing Python programs. It should be intuitive that any primitive operations like conditional, loops, and recursive as well as data structures such as lists, graphs, and maps can be simulated with TMs. For these purposes, we will often fix a particular encoding of mathematical objects as strings. The encoding of an object O shall be denoted as $\langle O \rangle$. The specific choice of encoding is of no particular interest to us.

4 Decidability

In this section, we study the fundamental question of what problems we can *solve*. Along the way, we will develop some powerful machinery that allows us to assert *what we cannot ever solve*.

Proposition 4.1. $A_{\text{DFA}} = \{\langle D, w\rangle \mid D \text{ is a DFA that accepts } w\}$ is decidable.

Proof. Simulate the DFA with a TM for $|w| < \infty$ steps. If the final step is an accept state, then accept; otherwise, reject. □

To demonstrate a language is decidable, it suffices to construct a particular TM that decides it.

Proposition 4.2. $E_{\text{DFA}} = \{\langle D\rangle \mid D \text{ is a DFA where } L(D) = \varnothing\}$ is decidable.

Proof. Let $G = (Q, E)$ be the directed graph of states where $E = \{(p,q) \mid \delta(p,a) = q \text{ for some } a \in \Sigma\}$. Perform DFS on the reverse graph G^R multiple times starting at each accept state. If q_0 is ever reached, accept; otherwise, reject. □

Proposition 4.3. $EQ_{\text{DFA}} = \{\langle D, D'\rangle \mid D \text{ and } D' \text{ are DFAs such that } L(D) = L(D')\}$ is decidable.

Proof. Recall that regular languages are closed under unions, intersections, and complements. Hence, the symmetric difference

$$(L(D) \backslash L(D')) \cup (L(D') \backslash L(D))$$

is recognized by some DFA $\tilde{D}$. Suppose T decides E_{DFA}. If T accepts $\langle \tilde{D}\rangle$, then accept; otherwise, reject. □

Clearly, the equivalence of DFAs and NFAs suggests we may replace "DFA" with "NFA" in the above statements with impunity. We now pose the same questions to context-free grammars. Whereas DFAs are simple enough that most of the natural questions one poses about them are decidable, this is unfortunately not the case for more powerful models of computation in general.

Lemma 4.4. Let G be a context-free grammar in Chomsky normal form and $w \in L(G) \backslash \{\epsilon\}$. Then, every derivation of w from G requires at most $2|w| - 1$ yields.

The proof is omitted due to relevance, though one may easily provide an induction argument.

Corollary 4.5. $A_{\text{CFG}} = \{\langle G, w\rangle \mid G \text{ is a context-free grammar that derives } w\}$ is decidable.

Proof. If $w = \epsilon$, check if $S \to \epsilon$ is a rule. If so, accept; otherwise, reject. If $w \neq \epsilon$, enumerate all possible derivations of length $2|w| - 1$ and see if w is ever derived. If so, accept; otherwise, reject. □

Corollary 4.6. Every context-free language is decidable.

Proposition 4.7. $E_{\text{CFG}} = \{\langle G\rangle \mid G \text{ is a context-free grammar such that } L(G) = \varnothing\}$ is decidable.

This construction is arguably much less obvious. Similar to constructing a decider for E_{DFA}, one needs to carefully examine how a string is derived from a grammar and perform a similar traversal.

Proof. We will recursively mark all symbols (both non-terminal variables and terminals) that derives terminals as follows. First, mark all terminals. Then, repeat the following until no new symbols are marked: whenever there is a rule $A \to U_1 \cdots U_k$ where each $U_1, \cdots, U_k \in V \cup \Sigma$ has been marked, we mark A also; if $S \to \epsilon$ is a rule, then we mark S and reject immediately.

After the loop terminates, if the start variable is marked, then reject; otherwise, accept. □

Surprisingly, EQ_{CFG} is *not* decidable, though we lack the machinery for this proof so far. We will therefore proceed to examine the same languages for TMs. Here, we encounter some first examples for undecidable languages and develop a powerful machinery—reduction—for doing so.

Realistically, what does it mean that a language is undecidable? For such a problem, we may assert that any conceivable algorithm is bound to fail, either by giving an incorrect answer or entering in a dead loop (and thus never answering) for some input. In other words, computers can't possibly solve such a problem.

That there exist unsolvable problems can be seen without even diving into the specifics of Turing machines.

Theorem 4.8 (Cantor)**.** *Suppose S is a set. Then, $|\mathcal{P}(S)| > |S|$.*

Proof. It suffices to show no map $f : S \to \mathcal{P}(S)$ is surjective. Suppose on the contrary that f is onto, and consider the set $Y \coloneqq \{x \in S \mid x \notin f(x)\} \in f[S]$. Fix $x_0 \in S$ such that $f(x_0) = Y$. If $x_0 \in f(x_0) = Y$, then $x_0 \notin f(x_0)$ by construction of Y; if $x_0 \notin f(x_0)$, then by definition of Y, we have $x_0 \in Y = f(x_0)$. We obtain a contradiction in both cases, which implies that f cannot be surjective. Hence, the cardinality of S is strictly less than that of S. □

Corollary 4.9. There exist unrecognizable and undecidable languages.

Proof. The finitude of Q, Σ, and Γ implies we may fix a canonical representation of Q as $\{1, \cdots, |Q|\}$, Σ as $\{1, \cdots, |\Sigma|\}$, and Γ as $\{1, \cdots, |\Gamma|\}$. It is therefore obvious that there are only countably infinitely many Turing machines up to renaming states and symbols. That is, the cardinality of the family of languages (up to renaming alphabet symbols) recognized by Turing machines is $|\mathbb{N}|$.

However, there are $|\mathcal{P}(\Sigma^*)| > |\Sigma^*| = |\mathbb{N}|$ languages over a non-empty alphabet, up to renaming alphabet symbols. Therefore, there must exist languages undecidable, and hence unrecognizable, by Turing machines. □

Of course, we can use the same idea for particular problems. This is where the real fun happens. How does one show something *can't possibly be solved* by a computer? Let's see an example in action.

Proposition 4.10. $A_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a Turing machine that accepts } w\}$ is undecidable.

Proof. Suppose on the contrary that some Turing machine T decides A_{TM}. We define a new TM D, which upon input $\langle M\rangle$ return the negation of T on $\langle M, \langle M\rangle\rangle$. Because T is a decider, so is D. Consider running D on $\langle D\rangle$. If it accepts, then T must reject $\langle D, \langle D\rangle\rangle$, or $\langle D\rangle \notin L(D)$. Because D is a decider, this implies D rejects $\langle D\rangle$. If it rejects, then T must accept $\langle D, \langle D\rangle\rangle$, which implies D accepts $\langle D\rangle$. In either scenario, a contradiction is deduced. Therefore, A_{TM} cannot be decidable. □

One must acknowledge how neat this proof is. By feeding to D its own representation, we always end up with a contradiction. Is this doable, however, without raising any self-referential paradoxes? Yes! Once we defined D completely, we can regard D as a piece of code—a string, which we can rightfully pass to a function (such as D) as an input.

The theoretical significance of decidability is also seen in the following result, linking it to recognizability.

Proposition 4.11. A language L a decidable if and only if L is recognizable and $\overline{L}$ is recognizable.

Proof. Suppose L is decidable by T. Then, L is trivially recognizable. To recognize $\overline{L}$, one simply constructs a Turing machine that returns the negation of the T on the same input, which is a decider because L decides.

Conversely, suppose both L and $\overline{L}$ are recognizable by T and T' respectively. Construct a decider D for L as follows. Upon input w, run T and T' on w alternatively—one step on T, then one step on T', and repeat. If T accepts at any point, then accept; if T' accepts at any point, then reject. Because either $w \in L(T) = L$ or $w \in L(T') = \overline{L}$, one of the two recognizers will accept in a finite amount of time. Therefore, D will decide L. □

Corollary 4.12. $\overline{A_{\text{TM}}}$ is not recognizable.

Proof. Clearly, A_{TM} is recognizable by simply simulating the input Turing machine on the input string. Because A_{TM} is undecidable, it is necessary that $\overline{A}_{\text{TM}}$ is unrecognizable to avoid contradiction to the Proposition above. □

5 Reducibility

In this section, we'll capitalize on the undecidable problems we have already shown to prove more problems are undecidable or unrecognizable. Our main tool will be *reducibility*. Informally, a language A is said to be reducible to, or simply reduces to, a language B, if we can solve (recognize/decide) A with a solution (recognizer/decider) for B. This means that A can't be harder than B since B is powerful enough to solve A. Symbolically, it is natural to write $A \leq B$ to illustrate this idea.

Leveraging this idea and plugging in some known unsolvable A, we can show that B is also unsolvable. Let's see this in action.

Proposition 5.1. $HALT_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a Turing machine that halts on } w\}$ is undecidable.

Proof. Suppose on the contrary that T is a decider for $HALT_{\text{TM}}$, with which we shall construct a decider D for A_{TM}. Upon input $\langle M, w\rangle$, let D first invoke T on $\langle M, w\rangle$. If T accepts, then return M on w; otherwise, reject immediately. Because T is a decider, if T accepts, then M is known to halt on w, and we may conclude that D decides A_{TM}. However, this languages is known to be undecidable, which leads us to a contradiction that invalidates our assumption. □

Of course, using the same technique as for A_{TM}, one could show directly that $HALT_{\text{TM}}$ is undecidable without resorting to reducibility. To really show how powerful reducibility is, then, let's take a look at some more examples.

Proposition 5.2. $E_{\text{TM}} = \{\langle M\rangle \mid M \text{ is a Turing machine such that } L(M) = \varnothing\}$ is undecidable.

Proof. Suppose on the contrary that T decides E_{TM}, with which we will construct a Turing machine D that decides A_{TM}. Upon input $\langle M, w\rangle$, we define the following TM $\tilde{M}$. Upon input x, accept x if M accepts w and reject otherwise. Let D

return T on $\langle \tilde{M} \rangle$. Note that the machine $\tilde{M}$ is not being run by D, so we need not worry about whether M halts. Now, by construction, D decides A_{TM}. □

References

[1] Michael Sipser. *Introduction to the Theory of Computation.* Cengage Learning, 3 edition, 2013.