Contents

Preliminiaries

Formal Language Theory

The entire subject is formalized upon the notation used in *formal language theory*.

Definition 0.1 (Alphabet)**.** An alphabet is a finite non-empty set of distinct symbols or glyphs or characters.

We most commonly will denote an alphabet with Σ. Some common alphabets can include $\{a, b\}, \{1\}, \{0, 1\}, \{a, \ldots, z, A \ldots, Z\}$. Let $\Sigma = \{a, b\}$ and consider the cartesian product

$$\Sigma^2 = \Sigma \times \Sigma = \{(a, a), (a, b), (b, a), (b, b)\}$$

Its elements are conventionally represented in tuple form, such as (a, a). Here, we will drop the cumbersome paranthesi and commas and let $\Sigma^2 = \{aa, ab, ba, bb\}$. These elements are called strings, or words.

Definition 0.2 (String, word)**.** A string or word is a finite sequence of letters from some alphabet. The length of the string is the number of symbols it contains.

Generalizing the previous example, Σ^n is all possible strings of length n. The set Σ^0 is defined as $\Sigma^0 = \{\varepsilon\}$, where ε is a special string of length zero called the empty string, $\varepsilon =$ "". It is different from the empty set $\emptyset$. It is analogous to the difference between an array of no elements, and a string of no length. They are of different types. For n a natural number, we define

$$w^n = \underbrace{ww \ldots w}_{n}$$

the word w concatenated with itself n times, with $w^0 = \varepsilon$. For example, $a(bc)^3d = abcbcbcd$ which is distinct from $ab^3c^3d = abbbcccd$.

We define

$$\Sigma^* = \bigcup_{i=0}^{\infty} \Sigma^i$$

which is the set of all strings.

Definition 0.3 (Language)**.** A language $L \subseteq \Sigma^*$ is any set of words over some alphabet.

Here are some examples of languages

- $L = \{aa, bb, abab, aaa, b\}$
- $L = \{w \in \Sigma^* \mid w \text{ begins with } a\}$
- $L = \{w \in \Sigma^* \mid \#a(w) \text{ is even}\}$
- $L = \{a^n \mid n \text{ is even}\}$
- $L = \{w \in \Sigma^* \mid \#a(w) \equiv 3, 4 \pmod 7\}$

- $L = \{w \in \Sigma^* \mid w \text{ is an encoding of a prime number}\}$

Its important you understand which way "the infiniteness" of an infinite language can go. There are no infinite length strings, each string must eventually terminate and have a specific length. But an infinite language has infinitely many words, and the length of words can be increasing, but each word is itself finite. It is analogous to how the natural numbers, $\mathbb{N} = \{0, 1, 2, ...\}$ is infinite, yet each number itself may be written with only finitely many digits.

Automata

An automata is a hypothetical model of a computer. We may study the limitations of certain automata, or contrast them to one another. We do not really care about the automata themselves, but what they can tell us about the kinds of problems they can solve.

We need the ability to first discuss what it means to solve a problem, and here we borrow tools from formal language theory. A decision problem a partition of Σ^* into the "good" and the "bad"; A language, and its complement. We give an automata a word, and it will either accept the string or reject it. We say that an automata M decides a language L if:

$$M \text{ on input } w \textit{ accepts} \iff w \in L$$
$$M \text{ on input } w \textit{ rejects} \iff w \notin L$$

We are concerned with what kinds of automata can decide what kinds of languages. There are two perspectives. First fix the machine, and note that each machine must define some language. Every machine has some behavior. Next is to fix the language and consider all possible machines which may decide it correctly. We are more concerned with the second perspective than the first.

Definition 0.4 (Decision Problem)**.** A Decision problem is a computational problem which can be phrased as a yes or no question.

Phrasing everything as decision problems simplifies the mechanics immensely, and lets us fully use the power of set theory. A decision problem can easily be formalized in set theory, since for any set, a word is either in the set, or not in the set. This is a binary relationship. Consider the language $\{w \in \Sigma^* \mid w \text{ is an encoding of a prime number}\}$ A machine to decide this language would take on input a word and simply have to accept or reject. We could then remark that this kind of automata would have the power of understanding prime numbers. A more cumbersome way would be to try to use *search problems*, which could be phrased like *on input n, output the nth prime.* If such a machine could perform such a task, we could remark that it too could comprehend prime numbers. But there is a diversity of ways different machines could output the nth prime number. By restricting the output of automata to be a simple boolean, we only need to ensure they have two small wired lights. This will help us compare and contrast them effectively.

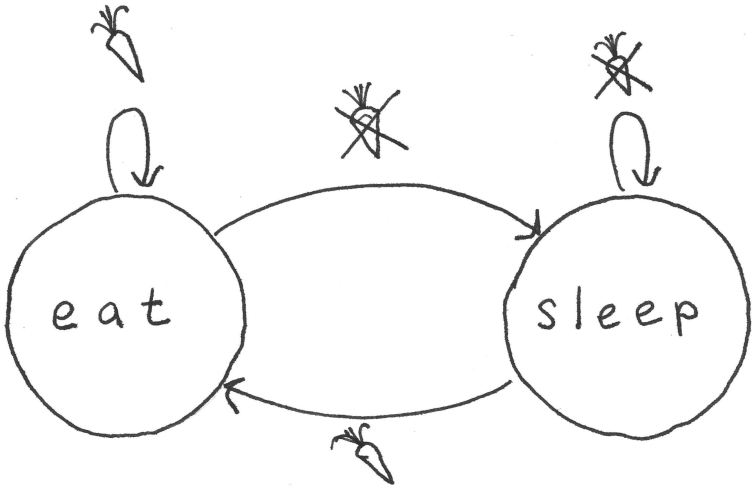
0.1 Exercises

1. Prove that a language is infinite if and only if it has no longest string.
2. Find a bijection from Σ^* to $\mathbb{N}$.
3. Let $x, z, y \in \Sigma^*$ with x, z not empty. Prove that $xy = yz$ if and only if there exists strings u, v and a natural number n such that $x = uv$, $y = vu$ with $y = (uv)^n u = u(vu)^n$
4. For any word w, denote w^R as the reversal of w. Prove that if $x, y \in \Sigma^*$ then $(xy)^R = y^R x^R$.

Regular Languages

1.1 Deterministic Finite Automata

Many systems and processes can be represented as a finite collection of modes, steps or "states of mind". Lets consider an example. My rabbit has exactly two braincells, and they have to take turns. The first braincell, she uses when she eats, and the second braincell, she uses when she sleeps. We may represent these two states of mind as two labelled nodes. We may then define *transitions* between those states of mind as outgoing arrows, on which a transition is acted upon sensory input. There is either food, or no food. She will sense food, wake up, and continue eating until there is none. We could represent the relationship between her states of mind with the following *state diagram.*



Many complex systems could be primitively modeled this way. Even processes which are assumed continuous can be arbitrarily discretized, such as the phases of the moon, the water cycle, the economy, and limbic system. Our first automata is motivated by this simple design.

Definition 1.1. A Deterministic Finite Automata (DFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$:

- $Q = \{q_0, \ldots, q_k\}$ is a non-empty finite set of *states.*
- Σ is the non-empty finite *alphabet*, usually $\Sigma = \{a, b\}$ or $\{0, 1\}$
- $\delta : Q \times \Sigma \to Q$ is the *transition function.* It is a well-defined finite function. Every state symbol pair in the input has a single output.
- $q_0 \in Q$ is the designated *start state.* We have to start somewhere.
- $F \subseteq Q$ is the set of *acceptance*, or final states. If a state is not accepting, we may say it is rejecting.

Let $w = w_1 ... w_n$ be a word. We say that D accepts w if there exists a sequence of states $s_0, ..., s_n$ such that:

- $s_0 = q_0$
- For $0 \leq i \leq n-1$ that $s_{i+1} = \delta(s_i, w_{i+1})$
- $s_n \in F$

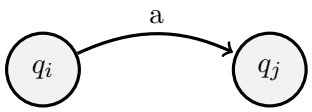
We say that D rejects w if D does not accept w. The language decided by D is the set of strings that it accepts, and it denoted as $L(D)$.

State Diagrams

Working with the formal definition can be a bit cumbersome as we shall see. We prefer to use *state diagrams*, which are a kind of graphical programming language, similar to that given in the bunny example.

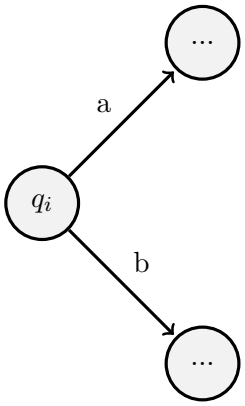
- The states, Q, will be denoted by circles with internal labels.

- The transition function, δ, will be denoted as arrows such that $\delta(q_i, a) = q_j$ would be drawn as

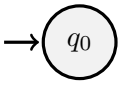


  Note that a transition may return to the same state, and we denote this as a self loop. If there are two transitions of the form $\delta(q_i, a) = q_j$ and $\delta(q_i, b) = q_j$, we use only one arrow with a label "a, b".

- The transition function is correct when each state has exactly $|\Sigma|$ outgoing transitions, one per symbol. For example, if $\Sigma = \{a, b\}$, then every state should have two outgoing arrows. One for a, and one for b. Either or both could perhaps be returning to same state.



- The start state will be denoted with a small arrow from nowhere. It conventionally is labelled as q_0.

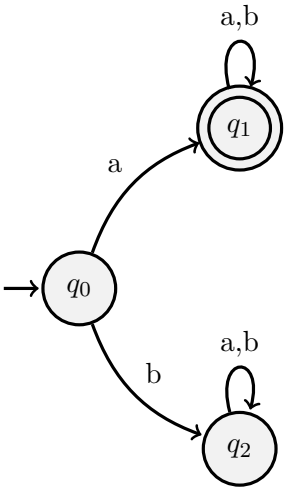


- The accepting states will be labelled with a double circle, and rejecting states as exactly those without.

The start state is allowed to be accepting.

Lets do several examples. One could draw DFAs and consider the languages that they decide. Instead, we enumerate several languages and consider the possible DFAs that decide them.

Example 1.1. $L_1 = \{w \in \Sigma^* \mid w \text{ begins with } a\}$



This diagram clearly communicates all five parts of the tuple, but if we were to state it using the formal definition, it would look like

- $Q = \{q_0, q_1, q_2\}$
- $\Sigma = \{a, b\}$
- The start state is q_0.
- The transition function δ can be encoded as the following table:

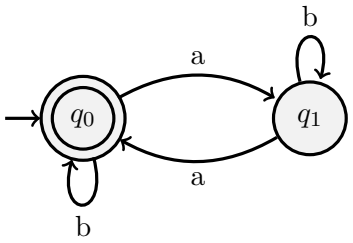
Q	Σ	Q
q_0	a	q_1
q_0	b	q_2
q_1	a	q_1
q_1	b	q_1
q_2	a	q_2
q_2	b	q_2

- $F = \{q_1\}$

The state diagram communicates all five parts more effectively, but know we may delegate to the formal tuple definition during proofs.

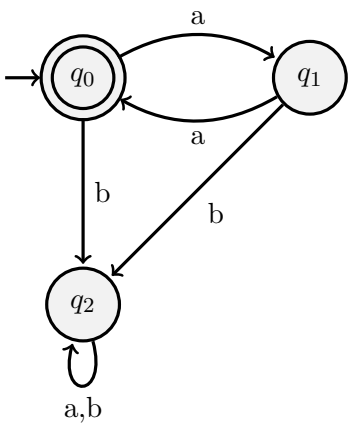
Consider a computation of this DFA on any word. It branches to two different states on the first letter. Once you enter states q_1 or q_2, you may not leave. Once you enter either of these two purgatories, the rest of the letters of the word are ignored. We denote q_1 as the good purgatory by making it a final state, and q_2 as the bad purgatory by making it a rejecting state.

Example 1.2. $L_2 = \{w \in \Sigma^* \mid \#a(w) \text{ is even}\}$



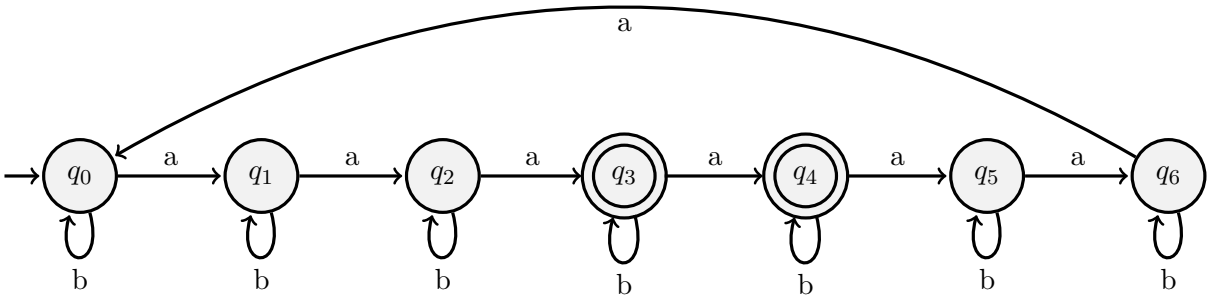
Here we have transitions to keep track modulo two the number of a's we have seen. Any computation of ε on a DFA will end on the start state. Therefore, a DFA accepts the empty string if and only if its start state is also an accepting state. Each state has a self loop upon seeing a b, which means they are ignored. What if we wanted to reject whenever we saw any b's, and not ignore them?

Example 1.3. $L = \{(aa)^n \mid n \in \mathbb{N}\}$



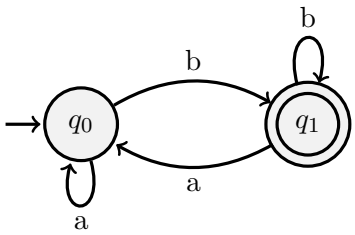
Now as soon as we see a b, we immediately enter purgatory and can never leave.

Example 1.4. $L = \{w \in \Sigma^* \mid \#a(w) \equiv 3, 4 \pmod 7\}$



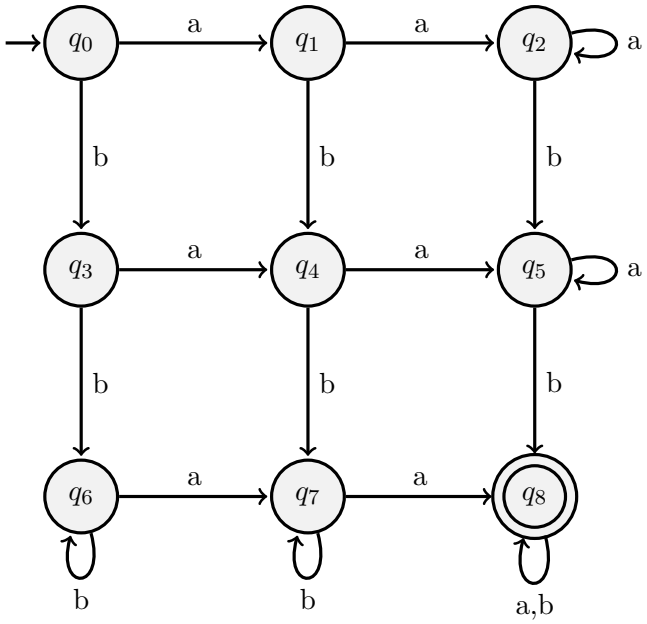
There is nothing too special about the number two. We may generalize the previous example to keep track of residues modulo any other number. Create one state per equivalence class, and simply transition between them on seeing an a, and ignore all b's. Going from one state to the next means you have seen an additional a. Going around the clock means you have seen a seven times.

Example 1.5. $L = \{w \in \Sigma^* \mid w \text{ ends with } b\}$



The DFA does not have any ability to rewind its input, jump around, or do any pre or post processing. It halts exactly when it runs out of symbols to read, but it doesn't know when that will happen. Like death, it must be vigilant and prepared for it at any moment. In this DFA, if we see a b, we transition to an accept state, prepared for the end, but if we see an a, we must transition away.

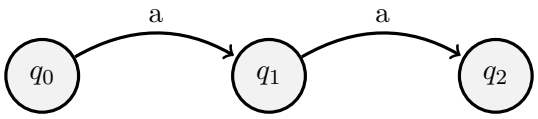
Example 1.6. $L = \{w \in \Sigma^* \mid w \text{ has atleast two } a\text{'s and atleast two } b\text{'s}\}$



To each state, we may correspond a *semantic meaning*. Here, there are several paths to the accept state. It has been drawn in such a way that the columns keep track of the number of a's you have seen, and the rows keep track of the number of b's. In fact, the unique paths to the accept state correspond to in which order you saw your required two a's and two b's. The fact that the shortest path to the accept state is of length four can also tell us that the shortest strings in this language must be of length four.

Programming Advice

A DFA is essentially a very limited kind of finite space program. There are only finitely many things it can keep track of at once, and it can also only interact with its input in a "read once only" way. When designing DFAs, it is helpful to think that each state has assigned to it a semantic meaning, and that a state can only be reached by certain strings which satisfy certain properties. For example, suppose we have a portion of a DFA which looks like the following.



You know you may only enter q_1 upon seeing an a. You know you may only enter q_2 not only upon seeing an a, but seeing an a from q_1, which you could only enter upon seeing an a. So q_2 may be entered only upon seeing aa. Each state corresponds this way like a line of code. You know a certain line of code may be hit by the control flow only if certain conditions are met. Our portion of a DFA could correspond to a portion of a program as:

```
if w[i] == 'a':
    if w[i+1] == 'a':
        *
```

You know the $*$ line will only be executed if certain conditions are met. Analogously, q_2 can only be entered by a string if a prefix of it meets certain conditions. It is helpful to think about each

state of a DFA as a line of code of some limited program, and the transitions between the states analogous to how the control flow moves between lines of code.

Regular Languages

What kinds of languages can DFAs decide? We don't yet know the problems they are capable of solving or not solving. We say a language is regular if and only if it is decided by a DFA.

Definition 1.2. We write the class of languages decidable by a DFA as $\mathscr{L}(DFA)$. The language $L \in \mathscr{L}(DFA)$ if and only if there exists a DFA to decide L. These are called the regular languages.

Note that a word is a finite sequence of symbols, a language is a (possibily infinite) set of words, and a class is a (possibily infinite) set of languages. A class is a set of sets of strings. We want to study the regular languages, and the only way for us to do so is by studying DFAs. What properties to the regular languages have?

Theorem 1.1. If $L \in \mathscr{L}(DFA)$, then $\overline{L} \in \mathscr{L}(DFA)$. The regular languages are closed under complement.

A DFA is a machine to tell you exactly what strings to accept, but by doing so, it also tells you exactly what strings to reject.

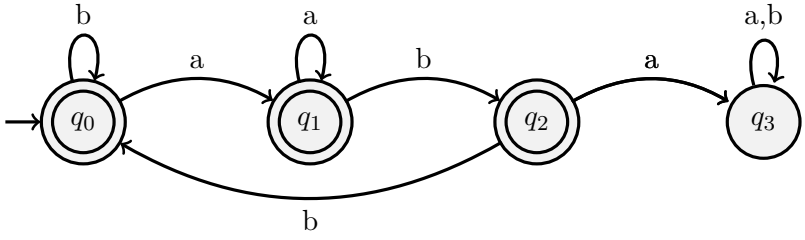
Proof. Let L be a regular language, then there exists a DFA $(Q, \Sigma, q_0, \delta, F)$ to decide L. Consider the DFA $(Q, \Sigma, q_0, \delta, Q \setminus F)$. It is identical to the first DFA, except that every previously accepting state is now rejecting, and every previously rejecting state is now accepting. If $w \in L$ then the first DFA will accept w, so the second DFA will reject w. If $w \notin L$, then the first DFA will reject w, so the second DFA will accept w. Therefore, the second DFA will except exactly and only the strings not in L, which is the complement $\overline{L}$. Since the second DFA is one which decides $\overline{L}$, then $\overline{L}$ has a DFA to decide it, and is therefore, regular. □

Often times, the core idea of such proofs are a construction of an automata of some sort. For some complex constructions, a proof of correctness could be done by induction on the length of the input. The correctness of the automata is often obvious, and such a wordy proof is often unnecessary. Analogously, many primitive algorithms also have omitted proofs of correctness, when the algorithm is simple enough that it is obvious. We don't really understand theorems until we understand their proofs, but computation is such a natural, human cognitive process, an example is almost as instructive.

Suppose we wanted to create a DFA for the language

$$\{w \in \Sigma^* \mid w \text{ does not contain the substring } aba\}$$

Rather than try to find some kind of positive characterization of strings with this property, lets just check if the string does contain aba as a substring. We reject those strings, and then accept everything else.



Theorem 1.2. Let $L_1, L_2 \in \mathscr{L}(DFA)$. Then $L_1 \cap L_2 \in \mathscr{L}(DFA)$. The regular languages are closed under intersection.

We may use one DFA to simulate two other DFAs simultaneously, and have our DFA accept if and only if the two DFAs it is simulating accept. Consider how DFAs are analogous to a very limited kind of program. Among its other limitations, it only uses constant memory. We may combine two constant memory programs into one (bigger) constant memory program. This is the intuition. Each state of our new DFA will correspond to a pair of possible states in two different DFAs. Computation on our DFA will correspond to computation on two other DFAs in parallel. We shall make our DFA accept exactly when the two DFAs it is simulating accept. For simplicity, suppose they have the same fixed input alphabet.

Proof. Let L_1 be decided by DFA $(Q_1, \Sigma, q_0^1, \delta_1, F_1)$ and L_2 be decided by DFA $(Q_2, \Sigma, q_0^2, \delta_2, F_2)$. We program a DFA called the cartesian product DFA $(Q, \Sigma, q_0, \delta, F)$ to decide $L_1 \cap L_2$ as follows:

1. $Q = Q_1 \times Q_2$
2. Σ is the same
3. $\delta : (Q_1 \times Q_2) \times \Sigma \to (Q_1 \times Q_2)$ such that

   $$\delta((q_i, q_j), a) = (\delta_1(q_i, a), \delta_2(q_j, a))$$

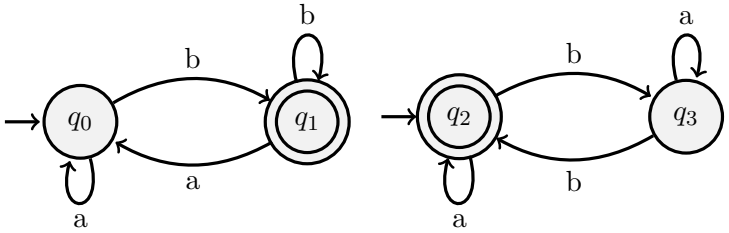
   for $q_i \in Q_1$ and $q_j \in Q_2$. The first DFA is simulated in the first coordinate, and the second DFA in the second coordinate.
4. $q_0 = (q_0^1, q_0^2)$
5. $F = F_1 \times F_2$

Let $L_1, L_2 \in \mathscr{L}(DFA)$. Then there exists DFAs D_1, D_2 to decide L_1, L_2. Let $w \in L_1 \cap L_2$, then $w \in L_1$ and $w \in L_2$. Then the computation of D_1 on w ends at some accepting state q_i, and the computation of D_2 on w ends at some accepting state q_j. Consider our cartesian product construction. The computation of D on w will end on state (q_i, q_j), which by our construction, is accepting. Thus D accepts w. Similarly, if $w \notin L_1 \cap L_2$, then the cartesian product DFA will not reach an accept state on w, and it will reject it. This DFA accepts exactly the strings in $L_1 \cap L_2$. Since $L_1 \cap L_2$ has a DFA to decide it, it is regular, and we see that $L_1 \cap L_2 \in \mathscr{L}(DFA)$. □

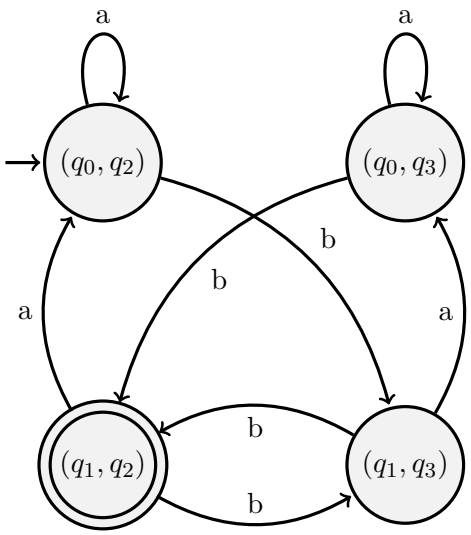
To demonstrate this construction, Lets proceed with an example.

$$L_1 = \{w \in \Sigma^* \mid w \text{ ends with a } b\}$$
$$L_2 = \{w \in \Sigma^* \mid \#b(w) \text{ is even}\}$$

Lets make two DFAs for these languages.



Out cartesian product DFA then looks like the following:



We may assign meaning to the states. State (q_1, q_2) means being in state q_1 in the first DFA and state q_2 in the second DFA simultaneously. You may only end on state q_1 if your string ends with b, and you may only end on state q_2 if you have seen an even number of b's at that point. So strings which end on state (q_1, q_2) are those which both end with b and have seen an even number of b's. If a string lands on state (q_0, q_2), then it has an even number of b's but ends with an a. If a string lands on state (q_1, q_3), then it ends with a b but has an odd number of b's. If a string lands on (q_0, q_3), then it has an odd number of b's and doesn't end with a b.

Notice that the transitions keep track of how the two DFAs would compute upon the word, and the accepting states keep track of how the two DFAs would accept or reject. If a string lands on states (q_0, q_2) or (q_1, q_3), then it is accepted by one DFA but not the other. What if you wanted the simulator DFA to accept if either DFA accepted, but not necessarily both?

Theorem 1.3. Let $L_1, L_2 \in \mathscr{L}(DFA)$. Then $L_1 \cup L_2 \in \mathscr{L}(DFA)$. The regular languages are closed under union.

Proof. Simply modify the construction from the previous proof, where the only difference is that the final states are constructed as

$$F = \{(q_i, q_j) \mid q_i \in F_1 \text{ or } q_j \in F_2\} = (Q_1 \times F_2) \cup (F_1 \times Q_2)$$

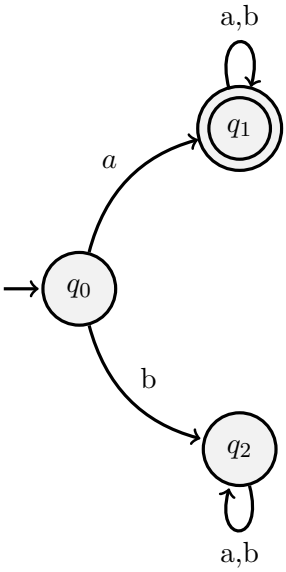
Then our cartesian product DFA would accept if any of its two simulated two DFA accepted, and it would reject if both DFAs rejected. □

In the previous example, our accepting states would then be $F = \{(q_0, q_2), (q_1, q_2), (q_1, q_3)\}$.

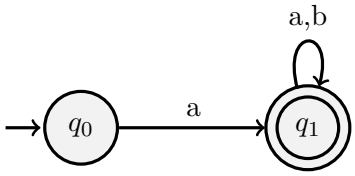
1.2 The Generalization of Nondeterminism

We noted that DFAs are weak. Let's try to generalize them. Recall that a DFA can be represented as a tuple $(Q, \Sigma, \delta, q_0, F)$. Given this definition, we wish to modify it to hopefully extend its power. The only useful thing we can extend is the way in which states interact with each other; the transition function δ. The rest of the device is static. We extend δ in the following three ways:

The first generalization we make is that of *implicit rejection*. We allow transitions to be undefined, and it is understood that undefined transitions implicitly reject. As an example, recall the following DFA which decides the language $\{w \in \Sigma^* \mid w \text{ begins with } a\}$.

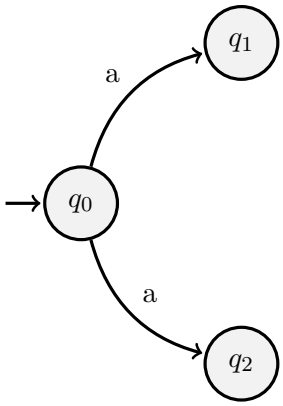


With implicit rejection, we could represent equivalently as



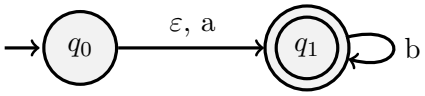
If we are at q_0 and we see b we would reject. This can be helpful for programming. Consider how well-defined a DFA is. Its like it has every edge case covered with all that try-catch nonsense. Implicit rejection allows us to lazily construct only the parts that we care about. Then "undefined behavior" results in immediate rejection. We can have an "if" without having to have a matching "else". We require the self loop on q_1, otherwise, this would implicitly reject all strings which are length two or more. Note that when we perform a complement of the accept states in a DFA that decides a language L, we get the complement of the language. The same does not hold here due to implicit rejection.

The second generalization we make is that of *nondeterminism*. We allow transitions of more than one of the same type. This means that you can have multiple outgoing transitions with the same input. For example:

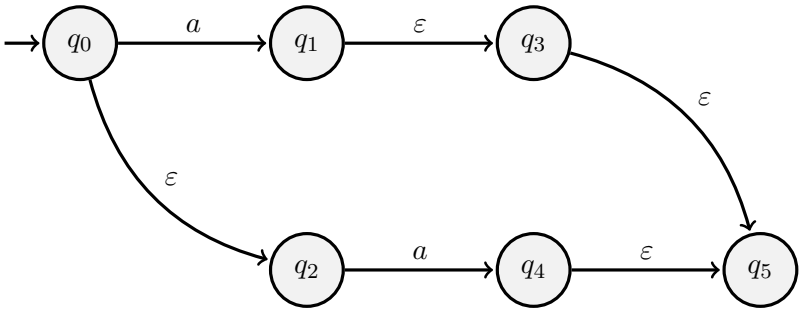


Consider the computation on a word beginning with an a. Which state are you in? q_1? q_2? You are in both! We define a nondeterministic computation to accept *if there exists* an accepting computation. For all possible states you could end up in, if at least one of them is accepting, then the NFA accepts the string. An NFA rejects a string if it doesn't accept, and this happens *if for all* computations, none are accepting. We shall expand on nondeterminism soon.

The third and final generalization we make is that of *ε-transitions*. taken "for free". For example:



a, ab, abb are some strings which are accepted. But now that we allow ε-transitions, b, bb, ε are also accepted. An accepting computation of the word bb would first take the ε transition from q_0 to q_1, then take the b transition twice to remain at q_1. While normally, each transition "costs" the next letter of the input, an ε-transition costs nothing. It is important to know that the choice to take it is not forced. A nondeterministic computation may choose not to take it. For example



On input of ε, this NFA will end on state q_0 and q_2 simultaneously. On input of a, it will end on states q_1, q_3, q_4, q_5. On input of aa, it would implicitly reject.

Formal Definition

Definition 1.3. A Nondeterminisitic Finte Automata (NFA) can be represented by a 5-tuple $(\Sigma, Q, q_0, \delta, F)$ where:

- $Q = \{q_0, \ldots, q_k\}$ is a non-empty finite set of *states*
- Σ is the non-empty finite *alphabet*, usually $\Sigma = \{a, b\}$ or $\{0, 1\}$

- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$ is the *transition function*. It is a well-defined finite function. Every state symbol pair in the input has a single output, as a set of states.
- $q_0 \in Q$ is the designated *start state*. We have to start somewhere.
- $F \subseteq Q$ is the set of *acceptance*, or final states. If a state is not accepting, we may say it is rejecting.

Let $w = w_1 \ ... \ w_n$ be a word. We say that N accepts w if for $m \geq n$, there exists there exists a sequence $x = x_1 \ ... \ x_m$ with each $x_i \in \Sigma \cup \{\varepsilon\}$ such that if all ε's were removed from x it would simply be equal to w and a sequence of states $s_0, ..., s_m$ where:

- $s_0 = q_0$
- For $0 \leq i \leq m-1$ that $s_{i+1} = \delta(s_i, x_{i+1})$
- $s_m \in F$

We say that N rejects w if there does not exist such a sequence. The language decided by N is the set of strings that it accepts.

The NFA accepts if *there exists* a computation paths and rejects if *all* computation paths are rejecting. Contrast the transition function of the DFA with the transition function of the NFA.

DFA: $\delta : Q \times \Sigma \to Q$

NFA: $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$

We allow a transition to be taken, upon reading no symbols. We model this by changing the transition function domain from $Q \times \Sigma$, to $Q \times (\Sigma \cup \{\varepsilon\})$. By having the co-domain be $\mathcal{P}(Q)$, the set of all subsets of states, we can allow a state symbol pair to map to any set of states. An implicitly rejecting transition would have the formal transition function map to the empty set $\emptyset$.

With these three new relaxations, we have defined a new kind of automata, the nondeterministic finite automata (NFA). On input a word, there may be multiple different possible computations, and we say an NFA accepts some string if *there exists* atleast one computation to an accepting state. It does not matter how many more rejecting computations there are. Non determinism is a biased power, in that it only needs one accepting computation to accept. But to reject, every computation must be rejecting.

Coping with Nondeterminism

Its important to understand nondeterminism and not just have deterministic coping strategies. Nondeterminism isn't real. You could not build a nondeterministic computer, but it doesn't matter. We may still study this unrealizable machine as a purely theoretical device. The following analogies may help in visualizing this power.

- **Graph Search** An NFA or DFA can just be thought of as a graph. A word computed by an NFA or DFA can be thought of as a path in the graph. You could easily determine if a DFA accepts a word by using the word as instructions on which path to take in the DFA, but to determine if an NFA accepts a word, you may have to employ a graph search algorithm, such as breadth first search or depth first search. Coming to a fork in the road, you explore down both paths until you find an accept state. Under this view, the power of nondeterminism is

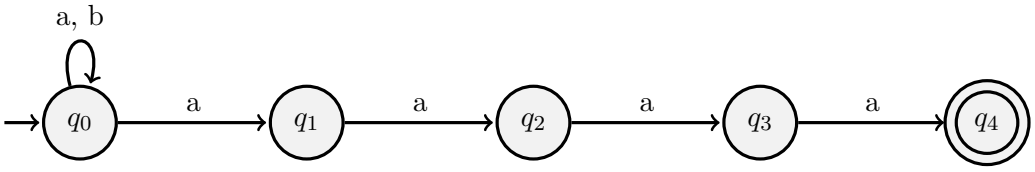
only how time is measured. The DFA on computation of a word of length n takes exactly n steps. An NFA also takes exactly n steps, but epsilon transitions take no time, and certain paths of the same depth are computed "in parallel", and their time is not double counted. You and I are deterministic. In order to determine if an NFA accepts a word, using pen and paper, necessarily may take more than n steps.

- **Lucky Coin** During your computation you come to a nondeterministic transition. Imagine you flip a lucky coin that tells you exactly which path to take. Through a purely imaginary way, you have divine information on which path will correctly lead you to an accept state. It is not random so much as it is omnipotent. You have faith in the coin, so you follow it. Somehow, you have the precognition to know where the answer is.
- **Alternate Timelines**[1] For each nondeterministic action, create multiple timelines. Each timeline consists of the what-if for each possible choice. As long as in one timeline, you reach an accept state, then the computation is accepting.

Examples

Lets show a few examples. We emphasize how to take advantage of nondeterminism in programming.

Example 1.7. $L_1 = \{w \in \Sigma^* \mid w \text{ ends with } aaaa\}$



Consider the computation of this machine on input a^7. If you are at q_0 and you read an a, you may choose to either stay at q_0 or move on to q_1. This word is accepted by the NFA because it may correctly guess exactly when it is four a's from the end and then choose to leave q_0. Another way is to consider all possible guesses of when to go to q_1 on seeing an a. If we guess too late, we will terminate on one of q_0, q_1, q_2, q_3 and not accept. If we guess too early, we will reach q_4, but then have more input to read, and must implicitly reject since q_4 has no outgoing transitions. Most of the computations will be rejecting but it doesn't matter, as there is atleast one accepting computation, one correct guess.

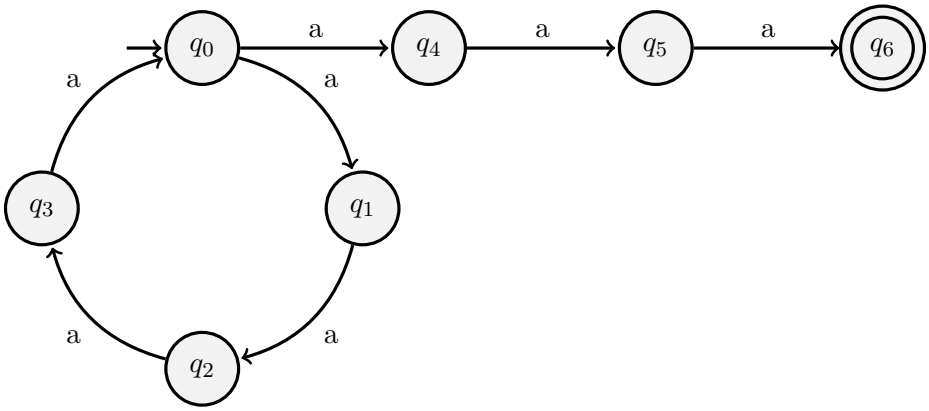
Let the delimiter symbolize when you non deterministically guess to go from q_0 to q_1. There are eight computations of a^7 on this NFA.

$\|a$	a	a	a	a	a	a	too early, implicit rejection from q_4
a	$\|a$	a	a	a	a	a	too early, implicit rejection from q_4
a	a	$\|a$	a	a	a	a	too early, implicit rejection from q_4
a	a	a	$\|a$	a	a	a	accepts
a	a	a	a	$\|a$	a	a	too late, rejects from q_3
a	a	a	a	a	$\|a$	a	too late, rejects from q_2
a	a	a	a	a	a	$\|a$	too late, rejects on q_1
a	a	a	a	a	a	a	too late, rejects on q_0

[1] Different science fictions have different rules for how time travel works. I am going off of the episode Remedial Chaos Theory from Community.

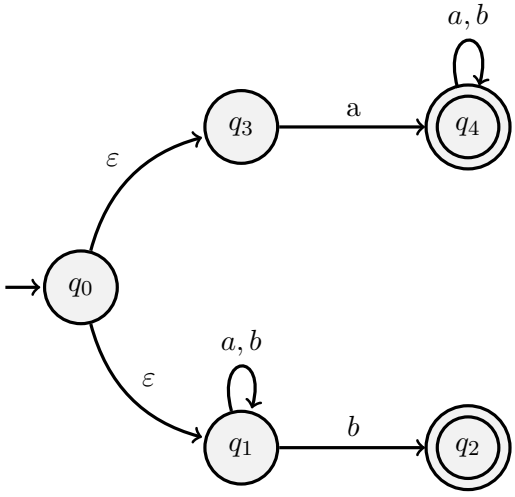
Nondeterminism allowed us to construct this NFA very easily. Just guess when you are four a's from the end and count them. Designing an NFA to accept the correct strings is usually easy, but you need to be careful that it also rejects the incorrect strings. A string is rejected when all of its computation paths are rejecting, and designing around this can be a bit trickier.

Example 1.8. $L = \{a^{4n+3} \mid n \in \mathbb{N}\}$
Lengths of the strings in this language form an arithmetic progression.



You non deterministically n, by choosing how many times you would go around the loop. Every time you reach q_0, do you choose to go to q_1 or q_4? If you choose to go to q_1, you must add another four to ever have a choice to reach an accept state. If you choose q_4, then you must add exactly another three before accepting.

Example 1.9. $L = \{w \in \Sigma^* \mid w \text{ begins with } a \text{ or ends with } b\}$ We need to accept strings if they accept any of two conditions. We will nondeterministically guess which condition to check.



If the string begins with a, then there is a computation on the above branch which accepts. If the string ends with b, then there is a computation on the below branch which accepts, where the prefix of the string is nondeterministically guessed.

Comparison with DFAs

Definition 1.4. We write the class of languages decidable by a NFA as $\mathscr{L}(NFA)$. The language $L \in \mathscr{L}(NFA)$ if and only if there exists a NFA to decide L.

We don't really care about comparing automata themselves, but comparing their *power*. Rather than compare NFAs and DFAs, we compare $\mathscr{L}(DFA)$ with $\mathscr{L}(NFA)$. What is the relationship between the powers of an NFA and DFA?

Theorem 1.4. $\mathscr{L}(DFA) \subseteq \mathscr{L}(NFA)$

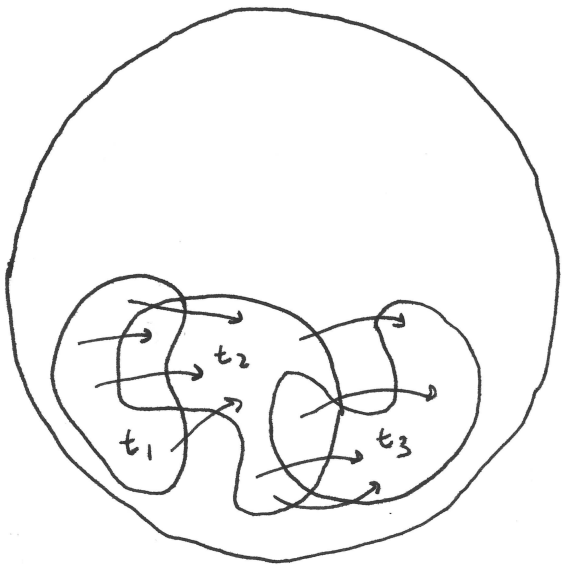
Every DFA is an NFA. An NFA has all these super powers, but there is no requirement to use them. Though it may be obvious just from the generalization that is nondeterminism, for exercise, we prove $\mathscr{L}(DFA) \subseteq \mathscr{L}(NFA)$.

Proof. Let $L \in \mathscr{L}(DFA)$. Then there exists a DFA to decide L. Note that this DFA is also an NFA, so there exists an NFA to decide L. Then $L \in \mathscr{L}(NFA)$. Since this is true for all $L \in \mathscr{L}(DFA)$, we see that $\mathscr{L}(DFA) \subseteq \mathscr{L}(NFA)$. □

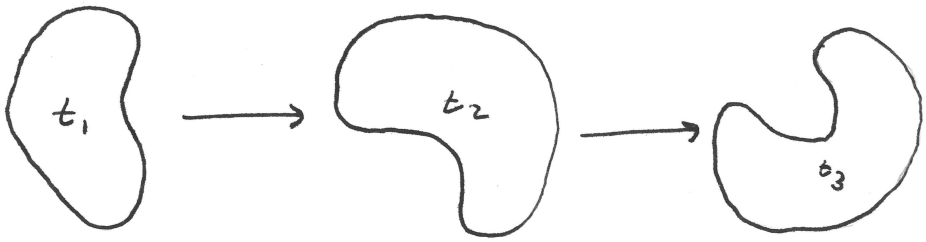
Do these generalizations give the NFA strictly more power? Or not?

Theorem 1.5. $\mathscr{L}(NFA) \subseteq \mathscr{L}(DFA)$

This should surprise you! We gave a normal computation device all this unrealistic unrealizable power. Yet, this power can be simulated using realizable methods. For any NFA, we will show how to simulate it on a DFA. This means that $\mathscr{L}(NFA) \subseteq \mathscr{L}(DFA)$. Combining the aforementioned point, we get $\mathscr{L}(DFA) = \mathscr{L}(NFA)$. This is called a *double set containment*, and is an argument we will use frequently. The proof strategy is as follows. We simulate an NFA on a DFA. Although an NFA may be in many states at once, it can only be in finitely many. Although the NFA moves *nondeterministically* between the states, it moves *deterministically* between the sets of states. This will motivate our powerset construction. We will construct a DFA to deterministically simulate how the NFA transitions between sets of states. For example, if we have an NFA with some large number of states, suppose at some step of the computation, it is at a set of states t_1, then after a single step, it is in a different state of states t_2, then after another step, t_3.



Although the NFA is in multiple states simultaneously, we may simulate this nondeterminism by deterministically moving between the set of states rather than the states themselves.

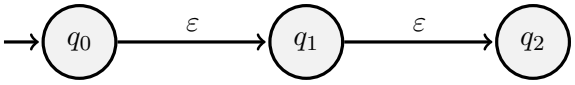


This is the key idea behind the simulation. To each possible set of state the NFA could be in, we assign one state of our DFA to represent each subset of the NFA. Then the NFA going between subsets of states can be simulated by our DFA going from just one state to another. This is called the powerset construction. There is also a small comment on economy. NFAs can be smaller. There exists regular languages with small NFAs but large DFAs, but they still have DFAs. The DFA may take cost to simulate the NFA, but it can still do so sucessfully, proving that the NFA is not more powerful. There is exponentially more subsets than elements, but that is still only a finite amount. There is also the issue of these epsilon transitions.

Definition 1.5. Let $reach : Q \to \mathcal{P}(Q)$ such that

$$reach(q_i) = \{q_i \text{ and any state reachable from } q_i \text{ by } \varepsilon\text{-transitions}\}$$

For example if you suppose we had a portion of an NFA as follows



Then $reach(q_0) = \{q_0, q_1, q_2\}$. We now proceed to prove that $\mathscr{L}(NFA) \subseteq \mathscr{L}(DFA)$

Proof. Let $L \in \mathscr{L}(NFA)$. Then there exists an NFA $N = (\Sigma, Q, q_0, \delta, F)$ to decide L. We construct an equivalent DFA $D = (\Sigma^{'}, Q^{'}, {q_0}', \delta', F')$ so that $L = L(N) = L(D)$.

- $Q' = \mathcal{P}(Q)$ For each possible subset of the states of the NFA, we creat one state of our DFA.
- $\Sigma' = \Sigma$
- ${q_0}' = reach(q_0)$ If there is an ε-transition from the start state of the NFA, then the computation need not necessarily begin at q_0 if this ε-transition is taken first. Then the start state of our DFA corresponds to the set of possible states in which the computation could begin in the NFA, which is those states reachable from q_0 in the NFA.
- For $S \subseteq Q$ any subset of states of the NFA and $a \in \Sigma$, we define

$$\delta^{'}(S, a) = \bigcup_{q \in S} reach(\delta(q, a))$$

  For S a state of the DFA, its outgoing transitions are defined to be the state corresponding exactly and only to the set of states of the NFA which you can go to on viewing the same symbol.

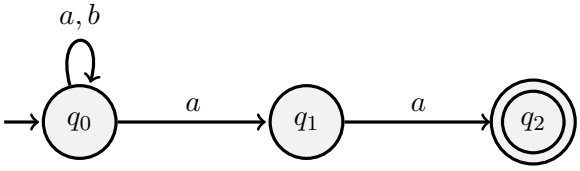
- $F' = \{S \subseteq Q \mid S \cap F \neq \emptyset\}$ Recall than an NFA accepts if there exists a computation which reaches an accept state. After computation on a word, you may be in several states at once, but if atleast one is accepting, the machine accepts. We set the accepting states of the DFA to be those which contain any accept state of the NFA.

Let $w \in L$, then on the computation of w, the NFA N reaches an accept state q_i. Consider D on input w. The DFA will reach a state marked as a set of states containing element q_i. By construction, this will be accepting, so D will accept w. Similarly, if $w \notin L$, then in all computation paths, N rejects w. Again consider D on input w. By construction, it will reach a state which is explicitly marked as non accepting, so D rejects w. Since D accepts exactly and only the same strings that N did, we see that $L \in \mathscr{L}(DFA)$, and thus $\mathscr{L}(NFA) \subseteq \mathscr{L}(DFA)$. □

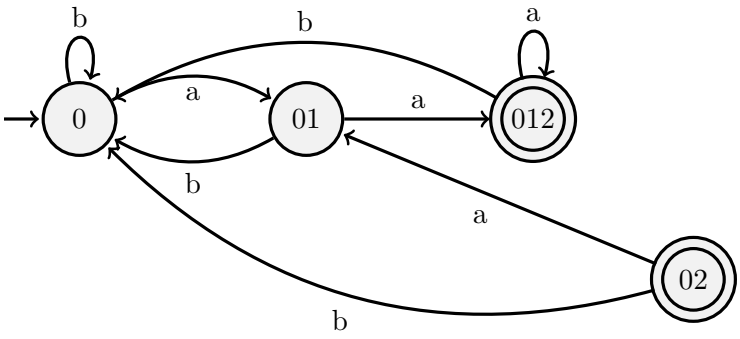
Examples

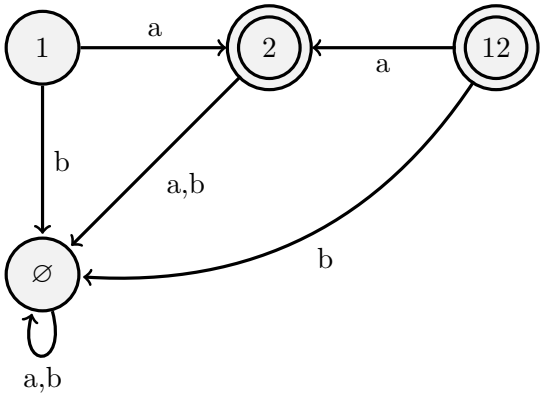
To further illustrate the powerset construction, we give some examples of converting NFAs into DFAs.

Example 1.10. Consider the language $L_2 = \{w \in \Sigma^* \mid w \text{ ends with } aa\}$. An NFA to decide this language could be.

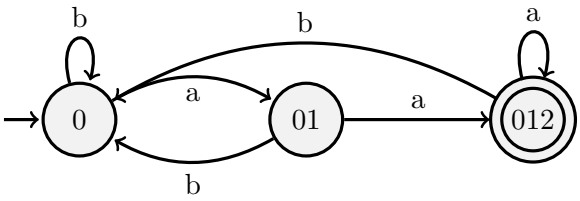


By following the process in the powerset construction exactly, we get the corresponding DFA.





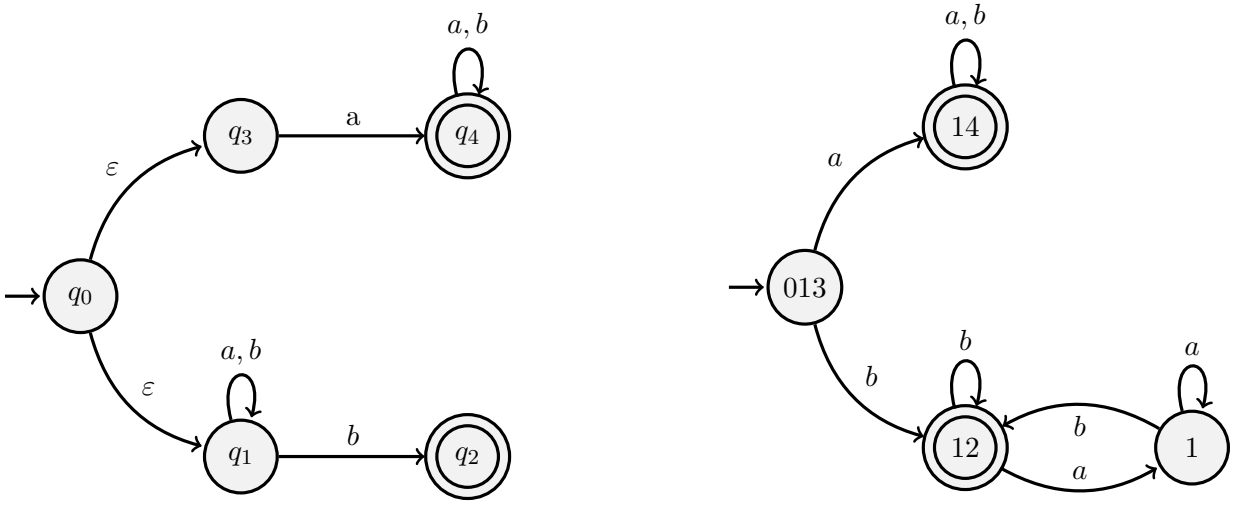
For simplicitly, rather than label a state as $\{q_0, q_1, q_2\}$, we will just label it as 012. We observe that there are unreachable states like q_{02} and an entire disconnected component. This process does not guarantee to give a minimal DFA, just an equivalent one. On cleaning up these unreachable states, we get the following DFA



Each state represents a superposition of the states in the NFA. A state being unreachable in the DFA could be interpreted to mean that its exact combination of states in the original NFA was unachievable. You cannot be in q_2 in the NFA without also being in q_0 and q_1.

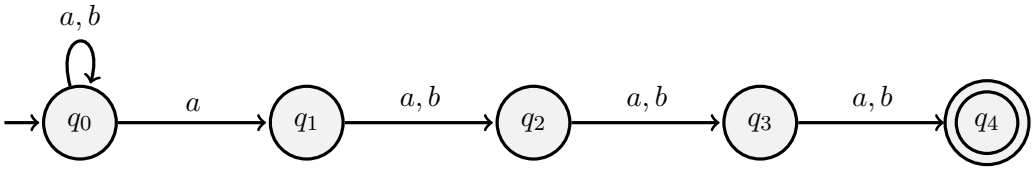
We do an example with ε-transitions. Recall example 1.9. There are five states, so the powerset construction should give us $2^5 = 32$ states. Rather than compute the 32 states first, then cut away those that are useless, we observe exactly what combinations of states are possible to be in simultaneously.

Example 1.11. $L = \{w \in \Sigma^* \mid w \text{ begins with } a \text{ or ends with } b\}$

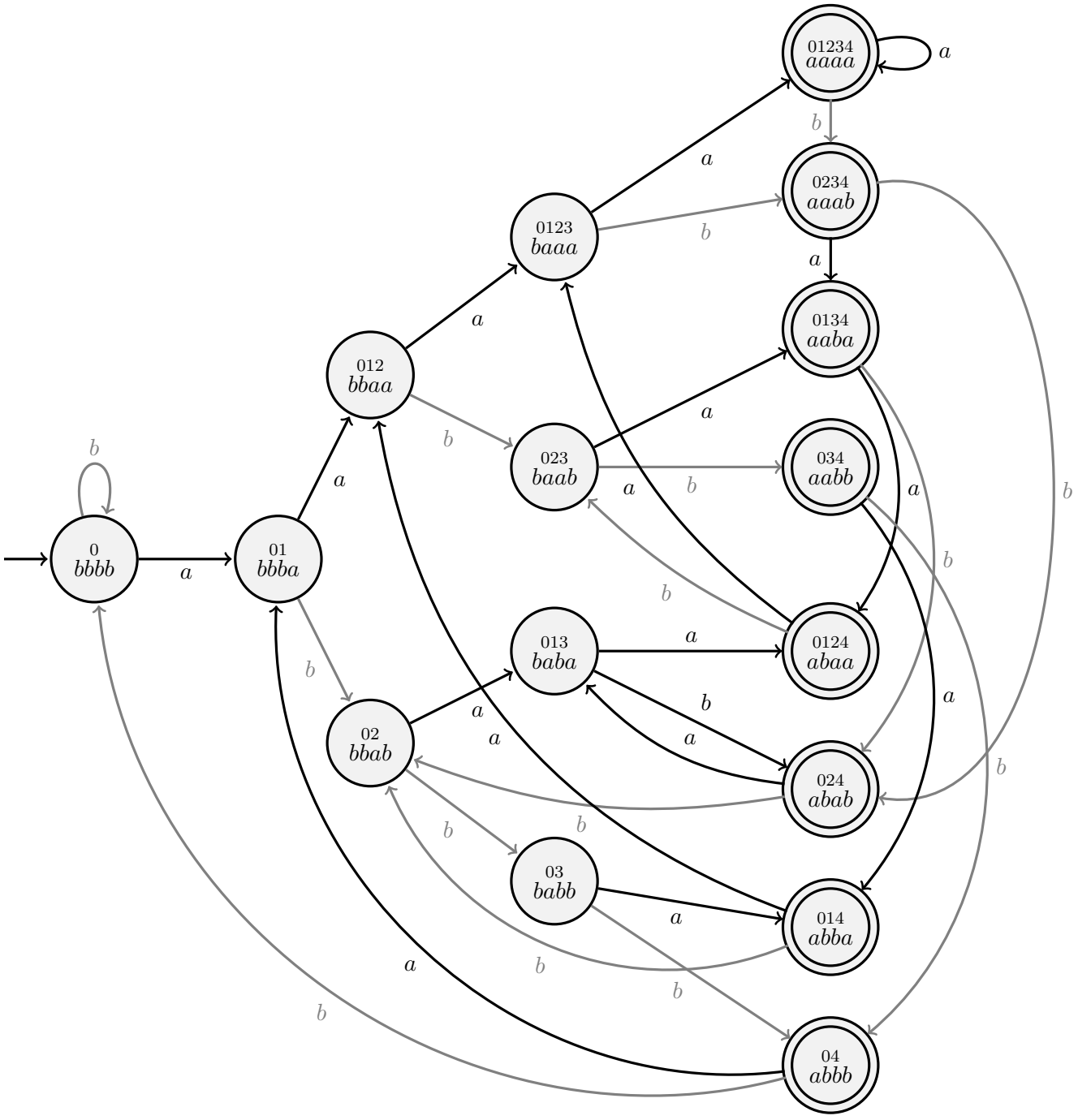


It is interesting to note that this DFA actually has less states than our original NFA. This is because our original NFA was not the smallest one.

Lets do another example. $L = \{w \in \Sigma^* \mid w \text{ has an } a \text{ as the fourth symbol from the right.}\}$. We could easily constrict the NFA similar as



Following the powerset construction, we get the following DFA.



Additional information has been given in this state diagram. For clarity, b transitions are shaded differently. Each state is not only given a sequence corresponding to the subset of states (023 corresponds to $\{q_0, q_2, q_3\}$), but also a word from Σ^4. Each string corresponds to the last four seen symbols. This is the semantic meaning we assign to each state. A DFA has a memory worse than a goldfish, and can only keep track of where it currently is. Upon a computation of "what symbol did I see three letters ago?", you end up needing one state for every possible word of length four,

of which there are $|\Sigma|^4 = 16$. This DFA is quite large and messy, and you can prove this language cannot be equivalently decided by a smaller DFA.

1.3 Regular Expressions

What does it mean to have a name? A name is a finite description. In some sense, the study we are undertaking is the study of what infinite objects have finite names? Every infinite regular language is an infinite object, that has a finite name; the DFA that decides it. Regular expressions are really a way to describe a regular language.

A regular expression is a single string which represents or describes a regular language. You can think of regular expressions as a kind of very limited programming language. Each regular expression is declarative of exactly what strings it wants to describe. First we define them, and then we will prove they correspond exactly and only to the regular languages.

Definition 1.6. We say that R is a **regular expression**, or regex, if R is one of the following:

- $\emptyset$ - empty set
- ε - empty string
- $a \quad \forall a \in \Sigma$
- R_i^*, $R_i R_j$ or $R_i \cup R_j$ where R_i, R_j are regular expressions.

Let R be a regular expression. We denote the language described by R as $L(R)$. We inductively define the languages described by regular expressions as follows.

- $L(\varepsilon) = \{\varepsilon\}$
- $L(\emptyset) = \emptyset$
- $L(a) = \{a\} \quad \forall a \in \Sigma$

Let R_i, R_j be regular expressions. Then

- $L(R_i \cup R_j) = L(R_i) \cup L(R_j)$

  The union of two regular expressions unions their languages. Here, there are two uses of the "$\cup$" symbol. The first is a literal symbol, in a syntactic sense. $R_i \cup R_j$ is a sequence of characters, a string, containing the "$\cup$" symbol. The second use is as a genuine set operation on languages; the union of two sets of strings.

- $L(R_i R_j) = L(R_i)L(R_j)$

  The concatenation of regular expressions concatenates their languages. Let L_i, L_j be languages. We define the concatenation of languages as

  $$L_i L_j = \{xy \mid \forall x \in L_i \text{ and } \forall y \in L_j\}$$

  It is all possible pairs of concatenations, no delimiter, and no choice on which strings are concatenated.

- $L(R_i^*) = L(R_i)^*$

  The star of a regular expression stars its language. Let L be a language. We define the Kleene star of a language as zero or more concatenations of that language with itself.

  $$L^* = \bigcup_{k=0}^{\infty} L^k = \{\varepsilon\} \cup L_i \cup L_i^2 \cup L_i^3 \cup ...$$

We will often conflate a regular expression R with the language it describes $L(R)$. If a regular expression is a name for a language, why not use it. We have a recursive, or inductive definition for a naturally recursive or inductive object. Each regular expression is a string, but it corresponds to a language, a (possibly infinite) set of strings.

Examples

Here are some examples of regular expressions. We often use Σ in regular expressions as a shorthand for the regular expression $(a \cup b)$ or whatever the alphabet may happen to be.

1. $a^* = \{a^n \mid n \in \mathbb{N}\} = \{\varepsilon, a, aa, aaa, \dots\}$
2. $\Sigma^* = (a \cup b)^*$ We introduced this as the definition of all strings, it is actually a regular expression for zero or more concatenations of any of the letters of the alphabet, which corresponds to all strings.
3. $a^*ba^* = \{a^iba^j \mid i, j \in \mathbb{N}\}$ = all strings which contain exactly one b.
4. $\Sigma^*b\Sigma^*$ = all strings with atleast one b. There can be more than one, but not zero.
5. $\Sigma^*abaabab\Sigma^* = \{$all strings with $abaabab$ as a substring$\}$
6. $(\Sigma\Sigma)^* = ((a \cup b)(a \cup b))^* = ((aa \cup ab \cup ba \cup bb))^*$ which is all strings of even length.
7. $\varepsilon(ab \cup ba) = \{ab, ba\}$
8. $(a \cup b)(b \cup c) = \{ab, bb, ac, bc\}$
9. a^*b^* any string where every a comes before every b.
10. $(ab)^*$ every a is followed by a b. and vice versa.
11. $\emptyset^* = \{\varepsilon\}$. By definition, $\emptyset^* = \emptyset^0 \cup \emptyset \cup \emptyset^2 \cup \dots$. All concatenations greater than one are empty, so this simplifies to $\emptyset^0$. Here, zero strings are concatenated together. This vacuously gives us one string of no length, thus $\emptyset^* = \{\varepsilon\}$.
12. $a^*\emptyset = \emptyset$ Since there are no elements in the empty set, the concatenation of it with anything is vacuously empty.

There are three perspectives on how to understand the correspondence between a regular expression and the language it describes. An automata may take input and provide output, but that is not the case here. The first perspective of a regular expression is that it is simply the name of the correct strings in the language. It describes the strings.

The second perspective is that the regular expression itself is some kind of string nondeterministic string, a string in "super position". After certain non-deterministic choices are made, it is casted down to being some deterministic string. It can only become strings in the language it describes. For example, we argue that $(a \cup b)^*$ produces all strings over Σ^* if $\Sigma = \{a, b\}$. Certainly every regular expression describes a set of strings so $L((a \cup b)^*) \subseteq \Sigma^*$ is obvious. We prove $\Sigma^* \subseteq (a \cup b)^*$. Each string in Σ^* has some length, Nondeterministically guess this length, say n. Thus

$$(a \cup b)^* \to (a \cup b)^n = (a \cup b)(a \cup b)...(a \cup b)$$

You now have n nondeterministic choices to make. Each one determines a letter of our string. This can allow us to construct every string of length n.

Regular Expressions Describe All Regular Languages

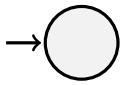
As you may suspect by their name, the languages that the regular expressions describe, are exactly and only the regular languages. Let $\mathscr{L}(REX)$ be the class of languages such that $L \in \mathscr{L}(REX)$ implies that there is a regular expression to describe L. We prove that $L\mathscr{L}(REX)$ if and only if L is regular, however we have a choice to make. We have proven that NFAs are equivalent in power to DFAs, so in proving this equivalence, we could choose to simulate with NFAs or DFAs. We shall choose NFAs. The nondeterminism of NFAs naturally and neatly simulates the nondeterminism of regular expressions as we shall see. We prove via a double set containment that $\mathscr{L}(REX) = \mathscr{L}(NFA)$.

Theorem 1.6. $\mathscr{L}(REX) \subseteq \mathscr{L}(NFA)$

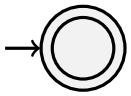
First we show that if a language is produced by a regular expression, then it is decided by an NFA. To prove that $\mathscr{L}(REX) \subseteq \mathscr{L}(NFA)$, we want to show that for each regular expression, there exists an equivalent NFA. Given that regular expressions are recursively defined, it is natural to choose to proceed by induction.

Proof. We proceed by structural induction upon the depth of the number of operations applied to form a regular expression. We first prove our base cases. Let the number of operations applied be zero, then the only possible regular expressions are themselves the base cases. We give NFAs for each of them.

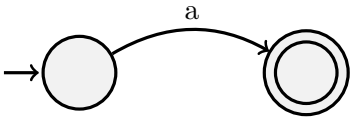
1. $R = \emptyset$



2. $R = \varepsilon$.

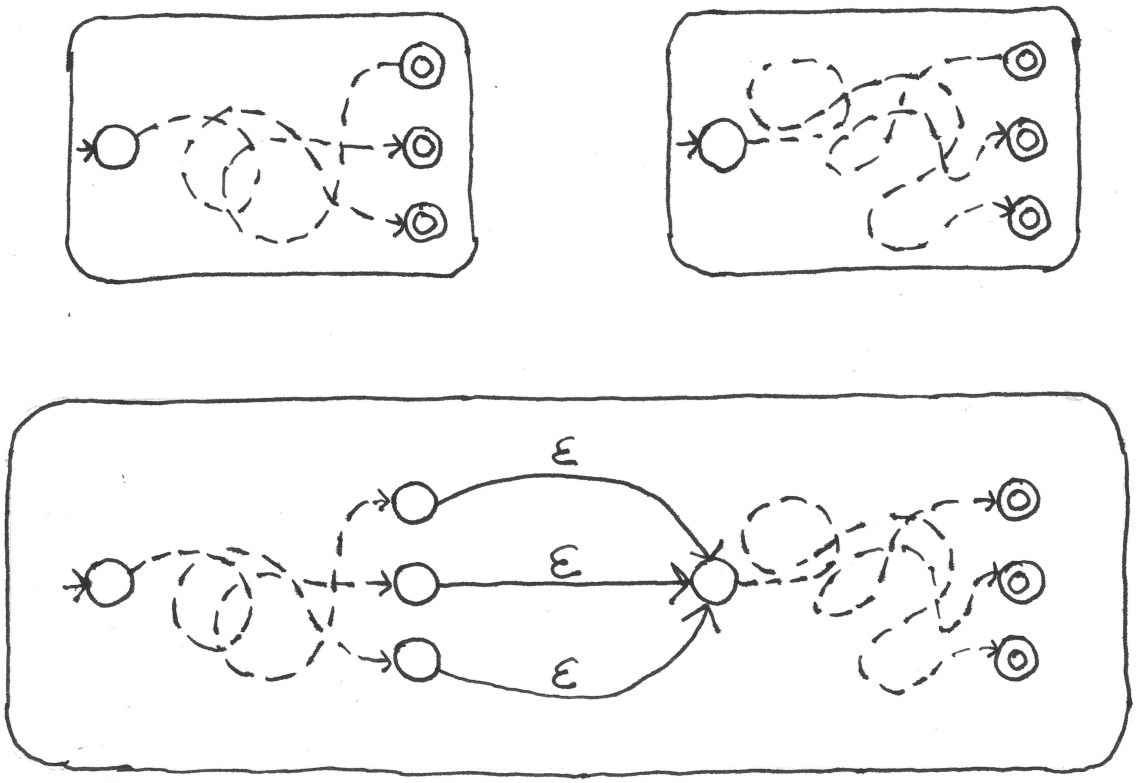


3. $R = a \in \Sigma$.



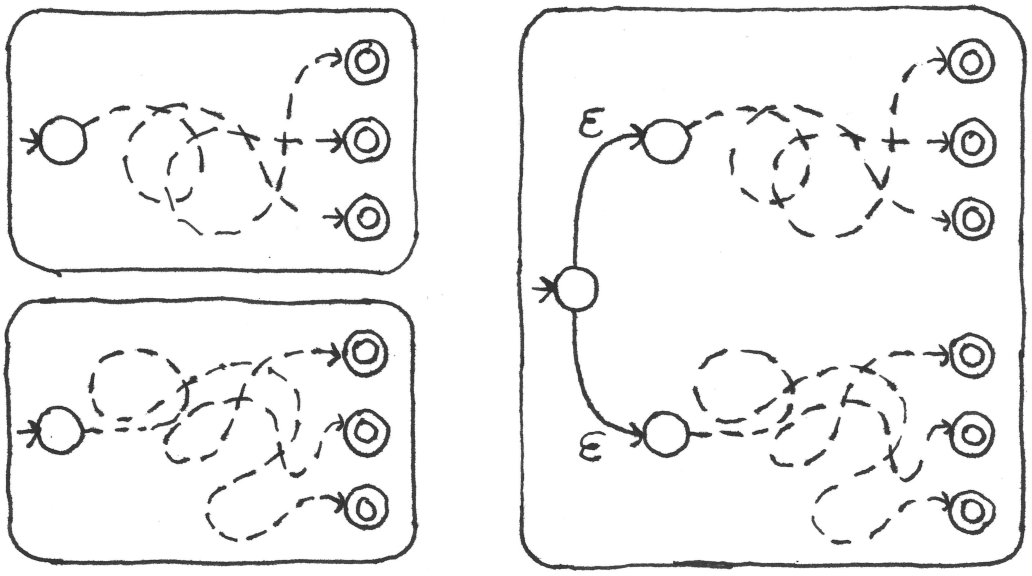
Next, we proceed with our inductive steps. Let R_i, R_j be regular expressions that decide regular languages, by structural induction, we assume that there exist NFAs N_i, N_j such that $L(N_i) = L(R_i)$ and $L(R_j) = L(N_j)$. We will prove R_i^*, R_iR_j, and $R_i \cup R_j$ also have NFAs to decide them. The proofs can be done graphically.

1. $R = R_iR_j$.



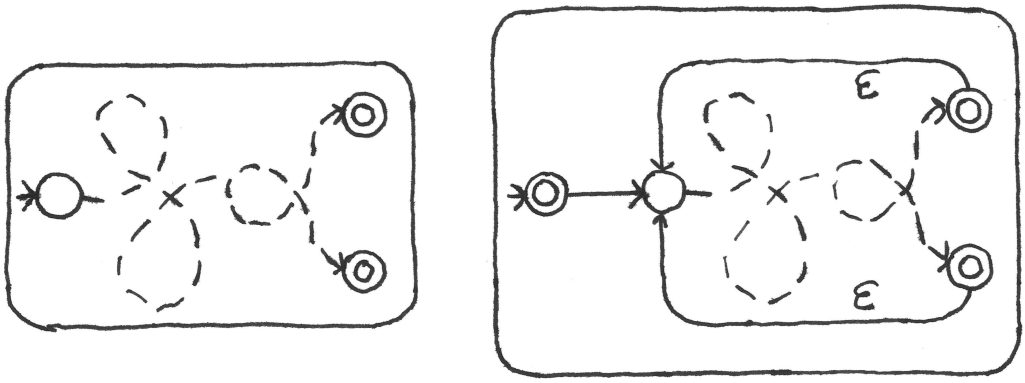
We remove final states F_i and $\forall f \in F_i$, add $\delta(f, \varepsilon) = q_j$ where q_j is the initial state of N_j. Consider a computation like a path through the NFA as a graph. To reach an accept state, you must go through the first NFA, then the second.

2. $R_i \cup R_j$.



Let q_i, q_j be the start states of N_i, N_j respectively. Add new start state q and $\delta(q, \varepsilon) = \{q_i, q_j\}$. Here you nondeterministically choose which NFA you wish to proceed on, so it decides the languages which reach the accepting states of either NFA, representing the union.

3. $R = R_i^*$.



We add new start state q', ε-transition from q' and all states of F to the old start state q, mark q' as accepting. Note we could not have just made the start state accepting, but must add a new state. You can traverse an arbirary number of times on the internal NFA so this corresponds to zero or more copies, which is the Kleene star operation.

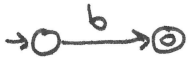
□

This proof not only shows every regular expression decides a regular language, but it gives a process to convert a regular expression into an NFA.

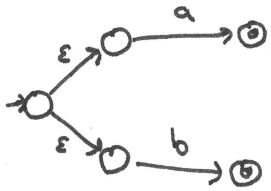
Example 1.12. Consider the following example for $a(a \cup b)^*a \cup b(a \cup b)^*b \cup a \cup b$. This is the regular expression for strings that begin and end with the same symbol.

a

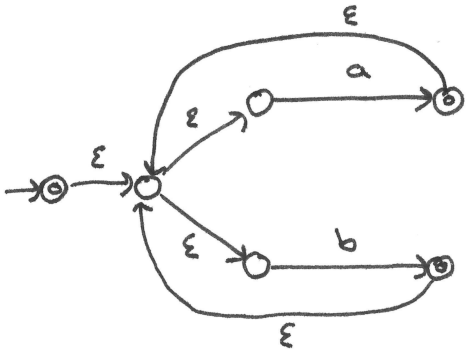
b



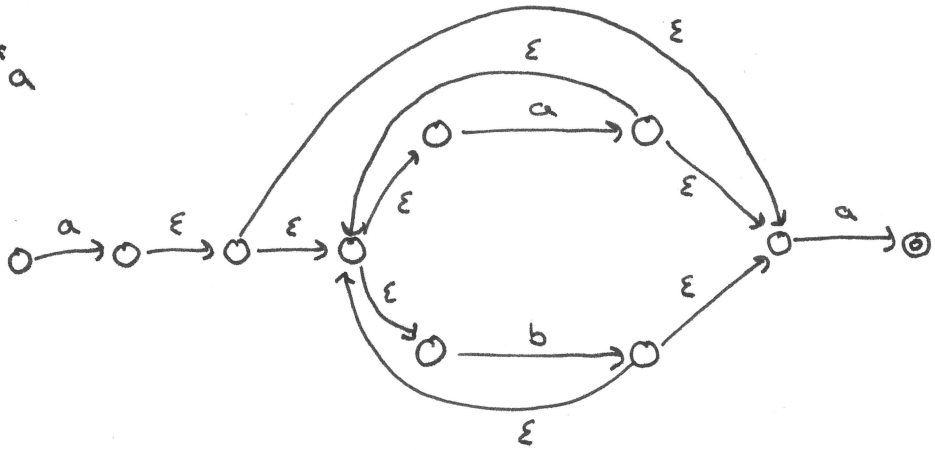
a ∪ b



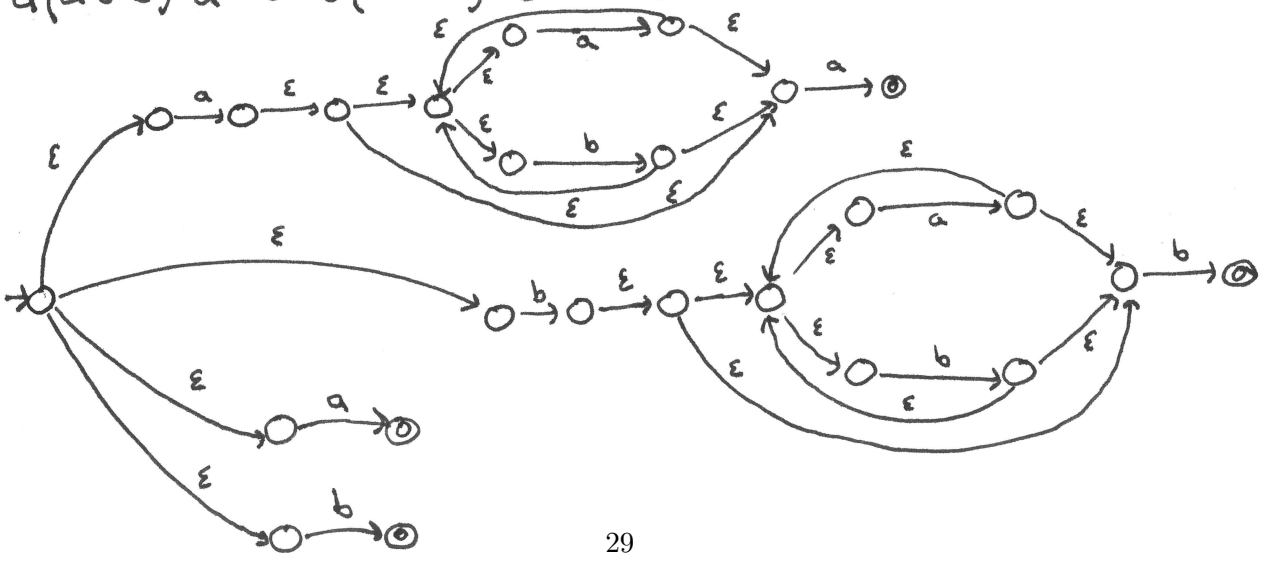
(a ∪ b)*



a(a ∪ b)*a



a(a ∪ b)*a ∪ b(a ∪ b)*b ∪ a ∪ b



There is in fact a third perspective on regular expressions, they are a short hand notation for the NFA, which are described by this process.

Theorem 1.7. $\mathscr{L}(NFA) \subseteq \mathscr{L}(REX)$

Proof. Let $L \in \mathscr{L}(NFA)$. Then there exists an NFA to decide L. We construct a regular expression to describe the strings which may traverse this NFA from the start state to an accept state. These are of course, exactly the strings that the NFA accepts. To do so, we generalize the definition of an NFA as follows.

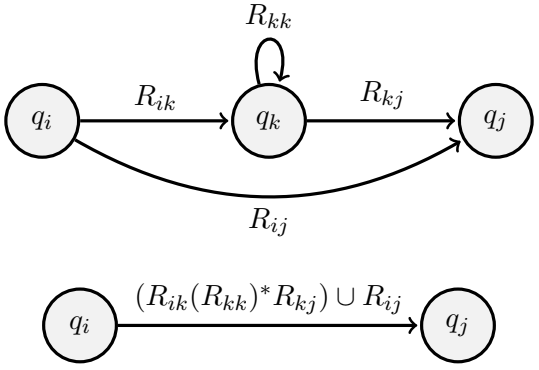
Definition 1.7. A **GNFA** is defined as an NFA with the following generalizations:

- Rather than an element of $\Sigma \cup \{\varepsilon\}$, transitions have an entire regular expression on them.
- The start state has no incoming transitions.
- The final state has no outgoing transitions.
- Every pair of states has a transition.

Taking a transition in a DFA is reading some single symbol off the front of the input. Taking a transition of a GNFA is nondeterministically choosing some prefix of the input which satisfies the regex on the transition. Note that every NFA can be immediately made into a GNFA by making the following changes:

- Add a new start state with an ε-transition to the old start state.
- Add one new final state, with ε-transitions from the old final states.
- For every pair of states with no transition between them, add a transition with the regular expression $\emptyset$.

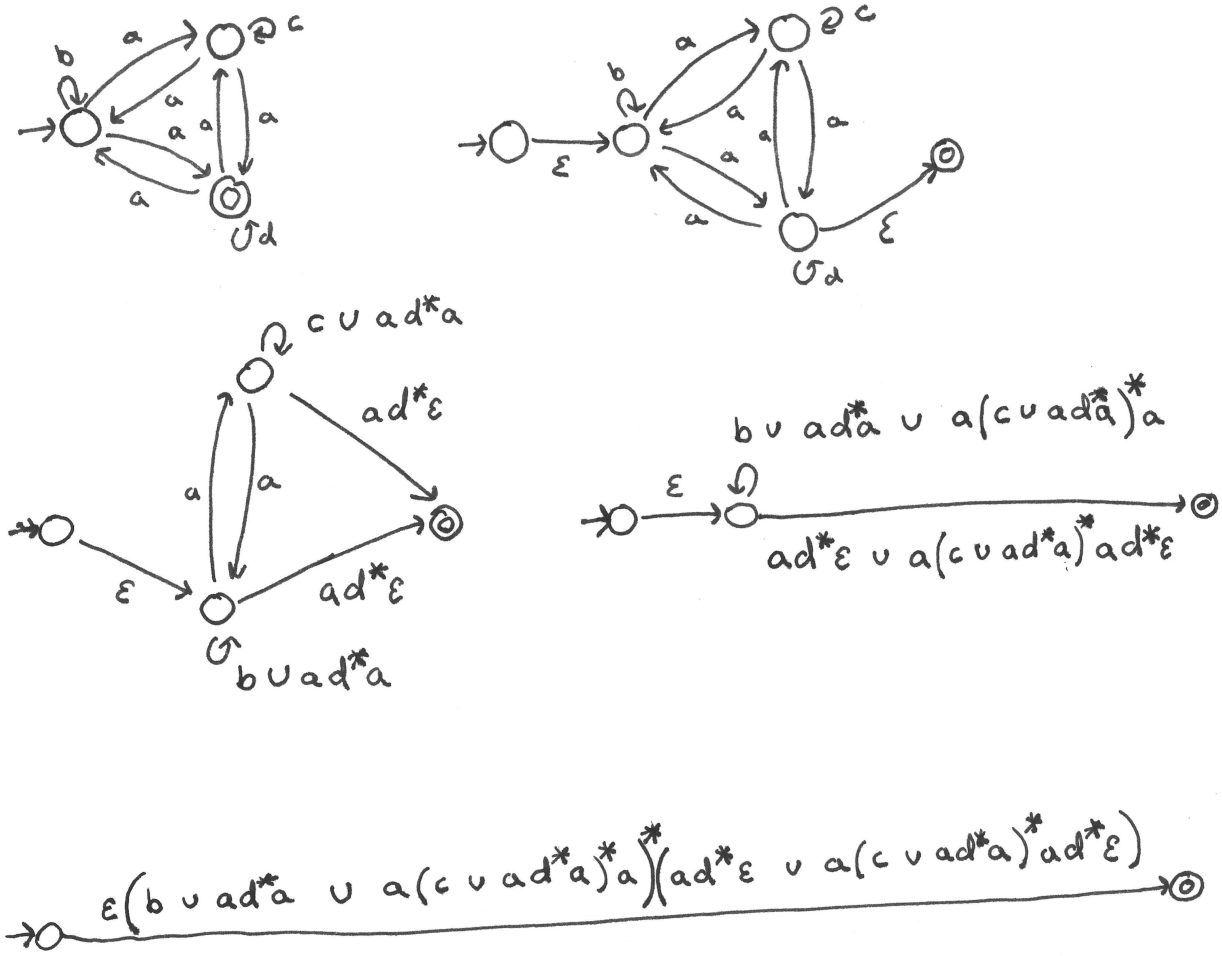
Clearly these modifications do not change the language that is decided. First we convert our NFA to this GNFA, then we convert this GNFA to our regular expression, proving, that every NFA has an equivalent regular expression. In our GNFA, we proceed to eliminate states, one at a time until two states and one transition remain. Our goal is to decrease the number of states, but increase the complexity of the transitions. The last transition left will be our desired regular expression. We eliminate states as follows. Suppose there are states q_i, q_k, q_k, and we wish to eliminate state q_k.



We update all paths that could use the state q_k with regular expresssions that describe this path, but without the state itself. Repeated application of this process will result in a GNFA with two states and one transition. The regular expression on this transition exactly describes the strings that would compute on the NFA from the start state to an accept state, thus it describes exactly the language decided by our original NFA. Since this can be done for any NFA, we see that each regular language can be described by a regular expression.

□

Example 1.13. Let us do an example. We convert an NFA with three states to a GNFA with five states to a regular expression.



The regular expression is quite long, and far from minimal. The process only guarantees to output a correct regular expression, not a nice one. An NFA may be organized as a two-dimensional state diagram, but a regular expression is one-dimensional. If it contains the same information, it may necessarily be longer.

1.4 Nonregular Languages

Not all languages are regular. Although DFAs and NFAs seem quite powerful, there are some immediate limitations. Our current goal is to prove that nonregular languages exist.

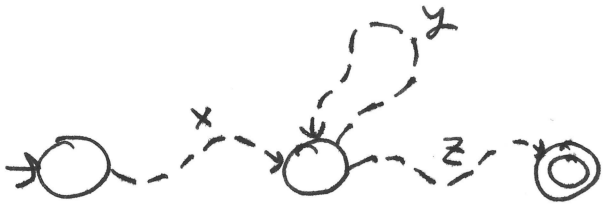
Consider the language $\{a^n b^n \mid n \in \mathbb{N}\}$. A DFA has a finite amount of states, and is only able to read the string left to right. It cannot read symbols it has previously. As it reads left to right, it somehow is tasked with memorizing an arbitrarily large amount of information, the number of a's, in order to match them to the number of b's. Note how this is a very different computation task than deciding $(ab)^*$, or a^*b^*, which can be computed using only a finite number of states. A DFA of say, 20 states may correctly decide if a string has the form $a^{20}b^{20}$, but this DFA must fail on a string of a large enough size, say $a^{100}b^{100}$. We can formally prove these languages to be nonregular by use of the pumping lemma. It follows from something relatively simple, the pigeonhole principle.

Definition 1.8 (Pigeonhole Principle)**.** If m pigeons are assigned to n holes with $m > n$ then some hole must have more than one pigeon.

If a DFA accepts an infinite language, then there are strings of arbitrarily long length that it must accept, yet it must do so only with finitely many states. The strings the DFA is tasked to accept are much much longer than the size of the DFA itself. The pigeonhole principle will apply to the computation on DFAs, which will show that regular languages have an interesting property.

Note each letter of our word takes not one state, but one transition. If we have p states, and compute on a word of length $\geq p$, then some state is visited twice in our computation path. We may not know where this loop is or how long it is, but we know that it must exist by the pigeonhole principle. The DFA is such a simple stupid device, if it accepts a long enough string, it must also accept if you were to take that string and repeat a substring of it arbitrarily many times. The Pumping Lemma is a formalization of this intuitive idea.

If w is long enough, some state in the computation is repeated. Then w can be partitioned into $w = xyz$ where y denotes a loop from some state back to itself. We could take this loop an arbitrary number of times. If the DFA accepts xyz, it must also accept xz and $xyyz$.



The Pumping Lemma

Theorem 1.8 (The Pumping Lemma)**.** Let L be an infinite language. If L is regular, then there exists a number p such that for each $w \in L$ with $|w| \geq p$, there exists a partition of $w = xyz$ where:

- $|xy| \leq p$
- $|y| > 0$
- $\forall i \in \mathbb{N}(xy^i z \in L)$

Proof. We prove that if L is an infinite regular language, then it can be pumped. If L is regular, then there exists a DFA $D = (Q, \Sigma, q_0, \delta, F)$ to decide L. Let $|Q| = p$ and let $w \in L$ be any word with length $|w| = n \geq p$ with $w = w_1 w_2 ... w_n$. Consider the sequence of states visited during the computation of D on w, and let these states be enumerated as $s_1, s_2, ..., s_{n+1}$ with $\delta(s_i, w_i) = s_{i+1}$ for $1 \leq i \leq n$. Note that a word of length n takes n transitions, but visits $n + 1$ states, and s_1 is

necessarily q_0. In the first p transitions, the $p+1$ states $s_1, ..., s_{p+1}$ are visited. By the pigeonhole principle, in $s_1, ..., s_{p+1}$, there must exist a state which has been visited twice. Let the first of these visits be s_i and the second of these visits be s_j. Since they are distinct, $i \neq j$. Consider the partition of $w = xyz$ into $x = w_1...w_{i-1}, y = w_i...w_{j-1}, z = w_j...w_n$. We demonstrate these choices of x, y, z satisfy our three conditions which require L to be pumpable.

- By the pigeonhole principle, we know a repetition must occur in the first $p+1$ visited states, so $j \leq p+1$ and since $|xy| = j - 1$ then $|xy| \leq p$.
- Since $i \neq j$, we know y is never the empty string, thus $|y| > 0$.
- The string x will take our DFA from q_0 to our repeated state q, y will take D from q back to q, and z will take D from q to an accept state. Upon reaching our repeated state q for the second time, instead of continuing on with z, we may just repeat the computation of y an arbitrary number of times before continuing on with z. Thus for each i, xy^iz will take D from q_0 to q_a. What is a path from the start state to an accept state if not exactly an accepted word?

□

You should think of i here as the number of times you may traverse the loop. Traversing it one time is the original string xyz. You may also traverse it zero times, so $xyz \in L \implies xz \in L$. You may traverse it twice, so $xyz \in L \implies xyyz \in L$, and so on. The term "pumping" refers to this property. A long enough string may have a substring of it "pumped" into it an arbitrary number of times. If $i = 0$, we refer to computing xz as "pumping down".

The pumping lemma is not itself useful to proving that a language is regular. Instead, we take its contrapositive: If an infinite language cannot be pumped, then it is not regular. Note that this is not an if and only if and there do exist some nonregular languages which can be pumped.

The pumping lemma has many moving pieces and can be tricky to apply. There are alternating existential and universal quantifications through out[2]. Of the proof techniques available to you, it certainly is the most cumbersome, and is surprisingly common to make a mistake on what you can or cannot choose. I suggest you use this proof template. Suppose that L is the language we want to prove is not regular.

1. Assume to the contrary, L is regular with pumping length p.
2. Choose some $w \in L$ such that $|w| \geq p$.
3. For all cases $w = xyz$ such that $|xy| \leq p$ and $|y| > 0$.
4. Choose any $i \neq 1$ and demonstrate that $xy^iz \notin L$.
5. Conclude that L cannot be pumped, and therefore L is not regular.

Let us go through the importance of each step.

- First, by assuming to the contrary that L is regular with pumping length p, we are supposing that there exists a DFA of p states. When we reach a contradiction, then no such DFA of p states can exist. Since p is general, this means that no such DFA can exist at all. We cannot fix p. If we did a pumping lemma proof with $p = 5$, this would conclude that there is no DFA of five states. It does not imply there is no DFA at all, as there may exist a DFA with more than five states to decide the language.

[2] $\exists p \forall w \exists x, y, z \forall i(...)$, thats four alternating quantifiers!

- We choose to pump some string in the language. By choosing $w \in L$, we know w brings our assumed DFA to an accept state, like a path in a graph. By requiring $|w| \geq p$, we meet the criterion required by the pigeonhole principle, and can guarantee the computation visits some state twice. It is not uncommon for us to choose strings with length much much larger than p. The only requirement is that its length is greater than or equal to. Choosing a good w will effect the proof greatly. A poor choice of w may make the proof very long, or even impossible.

- In the computation of w on our assumed to the contrary to exist DFA, we are guaranteed that there exists a loop somewhere by chosing $|w| \geq p$, but we don't know where. So we have to consider all possible cases of where this loop could be. We model this as considering all ways to partition w into the three parts $w = xyz$ subject to our two conditions on each case. Firstly that $|xy| \leq p$. This ensures that the occurance of a repeated state occurs somewhere before the end of what we denote as y. The second condition $|y| > 0$ ensures that this cycle is actually occuring. Note that $|y| = 0$ trivially ensures we could never reach a contradiction, since $\varepsilon^i = \varepsilon$ for any i.

- For each case, you only need to choose any $i \neq 1$ so that $xy^iz \in L$. Most of the time $i = 2$ works, we will show examples where it does not.

- Since we took a long enough string in the language, showed it was impossible to pump, then there cannot exist a DFA to decide L, and we must conclude that L must not be regular.

Let us proceed with some examples.

Example 1.14. $L = \{0^n1^n \mid n \in \mathbb{N}\}$ is not regular.

Proof. Assume to the contrary, L is regular with pumping length p. Let $w = 0^p1^p$ and notice that $w \in L$ and $|w| = 2p \geq p$. There is only one case since the first p characters in the string are all zeroes. Let $x = 0^a,\ y = 0^b,\ z = 0^{p-a-b}1^p$ subject to $|xy| = a + b \leq p$ and $|y| = b > 0$. Consider $i = 2$. Then

$$xy^iz = xy^2z = xyyz = 0^a0^b0^b0^{p-a-b}1^p = 0^{p+b}1^p$$

We know that $b > 0$, so the number of 0s does not equal the number of 1s since $p + b > p$. Thus, L cannot be pumped, and as a result, is not regular. □

The language 0^n1^n is the canonical example of a non-regular language. We choose w so that its length is a function of p so that $|w| \geq p$ is obvious. By choosing a good w, we can ensure that we reduce the number of cases required. The number of cases is technically a function of p, the number of ways $a + b \leq p$ subject to those conditions. We group these all into one case as the contradiction is identical. Note that then the substrings x, y, z also end up being a function of p. We only need to to show that one $i \neq 1$ gives a contradiction, so we choose a smallest and simplest one, that $i = 2$.

Example 1.15. $L = \{xx^R \mid x \in \Sigma^*\}$ is not regular.

Recall that x^R denotes the reveral of the string x. This language, xx^R then consists of the even length palindromes. For demonstration, we choose a worse w on purpose.

Proof. Assume to the contrary, L is regular with pumping length p. Let $w = 0^{p-1}110^{p-1}$. Confirm that $w \in L$ and $|w| = 2p \geq p$. The first p characters in the string are different, meaning there are several cases:

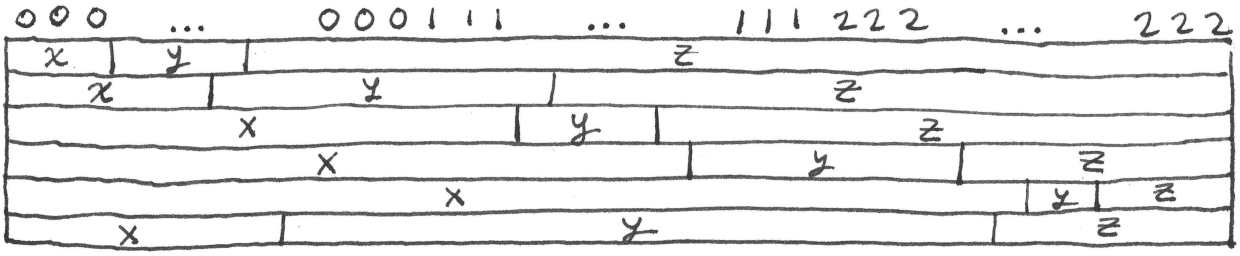
Case 1, y contains no 1s. Then let $x = 0^a,\ y = 0^b,\ z = 0^{p-1-a-b}110^{p-1}$ subject to $|xy| = a + b \leq p$ and $|y| = b > 0$. Choose $i = 2$. Then $xy^2z = xyyz = 0^a0^b0^b0^{p-1-a-b}110^{p-1} = 0^{p-1+b}110^{p-1}$. Since $b > 0$, we know that $p - 1 + b \neq p - 1$. Therefore, the two sections of 0s are unequal. If b makes $xyyz$ of odd length then we are done, so suppose $xyyz$ is of even length. If we were to split the string in half, the first half contains no 1s, and the second half contains two 1s, implying that this is not a palindrome.

Case 2, y contains a single 1. Then let $x = 0^a,\ y = 0^{p-1-a}1,\ z = 10^{p-1}$ subject to $|xy| = p \leq p$ and $|y| = p - 1 - a + 1 > 0$. Let us pump down. Consider $i = 0$. Then $xy^0z = xz = 0^a10^{p-1}$ Since there is only a single 1, this is never an even-length palindrome.

For both cases, the language could not be pumped. Therefore, L is not regular. □

Lets annotate this proof as well. We chose a poor w on purpose, resulting in more cases. There were two cases, whether or not xy contained a 1 or not. Had we increased the string length so that the initial block of 0's exceeded p, we would only have one case. For the second case, we chose $i = 0$. We call this "pumping down".

A very poor choice of w would have been $w = 0^p0^p$. Note that this is a simple even length palindrome, but it is too simple. This specific w can be easily pumped when y is of even length, and will not result in a contradiction. You want to choose w so that it is barely in the language, at the extremal conditions. Any small peturbation results in it no longer being in the language. Some choice of w do not allow you to complete the proof. Other choices of w incur a lengthy proof of many cases. Consider the language $\{0^n1^n2^n \mid n \in \mathbb{N}\}$. It is not regular for similar reasons as to $\{0^n1^n \mid \mathbb{N}\}$, and the proof is nearly identical. A good choice of string to pump would be $w = 0^p1^p2^p$. As a bad choice, consider $w = 0^{\lceil p/3 \rceil}1^{\lceil p/3 \rceil}2^{\lceil p/3 \rceil}$. The restriction of $|xy| \leq p$ has nearly no effect here, and a correct proof would involve six distinct cases.



Let us do another example with a good choice of w.

Example 1.16. $L = \{xx \mid x \in \Sigma^*\}$ is not regular.

This language consists of words which are themselves concatenated twice. It is not $\Sigma^*\Sigma^*$, but it contains strings like $abab, abaaba, aabbaabb, \varepsilon$ and so on.

Proof. Assume to the contrary, L is regular with pumping length p. Let $w = 0^p10^p1$ and notice that $w \in L$ and $|w| = 2p + 2 \geq p$. There is only 1 case since the first p characters in the string are all 0s, so let $x = 0^a,\ y = 0^b,\ z = 0^{p-a-b}10^p1$ subject to $|xy| = a + b \leq p$ and $|y| = b > 0$. Consider $i = 2$ so

$$xy^iz = xy^2z = xyyz = 0^a0^b0^b0^{p-a-b}10^p1 = 0^{p+b}10^p1$$

If $xyyz$ is of odd length we are done, so suppose it is of even length. Let $xy^2z = w_1w_2$ with $|u| = |v|$. Notice that u must end with a 0, but v must end with a 1, therefore, $u \neq v$ and $xyyz \notin L$. Thus, L cannot be pumped and is not regular. □

Lets do some unary examples.

Example 1.17. $L = \{1^{n^2} \mid n \in \mathbb{N}\}$ is not regular.

Proof. Assume to the contrary, L is regular with pumping length p. Let $w = 1^{p^2}$ and observe that $w \in L$ and $|w| = p^2 \geq p$ There is only 1 case since the first p characters in the string are all 1s. Let $x = 1^a,\ y = 1^b,\ z = 1^{p^2-a-b}$ subject to $|xy| = a + b \leq p$ and $|y| = b > 0$. Consider $i = 2$.

$$xy^2z = xyyz = 1^a1^b1^b1^{p^2-a-b} = 1^{p^2+b}$$

Since $b > 0$, $p^2 < p^2 + b$, thus $|1^{p^2}| < |1^{p^2+b}|$. Since $a + b \leq p$, $b \leq p$, thus

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

Together, we see that

$$|1^{p^2}| < |1^{p^2+b}| < |1^{(p+1)^2}|$$

Our pumped string xy^2z has length strictly between two consecutive perfect squares. Therefore, its length is not some perfect square is not an element of L. Thus, L cannot be pumped and is not regular. □

Example 1.18. $L = \{1^q \mid q \text{ is prime}\}$ is not regular.

Proof. Assume to the contrary L is regular with pumping length p. Let $w = 1^q$ where q is the next largest prime greater than p. By this definition, $w \in L$ and $|w| = q > p$. There is only 1 case since the first p characters in the string are all 1s. Let $x = 1^a,\ y = 1^b,\ z = 1^{q-a-b}$ subject to $|xy| = a + b \leq p$ and $|y| = b > 0$. Consider at $i = q + 1$. Then

$$xy^{q+1}z = 1^a1^{b(q+1)}1^{q-a-b} = 1^{q+qb} = 1^{q(1+b)}$$

Since $b > 0$, the length $q(1 + b)$ has a prime divisor which is not one or itself, and thus it is composite, not prime. We see that $xy^{q+1}z \notin L$ and thus, L cannot be regular. □

For this example, how did we know to choose $i = q + 1$? We worked it out before hand, solving for i such that $b(i - 1) + q$ would be composite, leading to a contradiction. Each pumping lemma proof should be done twice. Once to know the structure of the proof, and the second time formally.

Example 1.19. $L = \{0^n1^m \mid n \neq m\}$ is not regular.

Prior strategies will not work here, and we will need to be creative. We somehow need to pump to get the number of zeros and ones to be exactly equal.

Proof. Assume to the contrary L is regular with pumping length p. Consider $w = 0^p1^{p+p!}$. Observe that $w \in L$ and $|w| = p + p! > p$. We have one case, so let $x = 0^a, y = 0^b, z = 0^{p-a-b}1^{p+p!}$ subject to $a + b \leq p$ and $b > 0$. Consider $i = p!/b + 1$. Since $a + b \leq p$, we know $b \leq p$, and this implies that i is always a natural number. Then

$$xy^iz = xy^{p!/b+1}z = 0^a0^{p!/b+1}0^{p-a-b}1^{p+p!} = 0^{a+b(p!/b+1)+p-a-b}1^{p+p!} = 0^{p+p!+b-b}1^{p+p!} = 0^{p+p!}1^{p+p!}$$

which is clearly not in L, and therefore, L is not regular. □

Arguments via Closure

The pumping lemma is not the only way to prove a language is not regular. You may apply closure properties of the regular languages.

Definition 1.9 (Dyck Language)**.** Let the *Dyck language* be over the alphabet $\Sigma = \{(,)\}$ which consists of strings of valid, balanced parenthesis.

Some strings in this language include $(), ()(), ((())()), \varepsilon, ((()())())(())()$, and some strings not in this language include $((), ()(,)(, (((((.$

Theorem 1.9. The Dyck language is not regular.

There is a proof via the pumping lemma, but we shall prove it with closure instead.

Proof. Assume to the contrary the Dyck language D was regular. Since the regular languages are closed under intersection, then consider $D \cap (^*)^*$. The left side enforces that the number of opens equals the number of closes, and the right hand side enforces that all the opens come before all the closes. The intersection contains all strings which satisfy both of these properties and therefore is equal to $\{(^n)^n \mid n \in \mathbb{N}\}$, our canonical non-regular language, a contradiction. Therefore, the Dyck language is not regular. $\square$

Context-Free Languages

2.1 Context-Free Grammars

A grammar a computational model that we have not yet seen. An automata is tasked with decision. It takes on input and produces output. The set of strings it decides is exactly the ones it accepts. A regular expression is tasked with description, in that it is a description of a set of strings. In contrast to these, a grammar is tasked with *production* or *generation*. It is a defined set of rules which are capable producing a string according to those rules. It takes no input. Defined with the rules we give it, it will produce a string according to those rules.

Definition 2.1 (Context-Free Grammar)**.** A context-free grammar is a four tuple (V, Σ, R, S) such that:

- V is a finite non-empty set of *non-terminals* or *variables*. These are usually represented by capital letters such as$\{S, A, B, X, Y\}$.
- Σ is a finite non-empty set of *terminals* or our alphabet. These are usually represented by lower case letters such as $\{a, b\}$.
- R is a finite set of *productions* or *rules*. Each production is of the form $V \to (V \cup \Sigma)^*$. The left-hand side of the production will always be a single non-terminal and the right-hand side will be a string of (possibly zero) terminals and non-terminals.
- $S \in V$ is our designated start non-terminal.

We define a working string as an element of $(V \cup \Sigma)^* V (V \cup \Sigma)^*$, a string over $V \cup \Sigma$ which contains atleast one non-terminal. If we have a working string of the form $\alpha A \beta$ with $A \in V$ and $\alpha, \beta \in (V \cup \Sigma)^*$ and there exists a production of the form $A \to w$, we may apply a production to transform this working string into another as

$$\alpha A \beta \implies \alpha w \beta$$

Here, we write $\alpha A \beta \Rightarrow \alpha w \beta$ to indicate the application of a single production, and say $\alpha A \beta$ *yields* $\alpha w \beta$. We write $u \overset{*}{\Rightarrow} v$ to indicate the application of zero or more productions. We say a string w is *produced* by a grammar if there exists a sequence of working strings $w_1, ..., w_k$ such that

$$S \implies w_1 \implies w_2 \implies ... \implies w_k \implies w$$

We define the language produced by a context-free grammar G, as the set of strings it can produce $L(G) = \{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\}$. Note that since w contains no non-terminals, no more productions can be applied. A language is context-free if there exists a context-free grammar to produce it.

Note that a CFG takes no input, and only produces exactly and only the correct strings. This is why we say a grammar produces a string, but an automata decides or accepts a string.

Examples

Like a state diagram for an automata, you can specify all parts of the CFG by just giving the set of productions. It implicitly gives the terminals and non-terminals, and we always denote S as the start non-terminal, by convention.

Example 2.1. We give a CFG to produce $\{a^n b^n \mid n \in \mathbb{N}\}$.

$$S \to aSb$$
$$S \to \varepsilon$$

If we have two or more productions with the same beginning non-terminal, we may use "|" as a shorthand to represent these two productions more concisely.

$$S \to aSb \mid \varepsilon$$

This grammar still has two distinct productions. Most formally, we would specify this grammar as

$$G = (\{S\}, \{a, b\}, \{(S, aSb), (S, \varepsilon)\}, S)$$

but this quickly becomes cumbersome. Simply giving the production is sufficient. Let us say we want to produce *aaabbb*. The sequence of productions we follow is

$$S \implies aSb \implies a(aSb)b \implies aaSbb \implies aaaSbbb \implies aaabbb$$

We repeatedly apply the first production. Then applying the second production once will terminate our computation, as we no longer have any non-terminals. Notice that this CFG has to produce all strings of the form $a^n b^n$. Also observe that CFGs are non-deterministic, in that you have can have a several productions to choose from, each perhaps a different computation. With S in our working string, there are two productions you may apply, Between the productions $S \to aSb$ and $S \to \varepsilon$, one is non-deterministically chosen. This corresponds to how many a's and b's you wish to produce.

This was also our canonical example of a non-regular language. This should convince you atleast, that the class languages produced by context-free grammars, $\mathscr{L}(CFG)$, is not equal to the regular languages. Later we will show it is a strict super set.

Example 2.2. We give a CFG for $\{ww^R \mid w \in \Sigma^*\}$

$$S \to aSa \mid bSb \mid \varepsilon$$

This generates even length palindromes. As we apply productions, the same symbol is produced on the right and on the left of S. We can reuse the same idea to generate palindrome of odd length.

The productions $S \to a \mid b$ determine our middle character.

Example 2.3. We give a CFG for $\{w\Sigma w^R \mid w \in \Sigma^*\}$

$$S \to aSa \mid bSb \mid a \mid b$$

Example 2.4. There are many equivalent context-free grammars for Σ^*. We could produce the letters one at a time.

$$S \to aS \mid bS \mid \varepsilon$$

We could also pair up letters and produce them two at at time.

$$S \to aaS \mid abS \mid baS \mid bbS \mid a \mid b \mid \varepsilon$$

We need not produce them left to right either. We could produce them outside in.

$$S \to aSa \mid bSb \mid aSb \mid bSa \mid a \mid b \mid \varepsilon$$

Example 2.5. A CFG for 1^* could simply be

$$S \to 1S \mid \varepsilon$$

Example 2.6. The empty set $\emptyset$ is context-free. If a grammar produces no strings, not even ε, it is either trivial, or some how cannot eliminate non-terminals. A trivial one could be

$$S \to X$$
$$X \to S$$

A more complicated but equivalent one could also be

$$S \to 1S$$

Working strings of this CFG will increase in length, yet no string is produced since there is no $S \to \varepsilon$ for termination.

$$S \implies 1S \implies 11S \implies 111S \implies 1111S \implies 11111S \implies ...$$

Example 2.7. Recall our definition of the the Dyck language, which contains strings over $\Sigma = \{(,)\}$ of valid balanced parantheses. This language is context-free, and can be produced by the grammar G.

$$S \to (S) \mid SS \mid \varepsilon$$

Unlike our previous examples, the correctness of this grammar is not obvious. Why does it produce all and only balanced strings? Lets prove it.

Theorem 2.1. The grammar G defined by the productions $S \to (S) \mid SS \mid \varepsilon$ produces exactly and only valid balanced strings of parantheses.

Proof. Let D be the Dyck language, and G be the grammar as defined previously. We prove $L(G) = D$ by a double set containment.

First we prove $L(G) \subseteq D$. Let $S \overset{*}{\Rightarrow} w$. We wish to prove that w is balanced. We do so by induction on the number of productions it takes to produce w. Our base case is a single production. The only productions that have no non-terminals on the right-hand side is the $S \to \varepsilon$ rule, and we agree that ε is balanced. Now assume that if $S \overset{*}{\Rightarrow} u$ in k productions then u is balanced. Suppose $S \overset{*}{\Rightarrow} w$ in $k+1$ productions. Consider the first possible production. Since $|w| = k+1$, it could have only been one of the two productions $S \to SS$ or $S \to (S)$.

- In the first case, if the first production was of the form $S \to SS$, then our sequence of working strings appears as
$$S \implies SS \implies ... \implies w$$
Therefore, there exists u, v such that $w = uv$ and $S \overset{*}{\Rightarrow} u$ and $S \overset{*}{\Rightarrow} v$. Further, production of both u, v take $\leq k$ production steps. By our induction hypothesis, they are balanced. Since the concatenation of balanced strings is balanced, we see that so must be $uv = w$.

- In the second case, our first production was of the form $S \to (S)$, then our sequence of working strings appears as
$$S \implies (S) \implies ... \implies w$$
Therefore, there exists u such that $w = (u)$ and $S \overset{*}{\Rightarrow} u$. Since production of u takes $\leq k$ derivations, by our induction hypothesis, u is balanced. Therefore, so is $(u) = w$.

Next we prove $D \subseteq L(G)$. Let $w \in D$ be a balanced string. We prove that $S \overset{*}{\Rightarrow} w$ by induction on the length of w. Our base case is $|w| = 0$ and indeed $S \Rightarrow \varepsilon$. Assume that if $w \in D$ and $|u| < k+1$ then $S \overset{*}{\Rightarrow} u$. Consider $w \in D$ with $|w| = k+1$. We may assume that $|w| \geq 2$. So there exists a distinct first and last symbol. In fact, the first symbol of every balanced string is always an open, and the last symbol is always a close. So there exists a string u with length $k-1$ such that $w = (u)$. We have two cases.

- Our first case is if u is balanced. Since $|u| = k-1$, then by our induction hypothesis, $S \overset{*}{\Rightarrow} u$. Then $S \overset{*}{\Rightarrow} w$ using the $S \to (S)$ production first as

$$S \Rightarrow (S) \Rightarrow ... \Rightarrow (u)$$

- In the second case, we consider if u is not balanced. The first open and last close, do not match to each other. Since they must match to something, there exists balanced x, y such that $w = xy$. Then there exists a sequence of productions of w as

$$S \implies SS \Rightarrow ... \Rightarrow xy$$

We may conclude this grammar produces exactly and only balanced strings. □

Example 2.8. Many programming languages have their syntax specified via a CFG. We give a CFG for the well formed formulas of the propositional calculus. A proposition has a formal definition as follows. A term is either a propositional variable or a constant symbol t or f. All terms are propositions. If f_1, f_2 are propositions, then so are $(f_1 \vee f_2)$, $(f_1 \wedge f_2)$ and $\neg f_1$. Let our alphabet be $\Sigma = \{(,), \vee, \wedge, \neg, 1, t, f\}$ where 1^n is a representation of the nth propositional variable, instead of $p_1, p_2, ...$ A proposition has a natural recursive definition, so we build a CFG exactly to that idea. Our productions are then

$$\begin{aligned} &S \to (S \vee S) \mid (S \wedge S) \mid \neg(S) \mid A \mid B \\ &A \to 1A \mid 1 \\ &B \to t \mid f \end{aligned}$$

Recall that the logical and, or, and negation are complete for propositional logic. If we wanted to express $p \implies q$, instead we may derive $(\neg 1 \vee 11)$. We also have more paranthesis than necessary, but notice that they are cheap, and can only help with ambiguity. This way we do not need a PEMDAS style operator precedence rule to parse a proposition.

Example 2.9. We give a CFG for $\{w\#x \mid x \text{ contains } w^R \text{ as a substring }\}$. If x contains w^R as a substring, then $x = \Sigma^* w^R \Sigma^*$, so $w\#x = w\#(\Sigma^* w^R \Sigma^*) = (w(\#\Sigma^*)w^R)\Sigma^*$. We first will nondeterministically produce and match w with w^R, then we will produce the rest of x.

$$\begin{aligned} &S \to XY \\ &Y \to aY \mid bY \mid \varepsilon \\ &X \to aXa \mid bXb \mid \#Y \end{aligned}$$

This is an interesting grammar, as it shows the power of nondeterminism. You may have had to create some previous non-trivial deterministic algorithms in order to find the longest palindromic substring. You were looking for a needle in a haystack. Here through the power of nondeterminism, we can come at the problem from a different direction. First place the needle, then nondeterministically build all possible haystacks around it.

Example 2.10. As our final example, consider the language over $\Sigma = \{a, b, c, d\}$ as

$$\{w \in \Sigma^* \mid \#a(w) + \#b(w) = \#c(w) + \#d(w)\}$$

We provide a context-free grammar for this language. One idea is that every time something on the left hand side is produced, we ensure something on the right hand side is as well, with no enforcment on order.

$$S \to SaScS \mid SaSdS \mid SbScS \mid SbSdS \mid \varepsilon$$

Cumbersome, but technically correct. We know everytime we add something on the left, we must add something on the right, and vice versa. But we can use nondeterminism to guess for us which of a, b we add on the left and which of c, d we add on the right.

$$\begin{aligned} S &\to SLSRS \mid SRSLS \mid \varepsilon \\ L &\to a \mid b \\ R &\to c \mid d \end{aligned}$$

Relationship with Regular Languages

TODO, drawing

Every regular language is context-free, but not every context-free language is regular. We can prove the containment in two ways.

Theorem 2.2. Every regular language is context-free.

Proof. Let $\mathscr{L}(CFG)$ be the languages produced by context-free grammars. We prove

$$\mathscr{L}(REX) \subsetneq \mathscr{L}(CFG)$$

by structural induction. First we prove our base cases. We give context-free grammars for $\emptyset, \varepsilon, a, b$.

- $\emptyset$: $S \to S$
- ε: $S \to \varepsilon$
- a: $S \to a$
- b: $S \to b$

Now assume our induction hypothesis. Let G_i, G_j be two CFGs to produce L_i and L_j with start non-terminals S_i, S_j respectively. We prove that the context-free grammars are closed under union, concatenation, and star.

1. $L_i \cup L_j$: Copy all productions, add new start state S, and a new productions $S \to S_i \mid S_j$
2. $L_i L_j$: Similarly, with new start state S add production $S \to S_i S_j$
3. L_i^*: Similarly add new start state S and add productions $S \to S_i S \mid \varepsilon$

Since the context-free languages contain our bases cases and are closed under union, concatenation, and star, and we know the regular languages are the smallest such class with this property, we may conclude $\mathscr{L}(REX) \subseteq \mathscr{L}(CFG)$. □

Since we have given a context-free grammar for $\{a^n b^n \mid n \in \mathbb{N}\}$ and have proven via the pumping lemma it is not regular, we may add that the containment is strict. $\mathscr{L}(REX) \subsetneq \mathscr{L}(CFG)$.

Through a similar process to converting a regular expression into an NFA, you may apply this proof to convert a regular expression into a context-free grammar, thus concluding the proof that every regular language is also context-free. Later we will show CFLs are not closed under intersection or complement. This may be intuitive, if you observe the behavior of a CFG. It only knows how to grow correct strings. Given a grammar which produces only the right strings, it gives no idea on how to create a grammar to only produce the wrong ones.

We give a second proof that every regular language is context-free. We define a new kind of grammar to do so.

Definition 2.2 (Regular Grammar)**.** A regular grammar is a four tuple (V, Σ, P, S) defined exactly like a context-free grammar with the additional restriction. It may only have productions of the form $V \to \Sigma V \mid \Sigma \mid \varepsilon$. More specifically

$$A \to aB$$
$$A \to a$$
$$A \to \varepsilon$$

Where A, B are any non-terminals and a is any terminal.

Clearly every regular grammar is also context-free. We claim the regular grammars are equivalent to NFAs.

Theorem 2.3. The regular grammars produce exactly and only the regular languages.

Proof. Let $\mathscr{L}(RG)$ be the class of languages produced by regular grammars. We prove

$$\mathscr{L}(RG) = \mathscr{L}(NFA)$$

via double set containment. First we prove $\mathscr{L}(NFA) \subseteq \mathscr{L}(RG)$ Let $L \in \mathscr{L}(NFA)$. Then there exists an NFA $N = (Q, \Sigma, q_0, \delta, F)$ to decide L. We construct an equivalent regular grammar $G = (V, \Sigma, P, S)$ to simulate this NFA.

- For $Q = \{q_0, ..., q_k\}$ we have non-terminals $V = \{Q_0, ..., Q_k\}$
- Let the set of terminals of our grammar be the alphabet of our NFA.
- For q_0 the start state of our DFA, we designate our start non-terminal as Q_0
- For every transition of the form $q_j \in \delta(q_i, a)$, we add production $Q_i \to aQ_j$
- For every $q_f \in F$, we add production $Q_f \to \varepsilon$

We argue this grammar is equivalent to our NFA. Suppose that N accepts a string $w = w_1...w_n$. Then there exists a sequence of states traversed in our NFA during the accepting computation of w, say $s_0, ..., s_l$. In our constructed grammar, there exists an analogous derivation of the form

$$Q_0 \implies w_1 S_1 \implies w_1 w_2 S_2 \implies ... \implies w_1...w_n Q_f \implies w_1...w_n$$

and thus the grammar derives the string. If the NFA N rejects a word, then all computation paths reject. In our grammar, all possible derivations cannot produce $w_1...w_n$ since if there is a working

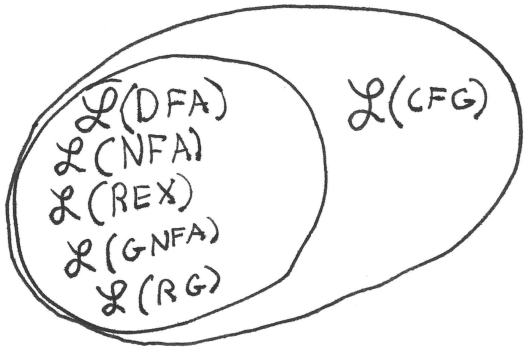
string of the form $w_1...w_nQ_i$, we explicitly do not have a $Q_i \to \varepsilon$ production. Thus we see this grammar and this automata are equivalent.

Next we prove $\mathscr{L}(RG) \subseteq \mathscr{L}(NFA)$, and the construction is nearly identical. Let $G = (V, \Sigma, P, S)$ be a regular grammar. We give an NFA $N = (Q, \Sigma, q_0, \delta, F)$ to simulate G.

- For $V = \{S, X_1, ..., X_k\}$ our non-terminals, form a set of states $Q = \{q_0, ..., q_k\}$.
- Associate q_0 with our start non-terminal.
- Let the alphabet of our NFA be the set of terminals of our grammar.
- For every production of the form $X_i \to aX_j$, add to our transition function $q_j \in \delta(q_i, a)$
- For every production of the form $X_i \to X_j$, add to our transition function $q_j \ \in \delta(q_i, \varepsilon)$
- For every production of the form $X_i \to \varepsilon$, add $q_i \in F$.

The correctness of our NFA follows from a near similar argument as before. Thus we may conclude that the regular grammars produce exactly and only the regular languages. □

Either proof should convince you that we are working with a strictly more powerful model of computation.



Chomsky Normal Form

Given a word w and a context-free grammar G, is $w \in L(G)$? This is a surprisingly non-trivial problem. Unlike an automata which reads the word as input and determines yes or no, a grammar must somehow nondeterministically produce only the correct strings. There is not an obvious way to determine that a string is produced by a context-free grammar. It is even less obvious to determine if a string is not produced by a context-free grammar. Chomsky normal form solves this.

Definition 2.3 (Chomsky Normal Form)**.** We say a CFG is in Chomsky Normal Form (CNF) if it has productions only of the form:

$$A \to BC$$
$$A \to a$$

where the A, B, C are any non-terminals, and the a is any terminal. Additionally B, C cannot be the start non-terminal, and the rule $S \to \varepsilon$ is present if and only if $\varepsilon \in L(G)$.

Any context-free grammar in CNF is still a context-free grammar. We describe a process to convert any grammar into Chomsky normal form.

1. Add a new start non-terminal S_0 and production $S_0 \to S$. Now every production will not have the new start non-terminal anywhere on the right-hand-side.

2. Delete and patch all $A \to \varepsilon$ productions. For example if you have productions $R \to uAv$ and $A \to \varepsilon$, replace them with the productions $R \to uAv \mid uv$.

3. Remove all unit productions of the form $A \to B$. For example if you have productions $A \to B, B \to C$, replace with production $A \to C$. These steps may need to be repeatedly applied until the grammar does not have the property that the step is trying to remove.

4. We remove all productions whose right hand side has length three or greater. Suppose we have a production of the form

$$A \to u_1 \dots u_k$$

where $u_1, ..., u_k$ are terminals or non-terminals. This production can equivalently be represented by a chain of productions with right hand sides of length exactly two. Add non-terminals $A_1, ..., A_{k-2}$ and the following productions.

$$\begin{aligned} A &\to u_1 A_1 \\ A_1 &\to u_2 A_2 \\ &... \\ A_{k-3} &\to u_{k-2} A_{k-2} \\ A_{k-2} &\to u_{k-1} u_k \end{aligned}$$

5. For each appearance of a terminal $a \in \Sigma$ in the right hand side of any production, replace it with non-terminal A and add production $A \to a$.

Note that the second step removes productions of length zero, the third removes productions of length one, and the fourth removes productions of length three or more. What is left is that if a production has a non-terminal on the right hand side, it has exactly two non-terminals. Now that we know every CFG can be put into Chomsky normal form, this restricted structure enables us to prove certain properties.

Theorem 2.4. Let w be a word of length n such that $n \geq 1$. Let G be a context-free grammar in CNF. If G produces w, it takes exactly $2n - 1$ productions.

Proof. We consider the sequence of working strings in reverse.

$$w_1 ... w_n \quad \overset{*}{\Longleftarrow}_1 \quad W_1 ... W_n \quad \overset{*}{\Longleftarrow}_2 \quad S$$

- The last productions (1) goes to n terminals from n non-terminals. At each production, exactly one non-terminal is replaced by exactly one terminal, so this takes n productions.

- For (2), to go from n non-terminals to one terminal, our start terminal, requires $n - 1$ productions. Every rule of a grammar in CNF takes one non-terminal, and adds exactly two, for a net of one non-terminal.

Combined, we see that a grammar in CNF form will take exactly $n + (n - 1) = 2n - 1$ productions to produce a word of length n for $n \geq 1$. This is the point of CNF, the limited structure allows us to have a guarantee for an algorithm. □

We can now determine for any context-free-grammar G and w if G could produce w. Convert your grammar to CNF, compute a list of all possible productions of exactly $2n-1$ steps. The word w is in this list if and only if $w \in L(G)$.

Example 2.11. We convert a grammar into CNF. Let us consider the grammar for $\{a^n b^n \mid n \in \mathbb{N}\}$

$$S \to aSb \mid \varepsilon$$

We add a new start-nonterminal S_0 and a $S_0 \to S$ production.

$$\begin{aligned} S_0 &\to S \\ S &\to aSb \mid \varepsilon \end{aligned}$$

Next, we patch out the $S \to \varepsilon$ rule.

$$\begin{aligned} S_0 &\to S \mid \varepsilon \\ S &\to aSb \mid ab \end{aligned}$$

We now patch out unit rules.

$$\begin{aligned} S_0 &\to aSb \mid ab \mid \varepsilon \\ S &\to aSb \mid ab \end{aligned}$$

Rules of length more than two are simplified by adding non-terminals.

$$\begin{aligned} S_0 &\to aX \mid ab \mid \varepsilon \\ S &\to aX \mid ab \\ X &\to Sb \end{aligned}$$

For each appearance of a terminal on the right hand side of a production, we replace it with a non-terminal.

$$\begin{aligned} S_0 &\to AX \mid AB \mid \varepsilon \\ S &\to AX \mid AB \\ X &\to SB \\ A &\to a \\ B &\to b \end{aligned}$$

This grammar is now in Chomsky normal form. To produce $aabb$, let us verify it takes $2n-1 = 7$ productions.

$$S_0 \overset{1}{\Longrightarrow} AX \overset{2}{\Longrightarrow} ASB \overset{3}{\Longrightarrow} AABB \overset{4}{\Longrightarrow} aABB \overset{5}{\Longrightarrow} aaBB \overset{6}{\Longrightarrow} aabB \overset{7}{\Longrightarrow} aabb$$