Automata and Languages

Amit Rajaraman

March 2020

Contents

1 Regular Languages **2**
- 1.1 Finite Automata . . . 2
- 1.2 Nondeterminism . . . 5
- 1.3 Regular Expressions . . . 9
- 1.4 Nonregular Languages . . . 13

2 Context-Free Languages **16**
- 2.1 Context-Free Grammars . . . 16
- 2.2 Pushdown Automata . . . 20

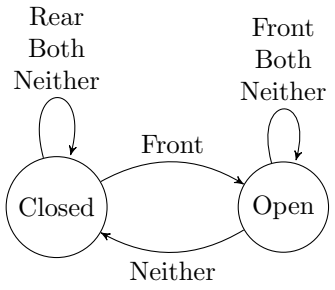
§1 Regular Languages

Before we proceed any further, a question we must ask is - what *is* a computer? The computers we use are probably too complicated to model as a mathematical system. So we shall try to create an idealized computer called a *computational model*. Like models in general, this model is realistic in some ways, and unrealistic in others.

1.1 Finite Automata

Finite automata are a good place to begin that are good models for computers with an extremely limited amount of memory.

Example. For starters, consider an automatic door controller. It has a front pad, a back pad and a door. The door can be either open or closed and each of the pads can be either pressed or not pressed. Using this, we can construct a "state diagram" to show how the state of the system proceeds:

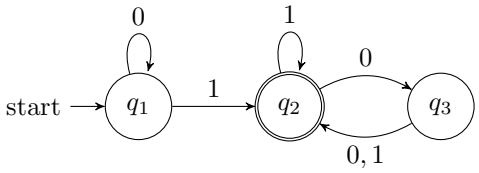


It can also be represented by the following table:

	Front	Back	Neither	Both
Closed	Open	Closed	Closed	Closed
Open	Closed	Closed	Open	Closed

Thinking of a finite automaton like this automatic door controller, which has only a single bit of memory, suggests standard ways to represent automata as a state transition graph or a state transition table.
Finite automata, and their probabilistic counterparts, *Markov Chains*, are very useful tools.
Let us look at another finite automaton to cement the idea before exactly defining what it is.

Example. Consider the following state diagram of an automaton M.



It has three "states", q_1, q_2 and q_3. The *start state*, q_1 is indicated as shown. The *accept state*, q_2 is indicated by the double circle. The arrows are called *transitions*. When the automaton receives some input string like `11001`, it processes the string and produces some output, *accept* or *reject*. For now, we will consider only yes/no questions like this one.
An already obvious question to ask is that what language of input strings give an accept output? We will answer this soon.

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

- Q is a finite set called the set of *states*.
- Σ is a finite set of input symbols called the *alphabet*.

- $\delta : Q \times \Sigma \to Q$ is the *transition function*.
- $q_0 \in Q$ is the *start state*.
- $F \subseteq Q$ is the *set of accept states*.

Accept states are also sometimes called *final states*.

Example. The finite automaton M we described earlier can be put in this format in the following way:

- $Q = \{q_1, q_2, q_3\}$
- $\Sigma = \{0, 1\}$
- δ is described as follows:

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

- q_1 is the start state, and
- $F = \{q_2\}$

If A is the set of all strings that machine M accepts, we say that A is the *language* of M and write $L(M) = A$. In our example,

$$L(M) = \{w \mid w \text{ contains at least one 1 and an even number of 0s follow the last 1}\}.$$

Definition 1.2. Let $M = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite automaton and let $w = w_1 w_2 \cdots w_n$ be a string where each $w_i \in \Sigma$. Then M *accepts* w if a sequence of states $r_0, r_1, \ldots, r_n$ in Q exist with three conditions:

1. $r_0 = q_0$,
2. $\delta(r_i, w_{i+1}) = r_{i+1}$ for $i = 0, 1, \ldots, n-1$ and
3. $r_n \in F$.

We say that M *recognizes* language A if $A = \{w \mid M \text{ accepts } w\}$.

Definition 1.3. A language is called a *regular language* if some deterministic finite automaton recognizes it.

We define three operations on languages, called *regular operations*, and use them to study the properties of regular languages.

Definition 1.4. Let A and B be languages. We define the following *regular operations*:

- *Union*: $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$.
- *Concatenation*: $A \circ B = \{xy \mid x \in A \text{ and } y \in B\}$.
- *Star*: $A^* = \{x_1 x_2 \cdots x_k \mid k > 0 \text{ and each } x_i \in A\}$

These operations are called regular operations as the class of regular languages are closed under these operations. We shall first show a proof that the class of regular languages is closed under the union operation.

We shall revisit this later and provide a much simpler proof.

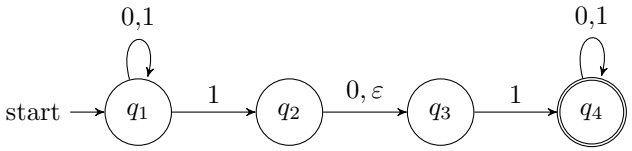
Proof. Let $M_1(Q_1, \Sigma_1, \delta_1, q_{01}, F_1)$ and $M_2(Q_2, \Sigma_2, \delta_2, q_{02}, F_2)$ recognize A_1 and A_2 respectively. Set $\Sigma = \Sigma_1 \cup \Sigma_2$.
We construct the finite automaton $M(Q, \Sigma, \delta, q_0, F)$ that recognizes $A_1 \cup A_2$ by setting $Q = Q_1 \times Q_2$, $\delta(q_1, q_2) = (\delta_1(q_1), \delta_2(q_2))$ for all $q_1 \in Q_1, q_2 \in Q_2$, $q_0 = (q_{01}, q_{02})$ and $F = \{(q_1, q_2) \mid q_1 \in F_1 \text{ or } q_2 \in F_2\}$.
It can be checked that M recognizes $A_1 \cup A_2$. ■

To prove that the class of regular languages is closed under concatenation, we must build an automaton that accepts a string if it can be split into two parts, the first of which is accepted by the first machine and the second of which is accepted by the second machine. To show how this can be done, we introduce a new concept called non-determinism.

1.2 Nondeterminism

So far, for every input symbol, the next state is exactly determined, that is, it is a *deterministic* machine. In a *nondeterministic* machine, several choices may exist for the next state at any point.
Every deterministic finite automaton is thus clearly a nondeterministic finite automaton as well.
We shall abbreviate "nondeterministic finite automaton" as NFA and "deterministic finite automaton" as DFA.

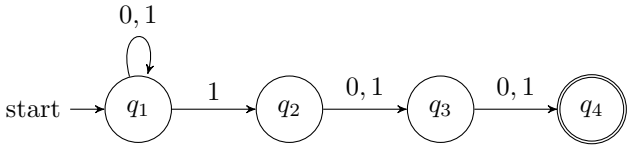
Example. The following is an NFA (and not a DFA):



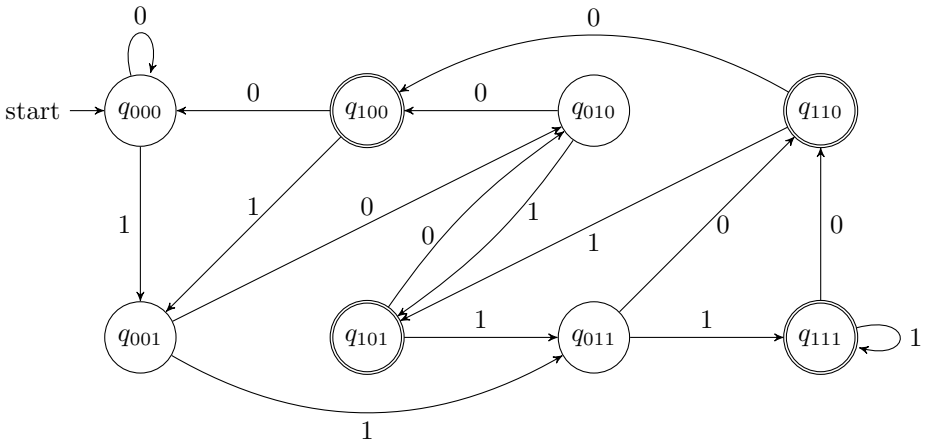
The difference between a DFA and an NFA is immediately apparent. In the above example, there are multiple arrows from q_1 corresponding to input 1 and no arrow from q_2 corresponding to 1. We also see the addition of an ε as input, which is not in the alphabet. How does an NFA compute? At each point with multiple paths, the machine splits into multiple copies of itself, each one following one of the possibilites in parallel. Each copy of the machine then continues as before. If there are subsequent choices, it splits again. If the next input symbol does not appear on any of the arrows exiting the current state, that copy of the machine dies. Finally, if *any* of these copies ends at an accept state, the NFA is said to accept the input string.
If a state with an ε exiting arrow is encountered, the machine splits into multiple copies,each one following one of the arrows, *without reading any input.* Nondeterminism is a sort of parallel computing where many "threads" are running simultaneously. The NFA splitting corresponds to the process of "forking" into several children, each proceeding separately. If any of these processes accepts, the entire computation accepts.
We can also think of an NFA as a tree of possibilities where the tree splits at each point where the machine has more than one choice.

But why are NFA's important? As we shall see shortly, every NFA can be converted to an equivalent DFA, and constructing NFA's is sometimes easier than constructing DFA's. An NFA is usually much smaller than its DFA counterpart and its functioning may be easier to understand.

Example. Let A be the language consisting of all strings over $\{0, 1\}$ containing a 1 in the third position from the end. The following NFA recognizes A.



The following DFA also recognizes A.



It is obvious that the DFA in the above example is far more complicated than the NFA.
Let us now formally define an NFA.

Definition 1.5. A nondeterministic finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

- Q is a finite set of states.
- Σ is a finite alphabet.
- $\delta : Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$ is the transition function.
- $q_0 \in Q$ is the start state, and
- $F \subseteq Q$ is the set of accept states.

Here Σ_ε is $\Sigma \cup \{\varepsilon\}$ and $\mathcal{P}(Q)$ is the power set of Q. Let $w = y_1 y_2 \cdots y_m$ be a string over the alphabet Σ. We say that a nondeterministic finite automaton N *accepts* w if we can write w as $w = y_1 y_2 \cdots y_m$ where each $y_i \in \Sigma_\varepsilon$ and a sequence of states $r = r_0, r_1, \ldots, r_m$ exists in Q such that:

- $r_0 = q_0$,
- $r_{i+1} \in \delta(r_i, y_{i+1})$ for $i = 0, 1, \ldots, m-1$ and
- $r_m \in F$.

Definition 1.6. We say that two machines are *equivalent* if they recognize the same language.

Theorem 1.1. Every nondeterministic finite automaton has an equivalent deterministic finite automaton.

Proof. Let us first do this for the case where there are no ε arrows. Then given an NFA $N = (Q, \Sigma, \delta, q_0, F)$ that recognizes language A, we can construct the DFA $M = (Q', \Sigma, \delta', q_0', F')$ such that

- $Q' = \mathcal{P}(Q)$. $\mathcal{P}(Q)$ is the power set of Q.
- For any $R \in Q'$ and symbol a,
$$\delta'(R, a) = \bigcup_{r \in R} \delta(r, a).$$
- $q_0' = \{q_0\}$.
- $F' = \{q \in Q \mid q \text{ contains at least one accept state}\}$.

Now we need to consider the ε arrows as well. For any $R \in Q'$, define

$$E(R) = \{q \mid q \text{ can be reached from } R \text{ by traveling along 0 or more } \varepsilon \text{ arrows}\}.$$

We can then modify the transition function as follows:

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

We must also modify the start state to $q_0' = E(\{q_0\})$.
It is clear that this construction will work and recognize language A, as can easily be verified from the definition. ■

Note that the size of the DFA created from the above construction equivalent to a given NFA has a number of nodes which is exponential in terms of the number of nodes of the NFA. It is thus clear why NFA's tend to be significantly more compact than their corresponding equivalent DFA's.

Corollary 1.2. A language is regular if and only if some nondeterministic finite automaton recognizes it.

Now that we have this much stronger corollary to determine if a language is regular, let us go back to continuing to prove that the class or regular languages is closed under the regular operations.

Theorem 1.3. The class of regular languages is closed under the union operation.

Proof. Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize A_1 and $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognize A_2 for (regular) languages A_1 and A_2. (We assume that they have the same alphabet Σ. If they have different languages Σ_1 and Σ_2, set $\Sigma = \Sigma_1 \cup \Sigma_2$).
Construct $N = (Q, \Sigma, \delta, q_0, F)$ to recognize $A_1 \cup A_2$ as follows.

- $Q = \{q_0\} \cup Q_1 \cup Q_2$. (Assume without loss of generality that Q_1 and Q_2 are disjoint)
- The state q_0 is the start state of N. (q_0 is not in Q_1 or Q_2)
- $F = F_1 \cup F_2$
- δ is defined as follows. For any $q \in Q$ and $a \in \Sigma_\varepsilon$,

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

Here we basically check separately whether a given string is accepted by M_1 or M_2, and accept if it is accepted by either. ■

Theorem 1.4. The class of regular languages is closed under concatenation.

Proof. Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize A_1 and $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognize A_2 for (regular) languages A_1 and A_2.
Construct $N = (Q, \Sigma, \delta, q_1, F_2)$ to recognize $A_1 \circ A_2$.

- $Q = Q_1 \cup Q_2$.
- Define δ such that for any $q \in Q$ and $a \in \Sigma_\varepsilon$,

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

We basically split if the part of the string read so far is accepted by N_1, check whether the remainder is accepted by N_2 and recurse.

■

Theorem 1.5. The class of regular languages is closed under the star operation

Proof. Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize A. Construct $N = (Q, \Sigma, \delta, q_0, F)$ as follows to recognize A^*.

- $Q = \{q_0\} \cup Q_1$. (q_0 is not in Q_1)
- $F = \{q_0\} \cup F_1$. This is done so that ε is in the resulting language.
- Define δ such that for any $q \in Q$ and any $a \in \Sigma_\varepsilon$,

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \text{ and } q \notin F_1 \\ \delta_1(q,a) & q \in F_1 \text{ and } a \neq \varepsilon \\ \delta_1(q,a) \cup \{q_1\} & q \in F_1 \text{ and } a = \varepsilon \\ \{q_1\} & q = q_0 \text{ and } a = \varepsilon \\ \emptyset & q = q_0 \text{ and } a \neq \varepsilon \end{cases}$$

It can be checked that this machine recognizes A^*. The idea is similar to that in the concatenation proof, but we keep looping on the same machine. ■

Exercise 1.1. Prove that the class of regular languages is closed under the intersection operation.

1.3 Regular Expressions

We use regular expressions to build up expressions describing languages, which are called regular expressions. An example is $(0 \cup 1) \circ 0^*$. This language describes all strings that start with a 0 or 1 and are followed by 0s. The concatenation symbol is usually omitted and understood implicitly, so the given expression can also be written as $(0 \cup 1)0^*$. Regular expressions have several obvious uses in computer science, like searching for

strings in a text that satisfy certain properties for instance.
Like how in arithmetic, there is a precedence order in the operations wherein we give $\times$ and higher precedence than $+$. Similarly, in regular expressions, the precedence order is star, then concatenation, and finally union. (unless parentheses are used to change the order)

Definition 1.7. We say that R is a *regular expression* if R is

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

Remark. Do not confuse the regular expressions ε and $\emptyset$. $\{\varepsilon\}$ is the language containing a single string, the empty string, whereas $\emptyset$ is the language containing no strings. Given a regular expression R, we use $L(R)$

to denote the language of R.
For convenience, we use R^+ to denote RR^*, that is, the language has all strings that are 1 or more concatenations of strings from R. So $R^+ \cup \{\varepsilon\} = R^*$. We also let R^k to denote the concatenation of k R's with each other.

Exercise 1.2. Describe the languages corresponding to the following regular expressions.

(a) $1^*\emptyset$

(b) $\emptyset^*$

(c) $(0 \cup \varepsilon)(1 \cup \varepsilon)$

(d) $(\Sigma\Sigma)^*$

(e) $(01^+)^*$

Solution.

(a) $\emptyset$

(b) $\{\varepsilon\}$

(c) $\{\varepsilon, 0, 1, 01\}$

(d) $\{w \mid w \text{ is a string of length } 2\}$

(e) $\{w \mid \text{every 0 in } w \text{ is followed by at least one 1}\}$

Exercise 1.3. Prove the following identities for any regular language R.

(a) $R \cup \emptyset = R$

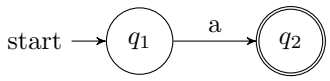
(b) $R \circ \{\varepsilon\} = R$

Regular expressions and finite automata are equivalent in their descriptive power, which may not be an immediately obvious fact. However any of them can be converted to the other.

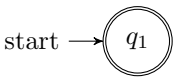
Lemma 1.6. If a language is described by a regular expression, it is regular.

Proof. We shall prove each of the 5 cases of the definition separately.

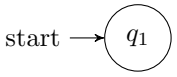
1. $R = a$ for some a in Σ. Then $L(R) = \{a\}$. The following NFA recognizes $L(R)$.



2. $R = \varepsilon$. Then $L(R) = \{\varepsilon\}$. The following NFA recognizes $L(R)$.



3. $R = \emptyset$. Then $L(R) = \emptyset$. The following NFA recognizes $L(R)$.



For the remaining cases, that is, $R = R_1 \cup R_2, R = R_1 \circ R_2$ and $R = R_1^*$, we use the constructions given in the proofs that the regular languages are closed under each of the regular operations. ■

We define a new type of automaton to help prove the next theorem, which states that if a language is regular, it can be described by a regular expression. We call this a generalized nondeterministic finite automaton (abbreviated as GNFA). The GNFA reads blocks of symbols from the input, not necessarily just one symbol at a time as in an ordinary NFA. The GNFA moves from one state to another by reading a block of symbols from the input, which may themselves constitute a string determined by the regular expression on that arrow.
For convenience, we also require that each GNFA always has a special form that meets the following criteria:

- The start state has transition arrows going to every other state but no arrows coming in from any other state.
- There is only one accept state, and there are arrows coming in to the accept state from every other state but no arrows going out to any other state. Further, the accept state is not the same as the start state.
- Except for the start and accept states, there are arrows going from every state to every other state and also from each state to itself.

To prove the theorem, what we do is find a way to convert any DFA to a special GNFA (with ≥ 2 states), and then repeatedly constructing an equivalent GNFA with 1 less state by "ripping" out a state. When we get a GNFA with just 2 states, it just has a single arrow from the start state to the accept state, with label equal to the required regular expression.

Define $\mathcal{R}$ to be the set of all regular expressions over the alphabet Σ.

Definition 1.8. A *generalized nondeterministic finite automaton* is a 5-tuple $(Q, \Sigma, \delta, q_{\text{start}}, q_{\text{accept}})$, where

- Q is the finite set of states,
- Σ is the input alphabet,
- $\delta : (Q - \{q_{\text{accept}}\}) \times (Q - \{q_{\text{start}}\}) \to \mathcal{R}$ is the transition function,

- q_{start} is the start state, and
- q_{accept} is the accept state.

A GNFA accepts a string w in Σ^* if $w = w_1 w_2 \cdots w_k$, where each w_i is in Σ^* and a sequence of states $q_0, q_1, \ldots, q_k$ exist such that

- $q_0 = q_{\text{start}}$,
- $q_k = q_{\text{accept}}$, and
- for each i, we have $w_i \in L(R_i)$, where $R_i = \delta(q_{i-1}, q_i)$. In other words, R_i is the expression on the arrow from q_{i-1} to q_i.

Lemma 1.7. Given any DFA, there exists an equivalent GNFA in the special form.

Proof. Consider a DFA $N = (Q, \Sigma, \delta, q_0, F)$, define a GNFA $N' = (Q', \Sigma, \delta', q_{\text{start}}, q_{\text{accept}})$ as follows.

- $Q' = Q \cup \{q_{\text{start}}, q_{\text{accept}}\}$, ($q_{\text{start}}$ and q_{accept} are not in Q)
- δ' is defined as follows. For any $q_i, q_j \in Q'$,

$$\delta'(q_i, q_j) = \begin{cases} \varepsilon & q_i = q_{\text{start}} \text{ and } q_j = q_0 \\ \varepsilon & q_i \in F \text{ and } q_j = q_{\text{accept}} \\ \{a : \delta(q_i, a) = q_j\} & \text{otherwise.} \end{cases}$$

We simply add a new start state with an ε arrow to the old start state, and a new accept state with ε arrows from each of the old accept states. If any arrows have multiple labels, we replace these with a single arrow whose label is the union of the previous labels. We add arrows labelled $\emptyset$ between states that had no arrows between them.
It can be checked that this is in fact a GNFA in special form. ■

We shall now exactly describe the process that we use to obtain a regular expression from a given GNFA. Given a GNFA G, we define $\texttt{CONVERT}(G)$ by the following algorithm.

1. Let k be the number of states of G.
2. If $k = 2$, then G must consist of a start state, an accept state, and exactly one arrow between them labelled with a regular expression R.

   Return R.
3. If $k > 2$, we select any state $q_{\text{rip}} \in Q$ different from q_{start} and q_{accept} and let G' be the GNFA $(Q', \Sigma, \delta', q_{\text{start}}, q_{\text{accept}})$ where

   $$Q' = Q - \{q_{\text{rip}}\},$$

   and for any $q_i \in Q' - \{q_{\text{accept}}\}$ and $q_j \in Q' - \{q_{\text{start}}\}$, let

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

   for $R_1 = \delta(q_i, q_{\text{rip}}), R_2 = \delta(q_{\text{rip}}, q_{\text{rip}}), R_3 = \delta(q_{\text{rip}}, q_j)$ and $R_4 = \delta(q_i, q_j)$.

   Return $\texttt{CONVERT}(G')$.

Lemma 1.8. For any GNFA G, $\texttt{CONVERT}(G)$ is equivalent to G.

Proof. We shall prove this by an induction on k, the number of states in G. Define G' as in the above recursive algorithm.

Basis. The basis case is $k = 2$. In this case, G only consists of a start state, an accept state, and a single arrow between them with the label describing all strings that G recognizes. Thus the expression is equivalent to G.

Inductive step. Assume it is true for $k-1$ states. Let $k > 2$. We shall show that G, which has k states, and G', which has $k-1$ states, are equivalent, that is, they recognize the same language. Suppose that G accepts a sequence w. Then in an accepting branch of the computation, G enters a sequence of states $q_{\text{start}}, q_1, q_2, \ldots, q_{\text{accept}}$.
If none of them is q_{rip}, G' also clearly accepts w. If there are any runs of q_{rip}, then removing those runs also yields an accepting string. This is because for bracketing sequences q_i, q_j around a run, the arrow between q_i and q_j has all strings from q_i to q_j through q_{rip}. So G' accepts w.
On the other hand, if G' accepts some sequence w, then the arrow from q_i to q_j in G' describes the collection of strings taking q_i to q_j in G, either directly or through q_{rip}. Clearly, G must also accept w. Thus G and G' are equivalent.

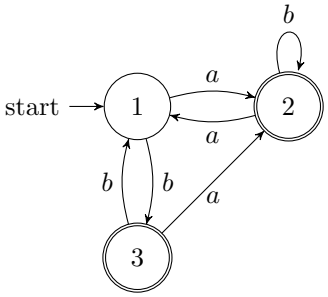
The induction hypothesis merely says that when the algorithm calls itself recursively on G', the resulting regular expression is equivalent to G' (because G' has $k-1$ states). As G is equivalent to G', G must also be equivalent to the resulting regular expression, namely `CONVERT`(G). ■

Theorem 1.9. A language is regular if and only if some regular expression describes it.

Proof. We have already proved one of the implications in 1.6.
To prove the other implication, we first use 1.7 to create an equivalent GNFA given any DFA. We then use 1.8 to obtain a regular expression that is equivalent to this GNFA, which is in turn equivalent to the initial DFA.

We thus have the two way implication. ■

Exercise 1.4. Find the regular expression corresponding to the following NFA.



Solution. The resulting regular expression from the above DFA using the algorithm used in 1.9 is

$$(a(aa \cup b)^*ab \cup b)((ba \cup a)(aa \cup b)^*ab \cup bb)^*((ba \cup a)(aa \cup b)^* \cup \varepsilon) \cup a(aa \cup b)^*$$

Exercise 1.5. For any string $w = w_1 w_2 \cdots w_n$, the *reverse* of w, written $w^{\mathcal{R}}$, is given by $w_n \cdots w_2 w_1$. For any language A, denote $A^{\mathcal{R}} = \{w^{\mathcal{R}} \mid w \in A\}$. Show that if A is regular, $A^{\mathcal{R}}$ is regular.

1.4 Nonregular Languages

As we have regular languages, the presence of *nonregular* languages is also expected, that is, languages that cannot be recognized by any finite automaton. For instance, consider the language $B = \{0^n1^n \mid n \geq 0\}$. We wouldn't expect to have a finite automaton that recognizes this language as it appears we'd need to find a way to count the number of 0s processed so far, and since this number is not bounded above and we only have a finite number of memory. Indeed, this language cannot be recognized by any finite automaton.

However, this immediately begs the question, what is a condition to determine whether a language is regular or nonregular?

Theorem 1.10 (The Pumping Lemma)**.** If A is a regular language, then there is a number p, called the *pumping length*, where if s is any string in A of length at least p, then s can be divided into three pieces, $s = xyz$, satisfying the following conditions:

1. For each $i \geq 0$, $xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

The proof of this theorem essentially relies on the pigeonhole principle. If I set p as the number of states, then I will have a segment in the middle which is just equal to a loop on some state. Since concatenating this segment with itself is still just a loop on that state, the theorem seems correct.

Proof. Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA recognizing A. Let p be the number of states of M.
Let $s = s_1s_2 \cdots s_n$ be a string in A of length $n \geq p$. Let $r_1, r_2, \ldots r_{n+1}$ be the corresponding states that M goes through while processing s, so $r_{i+1} = \delta(r_i, s_i)$ for $i = 1, 2, \ldots, n$. As M has p states, we have that there is at least one repeated state in the first $p + 1$ elements of the sequence (by the Pigeonhole principle). Let the indices of this repeated state be l, j, that is, $r_l = r_j$ where $l < j \leq p + 1$. Set $x = s_1s_2 \cdots s_{l-1}, y = s_ls_{l+1} \cdots s_{j-1}$ and $z = s_js_{j+1} \cdots s_n$. Now note that x takes M from r_1 to r_l, y takes M from r_l to r_l, and z takes M from r_l to r_{n+1}.
As y takes M from r_l to r_l, y^i for $i \geq 0$ will also take M from r_l to r_l and xy^iz will also be accepted by M.
As $l \neq j$, $|y| > 0$. And as $j \leq p + 1, j - 1 \leq p$ and $|xy| \leq p$. ■

Exercise 1.6. Prove that the language $B = \{0^n1^n \mid n \geq 0\}$ is nonregular.

Solution. If B is a regular language, then consider $s = 0^p1^p$, where p is the pumping length. Let us take two cases. (Here x, y, z represent the same x, y, z as in the Pumping Lemma)

- y is either only 0s (or only 1s). Then xy^2z will have more 0s (1s) than 1s (0s) and so it is not in B.
- y contains both 0s and 1s. Then xy^2z can have the same number of 0s and 1s, but they will be out of order and hence it will not be a member of B.

We arrive at a contradiction and hence B is not a regular language.

Exercise 1.7. Prove that $B = \{w \mid w$ has an equal number of 0s and 1s$\}$ is a nonregular language.

Hint. Show that 0^p1^p cannot be pumped.

Exercise 1.8. Prove the nonregularity of the following languages.

(a) $D = \{ww \mid w \in \{0, 1\}^*\}$.

(b) $E = \{1^{n^2} \mid n \geq 0\}$. This is a *unary* nonregular language.

(c) $F = \{0^i1^j \mid i > j\}$.

We shall now show another theorem that helps us determine when a language is regular.

Definition 1.9. Let x and y be strings and L be any language. We say that x and y are *distinguishable by* L if some string z exists such that exactly one of the strings xz and yz is in L. Otherwise, if for every string z, $xz \in L$ if and only if $yz \in L$, we say that x and y are *indistinguishable by* L.

If x and y are indistinguishable by L, we write $x \equiv_L y$.

Lemma 1.11. Given a language L, $\equiv_L$ is an equivalence relation.

Proof. This proof is trivial and is left as an exercise to the reader. ■

Definition 1.10. Let L be a language and X be a set of strings. We say that X is *pairwise distinguishable* by L if every two distinct strings in X are distinguishable by L.

Definition 1.11. Let L be a language. The *index* of L is defined as the maximum number of elements in any set that is pairwise distinguishable by L. The index of a language may be finite or infinite.

Theorem 1.12 (Myhill-Nerode Theorem)**.** A language L is regular if and only if it has finite index. Moreover, its index is the size of the smallest DFA recognizing it.

Proof. We shall first show that if a language L is recognized by a DFA with k states, it has index at most k. If there exists a set with $> k$ elements that is pairwise distinguishable, then by the Pigeonhole principle, we get that there exist two strings x and y in the set such that the state the DFA is in after processing these strings is the same. However, if this is the case, then for any string z, the DFA will accept xz if and only if it accepts yz. Thus, the index is at most k.

We shall now show the other direction, that is, if the index of L is a finite number k, it is recognized by a DFA with k states.
Let $X = \{x_1, x_2, \ldots, x_k\}$ be pairwise distinguishable by L. Construct a DFA $N = \{Q, \Sigma, \delta, x_0, F\}$ as follows. $Q = X$. $\delta(x_i, a) = x_j$ if $x_i a \equiv_L x_j$. Note that this x_j is unique as X is pairwise distinguishable and such an x_j exists as X has the *maximum* number of elements. x_0 is the unique x_i such that $x_i \equiv_L \varepsilon$. $F = X \cap L$. If a string $s \equiv_L x_j$ for some j, then the state of N after reading s will be x_j (Why?). Thus the language recognized by N is just L itself (from the definition of F).

Combining the above two, we get that a language is regular if and only if it has finite index. To prove the second part of the theorem, suppose that there is a DFA that recognizes the language with size less than the index. Then using the first part of the proof gives a contradiction. There clearly exists a DFA recognizing the language of size equal to the index from the second part of the theorem. This completes our proof. ■

Problems

Exercise 1.9. Let

$$\Sigma_3 = \left\{ \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \cdots, \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right\}.$$

A string in Σ_3 gives 3 rows. Consider each row to be a binary number and let

$$B = \{w \in \Sigma_3^* \mid \text{the bottom row of } w \text{ is the sum of the top two rows}\}$$

Show that B is regular.

Exercise 1.10. Consider the language

$$L = \{ww^{\mathcal{R}} \mid w \in (0 \cup 1)^*\}$$

Show that L is nonregular. (The meaning of $w^{\mathcal{R}}$ is described in 1.5.)

Exercise 1.11. Say that string x is a *prefix* of string y if a string z exists such that $xz = y$ and that x is a *proper prefix* of y if in addition, $x \neq y$. Let A be any language. Show that the class of regular languages is closed under the following two operations.

(a) $\texttt{NOPREFIX}(A) = \{w \in A \mid \text{no proper prefix of } w \text{ is in } A\}$

(b) $\texttt{NOEXTEND}(A) = \{w \in A \mid w \text{ is not the proper prefix of any string in } A\}$

Exercise 1.12. Let A be any language. Show that the class of regular languages is closed under the DROPOUT operation defined as follows.

$$\texttt{DROPOUT}(A) = \{xz \mid xyz \in A \text{ where } x, z \in \Sigma^*, y \in \Sigma\}.$$

Exercise 1.13. For languages A and B, let the shuffle of A and B be the language

$$\{w \mid w = a_1 b_1 \cdots a_k b_k, \text{ where } a_1 \cdots a_k \in A \text{ and } b_1 \cdots b_k \in B, \text{ each } a_i, b_i \in \Sigma^*\}.$$

Show that the classes of regular languages is closed under the shuffle operation.

Exercise 1.14. If A is any language, let $A_{\frac{1}{2}-}$ be the set of all first halves of strings in A defined as follows.

$$A_{\frac{1}{2}-} = \{x \mid \text{for some } y, |x| = |y| \text{ and } xy \in A\}$$

Show that if A is regular, so is $A_{\frac{1}{2}-}$.

Exercise 1.15. Let B and D be two languages. Write $B \Subset D$ if $B \subseteq D$ and D contains infinitely many strings that are not in B. Show that, if B and D are two regular languages where $B \Subset D$, then we can find a regular language C where $B \Subset C \Subset D$.

Exercise 1.16. Let $M = \{Q, \Sigma, \delta, q_0, F\}$ be a DFA and $h \in Q$ be called its "home". A *synchronizing sequence* for M and h is a string $s \in \Sigma^*$ where $\hat{\delta}(q, s) = h$ for all $q \in Q$ (Here $\hat{\delta}(q, s)$ is the state M ends up at when it starts at q and processes s). Say that M is *synchronizable* if it has a synchronizing sequence for some state h. Prove that if M is a k-state synchronizable DFA, it has a synchronizing sequence of length at most k^3.
This has in fact been improved by Kohavi (?, check again) to show that the length of the synchronizing sequence lies between $\frac{k^3-k}{6}$ and $\frac{k^2+k-4}{2}$. Černý conjectured in 1964 that this bound can be improved to $(k-1)^2$, a very tight bound, which remains one of the largest unsolved problems in Automata Theory at the time of writing this.

§2 Context-Free Languages

In this chapter, we shall present context-free grammars, that can describe certain features that have a recursive structure. They naturally arise from trying to understand the relationship of terms like nouns, verbs and prepositions in ordinary grammar, and their respective phrases which lead to a natural recursion. An important application of this is in most compilers and interpreters, which contain a parser that extracts the meaning of a program prior to compilation. A number of methods help construct this parser once a context-free language is available. Some even automatically generate the parser.
The collection of associated languages are *context-free languages*.

2.1 Context-Free Grammars

Example. The following is a context-free grammar. Call it G_1.

$$A \to 0A1$$
$$A \to B$$
$$B \to \#$$

The grammar consists of *substitution rules* or *productions*. Each rule has a symbol, called a *variable*, and a string separated by an arrow. The string contains variables and other symbols called *terminals*. One variable is designated as the start variable, and usually occurs on the left-hand side of the top-most rule. Using a grammar, we describe a language, called a context-free language, by generating each string of that language as follows.

1. Write down the start variable.
2. Find a variable that is written down and a rule that starts with that variable. Replace that variable with the right hand side of that rule.
3. Repeat step 2 until no more variables remain.

For example, G_1 generates 00#11 as follows.

$$A \implies 0A1 \implies 00A11 \implies 00B11 \implies 00\#11$$

The above information may also be represented pictorially by a *parse tree*.

In the above example, the two rules with A on the left-hand side can be merged into a single rule as $A \to 0A1 \mid B$, using the symbol "|" as an "or". To understand the link with the English language, we give a more illustrative example below.

Example.

$$\begin{aligned}
\langle \texttt{SENTENCE} \rangle &\to \langle \texttt{NOUN-PHRASE} \rangle \langle \texttt{VERB-PHRASE} \rangle \\
\langle \texttt{NOUN-PHRASE} \rangle &\to \langle \texttt{CMPLX-NOUN} \rangle \mid \langle \texttt{CMPLX-NOUN} \rangle \langle \texttt{PREP-PHRASE} \rangle \\
\langle \texttt{VERB-PHRASE} \rangle &\to \langle \texttt{CMPLX-VERB} \rangle \mid \langle \texttt{CMPLX-VERB} \rangle \langle \texttt{PREP-PHRASE} \rangle \\
\langle \texttt{PREP-PHRASE} \rangle &\to \langle \texttt{PREP} \rangle \langle \texttt{CMPLX-NOUN} \rangle \\
\langle \texttt{CMPLX-NOUN} \rangle &\to \langle \texttt{ARTICLE} \rangle \langle \texttt{NOUN} \rangle \\
\langle \texttt{CMPLX-VERB} \rangle &\to \langle \texttt{VERB} \rangle \mid \langle \texttt{VERB} \rangle \langle \texttt{NOUN-PHRASE} \rangle \\
\langle \texttt{ARTICLE} \rangle &\to \texttt{a} \mid \texttt{the} \\
\langle \texttt{NOUN} \rangle &\to \texttt{boy} \mid \texttt{girl} \mid \texttt{flower} \\
\langle \texttt{VERB} \rangle &\to \texttt{touches} \mid \texttt{sees} \\
\langle \texttt{PREP} \rangle &\to \texttt{with}
\end{aligned}$$

Strings in this grammar include

```
a boy sees
the boy touches a flower
the girl touches the boy with the flower
```

Exercise 2.1. Show two different ways in which the third string in the above grammar can be generated. Here, "two different ways" means two different parse trees, not two different derivations. (Why aren't these two the same?) Note the correspondence between these two ways and the two ways the string can be read.

Let us formalize our definition of a context-free grammar.

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

1. V is a finite set called the *variables*.
2. Σ is a finite set, disjoint from V, called the *terminals*.
3. R is a finite set of *rules*, with each rule being a variable and a string of variables and terminals. More precisely, R is a finite subset of $V \times (V \cup \Sigma)^*$, called the *set of rules*.
4. $S \in V$ is the *start variable*.

We shall abbreviate "context-free grammar" as CFG and "context-free language" as CFL.
If u, v and w are strings of variables and terminals, and $A \to w$ is a rule of the grammar, we say that uAv *yields* uwv, written as $uAv \Rightarrow uwv$.
We say that u *derives* v, written $u \overset{*}{\Rightarrow} v$, if $u = v$ or there is a sequence $u_1, u_2, \ldots, u_k$ of strings of terminals and variables exists for $k \geq 0$ such that

$$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_k \Rightarrow v$$

The *language* of the grammar is $\{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\}$.

Exercise 2.2. Put the two examples given above in the form given in the definition of a CFG.

Many CFLs are the union of simpler CFLs. These can then easily be combined to form a corresponding CFG by combining their rules and then adding a new rule $S \to S_1 \mid S_2 \mid \cdots \mid S_k$, where the S_is are the start variables of each of the CFGs.

Exercise 2.3. Construct a CFG which has corresponding language $\{0^n 1^n \mid n \geq 0\} \cup \{1^n 0^n \mid n \geq 0\}$

Solution. We can combine the two following CFGs:

$$S_1 \to 0S_1 1 \mid \varepsilon \text{ and } S_2 \to 1S_2 0 \mid \varepsilon$$

to get a grammar that generates the given language as follows.

$$\begin{aligned} S &\to S_1 \mid S_2 \\ S_1 &\to 0S_1 1 \mid \varepsilon \\ S_2 &\to 1S_2 0 \mid \varepsilon \end{aligned}$$

Theorem 2.1. Any regular language is a context-free language.

Proof. Let $N(Q, \Sigma, \delta, q_0, F)$ be a DFA that recognizes the given regular language. We convert N to a CFG $G(V, \Sigma, R, S)$ as follows.

- $V = Q$
- $R = \{q_i \to aq_j \mid q_i, q_j \in V \text{ and } \delta(q_i, a) = q_j\} \cup \{q_i \to \varepsilon \mid q_i \in F\}$
- $S = q_0$

We leave it to the reader to verify that this grammar generates the language that the DFA recognizes. ■

If a grammar generates a string in multiple ways, we say that the string is generated ambiguously from the grammar.

Definition 2.2. A derivation of a string w in a grammar G is a *leftmost derivation* if at every step the leftmost remaining variable is replaced.

Definition 2.3. A string w is derived *ambiguously* in a context-free grammar G if it has two or more different left-most derivations. Grammar G is *ambiguous* if it generates some string ambiguously.

Sometimes when we have an ambiguous grammar, we can find an unambiguous grammar that generates the same language. Some languages, however, can only be generated by ambiguous grammars. Such languages are called *inherently ambiguous*.

It is often very useful to represent context-free grammars in a simplified form.

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

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

where a is any terminal, A is any variable and B, C are any variable other than the start variable. In addition, we permit the rule $S \to \varepsilon$, where S is the start variable.

Theorem 2.2. Any context-free language can be generated by a context-free grammar in Chomsky normal form.

Proof. We can convert any CFG G into Chomsky normal form as follows.

1. Add a new variable S_0 and the rule $S_0 \to S$, where S was the original start variable. This is done to ensure that the start variable is not on the right side of any rule.
2. Next, we take care of all ε-rules, that is, rules with ε on the right side. We remove any ε-rule $A \to \varepsilon$, where $A \neq S$. Then for each occurrence of A on the right side of a rule, we add a new rule with that occurrence deleted. We repeat this until there are no ε rules not involving S.
3. We then handle all unit rules. We remove a unit rule $A \to B$. Then wherever a rule $B \to u$ appears, we add the rule $A \to u$ unless this was a unit rule previously removed. Here, u is a string of variables and terminals. We repeat this until there are no unit rules.
4. Given any rule $A \to u_1 u_2 \cdots u_k$, where $k > 2$ and each u_i is a variable or terminal symbol, with the rules $A \to u_1 A_1, A_1 \to u_2 A_2, \ldots,$ and $A_{k-2} \to u_{k-1} u_k$ (The A_is are new variables).
5. Finally, if we have any rule $A \to u_1 u_2$, we replace any terminal u_i with the new variable U_i and add the rule $U_i \to u_i$.

It can be checked that the language of this grammar is the same as that of the original grammar. ■

Exercise 2.4. Convert to the following CFG to Chomsky normal form.

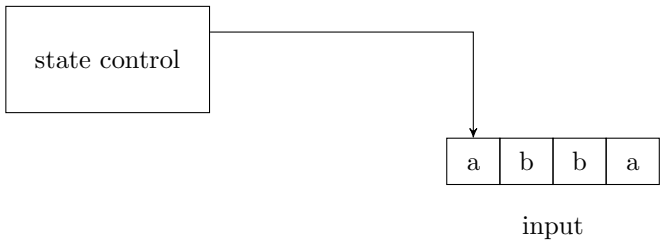
$$S \to ASA \mid aB$$
$$A \to B \mid S$$
$$B \to b \mid \varepsilon$$

Solution. Performing the algorithm given in the above proof yields the following CFG.

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

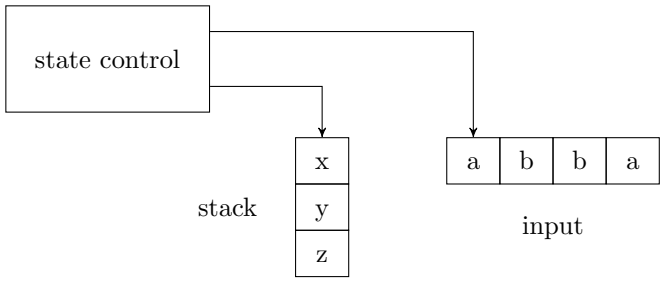
2.2 Pushdown Automata

We shall now introduce another computational model called a *pushdown automaton*. These automata are like DFAs, but they have an extra component called a *stack*, which provides additional memory beyond that of just the DFA. This allows the automaton to recognize certain non-regular languages. Pushdown automata are equivalent in power to context-free grammars.
The schematic of a finite automaton can be understood as follows:



The "state control" represents the states and the transition function, and the tape contains the input string, which is read character by character. The arrow points at the next character to be read.

Similarly, a pushdown automaton can be understood as follows.



In addition to the finite automaton-like structure it has, it also has a stack on which which symbols can be written down and read back later (The concept of a stack is hopefully familiar to the reader). This stack, which has infinite memory, is what enables the pushdown automaton to recognize languages such as $\{0^n 1^n \mid n \geq 0\}$ because it can store the number of 0s it has seen on the stack. The "pushdown" in pushdown automaton corresponds to the stack structure.

Unlike DFAs and NFAs, deterministic pushdown automata and nondeterministic pushdown automata are *not* equivalent. However, as nondeterministic pushdown automata are equivalent to context-free grammars, we shall focus on them for the remainder of this subsection.

Example. Let us attempt to construct the pushdown automaton corresponding to the language L described in 1.10. We do so as follows.

- Start in a state q_0 that represents a guess that we have not yet seen the end of the w in the definition of L. While in state q_0, we read symbols and push them onto the stack.
- At any time, we may guess that we have reached the end of w. Since the automaton is nondeterministic, we guess that we have reached the end of w by going to state q_1, and also stay in q_0 and continue to read inputs.
- Once in state q_1, we look at the input symbol and compare it to the topmost symbol on the stack. If they are the same, we pop it. Otherwise, the branch dies.
- If we empty the stack, then we have seen something of the form $ww^{\mathcal{R}}$, so we accept.

We shall now formally define a nondeterministic pushdown automaton.

Definition 2.5. A *nondeterministic pushdown automaton* is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ where Q, Σ, Γ, F are all finite sets, and

1. Q is the (finite) *set of states*,
2. Σ is the (finite) *input alphabet*,
3. Γ is the (finite) *stack alphabet* (this is the set of elements we can push onto the stack),
4. $\delta : Q \times \Sigma_\varepsilon \times \Gamma \to \mathcal{P}(Q \times \Gamma^*)$ is the *transition function*,
5. $q_0 \in Q$ is the *start state*,
6. $Z_0 \in \Gamma$ is a particular symbol called the *start symbol*, which initially appears on the stack, and
7. $F \subseteq Q$ is the *set of accept states*.

We shall abbreviate "Nondeterministic Pushdown Automaton" as NPDA.
In the above definition, the transition function is the main thing that is different from our usual definition of an NFA. In one transition, it:

1. consumes from input the symbol used in the transition (if the symbol is ε, then no input is consumed),
2. goes to a new state, and
3. replaces the symbol at the top of the stack with a string. Note that the string could also be ε, which means that we pop the stack.

Given this, the correspondence to the above definition is clear.
We require Z_0 in the above definition of a stack so that we can know when the stack is empty. Note that it is equivalent to use a specific symbol that we push in the beginning to signify the bottom of the stack. Some books use this definition of the NPDA instead. Next, we shall define what it means for an NPDA to recognize a string.

Definition 2.6. An NPDA $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is said to *accept* a string w if w can be written as $w = w_1 w_2 \cdots w_m$, where each $w_i \in \Sigma_\varepsilon$ and sequence of states $r_0, r_1, \ldots, r_m \in Q$ and strings $s_0, s_1, \ldots, s_m \in \Gamma^*$ exist that satisfy the following three conditions.

1. $r_0 = q_0$ and $s_0 = Z_0$.
2. For $i = 0, 1, \ldots, m-1$, we have $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, where $s_i = at$ and $s_{i+1} = bt$ for some $a \in \Gamma$ and $b, t \in \Gamma^*$.
3. $r_m \in F$.

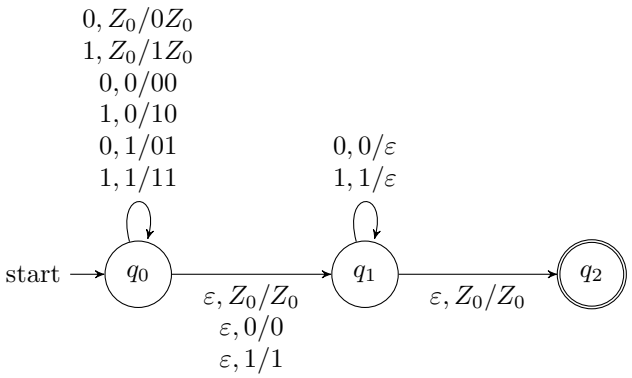
Exercise 2.5. Express 2.2 in the form given in the definition of an NPDA.

Solution. The PDA can be expressed as $P = (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, Z_0\}, \delta, q_0, Z_0, \{q_2\})$, where

$$\begin{aligned}
\delta(q_0, a, Z_0) &= \{(q_0, aZ_0)\} \quad \text{for all } a \in \{0, 1\} \\
\delta(q_0, a, b) &= \{(q_0, ab)\} \quad \text{for all } a, b \in \{0, 1\} \\
\delta(q_0, \varepsilon, a) &= \{(q_1, a)\} \quad \text{for all } a \in \{0, 1, Z_0\} \\
\delta(q_1, a, a) &= \{(q_1, \varepsilon)\} \quad \text{for all } a \in \{0, 1\} \\
\delta(q_1, \varepsilon, Z_0) &= \{(q_2, Z_0)\}
\end{aligned}$$

Similar to how we express NFAs and DFAs as graphs, NPDAs can also be expressed as graphs, where in addition to the way we draw the NFA structure, we also write what happens to the stack as follows. An arc labelled $a, X/\alpha$ from state q to p means that $(p, \alpha) \in \delta(q, a, X)$. That is, it tells what input is used (a), and the old and new tops of the stack (X and α respectively).

So for instance, the PDA described in 2.5 is depicted by the following diagram.



Exercise 2.6. Construct the NPDA that recognizes the language $L = \{a^i b^j c^k \mid i = j \text{ or } j = k\}$

It is also useful to represent a PDA at some point of time by a triple (q, w, γ), where q is the state, w is the remaining input, γ is the stack contents. We conventionally show that top of the stack at the left end of γ. Such a triple is called an *instantaneous description*, or ID of the automaton.

Let $V = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be an NPDA. Define $\vdash_P$, or just $\vdash$ when P is understood, as follows. Suppose $(p, \alpha) \in \delta(q, a, X)$. Then for all strings $w \in \Sigma^*, \beta \in \Gamma^*$, we write

$$(q, aw, X\beta) \vdash (p, w, \alpha\beta)$$

This notation, called the *"turnstile"* notation, represents the transition between different IDs of the NPDA. Note that w and β do not influence the transition, they are merely carried along.
We also use $\vdash_P^*$ or $\vdash^*$ to represent 0 or more moves of the NPDA. That is, $I \vdash^* I$ for any ID I and $I \vdash J$ if there exists a state K such that $I \vdash K$ and $K \vdash^* J$.

We shall call a sequence of IDs a *computation*. We have the following.

- If a computation is legal, then the computation formed by adding the same additional input string to the end of the input in each ID is also legal.
- If a computation is legal, then the computation formed by adding the same additional string below the stack of each ID is also legal.
- If a computation is legal, and some tail of the input is not consumed, we can remove this tail from each ID and the resulting computation will still be legal.

These three points just say that information that the NPDA does not look at does not affect its computation.

Theorem 2.3. If $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is an NPDA and $(q, x, \alpha) \vdash_P^* (p, y, \beta)$, then for any strings $w \in \Sigma^*, \gamma \in \Gamma^*$,

$$(q, xw, \alpha\gamma) \vdash_P^* (p, yw, \beta\gamma).$$

Proof. This proof is trivial and is left as an exercise to the reader. (Perform an induction on the number of steps in the sequence of IDs) ■

Using this notation, we can alternatively formulate the definition of the language recognized by a language as follows. Let $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be an NPDA. Then

$$L(P) = \{(q_0, w, Z_0) \vdash_P^* (q, \varepsilon, \alpha) \mid q \in F \text{ and } \alpha \in \Gamma^*\}$$

The above condition is called *acceptance by final state*, which is exactly what it is.

Definition 2.7. Let $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be an NPDA. We define

$$N(P) = \{w \mid (q_0, w, Z_0) \vdash^* (q, \varepsilon, \varepsilon)\}$$

The above set represents the set of input strings that when consumed, empty the stack as well. This is called *acceptance by empty stack*.
The following two theorems shows how the above two acceptances are intimately related.

Lemma 2.4. If $L = N(P_N)$ for some NPDA $P_N = (Q, \Sigma, \Gamma, \delta_N, q_0, Z_0)$, then there is an NPDA P_F such that $L(P_F) = L$.

Proof. We first set the start symbol of P_F as some $X_0 \notin \Gamma$. If we see X_0 on the stack for some input, then it means that P_N would empty the stack for that same input. We also set the start state of P_F as some $p_0 \notin Q$, whose sole purpose is to push Z_0 onto the stack and send it to q_0. Then, P_F simulates P_N, until the stack of P_N is empty. We also create another state $p_f \notin Q$, which is the (unique) accepting state of P_F. P_F goes to p_f if P_N would have emptied the stack for that input (that is, it has X_0 on the top of the stack). That is,

$$P_F = (Q \cup \{p_0, p_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta_F, p_0, X_0, \{p_F\})$$

where δ_F is given as follows.

1. $\delta_F(p_0, \varepsilon, X_0) = \{q_0, Z_0X_0\}$. This pushes Z_0 onto the stack and sends P_F to q_0.
2. $\delta_N(p, A, X) \subseteq \delta_F(p, a, X)$ for all $q \in Q, a \in \Sigma_\varepsilon$ and $X \in \Gamma$. This makes P_F behave like P_N.
3. $\delta_F(p, a, X)$ also contains $\{(p_f, \varepsilon)\}$ for $q \in Q, a = \varepsilon$ and $X = X_0$. This sends P_F to p_f if P_N would have emptied the stack for the same input.

We must show that $w \in L(P_F)$ if and only if $w \in N(P_N)$.

(If) This is reasonably straightforward. We have that $(q_0, w, Z_0) \vdash^*_{P_N} (q, \varepsilon, \varepsilon)$.

Using 2.3 gives $(q_0, w, Z_0X_0) \vdash^*_{P_N} (q, \varepsilon, X_0)$.

Since P_F has all the moves of P_N, we also have $(q_0, w, Z_0X_0) \vdash^*_{P_F} (q, \varepsilon, X_0)$. Along with the initial and final moves, we have

$$(p_0, w, X_0) \vdash_{P_F} (q_0, w, Z_0X_0) \vdash^*_{P_F} (q, \varepsilon, X_0) \vdash_{P_F} (p_f, \varepsilon, \varepsilon).$$

Thus P_F accepts w by final state.

(Only if) The first and third rules of δ_F give very limited ways to accept w by final state. We can only use the third rule at the last step and even then, it must have X_0 on the top of the stack. As X_0 only appears at the bottom-most position in the stack, and it must be inserted at the first step, any computation of P_F that accepts w must look like the above computation. Further, the entire computation except the first and last steps must be like a computation of P_N with X_0 below the stack. (Why?) We conclude that $(q_0, w, Z_0) \vdash^*_{\{} P_N\}(q, \varepsilon, \varepsilon)$, that is, $w \in N(P_N)$.

■

Lemma 2.5. If $L = L(P_F)$ for some NPDA $P_F = (Q, \Sigma, \Gamma, \delta_F, q_0, Z_0)$, then there is an NPDA P_N such that $N(P_N) = L$.

Proof. Similar to the previous proof, we introduce $p_0, p_f \notin Q$ and $X_0 \notin \Gamma$. Whenever P_F enters an accepting state after consuming input w, the corresponding system in P_N empties its stack. That is, let

$$P_N = (Q \cup \{p_0, p_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta_N, p_0, X_0)$$

where δ_N is given as follows.

1. $\delta_N(p_0, \varepsilon, X_0) = \{(q_0, Z_0X_0)\}$. This is the first step where Z_0 is pushed onto the stack so that P_N may behave like P_F.
2. $\delta_F(q, a, Y) \subseteq \delta_N(q, a, Y)$ for all $q \in Q, a \in \Sigma_\varepsilon$ and $Y \in \Gamma$. This makes it behave like P_F.

3. $\delta_N(q, \varepsilon, Y)$ also contains (p_f, ε) for $q \in F$ and $Y \in \Gamma$. If the P_F part of P_N is in an accepting state, it goes to p_f.

4. $\delta_N(p_f, \varepsilon, X) = \{(p_f, \varepsilon)\}$. This repeatedly pops the symbol on the stack until it is empty.

We must now prove that $w \in P_N$ if and only if $w \in P_F$. The ideas are similar to those used in the proof of 2.4 so we leave it as an exercise to the reader. ■

Theorem 2.6. Let L be a language. L is accepted by final state by some NPDA if and only if it is accepted by empty stack by some NPDA.

Proof. This follows directly from 2.4 and 2.5. ■