Automata and Languages

Amit Rajaraman

March 2020

Contents

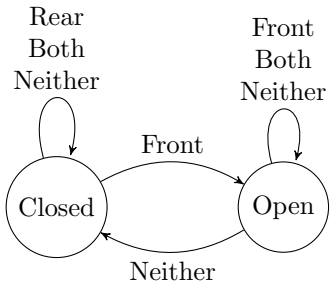
§1. Regular Languages

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:

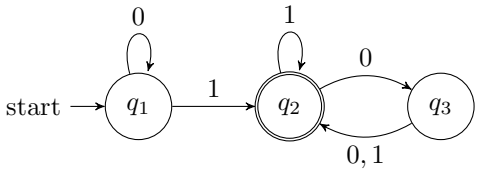


It can also be represented by the following table:

	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 obvious question to ask is: 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*: $A \circ B = \{xy \mid x \in A \text{ and } y \in B\}$.
- *Kleene Star*: $A^* = \{x_1 x_2 \cdots x_k \mid k \geq 0 \text{ and each } x_i \in A\}$

These operations are called regular operations as the class of regular languages are closed under these operations. Henceforth, we shall refer to the Kleene star operation as just the star operation.

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.

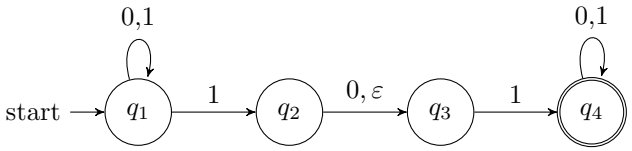
Proof. Let $M_1(Q_1, \Sigma_1, \delta_1, q_{01}, F_1)$ and $M_2(Q_2, \Sigma_2, \delta_2, q_{02}, F_2)$ recognize A_1 and A_2 respectively. Set $\Sigma = \Sigma_1 \cup \Sigma_2$.
We construct the finite automaton $M(Q, \Sigma, \delta, q_0, F)$ that recognizes $A_1 \cup A_2$ by setting $Q = Q_1 \times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



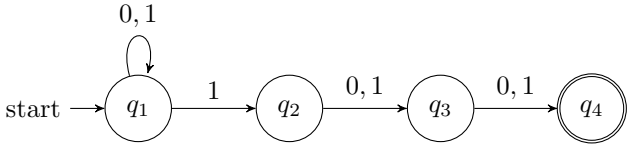
The difference between a DFA and an NFA is immediately apparent. In the above example, there are multiple arrows from q_1 corresponding to input 1 and no arrow from q_2 corresponding to 1. We also see the addition of ε as input, which is not in the alphabet.

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.*

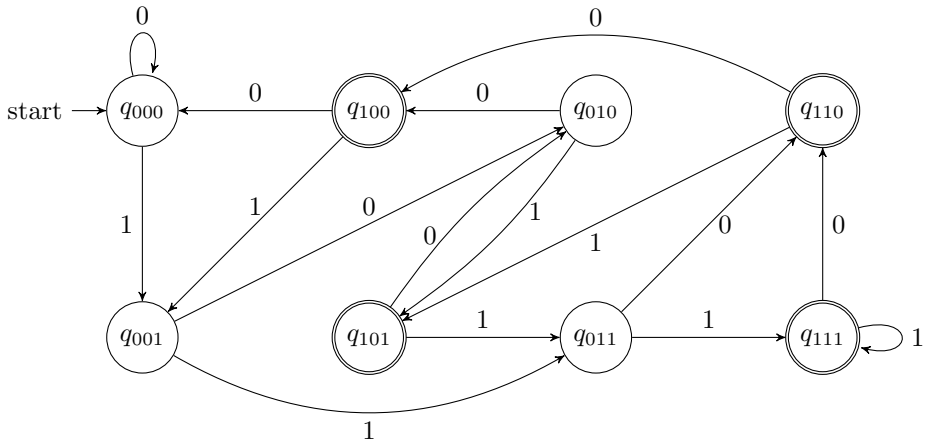
Nondeterminism is a sort of parallel computing where many "threads" are running simultaneously. The NFA splitting corresponds to the process of "forking" into several children, each proceeding separately. If any of these processes accepts, the entire computation accepts.
We can also think of an NFA as a tree of possibilities where the tree splits at each point where the machine has more than one choice.

But why are NFAs 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 plain as day 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.
In arithmetic, there is a precedence order in the operations wherein we give $\times$ 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 equal to

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, or
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 1.1

(a) $\emptyset$

(b) $\{\varepsilon\}$

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

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

(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) $R \cup \emptyset = R$

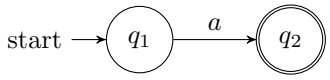
(b) $R \circ \{\varepsilon\} = R$

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.

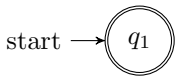
Lemma 1.6. If a language is described by a regular expression, it is regular.

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

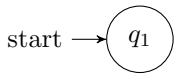
1. $R = a$ for some a in Σ. Then $L(R) = \{a\}$. The following NFA recognizes $L(R)$.



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.

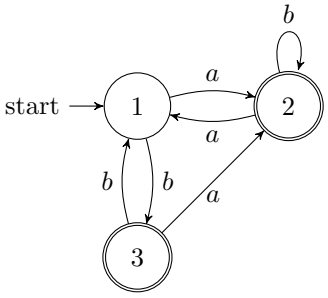
The induction hypothesis merely says that when the algorithm calls itself recursively on G', the resulting regular expression is equivalent to G' (because G' has $k-1$ states). As G is equivalent to G', G must also be equivalent to the resulting regular expression, namely `CONVERT`(G). ■

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 1.2

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 would need to count the number of 0s processed so far (which 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 1.3

If B is a regular language, then consider $s = 0^p1^p$, where p is the pumping length. Taking x, y, z to represent the same x, y, z as in the Pumping Lemma, since $|xy| \leq p$, y only consists of 0s. Then xy^2z will have more 0s than 1s so it is not in 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.

$$\begin{aligned} A &\to 0A1 \\ A &\to B \\ B &\to \# \end{aligned}$$

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 (completely different substitutions), not two different derivations (loosely, this means the same sequence of steps in a different order). (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 exists a sequence $u_1, u_2, \ldots, u_k$ of strings of terminals and variables some 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 2.1

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.

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

Lemma 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 of the DFA. ■

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 in a grammar G is a *leftmost derivation* if at every step the leftmost remaining variable is replaced.

Rightmost derivations are defined similarly.

Definition 2.3. Let (V, Σ, R, S) be a context-free grammar. Any string in $(V \cup \Sigma)^*$ that can be derived from the start symbol S is called a *sentential form*.

In the above definition, if there exists a leftmost derivation from S to the sentential form, it is called a *left-sentential form*. Right-sentential forms are defined similarly.

Definition 2.4. 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 2.5. 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.

$$\begin{aligned} S &\to ASA \mid aB \\ A &\to B \mid S \\ B &\to b \mid \varepsilon \end{aligned}$$

Solution 2.2

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

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

Definition 2.6 (Greibach normal form)**.** A context-free grammar is in *Greibach normal form* if every rule is of the form

$$C \to a\alpha$$

where C is a variable, a is a terminal, and α is some string in V^*. In addition, we permit the rule $S \to \varepsilon$, where S is the start variable.

Theorem 2.3. Any context-free language can be generated by a context-free grammar in Greibach normal form.

Proof. The proof of the above has three steps: eliminating any productions, left recurstion, and ordering the variables of V to eliminate non-Greibach rules.

1. Suppose we have the rule $A \to \alpha_1 B \alpha_2$. If we also have the rule $B \to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_r$, we shall replace the first rule with the rules $A \to \alpha_1 \beta_i \alpha_2$ for $1 \leq i \leq r$.

2. Suppose we have the rules $A \to A\alpha_i$ for $1 \leq i \leq r$ and $A \to \beta_i$ for $1 \leq i \leq s$, where the β_i do not begin with A. We introduce a new variable $B \in V$, and the rules $A \to \beta_i B$, and then the rules $B \to \alpha_i B \mid \alpha_i$.

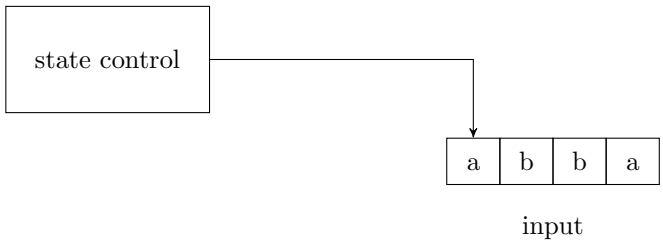
3. For example, the rules $A_1 \to A_2 A_3, A_2 \to A_3 A_1, A_3 \to A_1 A_2$ have a hidden left recursion (we can get $A_3 \to A_3 A_1 A_3 A_2$).

■

GNF is incredibly useful to show that any language recognized by a CFG is recognized by a PDA. Indeed, we may establish a correspondence between the non-terminals and the stack.

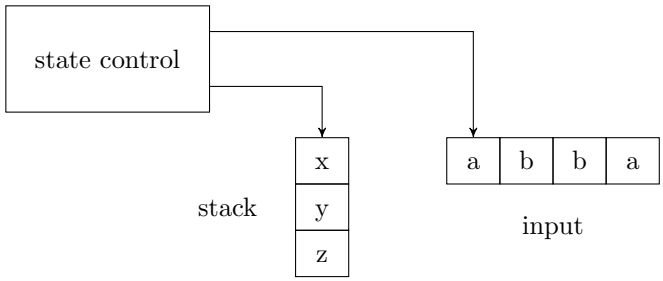
2.2. Pushdown Automata

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 2.7. 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 2.8. 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 $q_0, q_1, \ldots, q_m \in Q$ and strings $s_0, s_1, \ldots, s_m \in \Gamma^*$ exist that satisfy the following three conditions.

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

Some definitions instead accept when the final stack is empty (so there is no concept of a "final state" at all). We shall see in Lemmas 2.5 and 2.6 that both of these are equivalent.

Exercise 2.5. Express the example we described before defining an NPDA in the form given in the definition of an NPDA.

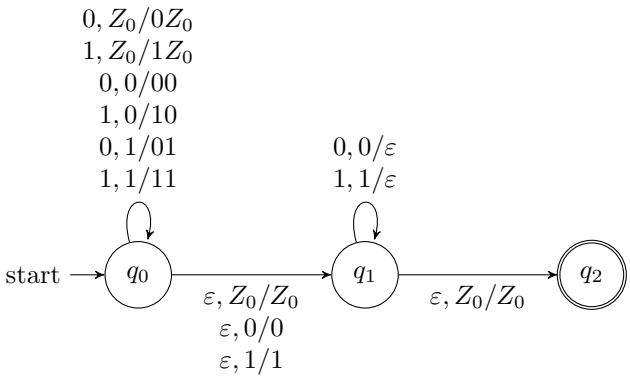
Solution 2.3

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, and γ is the stack contents. We conventionally show the 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 2.4. 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

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

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

Definition 2.9. 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 2.5. 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 2.4 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 2.6. 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.

The ideas used in proving that $w \in P_N$ if and only if $w \in P_F$ are similar to those used in the proof of 2.5 so we leave it as an exercise to the reader. ■

Theorem 2.7. 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 2.5 and 2.6. ■

2.3. Equivalence of Context-Free Grammars and Pushdown Automata

In this subsection, we shall show the equivalence of context-free languages and languages that are recognized (by final state) by some NPDA. To do so, we shall instead consider those languages that are recognized by empty stack by some NPDA.

Any left-sentential that is not a terminal string can be written as $xA\alpha$, where A is the leftmost variable, x is the string of whatever terminals appear to its left, and α is the string of terminals and variables that appear to its right. Then $A\alpha$ is called the *tail* of this left-sentential form. A left-sentential form that is a terminal string is said to have tail ε.

Given a CFG, we shall attempt to construct an NPDA that simulates its leftmost derivations. We shall do so by "guessing" the sequence of left-sentential forms that the CFG takes to reach a given terminal string w. The tail of each sentential form $xA\alpha$ appears on the stack (with A on top). x is represented by our having consumed of it from the input. That is, if $w = xy$, then the input consists of just y.

Now suppose the NPDA is in ID $(q, y, A\alpha)$ representing left-sentential form $xA\alpha$. It guesses the rule used to expand A as $A \to \beta$. Now again, the terminals at the start of the string $\beta\alpha$ need to be removed to expose the tail. These terminals are compared against the next input symbols, to ensure that our guess was correct. If it is not, then this branch of the NPDA dies. If we are able to guess a leftmost derivation of w, then we shall eventually reach the left-sentential form w. At that point, the stack is empty as the tail is ε and we accept by empty stack.

We shall now make the above informal construction rigorous as follows. Let $G = (V, \Sigma, R, S)$ be a CFG. Construct an NPDA P as follows:

$$P = (\{q\}, \Sigma, V \cup \Sigma, \delta, q, S)$$

where δ is defined by

$$\delta(q, \varepsilon, A) = \{(q, \beta) \mid (A, \beta) \in R\} \text{for each variable } A.$$

$$\delta(q, a, a) = \{(q, \varepsilon)\} \text{ for each terminal } a.$$

Lemma 2.8. Let G be a CFG and P be an NPDA constructed from G as above. Then $N(P) = L(G)$.

Proof. We shall prove that a string w is in $N(P)$ if and only if it is in $L(G)$.

(If) Let $w \in L(G)$. Then w has leftmost derivation

$$S = \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \Rightarrow \gamma_n = w.$$

We shall show by induction on i that $(q, w, S) \vdash_P^* (q, y_i, \alpha_i)$, where y_i and α_i are as follows.

Let α_i be the tail of γ_i and $\gamma_i = x_i\alpha_i$. Then y_i is the string such that $x_iy_i = w$.

Basis Step: For $i = 1$, we have $\gamma_1 = \alpha_1 = S$, $x_1 = \varepsilon$, and $y_1 = w$. We then trivially have $(q, w, S) \vdash^* (q, w, S) = (q, y_1, \alpha_1)$.

Induction: We shall assume that $(q, w, S) \vdash^* (q, y_i, \alpha_i)$ and show that $(q, w, S) \vdash^* (q, y_{i+1}, \alpha_{i+1})$. Since α_i is a tail, it begins with a variable A.

In the derivation, the step $\gamma_i \Rightarrow \gamma_{i+1}$ involves replacing A with some string β. Using the construction of P, the first part allows us to replace A at the top of the stack with β and the second part allows us to match terminals at the top of the stack with subsequent input symbols. Then we reach $(q, y_{i+1}, \gamma_{i+1})$, which represents the next left-sentential form γ_{i+1}.

Finally, note that $\alpha_n = \varepsilon$. Thus $(q, w, S) \vdash^* (q, \varepsilon, \varepsilon)$, which proves that P accepts w by empty stack.

(Only if) Here, we need to prove that if $(q, x, A) \vdash^* (q, \varepsilon, \varepsilon)$, then $A \overset{*}{\Rightarrow} x$. We shall do so by induction on the number of moves taken by P.

Basis Step: If only one move is taken, then the only possibility is that $A \to \varepsilon$ is a rule of G and $x = \varepsilon$. This implies that $A \Rightarrow \varepsilon$.

Induction: Suppose P takes $n > 1$ moves. The move taken in the first step must be of the first type in the definition of P. Let the rule that is substituted be $A \to Y_1 Y_2 \cdots Y_k$, where each Y_i is either a variable or a terminal. The next $n-1$ steps must consume x from the input and pop each of the elements $Y_1, Y_2, \ldots, Y_k$ from the stack. Let us break x as $x_1 x_2 \cdots x_k$, where x_1 is the portion of the input consumed until Y_1 is removed from the stack, x_2 is the next portion that is consumed until Y_2 is removed from the stack, and so on.

Then we have that $(q, x_i, Y_i) \vdash^* (q, \varepsilon, \varepsilon)$ for each i. As each $x_i < n$, we can use the inductive hypothesis to get that $Y_i \overset{*}{\Rightarrow} x_i$.

Then we have the following derivation:

$$A \Rightarrow Y_1 Y_2 \cdots Y_k \overset{*}{\Rightarrow} x_1 Y_2 \cdots Y_k \overset{*}{\Rightarrow} \cdots \overset{*}{\Rightarrow} x_1 x_2 \cdots x_k = x.$$

This completes the proof.

■

Now that we have gone one way from grammars to NPDAs, we shall provide a construction in the other direction to prove their equivalence.
That is, given any NPDA P that recognizes language $L(P)$, we must construct a CFG G that also recognizes language $L(P)$.

Lemma 2.9. Let $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$ be an NPDA. Then there is a context-free grammar G such that $L(G) = N(P)$.

Similar to the second part of the previous proof, let us have a sequence of symbols $Y_1, Y_2, \ldots, Y_k$ that must be popped. Let some input x_1 be read while Y_1 is popped. x_2 is the next portion that is consumed until Y_2 is popped, and so on. Note that here, by "popping", we do not mean a single step that is the usual meaning of popping. We instead mean possibly multiple steps that result in Y_1 being removed from the stack.
The variables of the CFG we shall construct will represent "events" that the NPDA changes from state p at the beginning to q when X is removed from the stack. This composite symbol is denoted $[pXq]$. Note that this is *a single variable*. We make this rigorous in the following proof.

Proof. We construct a CFG $G = (V, \Sigma, R, S)$ as follows. S is the special start symbol. We also have

$$V = \{S\} \cup \{[pXq] \mid p, q \in Q \text{ and } X \in \Gamma\}.$$

For all states p, $S \to [q_0 Z_0 p]$ is a rule. According to the intuition we provided, this generates all strings w that cause P to pop Z_0 when going from q_0 to p. That is, it represents all strings w that cause P to empty its stack.

Let $\delta(q, a, X)$ contain $(r, Y_1 Y_2 \cdots Y_k)$ where $a \in \Sigma_\varepsilon$ and $k \geq 0$ (in the case where $k = 0$, we take the pair (r, ε) instead).

Then for all lists of states $r_1, r_2, \ldots, r_k$, G has the rule

$$[qXr_k] \to a[rY_1 r_1][r_1 Y_2 r_2] \cdots [r_{k-1} Y_k r_k].$$

This represents that one way to go from q to r_k while popping X is to read a, then use some input to pop Y_1 while going from r to r_1, then read some more input that pops Y_2 while going from r_1 to r_2, and so on.

Let us now prove that the CFG we have constructed satisfies $L(G) = N(P)$, that is,

$$[qXp] \overset{*}{\Rightarrow} w \text{ if and only if } (q, w, X) \vdash^* (p, \varepsilon, \varepsilon).$$

(If) Suppose $(q, w, X) \vdash^* (p, \varepsilon, \varepsilon)$. We must show that $[qXp] \overset{*}{\Rightarrow} w$. We shall do so by induction on the number of steps.

Basis Step: Only one step is involved. Then $(p, \varepsilon) \in \delta(q, w, X)$ and w is in Σ_ε. By the construction of G, $[qXp] \to w$ is a rule, so $[qXp] \Rightarrow w$.

Induction: Let $n > 1$ steps be involved. The first step must look like

$$(q, w, X) \vdash (r_0, x, Y_1 Y_2 \cdots Y_k) \vdash^* (p, \varepsilon, \varepsilon)$$

where $w = ax$ for some $a \in \Sigma_\varepsilon$. It follows that

$$(r_0, Y_1 Y_2 \cdots Y_k) \in \delta(q, w, X).$$

By the construction of G, there is a rule

$$[q X r_k] \to a[r_0 Y_1 r_1][r_1 Y_2 r_2] \cdots [r_{k-1} Y_k r_k],$$

where $r_k = p$ and $r_0, r_1, \ldots, r_{k-1} \in Q$.

Let $x = w_1 w_2 \cdots w_k$, where each w_i is the input consumed when Y_i is popped. Then we have that $(r_{i-1}, w_i, Y_i) \vdash^* (r_i, \varepsilon, \varepsilon)$. We can use the inductive hypothesis on each of these steps to conclude that for each i, $[r_{i-1} Y_i r_i] \overset{*}{\Rightarrow} w_i$. With $r_k = p$, we may put these derivations together as follows to get the required result.

$$\begin{aligned}
[q X r_k] &\Rightarrow a[r_0 Y_1 r_1][r_1 Y_2 r_2] \cdots [r_{k-1} Y_k r_k] \\
&\overset{*}{\Rightarrow} a w_1 [r_1 Y_2 r_2][r_2 Y_2 r_3] \cdots [r_{k-1} Y_k r_k] \\
&\vdots \\
&\overset{*}{\Rightarrow} a w_1 w_2 \cdots w_k = w
\end{aligned}$$

(Only if) We must show that if $[qXp] \overset{*}{\Rightarrow} w$, then $(q, w, X) \vdash^* (p, \varepsilon, \varepsilon)$. We shall do so by induction on the number of steps in the derivation.

Basis Step: Only one step is involved. In this case, $[qXp] \to w$ is a rule. The only way this is possible is if there is a transition of P from q to p where X is popped. That is, $(p, \varepsilon) \in \delta(q, w, X)$. But then we have $(q, w, X) \vdash^* (p, \varepsilon, \varepsilon)$.

Induction: Let $n > 1$ steps be involved. Let $p = r_k$ and the first sentential form be as follows:

$$[q X r_k] \Rightarrow a[r_0 Y_1 r_1][r_1 Y_2 r_2] \cdots [r_{k-1} Y_k r_k] \overset{*}{\Rightarrow} w.$$

This is because $(r_0, Y_1 Y_2 \cdots Y_k) \in \delta(q, a, X)$. Break w as $w = a w_1 w_2 \cdots w_k$ such that $[r_{i-1} Y_i r_i] \overset{*}{\Rightarrow} w_i$ for each i. Then for all i,

$$(r_{i-1}, w_i, Y_i) \vdash^* (r_i, \varepsilon, \varepsilon).$$

Using 2.4, we have

$$(r_{i-1}, w_i w_{i+1} \cdots w_k, Y_i) \vdash^* (r_i, w_{i+1} w_{i+2} \cdots w_k, \varepsilon).$$

Putting all these together, we have

$$\begin{aligned}
(q, a w_1 w_2 \cdots w_k, X) &\vdash (r_0, w_1 w_2 \cdots w_k, Y_1 Y_2 \cdots Y_k) \\
&\vdash^* (r_1, w_2 w_3 \cdots w_k, Y_2 Y_3 \cdots Y_k) \\
&\vdots \\
&\vdash^* (r_k, \varepsilon, \varepsilon)
\end{aligned}$$

This completes our proof as $r_k = p$.

■

Theorem 2.10. The class of languages recognized by nondeterministic pushdown automata is equal to the class of languages generated by context-free grammars.

Proof. This is an immediate corollary of 2.9 and 2.8. ■

2.4. The Pumping Lemma

Lemma 2.11 (Pumping Lemma for Context-Free Languages)**.** If L is a context-free 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 five parts $s = uvwxy$, satisfying the following conditions:

1. for each $i \geq 0$, $uv^iwx^iy \in L$,
2. $|vx| > 0$, and
3. $|vwx| \leq p$.

§3. Turing Machines

3.1. Introduction

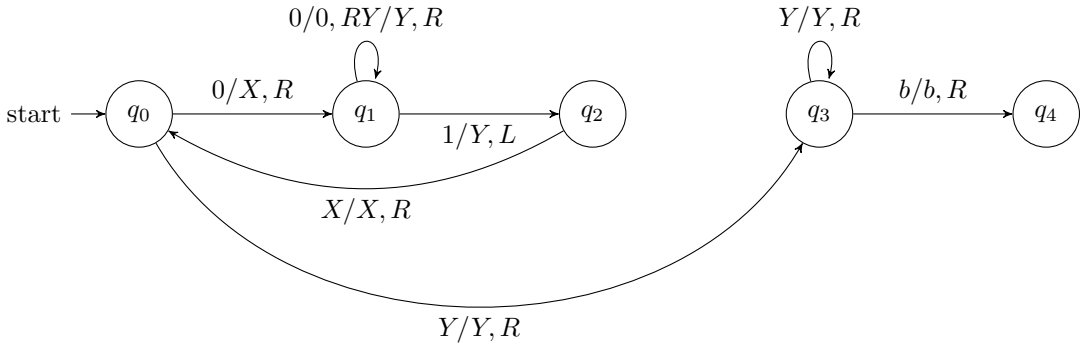
Before we get to the more rigorous part of Turing machines, we require some setup.
Recall that a set is said to be *countable* if it is in bijection with $\mathbb{N}$. For example, one may show that $\mathbb{Z}$, $\mathbb{N}^2$, and in general, $\mathbb{N}^k$ (for any fixed $k \geq 1$) is countable.
One may also show that a countable union of countable sets is countable. A consequence of this is that the set of all finite sequences in $\mathbb{N}$ is countable. However, it turns out that the set of all (countable) sequences in $\mathbb{N}$ is not countable!
Countable sets are also often referred to as "enumerable".

Inspired by this but with a more computer-science flavour, one may define the following.

Definition 3.1. A set S is said to be *computably enumerable* if there exists an algorithm that enumerates the members of S, that is, a sequence $s_1, s_2, s_3, \cdots$ (which is infinite if S is), such that any element of S is a member of the sequence.

One question that might immediately come to mind is: what is the difference between a computably enumerable set and an enumerable set?

Similar to how a NPDA is just an NFA with an additional stack to store information, a Turing machine is an NFA with an "input tape" – this is an infinitely long string. If the input word is $w_1 w_2 \cdots w_n$, this input tape is initially of the form $w_1 w_2 \cdots w_n bbb \cdots$, where b is a special "blank" symbol. If we see the input w_i when in state q_j, we do three things – move to a state q_k, replace the symbol w_i with some other symbol X, and move the tape left or right. Such a transition is represented by labelling the arrow from q_j to q_k as $w_i/X, R$ or $w_i/X, L$. If the machine "halts" (there are no moves possible) at a particular state, we accept if we are in an accepting state.
So, let us now get to an example of a Turing machine, constructing that one recognizes the (context-free) language $\{0^n 1^n : n \geq 1\}$. This Turing machine has tape alphabet $\{X, Y, 0, 1, b\}$.



Exercise 3.1. Construct a Turing machine that accepts

$$\{L = a^n b^n c^n : n \geq 1\}.$$

Definition 3.2 (Turing Machine)**.** A Turing machine is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, b, F)$, where

1. Q is the finite set of states,

2. $\Sigma \subseteq \Gamma \setminus \{b\}$ is the alphabet,
3. Γ is the tape alphabet,
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $b \in \Gamma$ is a special blank symbol, and
7. $F \subseteq Q$ is the set of final states.

There are several simple variations of Turing machines:

- Two-way infinite tapes,
- Multiple tape heads,
- Non-deterministic Turing machines,
- An output tape where we can only move right.

It turns out that all of these are equivalent to the usual Turing machine.

At any point of time, the Turing machine has a tape that contains a string of symbols with an infinite number of blanks on either side. Initially, this string is just the input, and the head is at the leftmost symbol.

Definition 3.3 (Instantaneous Description)**.** An *instantaneous description* of a Turing machine is a string of the form $\alpha q \beta$, where $\alpha, \beta \in \Gamma^*$ and $q \in Q$.

What this represents is that α, β are on the tape with an infinite number of blanks on either side, and q, the state the automaton is at, is just before the position pointed at by the head.
We further write that $\alpha q \beta \vdash \alpha' q' \beta'$ if a single step can take the first ID to the second.

Definition 3.4. A Turing machine is said to *accept* the word w if

$$q_0 w \vdash \cdots \vdash \alpha q_k \beta,$$

$q_k \in F$, and there is no move possible from the final ID (it *halts*).

Exercise 3.2. Construct a Turing machine that accepts

$$L = \{ww : w \in \{a, b\}^+\}.$$

In both DFAs and NPDAs, we had that the run time of the automaton is bounded above by some known quantity (the length of the word). In Turing machines however, this is not the case! A Turing machines is a decision procedure (outputting 'yes' or 'no') only if it halts for all inputs.
This is referred to as a *semi-decision procedure*.

Consider Hilbert's tenth problem – does a given polynomial (over possibly multiple variables) with integer coefficients have integer roots? Since it is possible to enumerate all possible integer tuples, it is possible to give a semi-decision procedure to solve this problem.
It in fact turns out that this problem is undecidable, as shown by Martin Davis, Yuri Matiyasevich, Hilary Putnam, and Julia Robinson.

Similar to how we had defined regular languages and context-free languages, we define the following class of languages.

Definition 3.5. A language $L \subseteq \Sigma^*$ is said to be *recursively enumerable* if it is recognized by some Turing machine.

An immediate question is: do there even exist languages that are not recursively enumerable? We shall answer this and deal with related questions in Subsection 3.5.

3.2. Type-0 Grammars

Type-0 grammars, usually referred to as *unrestricted grammars* are a far more powerful version of context-free grammars, with the difference being that the left-hand side of the rule can be a string instead of a single character. For example, if we have the rule $AB \to BA$, we can derive the string $ZBAXY$ from $ZABXY$. Consider the following grammar that recognizes the language

$$L = \{a^n b^n c^n : n \geq 1\}.$$

$$\begin{aligned} S &\to ABCS \mid ABCT_c \\ CA &\to AC \\ BA &\to AB \\ CB &\to BC \\ CT_c &\to T_c c \mid T_b c \\ BT_b &\to T_b b \mid T_a b \\ AT_a &\to T_a a \mid a. \end{aligned}$$

It turns out that Type-0 Grammars are equivalent to Turing machines!

3.3. Computable functions

Turing machines are not just restricted to recognizing languages. The same framework may be used to do several things:

- First is just language recognition which we have seen.
- Second is computation. Given a "simple" function from $\mathbb{N}^k \to \mathbb{N}$, we can begin the input tape with the input values separated by #s (to separate the input variables), and when the Turing machine halts, the tape has the output on it. This is easily extended to *partial* functions, which need not be defined on all of $\mathbb{N}^k$, by taking the output as valid iff it halts on an accepting state. A function that is not partial is said to be *total*. When a Turing machine is used in this manner, it is called a *transducer*.
- Third, we have enumeration. The tape begins empty, and an enumeration of a (countable) language is output. For example, if the language is $\{0^n 1^n : n \in \mathbb{N}\}$, the output is $01\#0011\#000111\#\cdots$ and the Turing machine never halts.

The second question presents an interesting question: what exactly does "simple" mean? More precisely, exactly which functions may be computed by Turing machines?

This leads to the subject of recursive function theory.
Before we move to the subject proper, let us define some functions and operators used.

Definition 3.6. Define the *primitive functions* as follows.

1. Given $k, n \in \mathbb{N}$, the *constant* function $C_n^k : \mathbb{N}^k \to \mathbb{N}$ is the constant function on k variables defined by
$$C_n^k(x_1, \ldots, x_k) = n$$
for all $x_1, \ldots, x_k \in \mathbb{N}$.
2. The *successor* function $S : \mathbb{N} \to \mathbb{N}$ is defined by
$$S(x) = x + 1$$
for all $x \in \mathbb{N}$.
3. Given $k, i \in \mathbb{N}$ with $i \leq k$, the *projection* function $P_i^k : \mathbb{N}^k \to \mathbb{N}$ is defined by
$$P_i^k(x_1, \ldots, x_k) = x_i$$
for all $x_1, \ldots, x_n \in \mathbb{N}$.

Further define the following operators.

1. The *composition* operator is defined as follows. Given a function $h : \mathbb{N}^m \to \mathbb{N}$ and m functions $g_1, \ldots, g_m : \mathbb{N}^k \to \mathbb{N}$, define the function
$$(h \circ (g_1, \ldots, g_m))(x_1, \ldots, x_k) = f(g_1(x_1, \ldots, x_k), \ldots, g_m(x_1, \ldots, x_k))$$
for all $x_1, \ldots, x_k \in \mathbb{N}$.
2. The *primitive recursion* operator is defined as follows. Given $k \in \mathbb{N}$, $g : \mathbb{N}^{k-1} \to \mathbb{N}$, and $h : \mathbb{N}^{k+1} \to \mathbb{N}$, define the function $\rho(g, h) : \mathbb{N}^k \to \mathbb{N}$ by
$$(\rho(g,h))(x_1, x_2, \ldots, x_k) = \begin{cases} g(x_2, \ldots, x_k), & x_1 = 0, \\ h(x_1 - 1, g(x_1 - 1, x_2, x_3, \ldots, x_n), x_2, x_3, \ldots, x_n), & \text{otherwise.} \end{cases}$$
3. The *minimization* operator is defined as follows. Given $k \in \mathbb{N}$, $f : \mathbb{N}^{k+1} \to \mathbb{N}$, the function $\mu(f) : \mathbb{N}^k \to \mathbb{N}$ is defined by
$$(\mu(f))(x_1, \ldots, x_k) = \min\{z : f(z, x_1, \ldots, x_k) = 0\}.$$
Note that the minimization operator need not be well-defined for a given f (the value of the function may be non-zero for all z). So, $\mu(f)$ is a partial function.

Using the above, we may define several classes of "computable" functions.

Definition 3.7. The class of *primitive recursive functions* is the smallest class of total functions that contains the primitive functions and is closed under composition and primitive recursion.
The class of *general recursive functions* is the smallest class of functions (total or partial) that contains the primitive functions and is closed under composition, primitive recursion, and minimization.
The class of *general recursive functions* is the smallest class of total functions that contains the primitive functions and is closed under composition, primitive recursion, and minimization.

For example, the function $f : (x, y) \mapsto x + y$ is defined by
$$f = \rho(P_2^2, S \circ P_2^3).$$

3.4. Recursively enumerable and recursive languages

Recall that recursively enumerable languages are those languages recognized by a Turing machine.

Proposition 3.1. There exist non-recursively enumerable languages over any alphabet Σ.

Proof. Clearly, it suffices to consider the case where $|\Sigma| = 1$, so let $\Sigma = a$. Observe that the set $\mathcal{P}(a^*)$ is not countable. Any Turing machine can be thought of as a string over $\{0, 1\}$, namely as the concatenation of the strings $0^i10^j10^k10^l10^s$, where for each transition from q_i to q_j with the lab $k/l, s$ (numbering the tape symbols and alphabet symbols ahead of time), with the transitions separated by 11s. The set of such strings is a subset of $\{0, 1\}^*$, so it (and thus the set of all Turing machines) is countable! The required follows. ■

These limits however, are only useful to us in the sense that we would like to know whether a given problem is within the limits or not. Indeed, the above argument may be used to show that there exist languages that cannot be recognized by any program (in C, say) either, because any program is a finite string of ASCII characters. Let us study our little nook of recursively enumerable languages a little more before moving to the limits. In fact, a program is weaker than a Turing machine since it has a finite amount of memory.

Definition 3.8 (Algorithm)**.** An *algorithm* is a Turing machine that halts for any input.
A language recognized by an algorithm is said to be a *recursive language*.

Nearly every language one might run into is recursive.

Theorem 3.2. The class of recursively enumerable languages is closed under union, intersection, concatenation and the Kleene star.
The class of recursive languages is closed under union, intersection, concatenation, the Kleene star, and complementation.

Proof. Let us begin by showing the closure of recursively enumerable languages under various operations.

1. Union. Create a two-tape Turing machine, with the two tapes simulating the Turing machines for each of the languages. Accept when either of the two simulations accepts – note that we forcibly halt the running of the other Turing machine after one accepts.
2. Intersection. Again, create a two-tape Turing machine, with each of the two tapes simulating the Turing machines for each of the languages. Accept when both simulations accept.
3. Concatenation. Create a two-tape Turing machine as follows. Given input w, non-deterministically guess a break as $w = xy$, move y to the second tape, and run the second Turing machine on y. Accept if both simulations accept.
4. Kleene star.The idea of this is identical to that of concatenation, except that we guess multiple breaks instead of just one.

Now, let us move on to the closure of recursive languages.

1. The construction for union and intersection is identical to that for recursively enumerable languages. Since both Turing machines halt for any input, so does the new Turing machine.
2. The construction for concatenation and Kleene star is identical as well.
3. Complementation. Let the original Turing machine run and accept iff it halts on a non-accepting state.

■

3.5. Limits of computation

A common theme of questioning is asking whether a given "object" has a certain "property". For example, given a graph G and an input number k, one may ask whether G has a clique of size k. In such a problem, the inputs (G, k) is referred to as an *instance* of the problem. Any instance of the problem may be encoded as a word over some alphabet, say $\{0, 1\}$, and based on this one can ask the question whether it is possible to construct a Turing machine that recognizes the words corresponding to all positive instances of the problem.

Definition 3.9. A problem is *decidable* if there is an algorithm to answer it. Otherwise, the problem is said to be *undecidable*.

As seen in Proposition 3.1, not every language is recognizable by a Turing machine.
In fact, one may use a diagonalization argument to construct a non-recursively enumerable language as

$$L_d = \{w : w \text{ is the } i\text{th string but is not recognized by the } i\text{th TM }\}.$$

One may also consider the language L'_d wherein instead of considering all strings, we only consider those strings that are valid TM codes (and reject everything else). This is defined by the slightly simpler-to-state

$$L'_d = \{w : \text{if } w \text{ is a TM code, it does not recognize itself}\}.$$

However, this is hardly interesting outside of the fact that it exists. Do there exist more useful undecidable languages?

Before moving to this, let us give an example of a language that is recursively enumerable but not recursive. Recall from earlier that any Turing machine may be translated to a string over $\{0, 1\}$. Consider the *universal Turing machine* that takes as input a Turing machine M and a binary string w, and accepts iff M recognizes w. M and w can be separated by 111 in the input. If the input is not of this desired form, we immediately reject. It is not too difficult to construct a Turing machine that recognizes this language – we maintain one tape for the initial input, one for the tape of (a simulation of) M, and a third to store the state of M.
We claim that the language L_u recognized by the above Turing machine is undecidable. Suppose instead that it was recursive, and let U be an algorithm that recognizes it. Given an input w, we may then check if it is in L'_d as follows.

1. Check if w is a valid TM code. If it is not, accept.
2. Feed $w111w$ into U.
3. Reject if U halts on an accepting state and accept otherwise.

This definitely halts (because U is an algorithm), so we have an algorithm for L'_d, which is a contradiction!

Definition 3.10 (Property)**.** A set of languages is called a *property* of languages. If a language belongs to a certain property, it is said to have that property.

Given a property P, let L_P be the language comprising TM codes of TMs M such that $L(M)$ has P.
Two obvious *trivial* properties are:

1. The "always false" property, which is the empty set.
2. The "always true" property, which has every language.

These are clearly decidable. Very surprisingly, these are the *only* decidable properties!

Theorem 3.3 (Rice's Theorem)**.** For any non-trivial property P that has a recursively enumerable language, L_P is undecidable.

Let us give some more setup before proving this result.

Definition 3.11 (Reduction)**.** A *reduction* from language L to L' is an algorithm that given a string w, converts it to a string x such that $x \in L'$ iff $w \in L$.

If a language L' is undecidable and we have a reduction from L to L', then the two algorithms together give an algorithm for L! That is, if there is a reduction from L' to L and L is decidable, then so is L'. The contrapositive of this is that if we have a reduction from L' to L and L' is undecidable, then so is L.
Observe that our earlier "reduction" of L'_d to L_u is not in fact a reduction in the sense of the above definition. Indeed, we had complemented the result of U which is not allowed by the above definition.

Proof of Rice's Theorem. Let us assume that $\emptyset$ does not have P. If it does, consider the complement of P instead – this works out because the class of recursive languages is closed under complementation.
We shall reduce L_u to L_P. Given a word w and Turing machine code M, we must generate a Turing machine code M' such that M recognizes w iff $L(M')$ has P.
Let L be a recursively enumerable language which has property P and M_L a TM that recognizes L. When given input x, M' works as follows:

1. It simulates M on w.
2. If accepted, it begins simulating M_L on x.
3. Accept iff M_L accepts x in the second step.

Why does this work?

- If M does not recognize w, then we never move past the first step, so the language of M' is $\emptyset$, which does not have P.
- If M recognizes w, then M' ends up accepting a word x iff M_L accepts x, that is, $x \in L$. As a result, if M recognizes w, the language of M' is just L, which does have P.

The Turing machine is just a filter that outputs a fixed language with P if M recognizes w and $\emptyset$ otherwise. The existence of the reduction implies that L_P is undecidable, completing the proof. ■

The important point to note is that while the Turing machine M' itself may not be algorithm (it is not), the *creation* of M' from M and w is.
While this is all good, we are yet to get an undecidable language that is not tied to some sort of Turing machine machinery! To give a more general example, consider the following.

Problem (Post's Correspondence Problem)**.** An instance of PCP is two lists $(w_1, w_2, \ldots, w_n)$ and $(x_1, x_2, \ldots, x_n)$ of strings over some alphabet Σ. The answer to this instance is yes iff there exists some non-empty sequence $i_1, i_2, \ldots, i_k$ of indices in $[n]$ such that $w_{i_1} w_{i_2} \cdots w_{i_k} = x_{i_1} x_{i_2} \cdots x_{i_k}$.

We also present a slightly modified version of the above for the sake of proving undecidability.

Problem (Modified Post's Correspondence Problem)**.** An instance of MPCP is two lists $(w_1, w_2, \ldots, w_n)$ and $(x_1, x_2, \ldots, x_n)$ of strings over some alphabet Σ. The answer to this instance is yes iff there exists some non-empty sequence $i_1, i_2, \ldots, i_k$ of indices in $[n]$ such that $w_{i_1} w_{i_2} \cdots w_{i_k} = x_{i_1} x_{i_2} \cdots x_{i_k}$ and $i_1 = 1$.

To prove undecidability, we shall give a reduction from L_u to PCP. This shall be done by reducing it to MPCP, then reducing MPCP to PCP. "Concatenating" the two reductions yields a reduction from L_u to PCP.

Theorem 3.4. MPCP is reducible to PCP.

Proof. Consider an instance $(w_1, \ldots, w_n)$ and $(x_1, \ldots, x_n)$ of MPCP. For each string v, refer to the ith character of v as v_i. The corresponding instance of PCP we use is of the form $(w'_0, w'_1, \ldots, w'_n, \$)$ and $(x'_0, x'_1, \ldots, x'_n, *\$)$ where

$$w'_0 = *(w_1)_1 * (w_1)_2 * \cdots * (w_1)_{\ell(w_1)} * \text{ and } x'_0 = *(x_1)_1 * (x_1)_2 * \cdots * (x_1)_{\ell(x_1)}.$$

For $i \geq 1$,

$$w'_i = (w_i)_1 * (w_i)_2 * \cdots * (w_i)_{\ell(w_i)} * \text{ and } x'_0 = *(x_i)_1 * (x_i)_2 * \cdots * (x_i)_{\ell(x_i)}.$$

If the MPCP instance is positive, it is clear that so is the PCP instance – we just need to insert $*$s everywhere and add a \$ at the end. If the PCP instance is positive, then observe that the sequence must begin with 0. Indeed, if it were to start with any other index $i \neq 0$, the first character of the w'_i would be $*$, but not that of the x'_i. That is, it forces the w_i corresponding to the string to be w_1, which is exactly what is required by MPCP. ■

Theorem 3.5. PCP is undecidable.

Proof. We shall give a reduction from L_u to MPCP. Suppose we are given a Turing machine M and a word w. Assume that

1. M has a semi-infinite tape.
2. M has exactly one accepting state that has no outward transitions. To assume this, we can add a transition to the new accepting state for each "missing" transition from each of the old accepting states (the symbol inputs for which it would halt at the accepting state). Let this accepting state be f.

The MPCP instance will simulate the instantaneous descriptions of the running of the Turing machine on w. The list of pairs of strings (w_i, x_i) is as follows.

1. $(\#, \#q_0w\#)$. This is the first string (we are forced to start with this by MPCP), wherein the second string starts off with the initial ID. q_0 is the start state of M.
2. $(\#, \#)$. This ends the ID currently being read.
3. (X, X) for any tape symbol X of M. This is the copying of a tape symbol from the previous ID to the next.
4. (qX, Yp) for each transition $\delta(q, X) = (p, Y, R)$. This represents q seeing the symbol X and moving to the right, replacing X with Y.
5. (ZqX, pZY) for each transition $\delta(q, X) = (p, Y, L)$ and any tape symbol Z. To move to the left, we also have to take into account the previous symbol and shift it to the right.
6. $(q\#, Yp\#)$ for each transition $\delta(q, B) = (p, Y, R)$. It is possible to replace a blank symbol immediately after the very end of the string and move to the right.
7. $(Zq\#, pZY\#)$ for each transition $\delta(q, B) = (p, Y, R)$. It is possible to replace a blank symbol immediately after the very end of the string and move to the left.
8. (Xf, f) and (fY, f) for all tape symbols X, Y. When we enter the accepting state, we must accept. This is done by destroying all the tape symbols until only f is left.
9. $(f\#\#, \#)$. When only f is left on the tape, we even out the two strings by adding this and stopping.

The idea is as follows: both strings attempt to create the list of the IDs separated by #s, except that the second string is ahead by one ID. To ensure string equality, the first string must copy the previous ID in the second string, and we can change the second string appropriately to convey that it is an ID. The role of the w_i is just to "read" the previous ID.
By construction, the given instance of MPCP is positive iff M recognizes w. ■

Exercise 3.3. Consider the problem Amibiguity, defined as follows. An instance of Ambiguity is a context-free grammar. The answer to this instance is yes iff the grammar is ambiguous.
Show that Ambiguity is undecidable.

Exercise 3.4. Consider the problem with each instance being a context-free grammar, with the answer being yes iff the language generated by the grammar is equal to all of Σ^*. Show that this problem is undecidable.

The above discussion demonstrates that some problems cannot expect to be solved at all. Even among the questions we can solve however, we can create a hierarchy based on how efficiently it is possible to solve it.

Definition 3.12. A Turing machine that given an input of length n, always halts within $T(n)$ moves is said to be $T(n)$-*time bounded*.

In the deterministic multitape case, the above roughly refers to an "$O(T(n))$ running time algorithm".

Definition 3.13. The class P of languages consists of the languages recognized by those deterministic Turing machines M that are $p(n)$-time bounded for some polynomial p. Such a Turing machine is said to be polynomial-time bounded (or polytime bounded).
The class NP of languages consists of the languages recognized by those nondeterministic Turing machines M that are $p(n)$-time bounded for some polynomial p. Here, the running time of a nondeterministic Turing machine is the maximum number of steps taken along any branch.

One of the Clay Institute's Millennium problems, the question of whether P = NP is an extremely important open question in computer science.
To resolve this debate, it is impractical (and impossible) to attempt to create a polynomial time scheme for every single problem in NP. A natural question to ask is: does there exist some smaller subclass of questions in NP which it suffices to consider?

Definition 3.14. A problem H is said to be *NP-hard* if if it is in P, then NP = P. More precisely, given any problem G in NP, there exists a polynomial time reduction from G to H.
A problem H is said to be *NP-complete* if it is in NP and it is NP-hard.

A complete problem "represents" every problem in NP. When we say that there is a polynomial time reduction from G to H, we mean that given any instance x of G, it is possible to construct a polytime algorithm that outputs an instance of y of H (it behaves as a transducer) such that H recognizes y iff G recognizes x. Why is it that the existence of a polynomial time reduction for any problem in NP implies that if $H \in$ P, then P = NP? Suppose that $H \in$ P and has a $p(n)$-time bounded polytime algorithm and we have a reduction that takes polynomial time $q(n)$. If H is given an instance of length n, then the reduction takes time $q(n)$ and further, the instance fed to H is of length at most $q(n)$. This instance is solved in at most $p(q(n))$, so the entire instance for H is solved in $q(n) + p(q(n))$, which is a polynomial.

Now, let us see a problem that is NP-complete.

Problem (The Satisfiability Problem)**.** An instance of SAT is a boolean formula over some variables. The answer to this instance is yes iff there exists some truth assignment to the variables such that the truth value of the entire formula is true.

We omit more precise definitions of "boolean formula" and "truth assignment", hoping that the reader has learnt these elsewhere. To encode a given boolean formula, we must have it as a string over a finite alphabet. The ith variable is encoded as the symbol x followed by i in binary. The propositional logic symbols $(,), \vee, \wedge, \neg$ are left as they are.

Theorem 3.6 (Cook's Theorem)**.** SAT is NP-complete.

Proof. It is not too difficult to see that SAT is in NP; we can "guess" a truth assignment to the variables in polynomial time then check if this truth assignment satisfies the formula in polynomial time.
Let L be a language in NP, and M a nondeterministic Turing machine recognizing L that has a single semi-infinite tape and a single accepting state f with no outward transitions. Let $p(n)$ be a polynomial time bound for M.
Given w, M, we must construct a boolean formula that is satisfiable iff M recognizes w. Let Γ be the union of the tape alphabet and state space of M (assume the two are disjoint). Our formula shall have $(p(n)+1)^2|\Gamma|$ variables.
Consider a sequence $I_0 \vdash I_1 \vdash \cdots \vdash I_k$ of IDs of M, where $k \leq p(n)$. Each I_j is of length at most $p(n)$, so assume that each is of length exactly $p(n)$ by padding with blanks at the right if necessary. Also assume that $k = p(n)$ by The symbol X_{ij} represents the jth position of the ith ID. It is constituted of $|\Gamma|$ variables $y_{ij,A}$ for each $A \in \Gamma$, wherein $y_{ij,A}$ is 1 iff $X_{ij} = A$.
The SAT formula is constituted of four parts:

1. Each position X_{ij} has a unique symbol.
   This is represented by
   $$\bigwedge_{\substack{0 \leq i,j \leq p(n) \\ X,Y \in \Gamma, X \neq Y}} (\neg y_{ij,X} \vee \neg y_{ij,Y}).$$
   This requires $O(p(n)^2 \log n)$ symbols to be written.

2. The first ID is the starting ID.
   This is represented by asserting that $X_{00} = q_0$, $X_{0i} = w_i$ for $1 \leq i \leq n$, and $X_{0i} = B$ for all $n < i \leq p(n)$. Each of these can be represented by appropriate $y_{0i,X}$ clauses. This requires $O(p(n) \log n)$ symbols.

3. There is an accepting state somewhere.
   This is merely the disjunction of the $y_{ij,f}$ for all valid i, j. This requires $O(p(n)^2 \log n)$ symbols.

4. The transitions from ID to ID are correct.

   - First, we have that if none of $X_{i-1,j-1}, X_{i-1,j}, X_{i-1,j+1}$ is a state, then X_{ij} is equal to $X_{i-1,j}$.
   - If $\delta(q, X)$ contains (p, Y, R) or (p, Y, L), and we have CqX at some point in the ID, then it changes appropriately. The encoding of this is very similar to that done in Theorem 3.5, so we omit it and skip the tedium. There is a special case to be considered at the "edges" of the ID.

■

Further note that upon minor modifications of the final part, the above proof in fact gives a boolean formula in *conjunctive normal form* (CNF): a conjunction of disjunctions of literals (variables or their negations). So, the problem CSAT of deciding the satisfiability of a CNF formula is NP-complete as well.
It is well-known that any CNF formula may be reduced to a 3-CNF formula (a CNF formula where each clause has three literals) in polynomial time that is satisfiable iff the original is. So, 3-CSAT is NP-complete as well.