**Notes on Computer Theory** Last updated: January, 2018

# Languages, regular languages, finite automata

*Content largely taken from Richards [1] and Sipser [2]*

## 1 Languages

An alphabet is a finite set of characters, which we will often denote by $\Sigma$. For example, $\Sigma = \{0, 1\}$ is an alphabet that we will frequently use. A *language* over some alphabet $\Sigma$ is a set of strings made up of characters from $\Sigma$. For example, $L_1 = \{0, 1, 00, 11\}$ is a language over the alphabet $\Sigma = \{0, 1\}$. An English dictionary is also a language, over the alphabet $\{a, \ldots, z, A, \ldots, Z\}$. However, languages need not be finite size: "the set of all binary strings ending in 0" is a language over $\Sigma = \{0, 1\}$. Clearly such a language is not as easy to formally describe, but we will address that issue later on.

It is useful to define a few set operators for languages. The union operator, $\cup$, is defined as with any other collection of sets. The concatenation operator, $||$, is so natural, we will often omit the operator all together: $L_1||L_2 = L_1L_2 = \{xy \mid x \in L_1 \wedge y \in L_2\}$. For any language $L$, we define $L^0 = \{\Lambda\}$, where $\Lambda$ is a special string, called the empty string. For $k > 0$, we recursively define $L^k = LL^{k-1}$. That is, we concatenate $L$ with itself $k$ times (and also include the empty string). Finally, we define the closure operator: $L^* = \cup_{i=0}^{\infty} L^i$.

At a high level, the fundamental question of computer theory is the following. Given a language $L$ and some string $x$, how hard is it to determine whether $x \in L$? (Or, as we will often phrase this question, how hard it is to *decide* the language $L$?) We all have some intuition of what this means. For example, given a string of english characters, one can determine whether it is a valid English word by scanning through the English dictionary, one word at a time. But anyone that still uses a paper dictionary knows that they can do better using binary search, and we may even have seen a proof that you require $O(\log n)$ comparison for a dictionary of size $n$. Such algorithmic questions aren't really our focus in this class, though. Rather, we are interested in characterizing whole *classes* of languages. A language that can be decided in polynomial time on a Turing machine is said to be in a class of languages that we call $\mathcal{P}$. But are there languages that are fundamentally easier to decide than these? Are there languages that can be proven to be strictly harder than those in $\mathcal{P}$? And, furthermore, why is the Turing machine the right model for computation? What if we consider other models? And why is time the right metric: how do memory and communication constraints impact what we can compute? Does access to good random sources help us to decide more languages? Moreover, why is deciding whether some string $x \in L$ the right place to focus our attention? We will look at almost all of these questions, and more, with the aim of gaining a deeper fundamental understanding of what is possible in our field and what is not.

# 2 Finite Automata

## 2.1 Regular Languages

To begin, we start with a very simple model of computation, and a simple class of languages, which are called the regular languages. The regular languages correspond to those generated by regular expressions. We formalize this class of languages recursively, as follows. $\mathcal{R}$ will denote the set of all regular languages over some alphabet, $\Sigma$.

1. $\emptyset \in \mathcal{R}$ and $\{\Lambda\} \in \mathcal{R}$.
2. $\forall \sigma \in \Sigma : \{\sigma\} \in \mathcal{R}$.
3. If $L \in \mathcal{R}$ then $L^* \in \mathcal{R}$.
4. If $L_1 \in \mathcal{R}$ and $L_2 \in \mathcal{R}$ then $L_1 L_2 \in \mathcal{R}$.
5. If $L_1 \in \mathcal{R}$ and $L_2 \in \mathcal{R}$ then $L_1 \cup L_2 \in \mathcal{R}$.

These languages are so common and useful, that we frequently use a special notation, called regular expressions in order to specify languages in this class. In this notation, the '{' and '}' are dropped, and union is denoted by '+'. For example, $(ab + c)^* = \{\Lambda, ab, c, abc, cc, cab, abab, \ldots\}$

## 2.2 Deterministic Finite Automata

A deterministic finite automata is a state machine that takes an input string and either accepts or rejects that string. It makes this decision by transitioning through a sequence of states, making exactly one transition for each character of the input string in a deterministic way. After the transitions are complete, it accepts if it has terminated in a state marked "accept", and it rejects if it has stopped in a state marked "reject". We will formalize this model of computing in a moment, but it is helpful to first demonstrate it by example. In the first example in Figure 1, there is a special start state, labeled 'A', and the state labeled 'C' has a circle around it, denoting that it is an accept state. We note that there can be multiple accept states, though that isn't the case in these two examples. Consider input string 'bc': the DFA reads the first character, and transitions from state 'A' to state 'B'. It then reads the second character and transitions into the accept state, 'C'. Because this is the last character of input, the machine terminates in the accept state, and we say that the machine accepts input 'bc'. Equivalently, we say that 'bc' is in the language of this DFA. Consider now input 'bcc': the last character causes a transition out of the accept state and into state 'D', where the machine now terminates. Because 'D' is not marked as an accept state, we say that the DFA rejects this input, or that the input is not in the language of this DFA. Looking more closely at state 'D', you can see that it is a sink: there are no transitions out of 'D', so no input that leads to state 'D' will ever be accepted. This is sometimes called a *trap state*, and usually we will leave such states out of our diagrams in order to simplify them. Removing 'D' and all of its edges, we will sometimes find that while processing an input string, there is no transition that can be made. In this case, we interpret this though we had transitioned to a trap state, and say that the machine rejects the input.

The language of the second DFA in Figure 1 is the language of the regular expression $(a + bb)^*$. (Written as a regular language, this would be $L^*$, where $L = \{a\} \cup \{bb\}$.) Note that $\Lambda$ is in this language, by the definition of the closure operator; to see why $\Lambda$ is accepted by the DFA, note

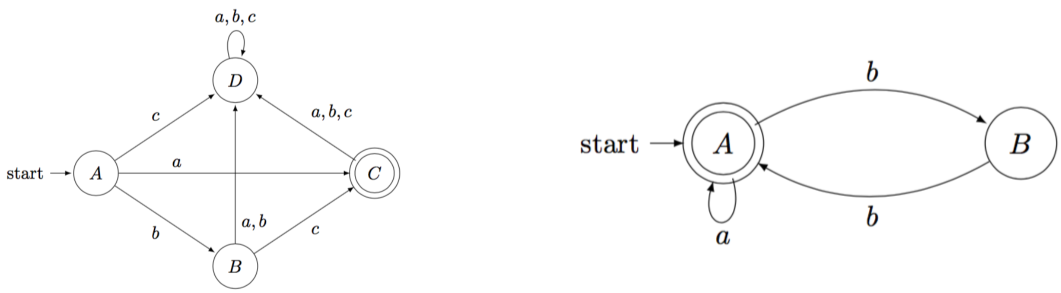



Figure 1: Two examples of DFAs (taken from Richards [1])

that the start state is also an accept state. We will shortly show that all regular languages can be decided by DFAs. Actually, we will show more: that the class of regular languages is equivalent to the class of languages decided by DFAs.

### 2.3 Formally Defining DFAs

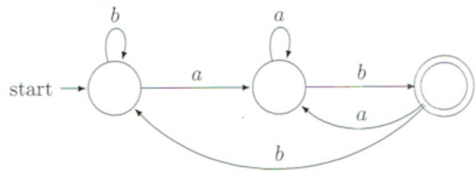
Formally, a DFA is defined by an alphabet $\Sigma$, a finite set of states, $Q$, a start state, $S \in Q$, a set of accept states, $\mathcal{A} \subseteq Q$, and a transition function $\delta : Q \times \Sigma \to Q$. Putting this together, $M = (\Sigma, Q, S, \mathcal{A}, \delta)$. Returning to the first example in Figure 1 (and ignoring the trap state), this DFA can be formally described by $(\Sigma = \{a, b, c\}, Q = \{A, B, C\}, A, \mathcal{A} = \{C\}, \delta)$, where $\delta$ is defined as:

$$\delta = \begin{array}{c|c|c|c|} & a & b & c \\ \hline A & C & B & \perp \\ \hline B & \perp & \perp & C \\ \hline C & \perp & \perp & \perp \\ \hline \end{array}$$

The formalism will help us to prove things about DFAs and the languages that they decide. Formally, for any alphabet $\Sigma$, and any $x \in \Sigma$, we can see that there exists a DFA deciding the language $\{x\}$: $(\Sigma, Q = \{A, B\}, A, B, \delta)$, where $\delta(A, x) = B$, and $\delta$ otherwise has output $\perp$. Informally, the DFA has a start state that is non-accepting, and a single accept state. The only transition allowed goes from the start to the accept state on input $x$. We can also define a DFA for $\{\Lambda\}$: it has a single state that is both the start state and an accept state, and it does not allow any transitions. So we can see that the languages defining the "base-cases" of the regular languages are all decidable by DFAs.

We now show that if a language $L_1$ is decided by DFA $M_1$, and a language $L_2$ is decided by DFA $M_2$, then there exists a DFA $M$ that decides $L_1 \cup L_2$. Intuitively, we construct $M$ so that it tracks the movement of the input through *both* $M_1$ and $M_2$, simultaneously. To do that, we create $|Q_1| \cdot |Q_2|$ states, and label each with a pair of names, one from $Q_1$ and one from $Q_2$. If $M$ is in state $(A, B)$, we can think of this as indicating that $M_1$ would currently, on this input, be in state $A$, while $M_2$ would currently be in state $B$. If $M$ halts in state $(A, B)$, we want to accept if either $A$ is an accept state for $M_1$, or if $B$ is an accept state for $M_2$.

To simplify the formal exposition, we'll assume $M_1$ and $M_2$ share the same alphabet; it is easy to see that this isn't necessary. Let $M_1 = (\Sigma, Q_1, S_1, \mathcal{A}_1, \delta_1)$ and let $M_2 = (\Sigma, Q_2, S_2, \mathcal{A}_2, \delta_2)$.



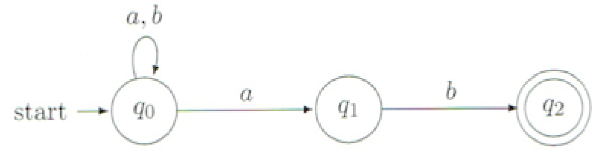




Figure 2: An example of an NFA (taken from Richards [1])

Then $M = (\Sigma, Q, S, \mathcal{A}, \delta)$ is defined as follows. $Q = \{(A, B) \mid A \in Q_1 \wedge B \in Q_2\}$. $S = (S_1, S_2)$, $\mathcal{A} = \{(A, B) \mid A \in \mathcal{A}_1 \vee B \in \mathcal{A}_2\}$, and $\delta((A, B), x) = (\delta_1(A, x), \delta_2(B, x))$.

To prove that $L(M) = L_1 \cup L_2$, we must show two things. First, we prove that if $w \in L(M)$, then $w \in L_1 \cup L_2$. We will write $w = w_1 \cdots w_k$, letting $w_i$ denote the $i$th character of $w$. Note that $w \in L(M)$ implies that there is a sequence of states in $M$, $(S_1, S_2), (A_1, B_1), (A_2, B_2), \ldots, (A_k, B_k)$ such that $\delta((S_1, S_2), w_1) = (A_1, B_1)$, $\delta((A_i, B_i), w_{i+1}) = (A_{i+1}, B_{i+1})$, and either $A_k \in \mathcal{A}_1$, or $B_k \in \mathcal{A}_2$. Without loss of generality, let's assume that $A_k \in \mathcal{A}_1$. It follows by the way $M$ was constructed that $\delta_1(S_1, w_1) = A_1$, and, for $i \in \{1, \ldots, k-1\}$, $\delta_1(A_i, w_{i+1}) = A_{i+1}$. Since $A_k \in \mathcal{A}_1$, it follows that $M_1$ accepts $w$, and that $w \in L_1 \cup L_2$. Secondly, we must show that if $w \in L_1 \cup L_2$, then $M$ accepts $w$. We leave this direction as an exercise. We will later come back to the other regular operators, closure and concatenation.

## 2.4 Non-deterministic Finite Automata (NFAs)

We consider a very useful relaxation in how we model finite automata. Although it was not made explicit, we previously did not allow any ambiguity in how our transitions were to be made: for any state $A$ and any input character $x$, we have, so far, allowed only a single transition from $A$ to be labeled with $x$. Relaxing that gives us a lot more flexibility in our design. Consider the two examples in Figure 2, again taken from Richards [1]. Both machines decide the same language: $\{w \in \{a, b\}^* \mid w \text{ ends in } ab\}$. The second example, which is non-deterministic, has an ambiguous transition out of the start state: on input 'a', the machine has a choice to make. It could either transition to state $q_1$, or it could stay in the start state. We say that this machine accepts an input if there exists some sequence of allowable transitions that ends in an accept state. Importantly, we only require *the existence* of some such sequence of transitions: we do not require that all allowable transitions result in acceptance, and we do not care how one might find such a sequence.

An even better example of where non-determinism helps ease the design of a finite automata is the following language. $L = \{x \in \{a, b\}^* \mid \text{the } k\text{th symbol from the last is 'a'}\}$, where $k$ is some fixed integer. We described an NFA for this language in class, and we will design a DFA for this language in the homework.

To formally define NFAs, we have to change the definition of our transition function. Whereas in DFAs, we have $\delta : Q \times \Sigma \rightarrow Q$, we now have to allow $\delta$ to map the same domain to a *set* of states, rather than to a single state. Formally, $\delta : Q \times \Sigma \rightarrow 2^Q$, where $2^Q$ denotes the power-set. Looking again at example 2 in Figure 2, we have

$$\delta = \begin{array}{c|c|c|} & a & b \\ \hline q_0 & \{q_0, q_1\} & \{q_0\} \\ \hline q_1 & \perp & \{q_2\} \\ \hline q_2 & \perp & \perp \\ \hline \end{array}$$

Additionally, it is helpful to allow $\Lambda$ transitions. These transitions allow the machine to move from one state to another without using up any of the input string. We don't bother to formalize this.

## 2.5 Equivalence of DFAs and NFAs

How much additional power does this non-determinism give us? It seems to make machine design a lot simpler, but does it allow us to decide a larger class of languages? It turns out that it does not: the set of languages decidable by NFAs is exactly the regular languages, just as for DFAs. We prove now that the two models are equivalent in this sense.

It is clear from the definitions that every DFA is also an NFA, so we only need to show that for every NFA, $M = (\Sigma, Q, q_0, \mathcal{A}, \delta)$, there exists a DFA, $M' = (\Sigma, Q', S', \mathcal{A}', \delta')$, such that $L(M') = L(M)$. The intuition is similar to the one above for showing a DFA that decides the union of two languages. We will create a new state for every possible subset of $Q$. We can think of a state $(q_1, q_5, q_7)$ as capturing the fact that $M$ could have followed paths leading to state $q_1, q_5$, or $q_7$. In this way, our DFA will keep track of all the possible places we could currently be in the NFA, given the input string seen so far. With that intuition, the state $(q_1, q_5, q_7)$ is an accept state if any of $q_1, q_5$, or $q_7$ are accept states for $M$. Formally, $Q' = 2^Q$, $S' = \{q_0\}$, $\mathcal{A}' = \{T \in Q' \mid \exists t \in T \text{ s.t. } t \in \mathcal{A}\}$, and $\delta'(T, x) = \bigcup_{q \in T} \delta(q, x)$.

We need to prove two things. For $w = w_0 \cdots w_{k-1}$, if $w \in L(M)$, then $w \in L(M')$, and if $w \in L(M')$, then $w \in L(M)$. We start with the first statement, letting $w \in L(M)$. Because $M$ is an NFA, we know that there exists some sequence of states, $q_0, q_1, \ldots, q_k$, such that, $q_{i+1} \in \delta(q_i, w_i)$, and $q_k \in \mathcal{A}$. Let $T_0, T_1, \ldots, T_k$, be the states in $M'$ such that for each $i \in \{0 \ldots, k\}$, $T_{i+1} = \delta'(T_i, w_i)$. We claim that $T_k \in \mathcal{A}'$. To show this, we first argue that $q_i \in T_i$. This clearly holds for $q_0$, since $T_0 = S' = \{q_0\}$. Assume it holds for $q_i$, and recall that by the definition of $\delta'$, $T_{i+1} = \bigcup_{q \in T_i} \delta(q, w_i)$. Since $q_{i+1} \in \delta(q_i, w_i)$, and $q_i \in T_i$, it follows that $q_{i+1} \in T_{i+1}$. Finally, since $q_k \in T_k$, and $q_k \in \mathcal{A}$, it follows that $T_k \in \mathcal{A}'$.

We now need to prove that if $w \in L(M')$, then $w \in L(M)$. Using the same notation, let $T_0, \ldots, T_k$ be the sequence of states such that $T_{i+1} = \delta'(T_i, w_i)$. We have to show that there exists some sequence of states in $Q$, $q_0, \ldots, q_k$, such that $q_{i+1} \in \delta(q_i, w_i)$, and $q_k \in \mathcal{A}$. We start by choosing $q_k$ and work backwards. To choose $q_k$, we note that because $T_k \in \mathcal{A}'$, then by definition of $\mathcal{A}'$, there is some $q_k \in T_k$ such that $q_k \in \mathcal{A}$. Choose any such $q_k$. For $i < k$, assume $q_{i+1}$ has already be chosen. Since $T_{i+1} = \bigcup_{q \in T_i} \delta(q, w_i)$, there exists some $q_i \in T_i$ such that $q_{i+1} \in \delta(q_i, w_i)$. Choose any such $q_i$, and repeat. Since $T_0 = \{q_0\}$, we can (and must) choose $q_0$ as our start state. This concludes the proof.

Actually, technically, we also have to show how to handle $\Lambda$-transitions when constructing $M'$. This is easily done. For each $q \in Q$, let $E(q)$ be the set of states that is reachable using only $\Lambda$-transitions. Then, instead of defining $\delta'(T, x) = \bigcup_{q \in T} \delta(q, x)$, we define it as $\delta'(T, x) = \bigcup_{q \in T} (\delta(q, x) \cup E(q))$. The rest of the proof would proceed as before.

### 2.6 Equivalence of Regular Languages and DFAs

Using NFAs, it becomes much easier to show that all regular languages can be decided by a DFA. We leave it as a homework problem to show that

1. if a language $L$ is decidable by a DFA, then $L^*$ is decidable by some NFA, $M$.
2. if $L_1$ and $L_2$ are decided by DFAs $M_1$ and $M_2$, then $L = L_1 L_2$ is decidable by some NFA, $M$.

To complete the proof that the class of regular languages and the class of languages decidable by DFAs are the same, we must also show that every language that is decidable by a DFA is regular. This is not a difficult proof, but we omit it in this class so that we can move on to other interesting things.

### 2.7 Some languages are not regular

There are some languages that cannot be decided by any finite automata. To demonstrate this, we first prove a useful lemma that is famously known as the pumping lemma.

**Lemma 1** *If $L$ is a regular language, then there exists a number $p$ such that for any $w \in L$ with $|w| > p$, $w$ can be divided into 3 strings, $w = xyz$ such that:*

1. $\forall i \geq 0, xy^i z \in L$
2. $|y| > 0$
3. $|xy| \leq p$

**Proof** Since $L$ is regular, we know that there exists some finite automata $M$ that decides $L$. We define $p$ to be the number of states in $M$. For $w = w_1, \ldots, w_n$, let $q_0, \ldots, q_n$ be the sequence of states that lead from start to accept on string $w$. Because $n > p$ (by assumption in the lemma statement), there must be some state in this sequence that is repeated (by the pigeonhole principal). Let $q_s$ be the first state on the list that appears more than once, and let $t$ be the index of the first repetition of $q_s$. (That is, $q_s = q_t$: they represent the same states, but appear in different places on this list.) Then we define $x = w_1 \cdots w_s$, $y = w_{s+1} \cdots w_t$, and $z = w_{t+1} \cdots w_n$. We now show that the three properties of the lemma are satisfied. For the first property, let's consider $i = 0$, so that we have input string $xz = w_1, \ldots, w_s, w_{t+1}, \ldots, w_n$. We know that the first $s$ characters will leave us in state $q_s$, since these are the same characters in the original input, $w$. Furthermore, since $q_s = q_t$, we know that $w_{t+1}, \ldots, w_n$ will transition us through $q_{t+1}, \ldots, q_n$, landing us is in the same accept state that results from processing the original $w$. For $i = 1$, it is true by assumption that $xyz \in L$. For $i > 1$, we claim that at the end of each repetition of $y$, we end in state $q_t$. This is certainly true at the end of the first repetition of $y$, by the way we defined $q_t$. Since $q_t = q_s$, the next repetition of $y$ transitions through $q_{s+1}, \ldots, q_t$, just as the first occurrence of $y$ did. Since the last repetition of $y$ leaves us in $q_t$, it follows that $z$ transitions us to accept state $q_n$.

The fact that $|y| > 0$ follows immediately from the definition of $y$. To see that $|xy| \leq p$, recall that $q_t$ is the first state to be repeated in the transition sequence. If $t > p$, it follows that there must be more than $p$ states in $M$, which violates our definition of $p$. ■

We now use the pumping lemma to show that $L = \{a^n b^n | n \geq 0\}$ is not regular. Suppose that $L$ is regular, and let $M$ be the DFA decides it. Let $p$ be the number of states in $M$. By the pumping lemma, we know that *any* string $w \in L$ with $|w| > p$ can be written as $xyz$ such that $y$ can be "pumped". Consider taking $j = \lceil p/2 \rceil$. We claim that $a^j b^j$ cannot be pumped. Specifically, regardless of how $y$ is chosen for this string, $xy^2z \notin L$. There are 3 cases to consider: 1) $y$ contains only $a$ values. In this case, $xy^2z$ has more $a$s than $b$s. 2) $y$ contains only $b$ values. In this case, $xy^2z$ has more $b$ values than $a$s. Finally, 3) if $y$ has both $a$s and $b$s, then note that $xy^2z$ has a substring in which some $a$s come after some $b$s.

We note that we could have also considered $j = p$, and the argument would have been simpler. But this argument is more "interesting", and demonstrates an important proof technique, called *case analysis*.

An important thing to pay attention to in the proof above is the ordering of quantifiers. The pumping lemma says that if $L$ is regular, then $\exists p$ such that $\forall w, |w| > p$, $\exists xyz = w$ where $x, y$,and $z$ satisfy the conditions of the lemma. To show that a language is NOT regular, we have to show that this statement is NOT true. That is, we have to show that $\forall p$, $\exists w, |w| > p$ such that $\forall xyz = w$, $x, y$, and $z$ fail to satisfy the criteria of the lemma. In particular, then, note that we don't know the value of $p$ we should use in the proof. Instead, for every possible value of $p$, we shown that there is some $w$ such that for each and every possible way of splitting up $w$, one of the criteria are not met.

# 3 Pushdown Automata

Push down automata are very similar to finite automata, but we equip the state machine with a stack for reading and writing data. The automata still operates by scanning the input, left to right, one character at a time. The automata terminates when it has read the last character of the input. An example can be seen below: transitions are labeled by $a, b/c$, where $a \in \Sigma$ is a value of the input, and $b, c \in \Gamma$, where $\Gamma$, called the "tape alphabet", is the set of characters that can pushed and popped from the stack. It is reasonable to assume that $\Sigma \subseteq \Gamma$. The notation $b/c$ means that you can take this transition if the character at the top of the stack is $b$, and, in doing so, you replace the $b$ with a $c$. Note that you can only take a transition $a, b/c$ if the next character of the input is $a$ AND the character at the top of the stack is a $b$. If we don't wish to put anything new onto the stack, we can use a transition of the form $a, b/\Lambda$. In this case, we would pop a $b$, and the number of elements on the stack would be reduced by one. Similarly, we also allow the machine to ignore the input or the stack. The former is denoted by $\Lambda, a/b$, and the latter is denoted by $a, \Lambda/c$. We can ignore both by $\Lambda, \Lambda/c$. Such transitions can be taken regardless of the values of the next input character and the character at the top of the stack. We also allow to push multiple characters onto the stack at once (though we do not allow multiple pops at once). For example, $a, b/bc$ would pop 1 $b$, push 1 $b$, and then push 1 $c$. If the stack initially only had content $b$, then, after this transition, it would contain $bc$; we will always write (left to right) the content of the stack from bottom to top.

**Empty stack:** We don't have any explicit mechanism for testing the stack to see if it is empty. Instead, if that is something we care to do, we can create a transition at the start that pushes a special symbol onto the stack, and we can later interpret as an indicator that the stack is empty. This can be seen in Figure 3 below, where \$ plays that role. Note that we do not read an input character in that transition; we only start processing input after we've initialized our stack.

**Termination:** The automata terminates when the last character of the input is read. It accepts if and only if it terminates in an accept state. Just as with finite state automata, we assume there is a trap state for rejecting that is not made explicit: if it is ever impossible to make a transition, and there is still input that hasn't been processed, then the machine is assumed to transition into a reject state and to stay there. Note that in the case where we ignore the input tape, we also delay termination by one transition. So, we can take many transitions of the form $\Lambda, a/b$, and these transitions do not "use up" any of the input. We will next walk through the example in Figure 3 below, which will demonstrate this point.

**Example:** We walk through the NPDA in Figure 3, using input $abba$.

- There is only one transition we can take from the start state. We transition to *even*, the input tape still holds $abba$, and the stack holds \$.
- Since the first input character is $a$, the only legal transition is to $>_a$. The input tape holds $bba$ and the stack holds $\$a$.
- Since the top of the stack is now $a$, the only legal transition is $b, a/\Lambda$, which leaves us in the same state. (Note we cannot take the transition labeled $\Lambda, \$/\$$, because the top of the stack was $a$.) This transition removes the $a$ at the top of the stack. The input tape now holds $ba$, and the stack now holds \$.
- The only legal transition is the one labeled $\Lambda, \$/\$$, to state *even*. The remaining input is *still* $ba$, since the $\Lambda$ in that transition does not use up an input character. This transition pops and pushes \$, so the stack still holds \$.
- We transition to state $>_b$, the input tape holds $a$ and the stack holds $\$b$.
- We transition using $a, b/\Lambda$, remaining in the same state. The input tape is now empty, and the stack now holds \$.
- We now have a choice to make. We can terminate and reject, or we can transition one more time using $\Lambda, \$/\$$ and then accept. Recall that the definition of non-determinism says that a string is in the language as long as there *exists* some sequence of choices that leads to accept. So, in this case, the string is in the language.

## 3.1 Formal notation for NPDAs

A NPDA can be denoted by $(Q, \Sigma, \Gamma, \delta, q_0, Q_A)$, where $Q$ is the set of states, $\Sigma$ is the input alphabet, $\Gamma$ is the tape alphabet (which might contain $\Sigma$), $\delta$ is a transition function, detailed below, $q_0$ is a special start state, and $Q_A \subseteq Q$ is a set of accept states. The function $\delta$ maps a state, an input character, and a character read from the stack, to a state and a sequence of characters to be written to the stack. However, in the non-deterministic case, note that it might map the same input onto multiple outputs. We therefore let the co-domain be the power set of $Q \times \Gamma^*$. Formally, then, $\delta$ is a function $\delta : Q \times \Sigma \times \Gamma \to 2^{Q \times \Gamma^*}$.

A machine accepts string $w$ if and only if $w$ can be written as $w_1 w_2 \cdots w_n$, where each $w_i \in \Sigma \cup \{\Lambda\}$, and there exists a sequence of states $r_0, r_1, \ldots, r_n$, $r_i \in Q$, and a sequence of strings $s_0, \ldots, s_n$, $s_i \in \Gamma^*$, such that

1. $r_0 = q_0$, $s_0 = \Lambda$, and $r_m \in Q_A$.

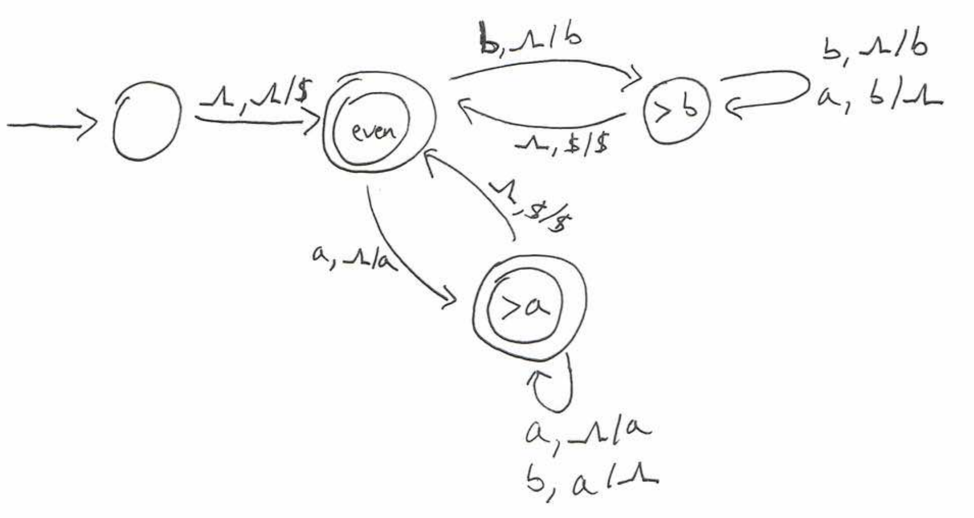



Figure 3: $M_L$ accepting language $L = \{(a+b)^* \mid$ there are an equal number of $a$s and $b$s$\}$

2. $\forall i \in \{1, \ldots, n\}$, $\exists \alpha \in \Gamma \cup \{\Lambda\}, \beta, \gamma, \in \Gamma^*$, such that $s_{i-1} = \alpha\gamma$, $s_i = \beta\gamma$, and $(r_i, \beta) \in \delta(r_{i-1}, w_{i-1}, \alpha)$

Intuitively, $w_1 \cdots w_n$ denote the input string, but possibly "padded" with internal $\Lambda$ values to account for places that we might take a transition that doesn't read any input. The first condition states that we start in the start state with an empty stack, and we terminate in an accept state. The second condition says that we transition through some valid sequence of states, maintaining valid stack content.

### 3.2 NPDAs and DPDAs are not equivalent

Unlike in the case of DFAs and NFAs, non-determinism in the case of push-down automata does in fact increase the expressiveness of the model. That is, there are language that can be decided by a NPDA that cannot be decided by a DPDA. An example of such a language is $L = \{a^n b^n \mid n \geq 0\} \bigcup \{a^n b^{2n} \mid n \geq 0\}$. We leave it as an exercise to show that $L$ can be decided by an NPDA. Also, we do not prove in this class that there is a pumping lemma, similar to the one for regular languages, which shows that certain languages cannot be decided by NPDAs. One such language is $L' = \{a^n b^n c^n \mid n \geq 0\}$. To prove that $L$ cannot be decided by any DPDA, we show that, if it were, then we can construct a PDA for $L'$, violating the pumping lemma.

Let $M_1 = (\{a, b\}, Q_1, q_0, \mathcal{A}_1, \delta_1)$ and $M_2 = (\{a, b\}, Q_2, S_2, \mathcal{A}_2, \delta_2)$ be identical copies of the machine that decides $L$, but with different state names. To construct $M'$ deciding $L'$, we start with $M' = (\{a, b, c\}, Q_1 \cup Q_2, q_0, \mathcal{A}_2, \delta')$, where, for $x \in \{a, b\}$, $\forall q \in Q_1$, $\delta'(q, x) = \delta_1(q, x)$, and $\forall q \in Q_2$, $\delta'(q, x) = \delta_2(q, x)$. We then make the following modifications to $\delta'$ in order to define its behavior on inputs of type $c$.

1. For each $q \in \mathcal{A}_1$, let $p_1 = \delta(q, b)$, and let $p_2$ be the equivalent state in $M_2$. Define $\delta'(q, c) = p_2$.
2. For every $q \in Q_2$, let $p_2 = \delta(q, b)$. Define $\delta'(q, c) = p_2$, and set $\delta'(q, b) = \mathsf{reject}$.

Intuitively, the modifications specified above are as follows. For each state that was an accepting state in $M_1$, if there was a transition out of that state labeled with character $b$, then we add a

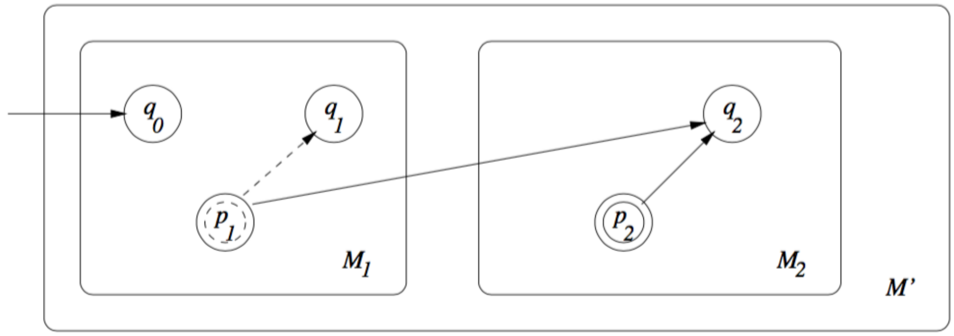



Figure 4: Machine $M'$, deciding language $L' = \{a^n b^n c^n \mid n \geq 0\}$, which we know to be impossible. Figure is taken from Richards [1].

transition labeled $c$ to the equivalent state in $M_2$. Then, for every $b$ transition in $M_2$, we replace it with a $c$ transition.

To prove that this decides $L'$, we start by arguing that if $w \in L'$, $M'$ ends in an accept state. To see this, consider what happens after $a^i b^i$ are processed. Since $a^i b^i \in L$, we know that at this point in the computation, $M'$ is in an accept state. Since $w \in L'$, the next $i$ characters are $c$, and because we're in an accept state, the first of these characters causes a transition to a state in $M_2$. From there, the transitions follow the last $i$ transitions of $\delta_2$, on input $a^i b^{2i}$.

On the other hand, if $M'$ accepts on some string $w$, we must argue that $w \in L'$. We first note that if $w$ does not begin either with $a^i b^i c$, or with $a^i b^{2i} c$, then $M'$ must reject. This is because all accept states are in $M_2$, so $w$ has to touch some state in $\mathcal{A}_1$, and then transition with a $c$ to $M_2$. Furthermore, if there are any characters other than $c$ after the transition to $M_2$ is made, then it is easy to see that $M'$ will reject: a $b$ will cause a reject explicitly, and an $a$ will cause a reject because $M_2$ does not allow an $a$ after the first appearance of a $b$. Suppose $w$ is of the form $a^i b^{2i} c^j$. If $M'$ accepts, it follows that $M_1$ would accept $a^i b^{2i+j}$, violating our assumption that $M_1$ decides $L$. So we have that $w$ is of the form $a^i b^i c^j$. Finally, if $j \neq i$, by the same previous argument, $M'$ must reject, or else $M_1$ would accept a string $a^i b^{i+j}$, where $0 \neq j \neq i$, violating our assumption about $M_1$. We conclude that $w$ is of the form $a^i b^i c^i$, as claimed.

## References


[1] D. Richards *Logic and Language Models for Computer Science*, third edition. World Scientific Publishing Co., 2018.

[2] M. Sipser. *Introduction to the Theory of Computation* (2nd edition). Course Technology, 2005.