Theory of Computation I: Introduction to Formal Languages and Automata

Noah Singer

April 8, 2018

1 Formal language theory

Definition 1.1 (Formal language)**.** *A formal language is any set of strings drawn from an alphabet* Σ*.*

We use ε to denote the "empty string"; that is, the string that contains precisely no characters. Note: the language containing the empty string, $\{\varepsilon\}$, is *not the same as* the empty language $\{\}$.

Definition 1.2 (Language operations)**.** *Given formal languages* A *and* B*:*

1. *The* ***concatenation*** *of* A *and* B*, denoted* $A \cdot B$ *or* AB*, is defined as* $\{ab \mid a \in A, b \in B\}$*, where* ab *signifies "string* a *followed by string* b*".*
2. *The* ***union*** *of* A *and* B*, denoted* $A \cup B$*, is defined as* $\{c \mid c \in A \vee c \in B\}$*.*
3. *The language* A *repeated* n *times, denoted* A^n*, is defined as*

$$A^n = \underbrace{A \cdot A \cdot A \cdots A \cdot A}_{n \text{ times}}$$

   if $n > 0$ *or* $\{\varepsilon\}$ *if* $n = 0$*.*

4. *The* ***Kleene star*** *of a language* A*, denoted* $A^\star$*, is defined as*

$$A^\star = \bigcup_{i \in \mathbb{N}} A^i = \{\varepsilon\} \cup A \cup A \cdot A \cup A \cdot A \cdot A \cup \cdots$$

.

Exercise .1: Language operations

1. Let $A = \{a, b\}$ and $B = \{c, d, e\}$. Compute the following:

   (a) $A \cdot B$
   (b) $A \cdot \emptyset$
   (c) $A \cdot \{\varepsilon\}$
   (d) $A \cup B \cup \emptyset$
   (e) $(BA) \cup A^3$
   (f) $A^\star$
   (g) $B^\star A^3$
   (h) $BA^\star B$
   (i) $(AB)^\star A$

2. Consider the operations $\cup$ and $\cdot$ on formal languages. In abstract algebra terms, are they associative and/or commutative? Do they have identities and/or inverses?

Definition 1.3 (Formal grammar)**.** *A formal grammar is a 4-tuple* $G = (N, \Sigma, P, S)$*, where:*

1. N *is a finite set of* ***nonterminal symbols****.*
2. Σ *is a finite set of* ***terminal symbols*** *and* $N \cap \Sigma = \emptyset$

3. *P is a set of* ***productions****, where* $P \subseteq ((\Sigma \cup N)^\star N (\Sigma \cup N)^\star) \times (\Sigma \cup N)^\star$.

4. *S is a special* ***start symbol*** *in* N.

These productions can be viewed as **rewriting rules** of the form

$$\alpha \to \beta$$

where α and β are strings of terminals and nonterminals and α has at least one nonterminal.

Definition 1.4 (Grammar derivation)**.** *Given two strings* $x, y \in (\Sigma \cup N)^\star$*, x "derives y in one step", written* $x \underset{G}{\Longrightarrow} y$*, iff*

$$\exists \alpha, \beta, \gamma, \delta \in (\Sigma \cup N)^\star : (x = \gamma\alpha\delta) \wedge (y = \gamma\beta\delta) \wedge ((\alpha \to \beta) \in P)$$

This just means that we can divide x three ways into $\gamma\alpha\delta$ and then divide y into $\gamma\beta\delta$–note that γ and δ are the same–where $\alpha \to \beta$ is a production of our grammar. Essentially, the grammar is "replacing α with β" to transform x into y.

If there's some sequence of $z_1, z_2, z_3, \ldots$ such that $x \implies z_1 \implies z_2 \implies \cdots \implies y$, then we say that x **derives** y. If y can be derived in a finite number of steps from S, then it is a **sentential form**; if y consists only of terminal symbols, then it is a **sentence** of the grammar G. (They're called "terminal" symbols because they're the end of the derivation.) *G determines a language, denoted $\mathcal{L}(G)$–precisely all the sentences that G derives!*

Definition 1.5 (Ambiguous grammar)**.** *A grammar is ambiguous iff it has a sentence that can be derived two different ways.*

We will examine this more later, but ambiguous grammars are undesirable because they admit multiple correct derivations–or **parses**–of the same sentence, which we certainly want to avoid if we are designing parsing algorithms.

Exercise .2: Grammars and languages

1. Describe the languages derived from the following grammars. Are any equivalent[a]? Are any ambiguous?

   (a) $S \to abc$

   (b) $S \to abS$
   $S \to \varepsilon$

   (c) $S \to aSb$
   $S \to \varepsilon$

   (d) $S \to S * S$
   $S \to S/S$
   $S \to S + S$
   $S \to S - S$
   $S \to (S)$
   $S \to \text{digit}$

   (e) $F \to T * T$
   $F \to T/T$
   $F \to T$
   $T \to T + T$
   $T \to T - T$
   $T \to (F)$
   $T \to \text{digit}$

2. Bonus: construct a grammar that derives $\{a^n b^n c^n \mid n \in \mathbb{N}\}$.

[a] More specifically, *weakly* equivalent in the sense that they derive the same language?

Now let's look at some specific types of grammars. Chomsky (1956) defined a general hierarchy of formal languages that we'll use throughout this activity:

Type	Grammar	Automaton	Productions
0	Unrestricted	Turing machine	$\alpha \to \beta$
1	Context-sensitive	Non-deterministic linear bounded automaton	$\gamma A \delta \to \gamma\alpha\delta$
2	Context-free	Non-determinsitic pushdown automaton	$A \to \alpha$
3	Regular	Finite automaton	$A \to a$, $A \to Ba$, $A \to B$, $A \to \varepsilon$

We'll introduce the automata later, but note how Type-3, the regular grammars, has the most restrictions on the productions that are allowed; type-0 has no restrictions and so includes all formal grammars.

2 Regular languages and finite automata

In computer science and discrete mathematics, an **automaton** is a mathematical model of a "machine". Given some input, it follows a defined sequence of steps to produce an output. Often, based on some predefined set of "decision rules", an automaton moves between various **states** depending on the input. Sometimes, the automaton will choose to **accept** an input string; in general, it may also halt or loop indefinitely.

Automata are intimately linked to formal language theory, because every automaton M **recognizes** a formal language of strings $\mathcal{L}(M)$–exactly the strings that the language accepts. Automata are important for two main reasons: 1) For discrete mathematics purposes, they are compact and finite representations of important kinds of formal languages, that give us insight into their properties, and 2) For computer science purposes, they are **models of computation**. Any particular automaton is, from the computer science standpoint, a **program**–a specific set of instructions for how to transform some input to some output.

The family of languages that a certain class of automata can recognize will often be strictly contained within a larger family recognized by a more general class of automata; thus, *the latter class is a more powerful model of computation than the former.* As you might have guessed, our ultimate goal in this activity is to prove the equivalence of different classes of languages, grammars, and automata.

Let's start by considering the simplest important class of automaton!

2.1 Definitions

Definition 2.1 (Deterministic finite automaton)**.** *A* ***determinsitic finite automaton*** *is a 5-tuple* $(Q, \Sigma, \delta, q_0, F)$*, where:*

1. Q *is a finite set of states that the DFA may exist in.*
2. Σ *is the* ***alphabet****, a finite set of symbols that the DFA reads in.*
3. $\delta : Q \times \Sigma \to Q$ *is the* ***transition function****, which maps the current state and the next input symbol to a new state.*
4. $q_0 \in Q$ *is the* ***start state*** *of the DFA.*
5. $F \subseteq Q$ *is the set of* ***accept states*** *of the DFA.*

Definition 2.2 (DFA acceptance)**.** *A DFA defined by* $(Q, \Sigma, \delta, q_0, F)$ ***accepts*** *a string* $w = a_1 a_1 \dots a_n$ *iff there exists some sequence of states* $r_0 r_1 \dots r_n$ *in* Q *where:*

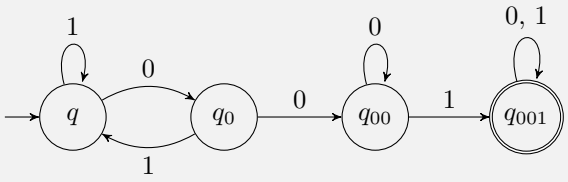
1. $r_0 = q_0$ *(*r_0 *is the start state).*
2. $r_{i+1} = \delta(r_i, a_{i+1})$*, for* i *between* 0 *and* $n-1$ *(*r_{i+1} *follows from* r_i *on input character* a_{i+1}*).*
3. $r_n \in F$ *(*r_n *is an accept state).*

Conceptually, the DFA is given an input string, "begins" in the start state and "follows" the transitions; each transition maps the current state and the current input symbol to a new state. The input symbol is then "consumed". If, when the input is entirely consumed, the DFA lands in an accept state, then the DFA accepts the string.

DFAs are nice because they're easy to visualize! The rules are simple: draw the states as labelled circles; draw transitions as arrows between states; draw the input state with an arrow coming in from nowhere; draw accept states with double lines instead of single lines.

Exercise .3: DFA practice

1. Consider the following DFA. What language does it recognize?



2. Construct DFAs which recognize the following languages ($\Sigma = \{0, 1\}$).
   (a) Binary strings of even length
   (b) Binary strings with exactly four 1's
   (c) Binary strings with an odd number of 1's
   (d) Binary strings divisible by 3

Now, let's look at a closely related type of automaton: the nondeterministic finite automaton (NFA). **Nondeterminism** is a property that some automata possess where the decision rules specify than more than one action to take. Their actions are not fully "determined"; they can explore many paths at the same time, and accept if any such path reaches an accept state. NFAs are also allowed to have so called ε-transitions, which occur without *any* input being consumed–it's useful for expressing branches in the program logic.

Definition 2.3 (Nondeterministic finite automaton)**.** *A **nondeterministic finite automaton** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where:*

1. *Q is a finite set of states that the NFA may exist in.*
2. *Σ is the alphabet, a finite set of symbols that the NFA reads in.*
3. *$\Delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$ is the transition function, which maps the current state and the next input symbol or ε to a new subset of states.*
4. *$q_0 \in Q$ is the start state of the NFA.*
5. *$F \subseteq Q$ is the set of accept states of the NFA.*

Definition 2.4 (ε-closure)**.** *The ε-**closure** of some state $q \in Q$, denoted $E(q)$, is the set of states reachable from q through only ε-transitions. In other words, it is the set of states $p \in Q$ such that there exists some sequence of states $q_0 q_2 ... q_k$ where:*

1. *$q_0 = q$*
2. *$q_{i+1} \in \Delta(q_1, \varepsilon)$*
3. *$q_k = p$*

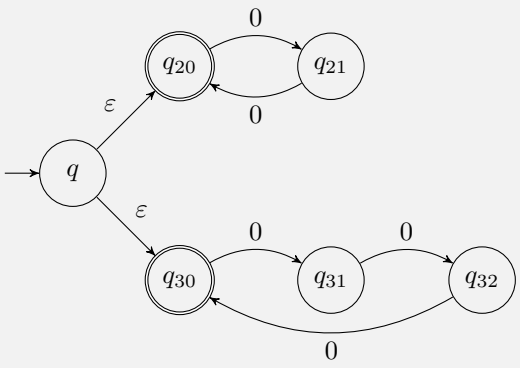
Similarly, we can define the ε-closure of a *set* of states as the union of their closures. ε-closure is simply a useful mathematical trick that we can employ to define and prove things more succinctly.

Definition 2.5 (NFA acceptance)**.** *An NFA defined by $(Q, \Sigma, \Delta, q_0, F)$ accepts a string $w = a_1 a_1 \dots a_n$ iff there exists some sequence of states $r_0 r_1 \dots r_n$ in Q where:*

1. *$r_0 = E(q_0)$*
2. *$r_{i+1} \in E(\Delta(r_i, a_{i+1}))$, for i between 0 and $n - 1$*
3. *$r_n \in F$*

Exercise .4: NFA practice

1. Consider the following NFA. Determine the language that it accepts, as well as the ε-closure of all states.



2. Construct both an NFA and DFA to recognize all binary strings containing the substring 010. Which is easier?

A final definition that's much more concise, convenient and intuitive:

Definition 2.6 (Regular language)**.** *The family of regular languages R is defined as the closure of the family of atomic languages $\{\emptyset, \{\varepsilon\}, \{s_0\}, \{s_1\}, \ldots\}$ for $s_i \in \Sigma$ under the operations of union, concatenation, and Kleene star.*

2.2 Equivalence

Theorem 2.1 (Powerset construction)**.** *Any non-deterministic finite automaton with n states can be converted into an equivalent deterministic finite automaton with up to 2^n states.*

To understand why this is true, consider that a DFA keeps track of only a *single* state at a time, while an NFA needs to keep track of *many* states at a time. But the NFA can only possibly be in a finite subset of its states at any given time! So the basic idea is to have a DFA state for every *set of states* the NFA could possibly be in. The DFA's initial state is the set containing only the NFA's initial state, which reflects the fact that the NFA can still only start in one place. Transitions from one DFA state to another on any given input symbol reflect all possible destinations from any state in the original set of state on that input symbol.

Proof. If the NFA has ε-transitions, we can consolidate the states by taking ε-closures until there are no more ε-transitions. Therefore, we assume that it has no ε-transitions.

For any NFA $N = (Q_N, \Sigma_N, \Delta_N, q_{0N}, F_N)$, we will specify an equivalent DFA M as follows:

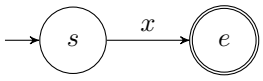
1. $Q_M = \mathcal{P}(Q_N)$
2. $\Sigma_M = \Sigma_N$
3. $q_{0M} = \{q_{0N}\}$
4. $\delta_M(S \in Q_M, x \in \Sigma) = \bigcup_{x \in S} \Delta(q, x)$
5. $F_M = \{\exists q(q \in S \wedge q \in F_N) \mid S \in \mathcal{P}(Q_N)\}$

□

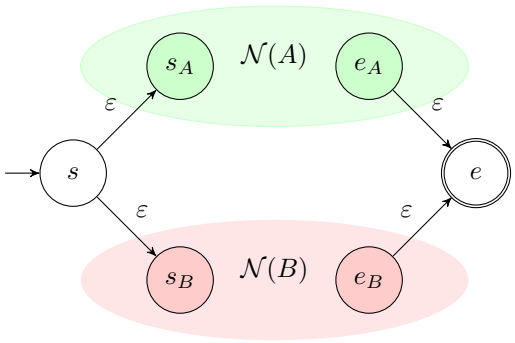
Theorem 2.2 (Thompson construction)**.** *All regular languages may be recognized by non-deterministic finite automata.*

Proof. Let $\mathcal{N}(A)$ denote the NFA corresponding to regular language A. We show that $\mathcal{N}(\{\varepsilon\})$ and $\mathcal{N}(\{x\})$ for $x \in \Sigma$ exist. We then show how to construct the NFA under each regular language operation and therefore our proof is complete by closure.

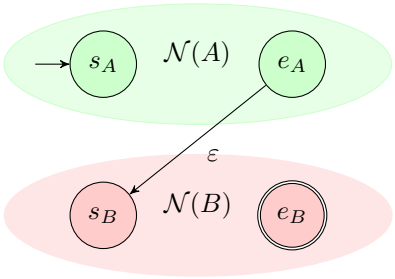
The NFA for $x \in \Sigma \cup \{\varepsilon\}$ is:



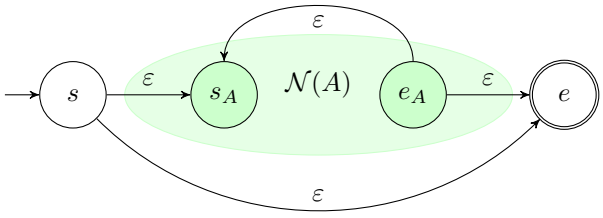
The NFA for $A \cup B$, where A and B are regular, is:



The NFA for $A \cdot B$, where A and B are regular, is:



The NFA for $A^\star$, where A is regular, is:



□

Theorem 2.3 (Kleene's algorithm)**.** *The language recognized by any given deterministic finite automaton is regular.*

Proof. Let the DFA $M = (Q, \Sigma, \delta, q_o, F)$ where $Q = \{q_0, q_1, \ldots, q_n\}$. Let R_{ij}^k denote the language of strings that take the DFA from state q_i to state q_j while only passing through states with numbers less than or equal to k (not including the starting and stopping states).

To start out, R_{ij}^{-1} should only contain the alphabet symbols that take q_i to q_j.

$$R_{ij}^{-1} = \begin{cases} \{x \in \Sigma \mid \delta(q_i, x) = q_j\} & i \neq j \\ \{x \in \Sigma \mid \delta(q_i, x) = q_j\} \cup \{\varepsilon\} & i = j \end{cases}$$

Now, consider some arbitrary $0 \leq k \leq n$, and states q_i and q_j. Consider any path from q_i to q_j that doesn't pass through states about k. All such paths either contain k or do not contain k. Thus,

$$R_{ij}^k = R_{ij}^{k-1} \cup (R_{ik}^{k-1} \cdot (R_{kk}^{k-1})^\star \cdot R_{kj}^{k-1})$$

where the first term represents passing from q_i to q_j while avoiding q_k while the second represents passing from q_i to q_k, back to q_k any number of times, and then to q_j.

Finally, the regular language that represents the entire DFA is

$$\bigcup_{q_i \in F} R_{0i}^n$$

and this language is certainly regular since we constructed it using only union, concatenation, and Kleene star. □

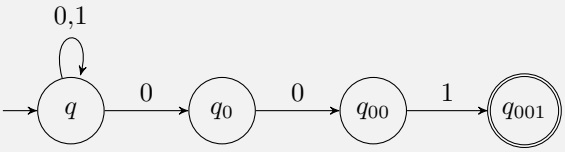
We've shown that any language an NFA recognizes, a DFA can recognize; that any regular language can be recognized by an NFA; and that every language that a DFA can recognize is regular. Thus, DFAs, NFAs, and regular languages are all equivalent!

Now, let's recall that regular grammars are those whose productions are of the forms $A \to a$, $A \to Ba$, or $A \to \varepsilon$. Note that there is a simple one-to-one correspondence between NFAs and regular grammars... you can figure this one out yourself!

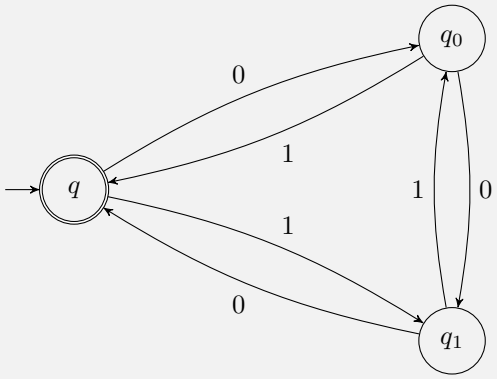
Exercise .5: Regular equivalences

In the following problems, follow the methods in the above proofs.

1. Convert the following NFA to a DFA. What language does it recognize?



2. Show that the following DFA recognizes a regular language.



3. Show that the following regular language is recognized by an NFA and construct a regular grammar that represents it.

$$\{0,1\}^\star \cdot \{010\}$$

4. Show that the following regular grammar derives a regular language.

$$S \to S0, S \to A, A \to B1, B \to C0, C \to D1, D \to D1, D \to \varepsilon$$

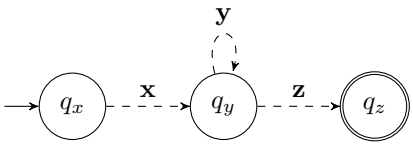
2.3 Pumping lemma

We have four equally powerful ways of describing regular languages. But why do we need grammars that are more general than regular languages? Well, it turns out that we can prove that some languages certainly are *not* regular.

Theorem 2.4 (Pumping lemma)**.** *For any regular language L, there exists some integer $p \geq 1$that for every string w where $|w| \geq p$, there are three strings x, y, z where:*

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

We say that the string y can be "pumped". Let's try and visualize it.



Proof. For some regular language A, there must exist a DFA M which recognizes it. Let p be the number of states of M. Let $w \in L$ be a string such that $|w| \geq p$. Then let the sequence of the start state and first p states that w takes through M be labeled $q_0, q_1, q_2, \ldots, q_p$. By the pigeonhole principle, since this sequence has $p + 1$ states in it, one state must have been visited twice. Let q_s denote this state. The section of w that takes M from the first occurrence of state q_s to the second is then y. Clearly, $|y| > 0$. $|xy| \leq p$ since $s \leq p$. The string y can also be repeated any number of times. The conditions of the theorem are therefore satisfied. □

Exercise .6: Pumping lemma applications

Show that the following languages are not regular.

1. $\{a^n b^n \mid n \in \mathbb{N}\}$
2. $\{a^{2^n} \mid n \in \mathbb{N}\}$