Automata Theory
Reading Project

Swayam Chube

Last Compiled: June 2, 2022

Contents

Chapter 1

Regular Languages

1.1 Finite Automaton

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

- Q: Finite set of states
- Σ: Finite set called *alphabet*
- $\delta : Q \times \Sigma \to Q$ is the *Transition Function*
- $q_0 \in Q$: start state
- $F \subseteq Q$: set of accept/final states

Definition 1.2 (Language of a DFA). An automaton D *accepts* a word $w \in \Sigma^*$ if the run of w on D ends in a final state. The *language* of an automaton $\mathcal{L}(D)$ is the set of all words accepted by D.

Definition 1.3 (NFA). A *non-deterministic finite automaton* is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

- Q: Finite set of states
- Σ: Finite set called *alphabet*

- $\delta : Q \times \Sigma_\varepsilon \to 2^Q$ is the *Transition Function*
- $q_0 \in Q$: start state
- $F \subseteq Q$: set of accept/final states

Notice that we allow ε-transitions in the definition of an NFA.

Definition 1.4 (Language of an NFA). An NFA N *accepts* a word $w \in \Sigma^*$ if there is atleast one run of w on N which ends in a final state. The language of N, $\mathcal{L}(N)$ is the set of all words accepted by N.

We say that two automatons, A_1 and A_2 are **equivalent** if and only if $\mathcal{L}(A_1) = \mathcal{L}(A_2)$.

Theorem 1.5 (Equivalence of NFA and DFA). Every non-deterministic finite automaton has an equivalent deterministic finite automaton.

Proof. Let $N = (Q, \Sigma, \delta_N, q_0, F)$ and define a function $E : 2^Q \to 2^Q$ given by

$E(R) = \{q \in Q \mid q \text{ can be reached from } R \text{ by travelling along 0 or more } \varepsilon \text{ transitions}\}$

We may now construct a deterministic finite automaton $D = (2^Q, \Sigma, \delta', q_0', F')$ where

$$F' = \{S \subseteq Q \mid S \cap F \neq \emptyset\}$$

$q_0' = E(\{q_0\})$ and

$$\delta'(R, a) = \{q \in Q \mid q \in E(\delta(r, a)) \text{ for some } r \in R\}$$

It isn't hard to showt that $\mathcal{L}(N) = \mathcal{L}(D)$. ■

Corollary 1.1. A language is regular if and only if there is a non-deterministic finite automaton accepting it.

Proof. Since each DFA is an NFA, for each regular language, there is a non-deterministic finite automaton accepting it. Conversely, for any language accepted by a non-deterministic finite automaton, there is a deterministic finite automaton accepting it and is therefore regular. ■

1.1.1 Properties of Finite Automata

In this section, we shall discuss the closure properties of regular languages.

Proposition 1.6. The class of regular languages is closed under the union operation. That is, if L_1 and L_2 are regular languages, then so is $L_1 \cup L_2$.

Proposition 1.7. The class of regular languages is closed under the complement operation. That is, if L is a regular language, then so is $\overline{L}$.

Corollary 1.2. The class of regular languages is closed under the union operation. That is, if L_1 and L_2 are regular languages, then so is $L_1 \cup L_2$.

Proposition 1.8. The class of regular languages is closed under the concatenation operation. That is, if L_1 and L_2 are regular languages, then so is $L_1 \circ L_2$.

Proposition 1.9. The class of regular languages is closed under the star operation. That is, if L is a regular language, then so is L^*.

1.2 Regular Expressions

Definition 1.10 (Regular Expression). R is said to be a regular expression over the alphabet Σ, if R is

1. a for some $a \in \Sigma$
2. ε - The language containing only the empty string
3. $\emptyset$ - The empty language
4. $R_1 \cup R_2$ where R_1 and R_2 are regular expressions
5. $R_1 \circ R_2$ where R_1 and R_2 are regular expressions

6. R^* where R is a regular expression

Sometimes, we may denote RR^* with R^+. We denote the language of a regular expression using $\mathcal{L}(R)$.

1.2.1 Equivalence with Finite Automata

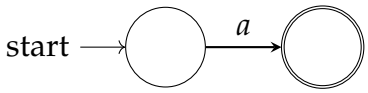
Theorem 1.11. A language is regular if and only if some regular expression describes it.

The proof of this theorem uses two lemmas.

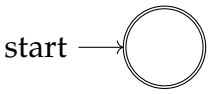
Lemma 1.12. If a language is described by a regular expression, then it is regular.

Proof. We shall give a constructive proof, by explicitly constructing the a non-deterministic finite automaton recognizing the language of the regular expression. We do this via an inductive method, by considering the six cases in the definition of a regular expression

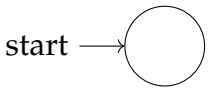
1. $R = a$ for some $a \in \Sigma$. Then, $\mathcal{L}(R) = \{a\}$ and the following NFA recognizes $\mathcal{L}(R)$



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



3. $R = \emptyset$. Then $\mathcal{L}(R) = \emptyset$ and the following NFA recognizes $\mathcal{L}(R)$



4. $R = R_1 \cup R_2$. The construction for this was discussed in the previous section.

5. $R = R_1 \circ R_2$. The construction for this was discussed in the previous section.
6. $R = R_1^*$. The construction for this was discussed in the previous section.

This completes the proof. ■

Lemma 1.13. If a language is regular, then there is a regular expression describing it.

Generalized NFAs

Definition 1.14 (GNFA). A *generalized non-deterministic finite automaton* is a 5-tuple $(Q, \Sigma, \delta, q_{\text{start}}, q_{\text{acc}})$ where

1. Q is the finite set of states
2. Σ is the input alphabet
3. $\delta : (Q - \{q_{\text{acc}}\}) \times (Q - \{q_{\text{start}}\}) \to \mathcal{R}$ is the transition function
4. q_{start} is the start state and
5. q_{acc} is the accept state

In simpler terms:

- The start state has transitions going to every other state but no arrow coming in from any other state
- There is only a single accept state, and it has arrows coming in from the other state but no arrows going to any other state. Furthermore, the accept state is not the same as the start state.
- Except for the start and accept states, one arrow goes from every state to every other state and also from each state to itself.

We shall now prove the second lemma by first converting the regular language into a DFA, then the DFA into a GNFA and finally the GNFA into a regular expression

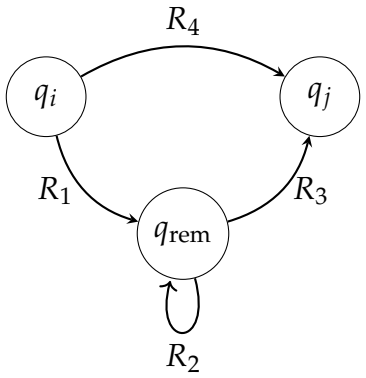
DFA to GNFA

This one's easy. Add a new start state with ε transitions to the old start state and a new accept state with ε transitions from all the old accept states. Finally, for each missing transition, add one, labelled with $\emptyset$.

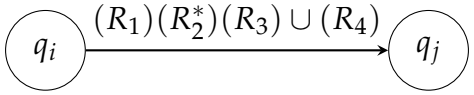
GNFA to Regular Expression

The idea here is to progressively construct a smaller GNFA at each step until one is left with a GNFA containing exactly two states, namely the start and the accept state. Then, the label of the edge joining them would be the Regular Expression equivalent to the language accepted by the GNFA.

We do this by first randomly picking out a state that we plan to remove, say q_{rem}. Now, we need to change the labels between every two nodes, so as to maintain the language accepted by the GNFA. Let us consider q_i and q_j with the following transitions:



The above is replaced by



This procedure may be written as a recursive algorithm. In the end, it should work.

1.3 The Pumping Lemma

This is a technique that is most commonly used to show that a language is non-regular.

Theorem 1.15 (Pumping Lemma). If L is a regular language, then there is a number p, called the pumping length where if s is any string in L of length at least p, then s may be divided into three pieces $s = xyz$ satisfying the following conditions:

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

Proof. Let $A = (Q, \Sigma, \delta, q_1, F)$ be a deterministic finite automaton recognizing L. Let $p = |Q|$, the number of states in A. If there are no strings in L of length atleast p, we are trivially done.

Let $s \in L$ have length $n \geq p$. Let us denote $q_i \in Q$ to be the state reached by the run of $s_1 \dots s_i$ on A. (Let us say, we begin with q_0) Then, the sequence $q_0, \dots, q_n$ has size $n + 1$ which is strictly greater than p. Therefore, due to the Pigeon Hole Principle, there must exist indices i and j such that $i < j$ and $q_i = q_j$. Let us now define $x = s_1 \dots s_i$, y = $s_{i+1} \dots s_j$ and $z = s_{j+1} \dots s_n$.

It isn't hard to see that $\hat{\delta}(q_i, y) = q_i$ and thus, $xy^iz \in L$ for all $i \geq 0$. It only remains to show that $|xy| \leq p$ but this is trivial since there must be a repetition in the first $p + 1$ states in the run, giving us that $j \leq p$. This completes the proof. ■

It is important to keep in mind that the Pumping Lemma is not a regularity test, that is, a language satisfying the conditions of the lemma need not be regular.

Example. Show that the language $L = \{a^nb^n \mid n \geq 0\}$ over the alphabet $\Sigma = \{a, b\}$ is not regular.

Proof. Assume L is regular. Let p be the pumping length of L. Consider the word $w = a^pb^p \in L$. We shall now take cases on the format of the substring y of w as dictated by the pumping lemma.

1. $y \in a^+$. Then, it is obvious that $xz \notin L$
2. $y \in b^+$. Then, it is obvious that $xz \notin L$
3. $y \in a^+b^+$. Then, $y = a^ib^j$ for some $i, j > 0$. Then, the word xy^2z is given by $a^pb^ja^ib^p \notin L$

This derives a contradiction. Thus, L is not regular. ■

Example. Show that the language $L = \{ww \mid w \in \{0,1\}^*\}$ over the alphabet $\Sigma = \{0,1\}$ is not regular.

Proof. Suppose L is regular. Let p be the pumping length of L. Consider the word $w = 0^p10^p1$. Then, y must be a substring of 0^p, due to the third condition of the lemma. Then, it is obvious that $xz \notin L$, deriving a contradiction. ■

1.4 Myhill-Nerode Theorem

These notes are taken from the 2019 offerring of the course CS310: Automata Theory at IIT Bombay.

1.4.1 DFA Minimization

Definition 1.16 (Equivalence of States). Let $A = (Q, \Sigma, \delta, q_0, F)$ be a DFA. $p, q \in Q$ are said to be *equivalent* if for each word w, $\hat{\delta}(p, w) \in F$ if and only if $\hat{\delta}(q, w) \in F$. States that are not equivalent are said to be *distinguishable*.

For a DFA $A = (Q, \Sigma, \delta, q_0, F)$ let $\mathcal{B}$ be the partition of equivalent states in A. We may then define the minimized DFA $A' = (\mathcal{B}, \Sigma, \delta', \mathcal{B}_0, \mathcal{B}_f)$ where

- For each $B, B' \subseteq \mathcal{B}$, $\delta'(B, a) = B'$ if there are $q \in B$ and $q' \in B'$ such that $\delta(q, a) = q'$.
- $\mathcal{B}_f = \{B \subseteq \mathcal{B} \mid B \cap F \neq \emptyset\}$
- $q_0 \in \mathcal{B}_0$

Proposition 1.17. Let A be a minimized DFA. Then, no DFA with fewer states than that of A recognizes $\mathcal{L}(A)$.

Proof. Suppose there is a DFA A' such that A' has fewer states than A and $\mathcal{L}(A) = \mathcal{L}(A')$. Then, by definition q_0, the initial state in A is equivalent to q_0', the initial state in A'. We shall show that for each $q \in A$, there is $q' \in A'$ such that $q \cong q'$.

We know that all states of A are reachable from its initial state, by construction. Then, there is a word $w \in \Sigma^*$ such that $\hat{\delta}(q_0, w) = q$. Let now, $q' \overset{\triangle}{=} \hat{\delta}'(q_0', w)$.

If $q \not\equiv q'$, then there is a word $\tilde{w}$ such that $\hat{\delta}(q, \tilde{w}) \in F$ but $\hat{\delta}'(q', \tilde{w}) \notin F'$ and thus, $\hat{\delta}(q_0, w\tilde{w}) \in F$ but $\hat{\delta}'(q_0', w\tilde{w}) \notin F'$, contradicting the fact that q_0 and q_0' are equivalent.

But since $|A| > |A'|$, due to the Pigeon Hole Principle, there exist states $q_1, q_2 \in Q$ that are equivalent to the same state q' in A and are therefore equivalent to one another. This contradicts the minimality of A, completing the proof. ■

1.4.2 Residual Languages

Definition 1.18 (Residual Language). Given a language $L \subseteq \Sigma^*$ and $w \in \Sigma^*$, the residual of L with respect to w is the language

$$L^w = \{u \in \Sigma^* \mid wu \in L\}$$

A language $L' \subseteq \Sigma^*$ is a residual of L if $L' = L^w$ for at least one $w \in \Sigma^*$.

Definition 1.19 (Language of a State). Let $A = (Q, \Sigma, \delta, q_0, F)$ be a DFA and $q \in Q$. The language recognized by q, denoted by $\mathcal{L}(A_q)$ is the language recognized by $A_q = (Q, \Sigma, \delta, q, F)$.

Proposition 1.20. Let $A = (Q, \Sigma, \delta, q_0, F)$ be a DFA recognizing a language L

1. For each $w \in \Sigma^*$, there is a $q \in Q$ such that $\mathcal{L}(A_q) = L^w$
2. For each $q \in Q$ reachable from q_0, there is a $w \in \Sigma^*$ such that $\mathcal{L}(A_q) = L^w$

Proof.

1. Let $q = \hat{\delta}(q_0, w)$ and $\widetilde{w} \in \mathcal{L}(A_q)$. Then, obviously, $w\widetilde{w} \in \mathcal{L}$ and therefore, $\widetilde{w} \in L^w$, giving us $\mathcal{L}(A_q) \subseteq L^w$. Conversely, for each $\widetilde{w} \in L^w$, we have that $w\widetilde{w} \in L$, equivalently, $\hat{\delta}(q_0, w\widetilde{w}) \in F$, that is, $\hat{\delta}(q, \widetilde{w}) \in F$, implying that $\widetilde{w} \in \mathcal{L}(A_q)$. This gives us $L^w \subseteq \mathcal{L}(A_q)$ and thus, $\mathcal{L}(A_q) = L^w$.
2. Let $w \in \Sigma*$ be such that $\hat{\delta}(q_0, w) = q$. We shall show that $\mathcal{L}(A_q) = L^w$. This proof is similar to the one in the previous case and is therefore omitted.

■

Definition 1.21 (Canonical Deterministic Automaton). Let $L \subseteq \Sigma^*$ be a language. The canonical deterministic automaton for L is $C_L = (Q_L, \Sigma, \delta_L, L, F_L)$ where

- Q_L is the set of residuals of L, that is, $Q_L = \{L^w \mid w \in \Sigma^*\}$
- $\delta_L(K, a) = K^a$ for each $K \in Q_L$ and $a \in \Sigma$
- $F_L = \{K \in Q_L \mid \varepsilon \in K\}$

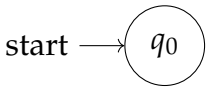
It is possible that Q_L is infinite and therefore, the CDA need not always be a DFA. But obviously $Q_L \subseteq \Sigma^*$ and is countable.

Example. Consider the language L given by the regular expression $L = aa^*b$, or equivalently $L = a^+b$. Construct the canonical deterministic automaton for L.

Proof. The residual languages are

- $L^\varepsilon = aa^*b$
- $L^a = a^*b$
- $L^{a^+b} = \varepsilon$
- $L^b = \emptyset$

This gives us the following automaton



■

Proposition 1.22. For a language L, $C_L = (Q_L, \Sigma, \delta_L, L, F_L)$ recognizes L.

Proof. Let $w \in \Sigma*$. We shall show by induction on the length of w that $w \in L$ if and only if $w \in \mathcal{L}(C_L)$. The base case with $|w| = 0$, or equivalently, $w = \varepsilon$ is trivial. Let $w = ax$ where $a \in \Sigma$ and $x \in \Sigma*$. Then, $w \in \mathcal{L}(C_L)$ if and only if $x \in \mathcal{L}(C_{L^a})$ if and only if $x \in L^a$ if and only if $ax \in L$. This completes the proof. ■

Proposition 1.23. If L is regular, then C_L is the unique minimum DFA upto isomorphism that recognizes L.

Proof. Let $A = (Q, \Sigma, \delta, q_0, F)$ be a DFA that recognizes L. Consider the mapping $\phi : Q \to Q_L$ such that $\phi(q) = \mathcal{L}(A_q)$. Since this function is surjective, we must have that $|Q| \geq |Q_L|$. Therefore, C_L is the minimal automaton for L.

Suppose A is also minimal. We shall show that A is isomorphic to C_L. Obviously $|Q| = |Q_L|$. And, for any $q \in Q$, $\delta(q, a) = q'$ if and only if $\mathcal{L}(A_{q'}) = \mathcal{L}(A_q)^a$ if and only if $\delta_L(\phi(q), a) = \phi(q')$. This establishes the isomorphism. ■

Theorem 1.24 (Myhill-Nerode Theorem). A language $L \subseteq \Sigma^*$ is regular if and only if L has finitely many residuals.

Proof. If L is not regular, then there is no DFA recognizing it, and thus the canonical deterministic automaton for L must be infinite, in other words, L has infinitely many residuals. Conversely, if L is regular, then there is a DFA recognizing it and thus, a minimal DFA recognizing it. Due to the previous theorem, this minimal DFA is isomorphic to the canonical deterministic automaton, implying that $|Q_L|$ is finite. This completes the proof. ■

Chapter 2

Context Free Languages

2.1 Context Free Grammars

Definition 2.1 (Context-Free Grammar). A *context-free grammar* is a 4-tuple (N, T, P, S) where

1. N is a finite set of *non-terminals*
2. T is a finite set of *terminals*
3. P is a finite set of *production rules* with each rule being a non-terminal followed by an arrow and a string of both non-terminals and terminals
4. $S \in N$ is the start variable

Consider the following grammar:

$$G = (\{A, B\}, \{0, 1\}, P, A)$$

where the production rules are given by

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

Note that the above rules may be written more compactly in the following form

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

We use a grammar to generate a string over $N \cup T$. The sequence of substitutions to obtain a string is called a *derivation*. For example, the following is a valid derivation of 000#111 in the grammar G

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111$$

Definition 2.2 (Yields, Derives, Language). If $u, v, w \in (N \cup T)^*$ and $A \to w$ is a rule of the grammar, we say that uAv *yields* uwv, written $uAv \Rightarrow uwv$. Similarly, we say u *derives* v, written $u \overset{*}{\Rightarrow} v$ if $u = v$ or if a sequence $u_1, \ldots, u_k$ exists for $k \geq 0$ and

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

The *language of the grammar* is formally given by

$$\{w \in T^* \mid S \overset{*}{\Rightarrow} w\}$$

For any $A \in N$, if $A \overset{*}{\Rightarrow} \alpha$ where $\alpha \in (N \cup T)^*$, then we say α is a *word* of A.

Example. Show that the language $L = \{a^n b^n \mid n \geq 0\}$, over the alphabet $\Sigma = \{a, b\}$ is context-free.

Proof. Consider the context-free grammar described by

$$G = (\{S\}, \Sigma, P, S)$$

where the production rules are:

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

Trivially note that $\mathcal{L}(G) = L$ and we are done. ■

Example. Show that all regular languages are context-free languages.

Proof. Let L be a regular language and $A = (Q, \Sigma, \delta, q_0, F)$ be a DFA recognizing L. And let $Q = \{q_0, \ldots, q_n\}$ where $n \geq 0$. For each $q_i \in Q$, introduce a non-terminal R_i and let the set of terminals be Σ. The start variable is R_0. Finally, add in the production rules $R_i \to aR_j$ for each $a \in \Sigma$ whenever $\delta(q_i, a) = q_j$ and the production rules $R_i \to \varepsilon$ whenever $q_i \in F$. ■

One may note that at each step of a derivation in a context-free grammar, we may be presented with many choices, since there may be more than one non-terminal in the string at that stage. These choices can be made predictable by imposing certain rules, such as:

Leftmost derivation Always expand the leftmost non-terminal. These are denoted by $\overset{lm}{\Rightarrow}$ and $\overset{lm*}{\Rightarrow}$

Rightmost derivation Always expand the rightmost non-terminal. These are denoted by $\overset{rm}{\Rightarrow}$ and $\overset{rm*}{\Rightarrow}$

2.1.1 Ambiguity

Definition 2.3 (Parse Tree). For a grammar $G = (N, T, P, S)$, a *parse tree* for G is a labelled tree with the following conditions:

- leaf label is $X \in N \cup T \cup \{\varepsilon\}$. If $X = \varepsilon$, the leaf has no siblings
- internal node label is $A \in N$
- if an internal node label is $A \in N$ and its children are $X_1, \ldots, X_n$ from left to right, then $A \to X_1 \ldots X_n \in P$

The *yield* of a parse tree is the word formed by all the leaves from left to right.

Definition 2.4. A string w is said to be derived *ambiguously* in a context-free grammar G if it has two or more different leftmost derivations. Grammar G is said to be *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 context-free languages however, can be generated only by ambiguous grammars. Such languages are called *inherently ambiguous*.

Example. Show that the language $\{a^i b^j c^k \mid i = j \vee j = k\}$ is inherently ambiguous.

2.1.2 Chomsky Normal Form

Definition 2.5. A context-free grammar is said to be in *Chomsky normal form* if every rule is of the form

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

where $a \in T$ and $A, B, C \in N$, except that B and C may not be the start variable. In addition, we permit the rule $S \to \varepsilon$ where S is the start variable.

Theorem 2.6. Any context-free language is generated by a context-free grammar in Chomsky normal form.

The proof of the above theorem will be by construction. We shall first add a new start variable (non-terminal). Then, we eliminate all ε-rules, of the form $A \to \varepsilon$ and all unit rules, of the form $A \to B$. In both cases, we patch up the grammar to be sure that it still generates the same language.

Proof. Introduce a new start variable S_0 and the rule $S_0 \to S$ where S was the original start variable. This change will guarantee that the start variable does not occur on the right-hand side of a production rule.

Next, we take care of all ε-rules. We remove a rule of the form $A \to \varepsilon$ where A is not the start variable. Then, for each rule of the form $R \to uAv$, add another rule $R \to uv$. Note that we do so for *each* occurrence of an A. That means, the rule $R \to uAvAw$ causes us to add $R \to uvAw \mid uAvw \mid uvw$. If we have the rule $R \to A$, we add $R \to \varepsilon$ unless we had previously reoved the rule $R \to \varepsilon$.

Next, we handle all unit rules. We remove a unit rule $A \to B$. Then, whenever a rule $B \to u$ appears, we add the rule $A \to u$ unless this was a unit rule previously removed. ($u \in (N \cup T)^*$). We repeat these steps until we eliminate all unit rules.

Finally, we convert all remaining rules into the proper form. We replace each rule $A \to u_1 \cdots u_k$ where $k \geq 3$ and each u_i is a variable or a terminal symbol with the rules $A \to u_1 A_1$ and so on up to $A_{k-2} \to u_{k-1} u_k$. The A_i's are fresh variables. We may replace any terminal u_i in the preceeding rule(s) with the new variable U_i and add the rule $U_i \to u_i$. ■

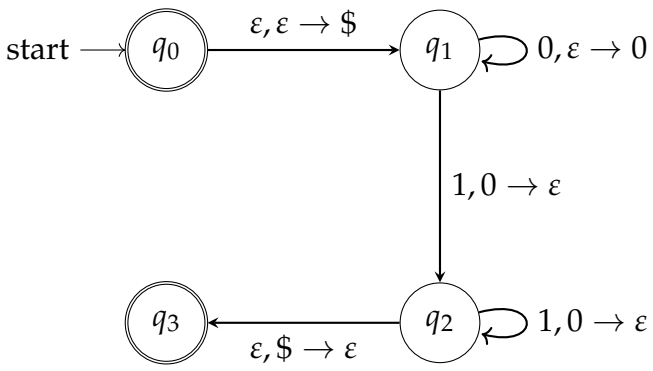
2.2 Pushdown Automata

These are similar to non-deterministic finite automata but have an extra component called a *stack*, which provides additional memory, thus making it more powerful than an NFA.

Definition 2.7 (Pushdown Automaton). A *pushdown automaton* is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where Q, Σ, Γ and F are all finite sets and

1. Q is the set of states
2. Σ is the input alphabet
3. Γ is the stack alphabet
4. $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to 2^{Q \times \Gamma_\varepsilon}$ is thet transition function
5. $q_0 \in Q$ is the start state
6. $F \subseteq Q$ is the set of all accept states

This is best illustrated by an example. Let us construct a Pushdown Automaton accepting the language $\{0^n 1^n \mid n \geq 0\}$. Indeed, consider the following state-transition diagram:



Remark. We may modify the definition of a pushdown automata by making it a 7-tuple, including a symbol Z_0, called the *start of stack* symbol. This does not change the power of the automaton, since this behaviour can be simulated with the 6-tuple pushdown automaton as is seen above. Here, \$ plays the role of the start symbol.

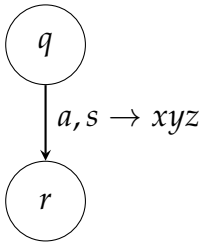
2.2.1 Equivalence with CFGs

Theorem 2.8. A language is context-free if and only if some pushdown automaton recognizes it.

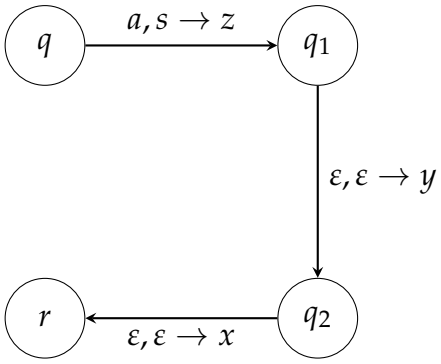
We shall prove both directions of the statement as two different lemmas.

Lemma 2.9. If a language is context-free, then some pushdown automaton recognizes it.

We shall use a shorthand while drawing the automaton. The notation $(r, u) \in \delta(q, a, s)$ is used to mean that when q is the state of the automaton, a is the next input symbol and s is the symbol on the top of the stack, the PDA may read the a and pop the s, then push the string u onto the stack and go on to the state r.



is the same as



Notice that the order of pushing the symbols onto the stack is reversed. That is, first z, then y and finally x.

The states of the automaton we wish to create are $Q = \{q_{\text{start}}, q_{\text{loop}}, q_{\text{accept}}\} \cup E$ where E is the set of extra states that are needed for implementing the shorthand

as shown above. We now examine 3 cases for constructing the automaton from the description of the grammar $G = (N, T, P, S)$. First, we initialize the stack to contain the symbols $\$S$ as $\delta(q_{\text{start}}, \varepsilon, \varepsilon) = \{(q_{\text{loop}}, S\$)\}$.

1. Top of the stack is a non-terminal. Let

$$\delta(q_{\text{loop}}, \varepsilon, A) = \{(q_{\text{loop}}, w) \mid \text{where } A \to w \in R\}$$

2. Top of the stack is a terminal. Let

$$\delta(q_{\text{loop}}, a, a) = \{(q_{\text{loop}}, \varepsilon)\}$$

3. Top of the stack is the marker \$. Let

$$\delta(q_{\text{loop}}, \varepsilon, \$) = \{(q_{\text{accept}}, \varepsilon)\}$$

This completes the proof of the lemma. We may now move on to proving the other direction.

Lemma 2.10. If a pushdown automaton recognizes some language, then it is context free.

Proof. ■

2.3 Pumping Lemma for CFLs

Theorem 2.11. If A is a context-free language, then there is a number p, the pumping length where, if s is any string in A of length at least p, then s may be divided into five pieces $s = uvxyz$ satisfying the conditions

1. for each $i \geq 0$, $uv^ixy^iz \in A$
2. $|vy| > 0$
3. $|vxy| \leq p$

Proof. Let $G = (N, T, P, S)$ be a CFG for the CFL A and b be the maximum number of symbols in the right hand side of a rule (suppose that $b \geq 2$). Then, one may

trivially note that any string of length atleast $b^h + 1$ must have a parse tree of height atleast $h + 1$.

Define $p = b^{|N|+1}$. Then any string s of length atleast p would have a parse tree of height atleast $|N| + 1$. Let us consider the tree τ with minimum number of nodes, in which case, there exists a path in the tree from root to leaf, containing atleast $|N| + 1$ internal nodes. Consequently, some non-terminal R appears more than once. Let R be the non-terminal that repeats in the lowest $|N| + 1$ internal nodes. Then, we have a division for the string s into $uvxyz$ as shown in the figure. It is then obvious that $uv^ixy^iz \in A$ for each $i \geq 0$. We need only show that conditions 2 and 3 hold.

Suppose both v and y were ε, then the parse tree obtained by substituting the smaller subtree for the larger would have fewer nodes than τ, a contradiction. Next, we know that the upper occurrence of R generates vxy but since R falls within the bottom $|N| + 1$ non-terminals, a tree of this height can generate a string of length at most $b^{|N|+1} = p$, giving us that $|vxy| \leq p$. This completes the proof. ■

Example. Show that the language

$$B = \{a^n b^n c^n \mid n \geq 0\}$$

over the alphabet $\Sigma = \{a, b, c\}$ is not context-free.

Proof. ■

Chapter 3

Turing Machines

This section of the notes are taken from both "Automata Theory and Computation" by Dexter Kozen and "Theory of Computation" by Michael Sipser.

Definition 3.1 (Turing Machine). A *deterministic* one-tape Turing Machine is a 7-tuple

$$T = (Q, \Sigma, \Gamma, \delta, q_{\text{start}}, q_{\text{acc}}, q_{\text{rej}})$$

- Q is a finite set of states
- Σ is a finite set called the input alphabet with $\sqcup \notin \Sigma$
- Γ is a finite set called the tape alphabet. Such that $\Sigma \in \Gamma$ and $\sqcup \in \Gamma$.
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function
- q_{start} is the start state
- q_{acc} is the accept state
- q_{rej} is the reject state such that $q_{\text{rej}} \neq q_{\text{acc}}$

The *only* halting states of a Turing Machine are q_{acc} and q_{rej}. That is to say, unless the Turing Machine reaches either one of the two states, it may not halt at all. And, whenever a Turing Machine halts on an input, it either accepts it or rejects it. A Turing machine is said to be a *decider* if it halts on all possible finite inputs.

- The set of all finite strings over Σ that a Turing machine accepts is said to be

the language of the Turing machine

- A language is said to be *recursively enumerable* if it is $\mathcal{L}(M)$ for some Turing macine M.
- A language is said to be *recursive* or *decidable* if it is $\mathcal{L}(M)$ for some *decider* M

A property P is said to be *decidable* if the set of all strings having property P is a *recursive* set. Similarly, P is said to be *semidecidable* if the set of strings having property P is a *recursively enumerable* set.

3.1 Equivalent Models

Turing Machines are a remarkably robust model of computation. There are several different flavors of Turing machines that are computationally equivalent.

3.1.1 Multi-tape Turing Machines

3.1.2 Two-Way Infinite Tape

3.1.3 Two Stacks

A machine with a two-way *read-only* input head and two stacks is as powerful as a Turing machine. This is possible by storing the tape contents to the left of the head on one stack and the tape contents to the right of the head on the other. The motion of the head is simulated by popping a symbol off one stack and pushing it onto the other.

3.1.4 Counter Automata

A *k-counter automaton* is a machine equipped with a two-way read-only input head and k integer counters. Each counter can store an arbitrary non-negative integer. In each step, the automaton can independently increment or decrement its counters and test them for 0 and can move its input head one cell in either direction. It *cannot* write on the tape.

We may simulate a stack using two counters. Suppose, without loss of generality that the the stack alphabet contains only two symbols, say 0 and 1. This is because we can encode finitely many stack symbols as binary numbes of fixed

length, say m; then pushing or popping one stack symbol is simulated by pushing or popping m binary digits. Then the contents of the stack can be regarded as a binary number whose least significant bit is on top of the stack. The simulation maintains this number in the first of the two counters and uses the second to effect the stack operations. To simulate pushing a 0 onto the stack, we need to double the value in the first counter. This is done by entering a loop that repeatedly subtracts one from the first counter and adds two to the second until the first counter is 0. We may then transfer the value back to the first counter. To push 1, the operation is the same except the value of the second counter is incremented once at the end. Similarly, one can encode popping.

Since a two-stack machine can simulate an arbitrary Turing machine, it follows that a 4-counter automaton can simulate an arbitrary Turing machine. However, it is possible to simulate a 4-counter automaton using a 2-counter automaton. When the 4-counter automaton has the value i, j, k, l in its counters, the two-counter automaton will have the value $2^i3^j5^k7^l$ in its first counter. It uses its second counter to effect the counter operations of the four counter automaton. FOr exxample, if the four-counter automaton wanted to add one to k, then the two counter automaton would have to multiply the value in the first counter by 5. This is done in the same way as above, adding 5 to the second counter for every 1 that we subtract from the first counter. Simulating a test for 0 is equivalent to testing divisibility by 2, 3, 5 or 7; which is trivial using only 2 counters.

It is important to note that the 1-counter automaton is not as powerful as an arbitrary Turing machine, although they can accept non-CFLs. For example, the set $\{a^nb^nc^n \mid n \geq 0\}$ can be accepted by a one-counter automaton.

3.1.5 Enumeration Machines

Define an *enumration machine* as follows. It has a finite control and two tapes, a read/write *work tape* and a write-only *output tape*. The work tape head can move in either direction and can read and write any element of Γ. The output tape head moves right one cell when it writes a symbol, and it can only write symbols in Σ. There is no input and no accept or reject state. In its start state, both tapes are blank. It moves according to its transition function like a Turing machine, occassionally writing symbols on the output tape as determined by the transition function. At some point it may enter a special *enumeration state*, which is just a distinguished state of its finite control. When that happens, the string currently written on the output tape is said to be *enumerated*. The output tape is then automatically erased and the output head moved back to the beginning of the tape,

while the work tape is left intact, and the machine continues from that point. Note that it is possible for a string to be enumerated *more than once*.

We shall now show that Enumeration machines and Turing machines are equivalent in computational power:

Theorem 3.2. The family of sets enumerated by enumeration machines is exactly the family of recursively enumerable sets. That is, a set is a language of some enumeration machine if and only if it is the language of some Turing machine.

Proof. We shall first show that given an enumeration machine E, we can construct a three-tape Turing machine M such that $\mathcal{L}(M) = \mathcal{L}(E)$. We first copy x to one of the three tapes. THen simulate E using the other two tapes where one of them functions as E's work tape and the other as its output tape. Then, for each string enumerated by E, M compares this string to x and accepts if they match.

To show the converse, we first enumerate $\Sigma*$ as $\{s_1, s_2, \ldots\}$. Then, for each $i \in \mathbb{N}$, we run M for i steps on each input $s_1, \ldots, s_i$. If any computations accept, print out the corresponding s_j. Thus, if M accepts some string s, eventually it will appear on the list generated by E and will be printed out at some instance, in fact, it would be printed out infinitely often. ■

Definition 3.3 (Algorithm). An *algorithm* is a Turing machine that halts for any input. That is, an algorithm is a *decider*.

Thus, a language is said to be recursive if and only if there is an algorithm for deciding it.

Chapter 4

Decidability and Undecidability

In this chapter we shall explore some decidable and undecidable languages.

4.1 Decidabiity

Proposition 4.1. The language

$$A_{\text{DFA}} = \{\langle B, w\rangle \mid B \text{ is a DFA accepting } w\}$$

is *decidable*.

Proof. The idea is simple, we simulate B on input w. If the simulation ends in an accept state, accept. If not, reject. We may represent this computation using a two-tape Turing machine. The first tape keeps track of B's state and the second keeps track of B's current position on the input w. ■

Similarly one can show that A_{NFA} is decidable. The only tweak to make to the proof is to first convert the NFA into an equivalent DFA, which can be done using the subset construction.

Proposition 4.2. The language

$$A_{\text{CFG}} = \{\langle G, w\rangle \mid G \text{ is a CFG that generates } w\}$$

is decidable.

The proof of this requires the following lemma:

Lemma 4.3. Let G be a grammar in Chomsky Normal Form. Then, for any $w \in \mathcal{L}(G)$, any derivation of w has $2|w| - 1$ steps.

Proof. Recall that the production rules for a grammar in Chomsky Normal Form look like $A \to BC$ or $A \to a$ where A, B, C are non-terminals and a is a terminal. Then, at each step of the derivation, the length of the string may increase by atmost 1. We may then keep a track of the number of termminals and non-terminals. Let n be the length of w. We begin in the configuration $(T, NT) = (0, 1)$ and finally end up in $(T, NT) = (n, 0)$. Note that for each T-derivation, the value of T increases by 1 but the length of the string remains the same while for an NT-derivation, the value of NT increases by 1 and so does the length of the word. Thus, we need exactly n T-derivations and $n - 1$ NT-derivations. This completes the proof. ■

Proof of Proposition. The algorithm is quite simple. We first convert G to an equivalent grammar in Chomsky Normal Form. Then, list all derivations with $2n - 1$ steps (obviously finite), where N is the length of w. If any of these derivations generate w, accept. If not, reject. ■

Finally, we have:

Theorem 4.4. Every context-free language is decidable.

Proof. We would be requiring the Turing machine that we designed in the previous problem, call it S. Let A be a context-free language and G be a CFG for A. Then, design a Turing machine M_G that, on input w, runs S on input $\langle G, w \rangle$. If this machine accepts, accept. If not, reject. ■

4.2 Undecidability

Proposition 4.5. Some languages are not Turing recognizable/recursively enumerable.

Proof. Since any Turing machine, M can be encoded into a string $\langle M \rangle$, over a finite alphabet Σ, the number of Turing machines must be countable. But, the set of all possible languages is $\mathcal{P}(\Sigma^*)$ which is known to be uncountable due to *Cantor's Diagonalization*. This finishes the proof. ■

Definition 4.6 (Universal Turing Machine). A *universal Turing machine* is one that takes as input the description of another Turing machine M and a string x, to be given as input to M and simulates the same. Then, the language of said universal Turing machine U is

$$\mathcal{L}(U) := \{\langle M, x\rangle \mid x \in \mathcal{L}(M)\}$$

Of course, the machine first checks its input $\langle M, x\rangle$ to make sure that M is a valid encoding of a Turing machine and that x is a valid encoding of a string over M's input alphabet. If not, immediately rejects. Obviously, if M accepts x, then the simulation would come to an end, if not, then the simulation may go on indefinitely. Thus, U *recognizes* $\mathcal{L}(U)$ but does not decide it.

Proposition 4.7. The language

$$A_{\text{TM}} = \{\langle M, x\rangle \mid x \in \mathcal{L}(M)\}$$

is not recursive/decidable.

Proof. The proof follows the same spirit as *Cantor's Diagonalization* for showing $|\mathbb{N}| < |\mathbb{R}|$. Suppose A_{TM} is decidable and let H be a decider for A_{TM}. Then, by definition:

$$H(\langle M, w\rangle) = \begin{cases} \text{accept} & \text{if } M \text{ accepts } w \\ \text{reject} & \text{if } M \text{ does not accept } w \end{cases}$$

Let us now construct a Turing machine D with H as a subroutine. This machine takes as input the encoding of a Turing machine M and runs H on the input $\langle M, \langle M\rangle\rangle$ and outputs the opposite of what H outputs. Recall that H is a decider and must outupt something. Then,

$$D(\langle M\rangle) = \begin{cases} \text{accept} & \text{if } M \text{ does not accept } \langle M\rangle \\ \text{reject} & \text{if } M \text{ accepts } \langle M\rangle \end{cases}$$

But this would imply

$$D(\langle D\rangle) = \begin{cases} \text{accept} & \text{if } D \text{ does not accept } \langle D\rangle \\ \text{reject} & \text{if } D \text{ accepts } \langle D\rangle \end{cases}$$

Which is an obvious contradiction. ■

A language is said to be *co-Turing recognizable* if it is the complement of a Turing-recognizable language.

Theorem 4.8. A language is decidable if and only if it is Turing recognizible and co-Turing recognizable.

Proof. Suppose a language A is decidable(recursive). Then, since the Turing machine M for A would halt on each input, we can trivially construct a Turing machine for $\overline{A}$ that simply outputs the negation of M on said input.

Conversely, let M_1 and M_2 be the Turing machines recognizing A and $\overline{A}$. We now construct a two-tape Turing machine M that runs both M_1 and M_2 in parallel (on both the tapes) and continues until one of them accepts. Now, any string w is either in A or $\overline{A}$ and thus computation on either one of the tapes must terminate and thus M is a decider for A. This completes the proof. ■

Corollary 4.3. $\overline{A_{\text{TM}}}$ is not Turing-recognizable.

Proof. If it were then due to the previous theorem, A_{TM} would be recursive, a contradiction. ■

Chapter 5

Reducibility

Informally speaking, a *reduction* is a way of converting one problem to another problem in such a way that a solution to the second problem can be used to solve the first problem.

Theorem 5.1 (Halting Problem). The language

$$HALT_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM halting on } w\}$$

is undecidable.

5.1 Rice's Theorem

Theorem 5.2 (Rice).