THEORY OF COMPUTATION

SIDDHANT CHAUDHARY

These are my course notes for the **THEORY OF COMPUTATION** course that I took in my third semester in CMI. The notes are mostly self contained. Throughout the document, the symbol ■ stands for QED.

Contents

Date: August 2020.

1. Languages

First, we begin with a formal definition of a *language*. This is an important idea as we move on to study models of computation.

Definition 1.1. Let Σ be a *finite* set. The Σ is also called the *alphabet*. Let Σ^* denote the set of all *finite* words of the symbols in Σ (this can be formally defined as the set of all sequences taking values in Σ where only finitely many terms belong to Σ, but we don't need to make that too formal here).

For instance, the alphabet of the English language is the set $\{A, B, C, D, ..., Z\}$ along with punctuation marks and case letters.

Definition 1.2. A *language* over Σ is a subset of Σ^*.

These are formal notions which will be useful when we formally define computational tools in further sections.

2. Finite Automatons

Definition 2.1. A *finite automaton* M is a 5-tuple $(Q, \Sigma, \Delta, Q_0, F)$ where the following hold:

(1) Q is a *finite* set of *states*.
(2) Σ is a finite alphabet.
(3) $\Delta \subset Q \times \Sigma \times Q$ is the transition relation.
(4) $Q_0 \subset Q$ is the set of *initial states*.
(5) $F \subset Q$ is the set of *final states* (also called *accepting states*).

The best way to visually represent finite automatons is via transition diagrams. I won't go through any examples here because this definition is fairly easy to understand.

2.1. Non-determinism.

Determinism in automatons in some sense refers to a well-defined and well-predictable automaton, i.e given any state and any symbol of the alphabet, there is exactly one edge going out of the state, and hence the output given by the automaton will be well-traceable. The same is not true for other automatons. We now formally define these notions.

Definition 2.2. Let M be a finite automaton. Then M is said to be a *deterministic finite automaton* (or DFA) if M has exactly one initial state and from each state and each symbol of the alphabet there is exactly *one* outgoing edge (or in other words Δ is a function denoted by $\delta : Q \times \Sigma \to Q$). If Δ is just a relation (and not necessarily a function), then M is called a *non-deterministic finite automaton* (or NFA).

The above definition makes precise the notion of an automaton being well-predictable.

Definition 2.3. Let M be a finite automaton, and let $w = a_1a_2...a_n$ be a word, where each $a_i \in \Sigma$. The *run* of M on w is defined as the walk

$$q_0a_1q_1a_2q_2...a_nq_n$$

where $q_i \in Q$, $q_0 \in Q_0$ and $(q_i, a_{i+1}, q_i) \in \Delta$ for all suitable i. Basically, this is one possible route that the automaton can take. A run is *accepting* if $q_n \in F$, and define

$$L(M) = \{w \in \Sigma^* | M \text{ has an accepting run on } w\}$$

We also say that M *recognizes* or *accepts* the language $L(M)$.

This motivates the following idea.

Definition 2.4. A language is said to be a *regular language* if some deterministic finite automaton recognizes it.

2.2. **Equivalence of NFAs and DFAs.** Now, we will show that all NFAs can be converted into a DFA. To make this more formal, given a language L which is accepted by an NFA, there exists a DFA which accepts the same language. This means that we can work with NFAs if that is easier.
Let $A = (Q, \Sigma, \Delta, F, Q_0)$ be an NFA that accepts a language L. Let $B = (Q', \Sigma, \delta', F', Q_0')$ be a DFA which is defined as follows: $Q' = \mathscr{P}(Q)$ (power set), and $Q_0' = \{Q_0\}$ (so we collapsed all initial state into one initial state). For $q' \in Q'$ and $a \in \Sigma$, define

$$\delta'(q', a) = \{q \in Q | \exists q_0 \in q' \text{ such that } (q_0, a, q) \in \Delta\}$$

i.e, there is an edge from a subset of Q to another subset of Q with label a iff. there is a state in the first set which has an outgoing edge with label a to a state in the second set. Next, we have

$$F' = \{S \subset Q | S \cap F \neq \phi\}$$

i.e, the set of final states is the set of all subsets of Q that contain a final state in the original automaton. Upto this point, this construction is pretty natural.
Now we prove that the languages accepted by both these automatons are the same, which will finish the goal of this section. Before proving the main theorem, we prove a lemma and introduce a new notation.

Definition 2.5. Let M be a finite automaton, and let q, q' be states. We use the notation $q \xrightarrow{w} q'$ if there is a run starting at q and ending at q' on the word w in M.

Lemma 2.1. *Let A be an NFA, and let B be the corresponding DFA obtained by the subset construction. Let Q_1 be a subset of Q such that*

$$Q_0 \xrightarrow{w} Q_1 \text{ (in the DFA } B)$$

for some word w. Then,

$$q_1 \in Q_1 \iff \exists q_0 \in Q_0 \text{ such that } q_0 \xrightarrow{w} q_1 \text{ in the NFA } A$$

In simple words, Q_1 is precisely the set of all those states which are reachable from some state in Q_0 on w.

Proof. We prove this by induction on the length of w. For the base case, suppose $w = \epsilon$, and suppose $Q_0 \xrightarrow{\epsilon} Q_0$ in B, implying that $Q_1 = Q_0$. The statement is then vacuously true, so the base case is established.

Suppose the statement is true for some fixed length of words, and consider a word w' of length one greater than this length. Let

$$w' = wa$$

where $a \in \Sigma$, so that w has length length smaller by one. Suppose in the DFA B we have

$$Q_0 \xrightarrow{w} Q_1 \xrightarrow{a} Q_2$$

and observe that $\delta'(Q_1, a) = Q_2$. We need to show that

$$q_2 \in Q_2 \iff \exists q_0 \in Q_0 \text{ such that } q_0 \xrightarrow{wa} q_2 \text{ in the NFA } A$$

First, suppose $q_2 \in Q_2$. So, there is some $q_1 \in Q_1$ such that $(q_1, a, q_2) \in \delta'$ (this is the definition of edges in B). Moreover, by inductive hypothesis, there is some $q_0 \in Q_0$ such that $q_0 \xrightarrow{q_1}$ in the NFA A. So, $q_0 \xrightarrow{w} q_1 \xrightarrow{a} q_2$ is also a run, proving one direction of the statement.

Conversely, suppose there is some $q_0 \in Q_0$ such that $q_0 \xrightarrow{wa} q_2$ is a valid run in the NFA A. We need to show that $q_2 \in Q_2$. Split the run as $q_0 \xrightarrow{w} q_1 \xrightarrow{a} q_2$. By the inductive hypothesis, we know that $q_1 \in Q_1$, and by the definition of edges in B, it follows that $q_2 \in Q_2$. This completes the induction proof. ■

Now, we prove the equivalence of DFAs and NFAs.

Theorem 2.2. *Every NFA is equivalent to a DFA.*

Proof. Let A be an NFA, and let B be the DFA obtained by the subset construction. We show that

$$L(A) = L(B)$$

First, let $w \in L(B)$, so that there is an accepting run $Q_0 \xrightarrow{w} Q_1$ in the DFA B, implying that Q_1 is a final state ($Q_1 \in F'$). Let $q_1 \in Q_1$ such that $q_1 \in F$ (by definition of finals states in B). By **Lemma** 2.1, there is some $q_0 \in Q_0$ such that $q_0 \xrightarrow{w} q_1$ is a run, and since $q_1 \in F$, it follows that $w \in L(A)$, so that $L(B) \subset L(A)$.

For the reverse inclusion, suppose $w \in L(A)$, so that there is a run $q_0 \xrightarrow{w} q_1$ with $q_0 \in Q_0$ and $q_1 \in F$. Let $Q_0 \xrightarrow{w} Q_1$ be the unique run in B (since B is a DFA). By the same **Lemma** 2.1, it follows that $q_1 \in Q_1$, and hence $Q_1 \in F'$, i.e $w \in L(B)$, implying that $L(A) \subset L(B)$. This completes the proof. ■

As a corollary, we see that

Corollary 2.2.1. *Any language is regular if and only if some NFA recognizes it.*

2.3. **Epsilon Transitions.** An *epsilon transition* between two states just means that you can jump from one state to another without consuming any input. So, FAs with epsilon transitions are NFAs where the alphabet is $\Sigma \cup \{\epsilon\}$. We will show that ϵ-NFAs have the same power as NFAs, and hence all these computational tools have the same power. Before continuing, we define the notion of an *epsilon closure*.

Definition 2.6. Let M be an ϵ-NFA. Let q_1, q_2 be states such that there is a path from q_1 to q_2 using only epsilon transistions. Then, we denote this by

$$q_1 \xrightarrow{\epsilon^*} q_2$$

Next, suppose $R \subset Q$. Define

$$\epsilon(R) := \{q \in Q | \exists r \in R \text{ such that } r \xrightarrow{\epsilon^*} q\}$$

and $\epsilon(R)$ is called the *epsilon closure* of R.

Theorem 2.3. *Any language recognized by an ϵ-NFA is also recognized by some NFA. Hence, a language is regular if and only if some ϵ-NFA recognizes it.*

Proof. Let A be an ϵ-NFA. We define a new NFA B as follows: B has the same states as A, and all non-ϵ transitions also remain the same. Next, if

$$u \xrightarrow{a} v \xrightarrow{\epsilon^*} w$$

where $a \in \Sigma$, then we add a transition (u, a, w) to B. Finally, the set of initial states in B is $\epsilon(Q_0)$, i.e it is the epsilon closure of the initial states of A. The final states of B are the same as the final states of A. It is then easy to argue that $L(A) = L(B)$. Hence, the proof is complete (the missing details are very minor). ■

2.4. **Operations on Regular Languages.** In this section, we will see what operations on regular languages as sets preserve their regularity. With the power of ϵ-NFAs in our hand, the task becomes much simpler than only having DFAs to work with.

Theorem 2.4. *Let R be the class of all regular languages over some alphabet.*

(1) *If $L_1, L_2 \in R$, then $L_1 \cup L_2 \in R$.*
(2) *If $L_1 \in R$, then $(L_1)^c \in R$.*
(3) *If $L_1, L_2 \in R$, then $L_1 \cap L_2 \in R$.*

Proof. For (1), take the disjoint union of the DFAs corresponding to L_1 and L_2 (note that the resultant FA in an NFA).

For (2), this is easily done by swapping the final and non-final states in the DFA (see **Problem 3.** in PSET-2).

(3) is true because

$$L_1 \cap L_2 = (L_1^c \cup L_2^c)^c$$

and the fact that (1) and (2) hold. ■

Definition 2.7. Let $A = (Q_1, \Sigma, \delta_1, F_1, Q_0^1)$ and $B = (Q_2, \Sigma, \delta_2, F_2, Q_0^2)$ be two FAs. Define the *cartesian product* of A, B to be an FA M with $Q = Q_1 \times Q_2$ over the same alphabet, and $Q_0 = Q_0^1 \times Q_0^2$. The set of final states of M is either $F_1 \times F_2$ or $F_1 \times Q_2 \cup Q_1 \times F_2$.

It is easy to see that the cartesian product of two FAs gives an FA which accepts either the union or the intersection of two languages. This gives another proof of the closure of regular languages under these operations. Next, we see two more important closure properties.

Theorem 2.5. *Let R be the set of all regular languages over some alphabet.*

(1) *If $L_1, L_2 \in R$ then $L_1 \cdot L_2 \in R$ (where the dot represents concatenation).*
(2) *If $L \in R$, define*

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

where $L^0 = \{\epsilon\}$. Then $L^* \in R$ (this operation is called the Kleene Star operation).

Proof. We give a construction in either case.

(1) Let M_1 and M_2 be DFAs for L_1 and L_2 respectively. Make a new DFA M as follows. Put M_1 and M_2 next to each other, and let the start state be the start state of M_1. From every final state of M_1, add an ϵ-transition to the start state of M_2, and let the final states be final states of M_2. This automaton M clearly recognizes $L_1 \cdot L_2$.
(2) Let M be a DFA accepting L. Add a new state to M, and make it the initial state, and also make it a final state (this ensures that ϵ is accepted by the new automaton as well). From this new state, and an ϵ-transition to the old initial state, and let all old final states remain final states. Moreover, from every old final state, add an ϵ-transition to the old initial state. Thus, M accepts L^*, and this completes the proof.

■

2.5. **Homomorphisms.** In this section, we see yet another closure property.

Definition 2.8. Let Σ, Γ be two alphabets. A *homomorphism* from Σ^* to Γ^* is a map $h : \Sigma \to \Gamma*$ which extends to a map on Σ^* which preserves the concatenation operation.

Homomorphisms between alphabets are very similar to homomorphisms between free groups, but not exactly. Throughout the results that follow, Σ and Γ will be two alphabets.

Theorem 2.6. *Let $L \subset \Sigma^*$ be regular, and suppose $h : \Sigma^* \to \Gamma^*$ is a homomorphism. Then, $h(L)$ is also regular. Hence, homomorphic images of regular languages are regular.*

Proof. Since L is regular, take a FA that accepts L, say M. Now, consider the following automaton: if there is an edge $q_1 \xrightarrow{s} q_2$ in M where $s \in \Sigma$, make a path between q_1 and q_2 in the new automaton such that the edges of the path are labeled with $h(s)$. The initial and final states of the new automaton are the same as those of M. Note that $h(s)$ could possibly be ϵ, which will form an ϵ-NFA. It is clear that the language of the new automaton is $h(L)$, completing the proof (the minor details are easy to verify). ■

Next is the question of taking inverse images under homomorphisms.

Theorem 2.7. *Let $h : \Sigma^* \to \Gamma^*$ be a homomorphism, and let $L \subset \Gamma^*$ be regular. Then, $h^{-1}(L)$ is also regular.*

Proof. This is very similar to the previous construction. Suppose M is an automaton that recognizes $L \subset \Gamma^*$. Consider the following automaton: if there is a path $q_1 \xrightarrow{h(s)} q_2$ in M for some $s \in \Sigma$, add an edge $q_1 \xrightarrow{s} q_2$ in the new automaton, and the initial and final states in the new automaton are the same as those in the original. In that case, it is clear that $h^{-1}(L)$ is recognizable, and this proves the claim. ■

2.6. **Quotients.** Here, we will see yet another closure property of regular languages.

Definition 2.9. Let $L_1, L_2 \subset \Sigma^*$ be any two languages. Define

$$L_1^{-1}L_2 = \{w | uw \in L_2 \text{ for some } u \in L_1\}$$

In simpler words, $L_1^{-1}L_2$ is the set of all words of L_2 with words of L_1 collapsed to ϵ.

Theorem 2.8. *Let $L_1, L_2 \subset \Sigma^*$ be any two languages such that L_2 is regular. Then, $L_1^{-1}L_2$ is also regular.*

Proof. Let M be a DFA accepting L_2. Now, we make a new FA M' as follows: put $M' = (Q, \Sigma, \Delta, F, Q_0')$, i.e M' is exactly the same as M except for its initial states. Let $q_0 \in Q_0$ be the initial state of M. Then, put

$$Q_0' = \{q \in Q : q_0 \xrightarrow{w} q \text{ for some word } w \text{ in } L_1\}$$

and it is easy to see that M' accepts $L_1^{-1}L_2$. This completes the proof. ■

Remark 2.8.1. Even though the above automaton M' recognizes $L_1^{-1}L_2$, determining Q_0' may not be easy to do (as computing the states reachable by traversing words of L_1 might not be easy if L_1 is not regular). However, if L_1 is known to be regular, we can use the product of the two automatons recognizing L_1 and L_2 to determine Q_0'.

2.7. **Rational Languages.** Upto this point, we have studied the closure properties of regular languages. From here, we will try and study another class of languages.

Definition 2.10. Let **RAT** denote the smallest class of languages that has the following properties.

(1) RAT contains all finite languages.
(2) RAT is closed under union (binary union, not arbitrary unions).
(3) RAT is closed under concatenation of languages.
(4) RAT is closed under the Kleene star operation.

Then, any language belonging to RAT is called a *rational language*.

A related notion is that of *rational* (or *regular*) *expressions*, which we will now define.

Definition 2.11. Fix an alphabet Σ. Call R a *regular expression* if R is equal to one of the following:

(1) ϕ, i.e the empty language.
(2) $\{a\}$, where $a \in \Sigma$. This represents a singleton language, and is denoted by a.
(3) $R_1 \cup R_2$, where R_1 and R_2 are regular expressions. This is denoted by $R_1 + R_2$, and represents the union of the two languages.

(4) $R_1 \cdot R_2$, where R_1, R_2 are regular expressions. This denoted the concatenation of the two languages.
(5) $(R_1)^*$, where R_1 is a regular expression.

Consider the class of all languages which are described by regular expressions. Clearly, this class satisfies the conditions of **Definition** 2.10, and hence this class is precisely RAT.

Theorem 2.9. *Any language belonging to RAT is regular.*

Proof. This can be proved by induction on the length of the regular expression describing the language. The base cases are true, because ϕ is regular, $\{a\}$ is regular for any $a \in \Sigma$ and $\{\epsilon\}$ is also regular. Moreover, the class of regular languages is closed under (binary) union, concatenation and Kleene star, and this proves the claim. ■

The natural question that follows is this: is the converse of the above theorem true? That is, can every regular language be described by some rational expression? We will investigate this question in the upcoming sections.

2.8. **GNFAs.** To answer the question asked at the end of the last subsection, we introduce a new type of automaton, called a *generalised non-deterministic finite automaton*. The basic idea is that in a GNFA, transitions are regular expressions, i.e we can go from one state to another via words in the language described by the regular expression labeling the transition. We make this precise as follows.

Definition 2.12. A *generalised non-deterministic finite automaton* M is a 5-tuple $(Q, \Sigma, \Delta, F, Q_0)$ where

$$\Delta \subset Q \times \mathscr{R} \times Q$$

where $\mathscr{R}$ is the set of all regular expressions over Σ.

Now, we will prove a surprising equivalence, which answers the question we asked before.

Theorem 2.10 (**Kleene's Theorem**)**.** *Any regular language is described by some regular expression. Hence, RAT is the set of all regular languages.*

Proof. Let L be a regular language, and let M be a DFA that accepts L. We will use a method called *state elimination* to convert M into a GNFA.

We assume that M has only one final state. If M has multiple final states, then we can make $|F|$ copies of M, each of which has only one final state, we we can take the disjoint union of these DFAs. So, if we find a regular expression for each of these DFAs, we can just add these expressions to get the final regular expression needed.

Now, suppose M has k states, where $k > 2$. We interpret M as a GNFA. So, there is some state $q \in M$ such that q is neither the initial state nor the final state. We will *eliminate* q to make a new GNFA M' as follows. Suppose there are states q_1, q_2 such that

$$q_1 \xrightarrow{e_1} q \xrightarrow{e_2} q_2$$

is a pair of transitions, where e_1, e_2 are regular expressions. Moreover, suppose $q_1 \xrightarrow{e_3} q_2$, $q \xrightarrow{e_4} q$ are also transitions in the GNFA (if these transitions don't

exist, then $e_3, e_4 = \phi$). So, in the new GNFA M', all old transitions will remain, and we add the transition

$$q_1 \xrightarrow{e_1 e_4^* e_2 + e_3} q_2$$

and it is clear why this preserves the language. We keep doing this until there are only two (or one, in case the initial state is also the final state) states left. If the initial state and the final state are the same, then the expression in the self-loop will be the regular expression for the given language. If the initial and final states are different, say q_0 and q_f, then suppose in the final GNFA, the transitions are as follows:

$$q_0 \xrightarrow{e_1} q_0$$
$$q_0 \xrightarrow{e_2} q_f$$
$$q_f \xrightarrow{e_3} q_f$$
$$q_f \xrightarrow{e_4} q_0$$

The regular expression for the language is

$$(e_1^* + e_2 e_3^* e_4)^* e_2 e_3^*$$

and this completes the proof. ■

2.9. **Using quotients to prove non-regularity.** In this short section, we will see a technique of proving that a language is not regular.

Proposition 2.11. *Let L be a regular language, and let $u, v \in \Sigma^*$. Let M be a DFA accepting L. If u, v reach the same state in M, then*

$$u^{-1}L = v^{-1}L$$

Proof. By the construction given in **Theorem** 2.8, we see that the same NFA accepts $u^{-1}L$ and $v^{-1}L$, and hence these two languages must be the same. ■

Corollary 2.11.1. *Suppose L is a regular language. If $u^{-1}L \neq v^{-1}L$, then u, v must reach different states in M.*

Now, we ask similar questions, and see that sometimes the answer is negative, as in the following examples.

Example 2.1. Suppose $u^{-1}L = v^{-1}L$. Does it follow that $u, v \in L$? Consider the following example. Let $L = \{ab, bb\}$, $u = a$ and $v = b$. So, we have

$$u^{-1}L = \{b\}$$

and

$$v^{-1}L = \{b\}$$

so that $u^{-1}L = v^{-1}L$. But clearly, $u, v \notin L$.

Example 2.2. Suppose $u, v \in L$. Does it follow that $u^{-1}L = v^{-1}L$? Again, the answer is no, by considering $L = \{b, bb\}$, $u = b$ and $v = bb$.

Now, we see an application of the above corollary.

Theorem 2.12. *Let L be a regular language. Then, L has finitely many quotients. In fact, the number of states in any DFA accepting L is bounded below by the number of quotients in L.*

Proof. Suppose L is a regular language. If L has infinitely many quotients, then we can find infinitely many words which reach *distinct states*, and hence no finite automaton will accept L, a contradiction. So, L must have finitely many quotients. The same observation shows that the number of states in any DFA accepting L is bounded below by the number of quotients. ■

Example 2.3. Let us show that the language

$$L = \{a^n b^n | n \geq 0\}$$

is *not regular* by this technique. Given any $n \in \mathbb{N}$, consider the word a^n. Let $n_1, n_2 \in \mathbb{N}$, and without loss of generality, suppose $n_1 < n_2$. Observe that $(a^{n_1})^{-1}L$ contains the word b^{n_1}, but the word b^{n_1} is *not* contained in $(a^{n_2})^{-1}L$. Hence, we have found infinitely many quotients, and hence this language is not regular.

Now, consider the following question: if the number of quotients of L is finite, is it regular?

Lemma 2.13. *For any u, v and any language L, we have*

$$(uv)^{-1}L = v^{-1}(u^{-1}L)$$

Proof. First, suppose $w \in (uv)^{-1}L$, and hence $uvw \in L$, implying that $vw \in u^{-1}L$, and hence $w \in v^{-1}(u^{-1}L)$. Conversely, suppose $w \in v^{-1}(u^{-1}L)$, implying that $vw \in u^{-1}L$, and hence $uvw \in L$, and hence $w \in (uv)^{-1}L$. This completes the proof. ■

Example 2.4. We try to enumerate all quotients of

$$(aa + bb)^* b = L$$

We have

$$\begin{aligned}
\epsilon^{-1}L &= L \\
a^{-1}L &= a(aa + bb)^* b = a \cdot L \\
b^{-1}L &= b(aa + bb)^{*b} + \epsilon = b \cdot L + \epsilon \\
(aa)^{-1}L &= a^{-1}(a^{-1}L) = L \\
(ab)^{-1}L &= b^{-1}(a^{-1}L) = \phi \\
(ba)^{-1}L &= a^{-1}(b^{-1}L) = \phi \\
(bb)^{-1}L &= L
\end{aligned}$$

It can be deduced that the residuals of L are

$$L, a \cdot L, b \cdot L + \epsilon, \phi$$

We now answer the question we asked before the example.

Theorem 2.14. *If a language L has finitely many quotients, then it is regular.*

Proof. We make a DFA as follows. Suppose the language L has quotients

$$L_1, L_2, ..., L_k$$

for some $k \in K$. Without loss of generality suppose $L_1 = L$. Make a DFA with k states each labelled by a label L_i for some $1 \leq i \leq k$. Make $L_1 = L$ the initial state. The final states are all those labels L_i which contain ϵ (the empty word).

The transitions are also clear: given any letter $a \in \Sigma$ and a state L_i, the transition from L_i through a goes to the state $a^{-1}L_i$. It is clear that L will be accepted by this automaton. This construction is called the *Nerode Automaton*. ■

Corollary 2.14.1. *If $u^{-1}L = v^{-1}L$, then u, v reach the same state in the Nerode Automaton for L. Moreover, if the number of quotients of L is finite, then the Nerode Automaton contains the least number of states required for any DFA (which is equal to the number of quotients).*

Proof. Any word u goes to the state $u^{-1}L$ in the Nerode automaton, so if $u^{-1}L = v^{-1}L$ then u, v go to the same state in the Nerode Automaton. The second claim is clear, because any DFA accepting L must contain atleast K states, where K is the number of quotients of L. ■

2.10. **Two equivalence Relations.** In this section, we will see two new equivalences for words over an alphabet.

Definition 2.13. Let L be any language. For words $u, v \in \Sigma^*$, we say

$$u \equiv_L v$$

if $u^{-1}L = v^{-1}L$. u and v in this case are said to be L*-equivalent*.

Proposition 2.15. *L-equivalence is an equivalence relation. Moreover, if $u \equiv_L v$, then $ua \equiv_L va$ for any word a, i.e $\equiv_L$ is a congruence.*

Proof. That the relation is an equivalence relation is clear. To show that it is a congruent, suppose $u \equiv_L v$, and let a be any word. Suppose w is a word such that $uaw \in L$, which implies that $aw \in u^{-1}L$, and hence $aw \in v^{-1}L$, and hence $vaw \in L$, which means that $w \in (va)^{-1}L$, implying that $(ua)^{-1}L \subset (va)^{-1}L$. Swapping the role of u and v, we see that this is actually an equality. ■

Remark 2.15.1. This relation is called the *Myhill-Nerode* relation.

Definition 2.14. Let A be a DFA. We say that $u \equiv_A v$ if u and v reach the same state in A. This is called A-equivalence.

Proposition 2.16. *Let A be a DFA. Then A-equivalence is an equivalence relation. Like $\equiv_L$, $\equiv_A$ is also a congruence.*

Proof. It is clear that $\equiv_A$ is an equivalence relation. Now, if u and v reach the same state in A, it must be true that ua and va reach the same state, because A is a DFA. Hence, $\equiv_A$ is a congruence. ■

Proposition 2.17. *Let A be a DFA, and let L be the language accepted by A. Then if $u \equiv_A v$, then $u \equiv_L v$ (the converse need not be true). So, $\equiv_A$ refines $\equiv_L$. Morever, if A is the Nerode Automaton for L, then $\equiv_A$ and $\equiv_L$ are the same relations.*

Proof. The claim that $\equiv_A$ refines $\equiv_L$ is just **Proposition** 2.11. By **Corollary** 2.14.1, it follows that if A is the Nerode Automaton then these two relations coincide. ■

An immediate corollary of this is the *Myhill-Nerode Theorem*.

Corollary 2.17.1 (*Myhill-Nerode Theorem*)**.** *Let L be a language. Then, L is regular if and only if $\equiv_L$ has finitely many equivalence classes. Moreover, the least number of states in a DFA accepting L is equal to the number of equivalence classes in $\equiv_L$.*

Proof. Suppose L is regular. By **Theorem** 2.12, we know that L has finitely many quotients. Let N be the Nerode Automaton for L. By **Proposition** 2.17, it follows that $\equiv_N$ and $\equiv_L$ are the same relations, and hence $\equiv_L$ has finitely many equivalence classes. Conversely, if $\equiv_L$ has finitely many classes, then it is clear that L has finitely many quotients, and hence L is regular by **Theorem** 2.14. In any case, we see that $\equiv_N$ and $\equiv_L$ are the same relations. The second part of the corollary is easy. ■

Using these ideas, we give another construction. Suppose L is a regular language, so that $\equiv_L$ has finitely many quotients, and this number is equal to the number of states for the Nerode Automaton N for L. Make a DFA M as follows: the states of M are equivalence classes of $\equiv_L$, and the transitions are as per the congruence. A state is final if it contains any word of L (and in that case, all words in that state will be words of L), and the initial state is the equivalence class containing ϵ. It is easy to see that this automaton recognizes L. In fact, as we see in the following proposition, this automaton is *isomorphic* to N.

Proposition 2.18. *Let A be a DFA for L with the same number of states as the Nerode Automaton for L. Then A is isomorphic to N. So, N is the unique DFA with the least number of states that accept L.*

Proof. (Details missing). The rough idea is as follows. Let q_0 be the initial state of A, and let q_0' be the initial state of N. The isomorphism is as follows. If $q_0 \xrightarrow{w} q$ in A and $q_0' \xrightarrow{w} q'$ in N, then map $q \to q'$. The only thing left to prove is that this is an isomorphism, but I won't do that here. ■

So the Nerode Automaton gives us a new technique to see whether two regular languages are the same. If we can show that their Nerode Automatons are isomorphic, then we are done.

2.11. **DFA Minimization.** In this section, we will see a technique of minimizing DFAs. First, let us prove a simple fact which will be the key idea in this process.

Proposition 2.19. *Let L be a regular language, and let A be any DFA accepting L such that all states of A are reachable from the initial state of A. Let N be the Nerode Automaton for L. If A has strictly more states than N, then there are states q_1, q_2 in A such that*

$$Lq_1 = Lq_2$$

where Lq is defined as

$$Lq = \{w | w \text{ is accepted in } A \text{ if } q \text{ was made the initial state}\}$$

Proof. Since every state q in A is reachable from the initial state of A, we know that Lq is a quotient of L. More specifically, if u is a word that reaches q, then

$$Lq = u^{-1}L$$

Now, since A has more number of states than N, by the pigeonhole principle, it must be true that $Lq_1 = Lq_2$ for distinct states q_1, q_2 in A, because the number of quotients of L is equal to the number of states in N. This completes the proof. ■

Remark 2.19.1. In N, $Lq_1 \neq Lq_2$ for distinct q_1, q_2, which follows from **Proposition** 2.17.

Next, we will see that states for which $Lq_1 = Lq_2$ is true, can be *merged* together.

Proposition 2.20. *Let A be a DFA accepting L, and suppose A has states q_1, q_2 such that*

$$Lq_1 = Lq_2$$

Make a new DFA A' with one less state, where q_1, q_2 are merged to form the state (q_1, q_2). If there is an edge $q \xrightarrow{a} q_1$ or $q \xrightarrow{a} q_2$ in A, make an edge $q \xrightarrow{a} (q_1, q_2)$ in A'. If there are edges $q_1 \xrightarrow{a} q_1'$ and $q_2 \xrightarrow{a} q_2'$ in A, make an edge $(q_1, q_2) \xrightarrow{a} q_1'$ in A'. If one of q_1, q_2 is initial or final, make (q_1, q_2) initial or final respectively. Then, A' is a valid DFA with

$$L(A) = L(A')$$

Proof. The only key fact we need to verify here is that

$$Lq_1' = Lq_2'$$

where q_1', q_2' are as given above. This fact is easy to verify, and I will leave it. The claim $L(A) = L(A')$ immediately follows from this. ■

The above two propositions give us a DFA minimisation algorithm, which goes as follows:

(1) Let A be the start DFA. Delete any state which is not reachable from the initial state.
(2) If $Lq_1 = Lq_2$ for distinct states, combine them as in **Proposition** 2.20.
(3) Repeat (2) until it can't be repeated.

Proposition 2.19 immediately tells us that when the algorithm stops, the number if states in the DFA is exactly equal to the number of states in the Nerode Automaton. **Proposition** 2.18 implies that in the end, we get the Nerode Automaton.

2.12. **Partition Refinement.** In this small section, I will discuss another technique of DFA minimisation, and I won't prove correctness here. This technique is called *partition refinement* (This discussion is very informal. I may want to prove these rigorously at some point).
The basic idea is as follows. Suppose we are given a DFA M for a language, which we want to minimize. So, we put all states in one class. Clearly, final and non-final states cannot belong to the same class, as final states accept the word ϵ, but non-final states don't. So, we *refine* this class into two classes, one for the non-final states and one for the final states. We continue this process further; if at any stage, there are two members in the same class, which have a transition into different classes, we split those states again, i.e the two states must belong to different classes. We keep doing this until this cannot be done, and the final DFA will be the minimal DFA.

2.13. **Pumping Lemma.** This is a very useful tool to prove non-regularity for some languages.

Theorem 2.21 (**Pumping Lemma**). *Let M be a DFA with n states, and suppose M accepts a word of length $\geq n$. Then, w can be written as*

$$w = xyz$$

where $y \neq \epsilon$ such that xy^kz is accepted by M for all $k \in \mathbb{N} \cup \{0\}$. Moreover, x, y can be chosen in a way which satisfies $|xy| \leq n$. In that case, the language $L(M)$ is infinite.

Proof. Refer to the solution of **Problem 1** of PSET-2. ■

3. Context Free Languages

We will begin by defining the notion of *context-free grammars*.

Definition 3.1. A *context-free grammar* G is a four tuple (N, Σ, S, P) where the following hold:

(1) N is a finite set. Each member of N is called a *variable* or a *non-terminal*.
(2) Σ is a finite set, and each member is called a *terminal*.
(3) $S \in N$ is a starting variable.
(4) P is a finite set of *productions*, i.e $P \subset N \times (N \cup \Sigma)^*$.

Definition 3.2. Let $\alpha, \beta \in (N \cup \Sigma)^*$. We say that β is *derived from α in one step* if

$$\alpha = \alpha_1 X \alpha_2$$

for some $\alpha_1, \alpha_2 \in (N \cup \Sigma)^*$, $X \in N$ such that

$$\beta = \alpha_1 \gamma \alpha_2$$

where $(X, \gamma) \in P$, and this is denoted by $\alpha \to \beta$. More generally, β is said to be derived from α if there is a sequence of derivations

$$\alpha = \alpha_0 \to \alpha_1 \to ... \to \alpha_k = \beta$$

and this is denoted by $\alpha \xrightarrow{*} \beta$.

Definition 3.3. Let $G = (N, \Sigma, S, P)$ be a context-free grammar with starting variable S. The *language of the grammar* $L(G)$ is defined as

$$L(G) := \{w \in \Sigma^* | S \xrightarrow{*} w\}$$

If a language L is the language of some context-free grammar, then L is said to be a *context-free language*.

Example 3.1. Consider the context free grammar G where the productions are

$$S \to aSb \mid \epsilon$$

The claim is that

$$L(G) = \{a^n b^n | n \geq 0\}$$

which can easily be proven by induction. Note that, this language is known to be not regular, but it is context-free.

Example 3.2. Let G be given by

$$S \to aSb \mid bSa \mid SS \mid \epsilon$$

The claim is that

$$L(G) = \{w \mid |w|_a = |w_b|\}$$

where $\Sigma = \{a, b\}$.

First, we show that every word generated by G has equal number of a's and b's. Suppose $\alpha \in (N \cup \Sigma)^*$ such that $S \xrightarrow{*} \alpha$, and we show that the number of a's in α is equal to the number of b's in α. Suppose S derives α in zero steps. Then, $\alpha = S$, so the base case is true. For the inductive step, suppose S derives α in n steps, say

$$S \to \alpha_n \to \alpha$$

By inductive hypothesis, the number of a's in α_n is equal to the number of b's in α_n. Now, any transition from α_n to α adds the same number of a's and b's. This completes the induction proof.
Next, we prove the converse, i.e every word with equal number of a's and b's can be generated by G. If w has equal number of a's and b's then one of the following is true.

(1) $w = \epsilon$.
(2) $w = aw_1b$ where $|w_1|_a = |w_1|_b$.
(3) $w = bw_1a$ where $|w_1|_a = |w_1|_b$.
(4) $w = w_1w_2$ and $w_1, w_2 \neq \epsilon$ with $|w_1|_a = |w_1|_b$ and $|w_2|_a = |w_2|_b$.

Now, we can prove the converse by induction on the length of w. The base case is when $w = \epsilon$, which is clearly accepted by G. Suppose $w = a_1a_2...a_{2n}$. So, one of the cases (2), (3) or (4) must apply. Suppose (2) applies. So

$$w = a(a_2...a_{2n-1})b$$

By induction hypothesis, $S \xrightarrow{*} a_2...a_{2n-1}$ and then we have

$$S \to aSb \xrightarrow{*} aa_2...a_{2n-1}b$$

and similarly case (3) can be handled. For case (4), suppose $w = w_1w_2$. By inductive hypothesis, $S \xrightarrow{*} w_1$ and $S \xrightarrow{*} w_2$. So, we have

$$S \to SS \xrightarrow{*} w_1S \xrightarrow{*} w_1w_2$$

and hence this case is also proven. So this completes the proof by induction.

Example 3.3. Let G be given by

$$S \to aSa \mid bSb \mid a \mid b \mid \epsilon$$

It is easy to see that this grammar generates the set of all *palindromes* over $\{a, b\}$.

Example 3.4. Let G be given by

$$S \to aSb \mid T$$
$$T \to cTd \mid \epsilon$$

The claim is that

$$L(G) = \{a^nc^md^mb^n | n \geq 0, m \geq 0\}$$

and this is not hard to prove.

Example 3.5. Let G be the grammar given by

$$S \to S_1S_2$$
$$S_1 \to aS_1b \mid \epsilon$$
$$S_2 \to cS_2d \mid \epsilon$$

and the claim is that

$$L(G) = \{a^n b^n c^m d^m | n \geq 0, m \geq 0\}$$

Again, this is not very hard to prove.

Example 3.6. Consider the grammar G given by

$$S \to (S) \mid SS \mid \epsilon$$

and we claim that $L(G)$ is the *Dyck Language* (which was introduced in PSET-4), where the alphabet is $\{(,)\}$. It is a good exercise to prove this.

The next theorem gives the relationship of these grammars to regular languages.

Theorem 3.1. *Every regular language is context free.*

Proof. Let L be a regular language, and let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA that accepts L. Using this DFA, we will construct a context free grammar that accepts L. Let G be a context-free grammar as follows: the variable or non-terminals for G is the set Q (set of states), the set of terminals is simply Σ and the starting non-terminal S is q_0 (the starting state). The set of productions is

$$P = \{q \to aq' | q \xrightarrow{a} q' \text{ is a transition in M}\} \cup \{q \to \epsilon | q \in F\}$$

In other words, transitions have been converted to the suitable productions, and each final state has a production going to ϵ. It is clear that the language of this grammar is L, and this completes the proof. ■

Motivated by the grammar in the above proof, we have the following definitions.

Definition 3.4. Let $G = (N, \Sigma, S, P)$ be a context-free grammar. If every RHS in a production has atmost one non-terminal, then G is said to be a *linear grammar*. Moreover, if the a linear grammar G has the property that if there is a non-terminal in the RHS of a production, it appears as the last letter, then G is said to be a *right linear grammar*.

So, via **Theorem** 3.1, we have shown that every regular language is recognized by a right-linear grammar. It turns out that the converse is also true.

Theorem 3.2. *A language is regular if and only if it has a right linear grammar.*

Proof. One direction is just **Theorem** 3.1. For the converse, see **Problem 3** of PSET-6. ■

Remark 3.2.1. One can similarly define the notion of a *left linear grammar*, and similar arguments hold for those as well. More information can be found in PSET-6.

As seen in **Example** 3.1, linear grammars may generate non-regular languages.

3.1. **Closure Properties.**

Let us now explore some closure properties of CFLs.

Theorem 3.3. *Suppose L_1, L_2 are CFLs. Then the following hold.*

(1) $L_1 \cup L_2$ *is a CFL.*
(2) $L_1 \cdot L_2$ *is a CFL.*
(3) L_1^* *is a CFL.*

Proof. Let $G_1 = (N_1, \Sigma, S_1, P_1)$ and $G_2 = (N_2, \Sigma, S_2, P_2)$ be context-free grammars for L_1, L_2 respectively. Without loss of generality assume that $N_1 \cap N_2 = \phi$, i.e the set of non-terminals in both grammars are completely different.

(1) Consider the context free grammar G as follows. Let $S \notin N_1 \cup N_2$ be a new symbol, and let S be the starting non-terminal for G. Let the set of productions be
$$\{S \to S_1 \mid S_2\} \cup P_1 \cup P_2$$
and clearly the set of non-terminals for G is $N_1 \cup N_2 \cup \{S\}$. It is clear that the language of G is $L_1 \cup L_2$.

(2) Consider the construction given in (1), with the only difference being the production $S \to S_1 \mid S_2$ is replaced by $S \to S_1 S_2$.

(3) For the Kleene star operation, consider the grammar G as follows. Let $S \notin N_1$ be a new symbol, and let this be the starting symbol for G. The set of all non-terminals is simply $N_1 \cup \{S\}$, and let the set of productions be
$$\{S \to S_1 S \mid \epsilon\} \cup P_1$$
and it is clear that G accepts the language L_1^*.

■

Remark 3.3.1. In the proof of (3) above, we added a new state S instead of just adding the production $S_1 \to S_1 S_1 \mid \epsilon$. It needs to be pointed out that this is an important step, and just adding this production may accept words which are not in the Kleene Star operation. For instance, consider the language in **Example** 3.1. The productions were $S_1 \to aS_1b \mid \epsilon$, and the language accepted was $\{a^n b^n | n \geq 0\}$. Now, suppose we add the production $S_1 \to S_1 S_1$. Then, observe the following sequence of derivations

$$S_1 \to S_1 S_1 \to aS_1 baS_1 b \to aS_1 S_1 baS_1 b \to aaS_1 baS_1 bbaS_1 b \to ... \to aababbab$$

and this word is clearly not in the Kleene Star of this language. The problem is that S_1 is able to duplicate itself at any point of time.

Now, here are two important examples highlighting the limitations of context-free grammars.

Example 3.7. $L = \{a^n b^n c^n | n \geq 0\}$. We will show in the subsequent sections that this language is *not* a CFL (go to **Example** 3.13 to see the proof).

Observe that this language can be written as the intersection of the two languages
$$L_1 = \{a^n b^n c^* | n \geq 0\}$$
and
$$L_2 = \{a^* b^n c^n | n \geq 0\}$$
and observe that both L_1, L_2 are CFLs. So, it follows that CFLS *are not closed under intersections*, and moreover since they are closed under unions, it follows that CFLs are also *not closed under complementation*.

Example 3.8. $L = \{ww | w \in \Sigma^*\}$. Go to **Example** 3.14 to see a proof of why this language is not a CFL.

Example 3.9. Here, we will explicitly show by example that CFLs are not closed under complementation. Let L be the language in **Example** 3.8, and we consider L^c. We will show that L^c is a CFL, and since L is not, it will prove our claim. Observe that

$$L^c = \{\text{odd length words}\} \cup \{\text{even length words that are not of the form } ww\}$$

Observe that the set of all odd length words is a regular language, and hence it is a CFL by **Theorem** 3.1. Since CFLs are closed under union, it is enough to show that the language

$$L_1 = \{\text{even length words that are not of the form } ww\}$$

is a CFL. Observe that any word w in L_1 is of one of the following forms

$$w = xaxyby \text{ for some words } x, y$$
$$w = xbxyay \text{ for some words } x, y$$

and this is not hard to prove. So, the context free grammar for L_1 is the following.

$$\begin{aligned} S &\to S_aS_b \mid S_bS_a \\ S_a &\to aS_aa \mid bS_ab \mid aS_ab \mid bS_aa \mid a \\ S_b &\to aS_ba \mid bS_bb \mid aS_bb \mid bS_ba \mid b \end{aligned}$$

and this completes the proof.

Proposition 3.4. *If L is a CFL and R is a regular language, then $L \cap R$ is also a CFL.*

Proof. The proof is given in a subsection below. ■

Example 3.10. Consider the language

$$L = \{w | |w|_a = |w|_b = |w|_c\}$$

and we show that this is *not* a CFL. Suppose L is a CFL. Consider the language $R = a^*b^*c^*$, which is regular. So, by **Proposition** 3.4 we see that

$$L \cap R = \{a^nb^nc^n | n \geq 0\}$$

is a CFL, which contradicts **Example** 3.7. So, L is not a CFL.

3.2. **Homomorphisms.** Let us now see whether CFLs are closed under homomorphisms and inverse homomorphisms.

Theorem 3.5. *Let Σ,Γ be two alphabets, and let L be a CFL over Σ. Suppose $h : \Sigma \to \Gamma^*$ is a map, and without loss of generality regard h as a homomorphism $h : \Sigma^* \to \Gamma^*$. Then, $h(L)$ is a CFL over Γ.*

Proof. Let $G = (N, \Sigma, S, P)$ be a context-free grammar that accepts L. Consider the grammar $G' = (N, \Gamma, S, P')$ as follows. If $S \to w$ is any production in G, where $w \in (N \cup \Sigma)^*$, let w' be the word in $(N \cup \Gamma)^*$ where each occurrence of a letter $a \in \Sigma$ in w is replaced by $h(a)$ in w'. It is clear that the grammar G' accepts $h(L)$, completing the proof. Proof is incomplete. Need to fix it. ■

Theorem 3.6. *Let Σ, Γ and h be as in the above theorem. Let L be a CFL over Γ. Then, $h^{-1}(L) \subset \Sigma^*$ is a CFL over Σ.*

Proof. This will be a bit hard to prove by using only CFGs. See the proof of **Proposition** 3.15 using PDAs. ■

Example 3.11. Consider the language

$$L = \{a^n b^n c^n d^n | n \geq 0\}$$

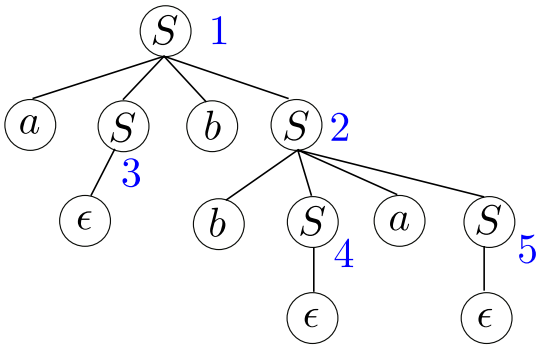
Consider the homomorphism that fixes a, b, c and sends $d \to \epsilon$. By **Example** 3.7, the language $\{a^n b^n c^n | n \geq 0\}$ is not a CFL, and hence by **Theorem** 3.5, the language L is not a CFL.

Example 3.12. Consider the grammar G as follows.

$$S \to aB \mid bA \mid SS \mid \epsilon$$
$$A \to aS \mid bAA$$
$$B \to bS \mid aBB$$

(Two questions to be answered. What is the language generated here and does the language change if we replace $A \to aS$ by $A \to a$ and $B \to bS$ by $B \to b$. What if we drop $S \to SS$? Apparently yes. Conjecture is this grammar generates words with equal number of a's and b's.)

3.3. **Derivation Trees.** This section will be a bit informal, but all of the discussion can be easily formalised. We will mostly work with a single example.



Consider the following context free grammar.

$$S \to aSbS \mid bSaS \mid \epsilon$$

It is easy to see that the word $abba$ is accepted by this grammar, and we see that

$$S \to aSbS \to aSbbSaS \to abbSaS \to abbaS \to abba$$

is a valid sequence of derivations for this word. Now, this sequence of derivations can be represented by drawing the tree given above, and this tree is called a *derivation tree*. The numbers $1 - 5$ written next to the non-leaf nodes represent the order in which those terminals were rewritten in the derivation of the word. So this means that every sequence of derivations has a unique derivation tree, but the converse is not true; the same derivation tree can represent two distinct sequences of derivations. For instance, in this example, we can switch the orders of 4 and 5, to get a different derivation for the same word. However, there is a *canonical derivation* for every derivation tree, which we will now define.

Definition 3.5. Suppose in a grammar, there is a sequence of derivations for a word w. The *left-most derivation* for w is that derivation in which at every sequence, the *left-most* non-terminal is rewritten. Equivalently, it is the derivation corresponding to the derivation tree of w which is obtained by DFS on the tree, where children of a node are visited from left to right.

For the derivation tree given above, the left most derivation is

$$S \to aSbS \to abS \to abbSaS \to abbaS \to abba$$

It then follows easily that every derivation tree has a unique left-most derivation, and hence there is a bijection between derivation trees and left-most derivations.

Remark 3.6.1. Analogously, one can define the notion of the *right-most derivation*, and similar results hold.

Definition 3.6. A context-free grammar is said to be *ambiguous* if some word has multiple derivation trees, or equivalently, if some word has multiple left-most derivations.

3.4. **The Emptiness Problem.** Now, we will describe two algorithms to see if, given a context free grammar G, $L(G)$ is empty. First, we will prove a simple fact about the size of derivation trees.

Proposition 3.7. *Let G be a context free grammar, and suppose $L(G) \neq \phi$, i.e there is a derivation tree T generating a word. Then, there is some derivation tree T' generating a word in $L(G)$ such that the height of T' is bounded above by $|N|$, where N is the set of non-terminals in G.*

Proof. This proof idea is very intuitive. Suppose there is some tree T generating a word in $L(G)$, and hence all the leaves of T are terminals. Suppose there is some path in T from the root to a leaf such that a non-terminal repeats in the path. Let T_1 be the sub-tree rooted at the second occurrence of the non-terminal in the path. Replace the sub-tree rooted at the first occurrence of the non-terminal by T_1, and let the new tree be T'. It is then clear that T' is a valid derivation tree generating some word in $L(G)$. We can keep repeating this procedure until all the non-terminals on every path from the root to a leaf are distinct. In that case, the height of the resultant tree is atmost $|N|$. This completes the proof. ■

Remark 3.7.1. The above fact gives us a simple algorithm to solve the emptiness problem. Given a grammar G, enumerate all possible derivation trees of height atmost $|N|$, where N is the set of terminals of G. If some tree has all its leaves as terminals, then $L(G)$ is non-empty. This algorithm is not very efficient however.

We shall now see a more efficient algorithm to solve the same problem.

Definition 3.7. Let $G = (N, \Sigma, S, P)$ be a context free grammar. Let $N_0 = \Sigma$, the set of terminals. For each $i \in \mathbb{N}$, we inductively define

$$N_i = \{x \in N | x \to \alpha \text{ for some } \alpha \in (N_0 \cup ... \cup N_{i-1})^*\}$$

We will now prove another simple fact, which will lead to a simpler algorithm for the emptiness problem.

Proposition 3.8. *Let G be a context free grammar, and let $N_i : i \in \mathbb{N} \cup \{0\}$ be as defined in* **Definition** 3.7. *Then, a non-terminal $x \in N$ generates a word $w \in \Sigma^*$ if and only if $x \in N_i$ for some $i \in \mathbb{N}$.*

Proof. First, suppose $x \in N$ is a non-terminal that generates a word in Σ^*. Let T be the corresponding derivation tree, so that all leaves of T are non-terminals, i.e each leaf is in N_0. Remove the leaves of T to obtain a new tree T'. Then, it is clear by definition that all leaves of T' are in N_1. Continue to do this until the tree is non-empty. It is then clear that x, which is the root of T, belongs to N_i for some $i \in \mathbb{N}$.

Conversely, suppose $x \in N$ is a non-terminal such that $x \in N_i$ for some $i \in \mathbb{N}$. If $i = 1$, it is clear that x generates a word in Σ^*. If $i > 1$, then we know that $x \to \alpha$ for some $\alpha \in (N_0 \cup ... \cup N_{i-1})^*$, i.e each non-terminal in α belongs to N_{i-1}. By induction, each non-terminal in α generates a word in Σ^*, and hence x generates a word in Σ^*. This completes the proof. ■

Remark 3.8.1. Observe that $N_1 \subseteq N_2 \subseteq N_3 \subseteq ...$, i.e this is an increasing chain of sets. Moreover, it is not hard to see that if $N_i = N_{i+1}$ for some $i \in \mathbb{N}$, then $N_k = N_i$ for all $k \geq i$. Finally, since $|N| < \infty$, the largest such i cannot be greater than $|N|$. This gives us yet another algorithm to solve the emptiness problem; $L(G) \neq \phi$ if and only if $S \in N_{|N|}$, where S is the starting non-terminal of G, and N is the set of non-terminals.

3.5. **Membership Problem.** Here, we explore the following question: given G and a word w, is it true that $w \in L(G)$? There are two main difficulties while solving this problem. The first is ϵ-productions. If there is an ϵ-production, then a derivation can decrease the length of the word, which we don't want. Secondly, if there is a production of the form $X \to Y$ where X, Y are non-terminals, then again a derivation involving this production won't increase the length of the word. We will see how to remedy these situations. We begin with a simple definition.

Definition 3.8. Any production of the form $X \to Y$ where $X, Y \in N$ is called a *unit production*.

Theorem 3.9. *If L is a CFL then $L - \{\epsilon\}$ has a CFG without ϵ-productions and unit productions.*

Proof. Let G be a grammar for L, let $G = G_0$ and let set $i = 0$. We do the following algorithm.

(1) Suppose $Y \to \epsilon$ and $X \to \alpha Y \beta$ are productions in G_i, where $\alpha, \beta \in (N \cup \Sigma)^*$, such that $X \to \alpha\beta$ is *not* a production in G_i. Then, add a production $X \to \alpha\beta$ in the new grammar G_{i+1}. It is clear that $L(G_i) = L(G_{i+1})$ in this case. Repeat the same algorithm starting with the grammar G_{i+1}.
(2) Suppose (1) cannot be applied to G_i. If there are productions $X \to Y$ and $Y \to \alpha$ in G_i where $\alpha \in (N \cup \Sigma)^*$ such that $X \to \alpha$ is *not* a production in G_i, then add the production $X \to \alpha$ in a new grammar G_{i+1}. Repeat the algorithm with G_{i+1}.
(3) Suppose neither (1) nor (2) can be applied to G_i. Then halt the algorithm.

Note that at each step, at most one production is added. Moreover, if j is the length of the largest RHS of a production in G_0, then for any i, the length of the largest RHS of a production in G_i is atmost j (infact equal to j, since the old productions are retained). Since there are only finitely many productions with length of the RHS of any production atmost j, the above algorithm halts. Let k be the index where the algorithm halts, i.e G_k is the final grammar with $L(G_0) = L(G_k)$, and observe that $G_0 \subseteq G_k$ with respect to production containment.

† Now, we show that if $w \neq \epsilon$ is derivable in $G = G_0$, then it is derivable in G_k without using any ϵ or unit productions.

First, suppose w is derivable in G, and suppose some unit production $X \to Y$ was used, where $X, Y \in N$. Since $w \in \Sigma^*$, some production of the form $Y \to \alpha$ is used. Moreover, by the definition of G_k, we know that $X \to \alpha$ *is* a production in G_k, and hence in G_k, the word w can be derived *without* using the unit production $X \to Y$. So what we have shown is that the *smallest* derivation tree of w in G_k does not contain any unit production. Next, suppose $w \neq \epsilon$ is derivable in G. Also, suppose some ϵ-production of the form $Y \to \epsilon$ is used in the derivation, and consider the derivation tree for this derivation. Since $w \neq \epsilon$, $Y \neq S$, where S is the starting non-terminal. Hence, Y has *some* parent X in the derivation tree. Suppose, for this parent X, the production $X \to \alpha Y \beta$ was used. By the definition of G_k, we know that $X \to \alpha\beta$ *is* a production in G_k, and hence w can be derived in G_k *without* using the production $Y \to \epsilon$. Again, we have just shown that for such w, the *smallest* derivation tree in G_k is free of any ϵ-productions. This proves the † statement above.

Finally, let G'_k be the grammar G_k, with all ϵ and unit productions removed. So, we have shown above that

$$L(G) - \{\epsilon\} = L(G_k) - \{\epsilon\} = L(G'_k)$$

and hence G'_k is the required grammar for $L(G) - \{\epsilon\}$, completing the proof. ■

3.6. **Chomsky Normal Form (CNF).** This turns out to be a very useful form for CFLs.

Definition 3.9. A grammar G is said to be in *Chomsky Normal Form* if each production is of one of the following forms:

$$X \to YZ$$
$$X \to a$$

where $X, Y, Z \in N$ and $a \in \Sigma$ above, and also $S \to \epsilon$ is allowed if necessary.

Theorem 3.10. *Every context-free grammar is equivalent to a grammar in Chomsky Normal form.*

Proof. Let G be a grammar, and let $L(G)$ be the language of the grammar. By **Theorem** 3.9, we know that $L(G) - \{\epsilon\}$ has a grammar without any ϵ or unit productions. Let G' be this grammar. Now, suppose $\Sigma = \{a_1, ..., a_n\}$ is the set of terminals. For each terminal a_i, we add a new non-terminal A_i to G' and add the production $A_i \to a_i$. In any RHS of any other production in G', we replace a_i by the non-terminal A_i. So now, G' has the property that if $X \to \alpha$ is a production, then $|\alpha| = 1$ implies that $\alpha \in \Sigma$, and if $|\alpha| > 1$, then α consists of *only* non-terminals. Finally, suppose

$$X \to Y_1 Y_2 ... Y_k$$

is a production in G', where $Y_1, Y_2, .., Y_k \in N$ are non-terminals. Replace this production with the following set of productions:

$$X \to Y_1 K_1$$
$$K_1 \to Y_2 K_2$$
$$...$$
$$K_{k-2} \to Y_{k-1} Y_k$$

where $K_1, ..., K_{k-2}$ are *new* non-terminals which have been added to G'. Finally, if $\epsilon \in L(G)$, we add the production $S \to \epsilon$ in G'. So, it follows that G' is a grammar for $L(G)$, and clearly G' is in Chomsky Normal form, completing the proof. ■

Remark 3.10.1. The algorithm that we have described above is actually not efficient, and infact exponential in the worst-case running time. As an example, suppose we have the following grammar.

$$X \to X_1 X_2 ... X_k$$
$$X_i \to \epsilon$$

where $1 \leq i \leq k$. As in **Theorem** 3.9, we see that we have to add every production of the form $X \to X_{j_1} ... X_{j_m}$ where $\{j_1, ..., j_m\} \subset \{1, ..., k\}$, i.e we have to add exponentially many productions corresponding to each subset of $\{1, ..., k\}$. However, it turns out that the CNF of a grammar can be obtained in polynomial time. Infact, if we *first* introduce new non-terminals so that every RHS of a production contains atmost two non-terminals and then proceed as in **Theorem** 3.9, the algorithm will be much faster.

3.7. **Membership Problem Continued.** We can use the Chomsky Normal Form to solve the membership problem in polynomial time. We will now describe an algorithm for the same.
Let $G = (N, \Sigma, S, P)$ be a context free grammar and let $w \in \Sigma^*$ be any non-empty word. Checking whether a context free grammar generates the empty word is very similar to the emptiness problem, and for more details check problem **4.** of PSET-7.
We can assume that G is in CNF (and as mentioned in **Remark** 3.10.1, CNF can actually be obtained in polynomial time), and we can also assume that the CNF does not contain any ϵ-production, because we are only interested in non-empty words. The problem is to determine whether $S \xrightarrow{*} w$. Throughout the discussion that follows, we assume that $w = a_1 a_2 ... a_n$ with $n \geq 1$ and $a_i \in \Sigma$ for each i.
Suppose X is a non-terminal, and we want to determine whether $X \xrightarrow{*} w$. Since the grammar is in CNF, the algorithm is actually not very hard. If $|w| = 1$, i.e $w \in \Sigma$, this is equivalent to checking whether $X \to w$ is a production in G, i.e whether $X \to w \in P$. If $|w| > 1$, then $X \xrightarrow{*} w$ if and only if there are non-terminals Y, Z such that $X \to YZ \in P$ and $Y \xrightarrow{*} a_1 ... a_i$ and $Z \xrightarrow{*} a_{i+1} ... a_n$, where $1 \leq i < n$ (notice that $i < n$, because the CNF cannot produce the empty word). So, we have the following *dynamic programming* algorithm.

```
/*Suppose the non-terminals of G are X_1,...,X_k
and suppose $X_1$ is the starting non-terminal of G.
*/
/*Let Derives(i , j , k) be 1 if X_i can generate
```

```
the word a_ja_{j + 1}...a_k, and let it be 0 otherwise. So, the
solution to the membership problem is Derives(1 , 1 , n).
*/
/*Initialise all Derives(i , j , k) to be 0*/
Derives(i , j , k) =
    if (j = k) then
        if(X_i -> a_j is a production in G)
            Derives(i , j , j) = 1
    else
        for each l with j <= l < k
            for each production X_i -> X_pX_q in G
                if Derives(p , j , l) and Derives(q , l + 1 , k)
                    Derives(i , j , k) = 1
```

It is clear that the above dynamic programming algorithm can be implemented in a bottom-up fashion as well, for more efficiency. Moreover, the table `Derives` contains $|N| \times n^2$ entries, and the algorithm simply fills up these entries bottom up. From the algorithm, it is clear that the complexity is $O(|N| \times n^2 \times n \times |P|) = O(|N| \times n^3 \times |P|)$. This algorithm is known as the *Cocke-Younger-Kasami*(CYK) algorithm. Using the standard techniques of dynamic programming, if the word is derivable in the grammar, we can also find a derivation/derivation tree for the same.

Remark 3.10.2. Observe that, in the above algorithm, if the grammar(in CNF) is fixed, then the complexity is $O(n^3)$.

Remark 3.10.3. The CYK algorithm is *not* the most efficient algorithm for parsing context free grammars. In fact, for some classes of CFGs, there are linear time parsing algorithms as well.

3.8. **Revisiting A Closure Property.** In this section, we will give a construction to prove the claim in **Proposition** 3.4.

Proof of **Proposition** 3.4. Let $G = (N, \Sigma, S, P)$ be a context free grammar accepting the language $L(G)$, and suppose G is in Chomsky Normal Form. Let R be a regular language accepted by a DFA $A = (Q, \Sigma, \delta, s, F)$, so that $R = L(A)$. We will construct a CFG for $L(G) \cap L(A)$ as below.

The non-terminals of the new grammar will be of the form $X(p, q)$, where $X \in N$ and $p, q \in Q$. Our goal will be to have the non-terminal $X(p, q)$ generate precisely those words w such that $X \xrightarrow{*} w$ in G and $p \xrightarrow{w} q$ in A, i.e

$$(*) \qquad \forall X \in N \text{ and } p, q \in Q\,,\ X(p,q) \xrightarrow{*} w \iff X \xrightarrow{*} w \text{ and } p \xrightarrow{w} q$$

So, we add productions in our new grammar as per the following.

(1) If $X \to a \in P$ and if $p \xrightarrow{a} q \in \delta$ then add the production $X(p, q) \to a$ in the new grammar.
(2) If $X \to YZ \in P$ then for every $p, q, r \in Q$, add the production $X(p, q) \to Y(p, r)Z(r, q)$ in the new grammar.
(3) Add a start symbol S' in the new grammar with a production $S' \to S(s, f)$ for every $f \in F$, where S is the start non-terminal of G and s is the initial state of A.

Now, we will prove the claim $(*)$ above, and that will complete the proof of the theorem. First we show that $X(p, q) \xrightarrow{*} w$ implies that $X \xrightarrow{*} w$ and $p \xrightarrow{w} q$, and we

do so by induction on the size of the derivation. For the base case, the derivation has size 1, and clearly in that case $X(p,q) \to w$ is a production, meaning that $|w| = 1$. By the definition of the productions in the new grammar, it is clear that $X \to w \in P$ and $p \xrightarrow{w} q \in \delta$, and hence the base case is true. For the inductive case, suppose the length of the derivation $X(p,q) \xrightarrow{*} w$ is n where $n > 1$, so the derivation is of the form

$$X(p,q) \to Y(p,r)Z(r,q) \xrightarrow{*} w_1 w_2 = w$$

where $Y(p,r) \xrightarrow{*} w_1$ and $Z(r,q) \xrightarrow{*} w_2$, and both these derivations have size atmost $n-1$. So by the inductive hypothesis, we know that $Y \xrightarrow{*} w_1$, $Z \xrightarrow{*} w_2$, $p \xrightarrow{w_1} r$ and $r \xrightarrow{w_2} q$. All of these facts together imply that $X \xrightarrow{*} w_1 w_2 = w$ and $p \xrightarrow{w_1 w_2 = w} q$, and the proof is complete by induction. The converse is also proven by reversing these arguments, and this completes the proof. ■

3.9. **Pumping Lemma for CFLs.** Now we will prove a useful pumping lemma for CFLs, which allow us to prove that certain langauges are not CFLs.

Theorem 3.11. *Let L be a CFL. Then, there is a number p (called the pumping length of L), where if w is any word in L such that $|w| \geq p$, then w can be written in the form $w = uvxyz$ such that the following hold.*

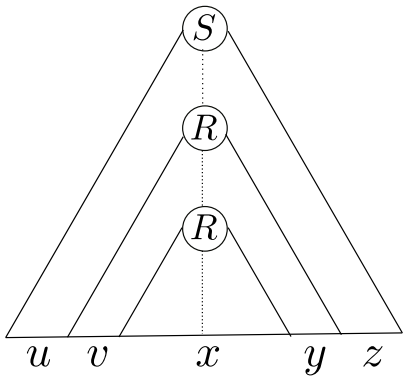
(1) $uv^i xy^i z \in L$ *for each* $i \geq 0$.
(2) $|vy| > 0$, *i.e atleast one of* v *or* y *is not empty.*
(3) $|vxy| \leq p$.

Proof. Let G be a CFG for L, and let b be the *maximum* number of symbols (terminals or non-terminals) occuring on the RHS of a production in G. Without loss of generality, we can assume that $b \geq 2$ (otherwise we can always add a garbage non-terminal to our grammar). Hence, *any* derivation tree in G of height h can generate a word of length atmost b^h, since the maximum number of leaves possible in a derivation tree of height h is b^h. Consequently, *any word* generated by G of length atleast $b^h + 1$ must have a derivation tree of height atleast $h+1$. This is a key observation that we are going to use.

Put $p = b^{|N|+1}$, where $|N|$ is the number of non-terminals in G. By the remarks in the above paragraph, any word in L of length atleast p must have a derivation tree of height atleast $|N|+1$, since $b^{|N|+1} \geq b^{|N|} + 1$.

Now, let $w \in L$ such that $|w| \geq p$. Let T be a derivation tree in G for w with the *minimum number of nodes* (we take the minimum number of nodes to guarantee condition (2), as we shall see below †). Clearly, T has height atleast $|N|+1$. So, there is a path from the root of T to a leaf of height atleast $|N|+1$, i.e there are atleast $|N|+2$ vertices in this path. The leaf must be a terminal, and hence the rest of the vertices in this path are non-terminals, and there are $|N|+1$ of such vertices. So, by the pigeon-hole principle, atleast one non-terminal R repeats in this path. The picture of the derivation tree is as follows.

So we can break the word as $w = uvxyz$ as in the above derivation tree, and it is clear by cutting and pasting subtrees that $uv^i xy^i z \in L$ for each $i \geq 0$. This proves the condition in (1). † Next, we show that $|vy| > 0$. For the sake of contradiction, suppose $|vy| = 0$, i.e $v = y = \epsilon$. However, by cutting and pasting subtrees again, this shows that there is a derivation tree with *strictly less* number of nodes than in T that generates the word w, a contradiction. So, $|vy| > 0$.



Finally, we guarantee condition (3). Take all those paths from the root to leaves in T where some non-terminal repeats in the path, and take the path of maximum length, and hence the length of this path is $\geq |N| + 1$. Choose R to be among the last $|N| + 1$ vertices in this path. Hence, it follows that the maximum length of vxy can be $b^{|N|+1} = p$, i.e $|vxy| \leq p$, and this completes the proof. ■

Example 3.13. Let us revisit **Example** 3.7, and let us show that the language $L = \{a^n b^n c^n | n \geq 0\}$ is not a CFL, and it is quite easy to show this using the pumping lemma. As a consequence, the language

$$L' = \{w \mid |w|_a = |w|_b = |w|_c\}$$

is not context-free, because if L' was context free, the intersection $L' \cap a^* b^* c^* = L$ would be context free by **Proposition** 3.4, which is a contradiction.

Example 3.14. Let us show that the language

$$L = \{a^n b^m a^n b^m | n \geq 0, m \geq 0\}$$

is not a CFL. To be completed, see lecture 16. Consequently, it follows that if

$$L_3 = \{ww \mid w \in \{a, b\}^*\}$$

then L_3 is not a CFL, because

$$L = L_3 \cap a^* b^* a^* b^*$$

where we are applying **Proposition** 3.4 again. This proves the claim in **Example** 3.8.

3.10. **Greibach Normal Form.** This is another useful normal form that we shall study.

Definition 3.10. A grammar is in *Greibach Normal Form* (or GNF) if every production is of the form

$$X \to a\alpha$$

where $a \in \Sigma$ and $\alpha \in N^*$. So, GNF doesn't contain any ϵ or unit production.

Before proving the existence of GNFs for languages, we introduce a new notion.

Definition 3.11. Let G be a CFG, and let the non-terminals of G be listed as $X_1, ..., X_k$. Then, G is said to be *ordered* if all productions are of the form

$$X_i \to A\alpha$$

where $\alpha \in (N \cup \Sigma)^*$ and $A \in \Sigma \cup \{X_{i+1}, ..., X_k\}$. In simple words, any production begins with a terminal or a non-terminal with a higher index.

Proposition 3.12. *Any ordered CFG G has a GNF.*

Proof. The idea here is very similar to the Gaussian Elimination method of solving linear equations. Suppose the grammar G is ordered as $X_1, ..., X_k$. Then by the definition of an ordered grammar, every production of X_k must begin with a terminal. Now, look at all the productions of X_{k-1}. Since G is ordered, all productions must begin with either a symbol in Σ or the non-terminal X_k, i.e all productions of X_{k-1} are of the form

$$X_{k-1} \to a\alpha \text{ or } X_{k-1} \to X_k\alpha$$

where $\alpha \in (N \cup \Sigma)^*$ and $a \in \Sigma$. Suppose a production of the form $X_{k-1} \to X_k\alpha$ exists, and let $X_k \to \beta_1 \mid \beta_2 ... \mid \beta_m$ are all the productions of X_k. Then, *replace* the production $X_{k-1} \to X_k\alpha$ by the productions

$$X_{k-1} \to \beta_1\alpha \mid ... \mid \beta_m\alpha$$

(and the new grammar with these productions continues to accept the same language). So, all productions of X_{k-1} and X_k now begin with a terminal. Inductively, we can repeat the same procedure with the productions of the non-terminals $X_{k-2}, ..., X_1$ and hence we obtain a grammar where all productions of every non-terminal begins with a terminal.

Finally, to obtain GNF, consider a production $X_i \to s\alpha$ with $s \in \Sigma$. If α contains any terminals, we can replace them by appropriate non-terminals. This will give us the GNF. ■

Theorem 3.13. *If L is a CFL then $L - \{\epsilon\}$ has a grammar in GNF.*

Proof. Let $G = (N, \Sigma, P, X_1)$ be a grammar in CNF for $L - \{\epsilon\}$ without any ϵ or unit productions (possible by **Theorem** 3.10). By **Proposition** 3.12, it is enough to convert G to an equivalent ordered grammar. Suppose the non-terminals of G are $X_1, ..., X_k$, where X_1 is the starting non-terminal. Let us add new non-terminals $X_{-k}, X_{-k+1}, ..., X_{-1}$ to the grammar, and these non-terminals have no productions so far. We will see that in the end, the grammar that we obtain will be *ordered* with the ordering

$$X_{-k}, ..., X_{-1}, X_1, ..., X_k$$

Starting with $i = 1$, we perform the following algorithm.

(1) Consider all the productions of X_i. If $X_i \to A\alpha$ is a production, then $A \in \Sigma \cup \{X_1, ..., X_k\}$. Now if $A \in \Sigma \cup \{X_{i+1}, ..., X_k\}$, then in the ordered grammar, these productions are allowed. However, if $A \in \{X_1, ..., X_i\}$ then these productions cannot be allowed in the ordered grammar. If $i > 1$, then starting from $j = 1$ all the way to $j = i - 1$, do the following (inductively we assume that all productions of X_j satisfy the conditions of an ordered grammar):
   (a) Let $X_j \to \beta_1 \mid ... \mid \beta_m$ be all the productions of X_j.
   (b) If $X_i \to X_j\alpha$ is a production, then *replace* this production by the following productions

   $$X_i \to \beta_1\alpha \mid ... \mid \beta_m\alpha$$

   (and again, the modified grammar accepts the same language). This step ensures that all the RHS of productions of X_i now start with a symbol in $\Sigma \cup \{X_{j+1}, ..., X_k\}$.

(c) Put $j = j + 1$. Go to step (a).

After doing these steps, *each* RHS of a production of X_i now begins with a symbol in $\Sigma \cup \{X_i, ..., X_k\}$. The job is still not done, as we don't want X_i to be in this list. We handle this in step (2).

(2) This step is called *left-recursion elimination*. After step (1), let all the productions of X_i be of the following forms:

$$(***) \qquad \begin{aligned} X_i &\to X_i\alpha_1 \mid X_i\alpha_2 \mid ... \mid X_i\alpha_m \\ X_i &\to \beta_1 \mid ... \mid \beta_l \end{aligned}$$

where each β_j begins with a symbol in $\Sigma \cup \{X_{i+1}, ..., X_k\}$. In simple words, the first set of productions are what we want to eliminate, and the second set of productions are what are acceptable in the ordered grammar. Using derivation trees, we see that

$$X_1 \xrightarrow{*} \alpha X_i \beta \xrightarrow{*} w$$

iff

$$X_1 \xrightarrow{*} \alpha X_i \beta \xrightarrow{*} \alpha\beta_j\alpha_{i_j}\alpha_{i_{j-1}}...\alpha_{i_1}\beta \xrightarrow{*} w$$

where $\beta_j \in \{\beta_1, ..., \beta_l\}$ and $\alpha_{i_1}, ..., \alpha_{i_j} \in \{\alpha_1, ..., \alpha_m\}$. So, the non-terminal X_i is always used to derive a word in

$$(\beta_1 + ... + \beta_l)(\alpha_1 + ... + \alpha_m)^*$$

within any derivation. Here is another way of generating the same words without the productions in $(***)$: *replace* all those productions with the following set of productions.

$$\begin{aligned} X_i &\to \beta_1 X_{-i} \mid \beta_2 X_{-i} \mid ... \mid \beta_l X_{-i} \\ X_{-i} &\to \alpha_1 X_{-i} \mid \alpha_2 X_{-i} \mid ... \mid \alpha_m X_{-i} \mid \epsilon \end{aligned}$$

Then, all productions of X_i now begin with a symbol in $\Sigma \cup \{X_{i+1}, ..., X_k\}$. Moreover, observe that $\alpha_j \neq \epsilon$ for any $1 \leq j \leq m$ (as the original grammar G does not contain any unit productions), hence no unit productions are introduced. Also, the productions involving X_{-i} also satisfy the ordered grammar conditions.

(3) Put $i = i + 1$ and continue with step (1) if $i + 1 \leq k$. Otherwise halt.

After doing all these steps, we obtain an *ordered* grammar with some ϵ-productions. Observe that the only non-terminals which can have ϵ-productions belong to the set $\{X_{-1}, ..., X_{-k}\}$. So, as in **Proposition** 3.12, we can convert this grammar to GNF even though there are ϵ-productions. Finally, eliminate the ϵ-productions, and the resultant grammar will still be in GNF. This completes the proof. ■

Proof 2 of **Theorem** 3.13. There is another proof of the existence of GNFs. See lecture 17 from the 50:00 mark and from the beginning of Lecture 18 to its 30:00 mark. This proof is apparently in Kozen's book. ■

3.11. **Push-Down Automatons.** In this section, we will introduce machines which recognize context-free languages. These are the so called *pushdown automata*. Intuitively, a *pushdown automaton* is like an NFA which has memory, and the memory storage is in the form of a *stack*. The automaton, by reading letters or ϵ, can move to a different state, and can replace the top of a stack by a new

word (note that in a stack, we can only access its top). We formalize this notion below.

Definition 3.12. A *pushdown automaton* or PDA is a tuple $(Q, \Sigma, \Gamma, \delta, s, \perp, F)$ where the meanings of the symbols are as follows.

(1) Q is the set of states of the PDA.
(2) Σ is the *alphabet*.
(3) s is the initial state of the PDA.
(4) F is the set of final states of the PDA.
(5) Γ is the *stack alphabet*.
(6) $\perp$ is the *initial stack symbol*.
(7) δ is the set of *transitions*, i.e $\delta \subset (Q \times \Sigma \cup \{\epsilon\} \times \Gamma) \times (Q \times \Gamma^*)$ and δ is a *finite subset*. More precisely, any transition is of the form $(q, v) \xrightarrow{a} (q', w)$, which means that if the current state of the PDA is q and the top stack symbol is v, then by reading a (where $a \in \Sigma \cup \{\epsilon\}$), we go to state q' and replace the top of the stack by the *word* w.

Remark 3.13.1. Note that by this definition, PDAs are non-deterministic machines. This is because δ is defined to be an *arbitrary* subset of the set $(Q \times \Sigma \cup \{\epsilon\} \times \Gamma) \times (Q \times \Gamma^*)$ and it does *not* have to be a function. Moreover, PDAs are allowed to non-deterministically switch states without reading any letter.

Remark 3.13.2. Another point to be noted is that *if* the stack becomes empty, the automaton has to stop the computation. This is inherent in our definition of δ.

Definition 3.13. A pair (q, γ) where $q \in Q$ and $\gamma \in \Gamma^*$ is called a *configuration* of the PDA. Given configurations (q_1, γ_1) and (q_n, γ_n) and a word $w = a_1 ... a_k$, we write

$$(q_1, \gamma_1) \xrightarrow{w} (q_n, \gamma_n)$$

and call this a *run* if there is a sequence of transitions

$$(q_1, \gamma_1) \xrightarrow{a_1} (q_2, \gamma_2) \xrightarrow{a_2} ... \xrightarrow{a_n} (q_n, \gamma_n)$$

and note that in any transition, only the *first* letter of γ_i is used in a transition (i.e only the top of the stack is removed). The configuration $(s, \perp)$ is called *the initial configuration* of the PDA, and a configuration of the form (q, γ) with $q \in F$ and $\gamma \in \Gamma^*$ is called an *accepting configuration*.

Definition 3.14. A word w is *accepted* in the PDA if there is a run

$$(s, \perp) \xrightarrow{w} (q, \gamma)$$

for some accepting configuration (q, γ). If A is a PDA, we define

$$L(A) := \{w \mid w \text{ has an accepting run in } A\}$$

Remark 3.13.3. Observe that, the only rule to accept a word is that it must end up in a final state, and we don't care about what is going on in the stack.

Remark 3.13.4. Note that every PDA is equivalent to one which has only one final state. This can be easily done as follows: if A is a PDA, add a new state t to A. Then, from every final state q_f in A, add an ϵ-transition from q to the new state t (no matter what the top of the stack is), and in this new PDA, make t the *only* final state.

Remark 3.13.5. We can arrange our PDA so that reaching the final state also *empties* the stack. This is again easy, and is very similar to the construction in the previous remark; we add a new state t which will be our only final state, and we can ensure that t, if reached, can also empty the stack. This is accomplished by just adding transitions from t to itself which pops the stack.

3.12. **Closure Properties of PDAs.** As usual, we will review some closure properties that PDAs have.

Proposition 3.14. *Let $A_1 = (Q_1, \Sigma, \Gamma_1, \delta_1, s_1, \perp_1, F_1)$ and $A_2 = (Q_2, \Sigma, \Gamma_2, \delta_2, s_2, \perp_2, F_2)$ be two PDAs, and without loss of generality suppose all components of these PDAs are pairwise disjoint (except Σ). Then the following hold.*

(1) *$L(A_1) \cup L(A_2)$ is recognized by some PDA.*

Proof. In each case we shall give a construction of a PDA.

(1) We will construct a new PDA A as follows. We add a new start state s to the PDA and add a new stack symbol $\perp$ to the PDA. So,

$$A = (Q_1 \cup Q_2 \cup \{s\}, \Sigma, \Gamma_1 \cup \Gamma_2 \cup \{\perp\}, \delta, s, \perp, F_1 \cup F_2)$$

and the set of transitions δ is simply

$$\delta = \delta_1 \cup \delta_2 \cup \{(s, \perp) \xrightarrow{\epsilon} (s_1, \perp_1), (s, \perp) \xrightarrow{\epsilon} (s_2, \perp_2)\}$$

and it is clear that the new PDA A will accept the language $L(A_1) \cup L(A_2)$. ■

Proposition 3.15. *Suppose L is a language over Σ recognized by some PDA A, and suppose $h : \Delta^* \to \Sigma^*$ is a homomorphism. Then, $h^{-1}(L) \subset \Delta^*$ is also recognized by some PDA. In particular, this proves* **Theorem** 3.6*, as we will show ahead that CFLs are equivalent to PDAs.*

Proof. Let $A = (Q, \Sigma, \Gamma, \delta, s, \perp, F)$ be a PDA recognizing L. We will make a new PDA $A_{h^{-1}}$ as follows. Let Suffixes$(h(\Delta))$ denote the set of all suffixes of words in $h(\Delta)$. Let the PDA $A_{h^{-1}}$ be as follows.

$$A_{h^{-1}} = (Q \times \text{Suffixes}(h(\Delta)), \Delta, \Gamma, \delta_1, (s, \epsilon), \perp, F \times \{\epsilon\})$$

Now, we come to the transitions in δ_1. First, let all transitions of the form

$$((q, \epsilon), X) \xrightarrow{a} ((q, h(a)), X)$$

be added to $A_{h^{-1}}$ (and these transitions just mean that if we read a symbol $a \in \Delta$, we will stimulate the old PDA A on the word $h(a) \in \Sigma^*$). Now, suppose $(q, X) \xrightarrow{c} (q', \gamma)$ is a transition in A. Then for any $x \in \Sigma^*$ such that $cx \in$ Suffixes$(h(\Delta))$, add the transition

$$((q, cx), X) \xrightarrow{\epsilon} ((q', x), \gamma)$$

to the PDA $A_{h^{-1}}$ (and these transitions juts mean how we are stimulating the old PDA A). Hence, the PDA $A_{h^{-1}}$ accepts the language $h^{-1}(L)$, completing the proof (A lot of details are missing, but they are relatively easy to prove). ■

Remark 3.15.1. The above proof may be hard to understand, but the idea is very simple. If we have a word $a_1 ... a_k \in \Delta^*$, to see whether it belongs to $h^{-1}(L)$, we just have to stimulate the old PDA on the word $h(a_1)...h(a_k)$. This is exactly what we are doing above.

3.13. **Another Mode of Acceptance.** Observe that by our definition, a word is accepted in a PDA if the word reaches a final state via some run. However, we also have the power of a stack with us, and we can ask the following question: suppose we change the definition of acceptance to that wherein a word is accepted if it empties the stack. Then is this notion stronger/weaker or equivalent to our old definition of acceptance?

Definition 3.15. Let A be any PDA, and define the language

$$N(A) := \{w \mid (s, \perp) \xrightarrow{w} (q, \epsilon) \text{ for some } q \in Q\}$$

and this is the new idea of acceptance that we mentioned in the above paragraph.

For this section, we will give another definition.

Definition 3.16. Let $A = (Q, \Sigma, \Gamma, \delta, s, \perp, F)$ be a PDA. Define a new PDA A' as

$$A' = (Q \cup \{t\}, \Sigma, \Gamma, \delta', s, \perp, \{t\})$$

with

$$\delta' = \delta \cup \{(f, X) \xrightarrow{\epsilon} (t, X) \mid f \in F, X \in \Gamma\} \cup \{(t, X) \xrightarrow{\epsilon} (t, \epsilon) \forall X \in \Gamma\}$$

Then it is easily seen that $L(A) = L(A')$, and a word w is accepted in A' if and only if $(s, \perp) \xrightarrow{w} (t, \epsilon)$ in A', i.e if and only if w empties that stack and reaches the final state t (see **Remark** 3.13.5).

Proposition 3.16. *Let A be any PDA, and let A' be constructed as in* **Definition** 3.16*. Then $L(A') \subseteq N(A')$.*

Proof. This easily follows from the comments in **Definition** 3.16, because a word w is accepted in A' if and only if it empties the stack. ■

Remark 3.16.1. This inclusion is *not* true in general.

Proposition 3.17. *Let A be any PDA, and let A' be constructed as in* **Definition** 3.16*. Suppose A never empties its stack. Then $N(A') \subseteq L(A')$, and from* **Proposition** 3.16 *we get that*

$$L(A') = N(A')$$

Proof. Suppose $w \in N(A')$, and hence we know that $(s, \perp) \xrightarrow{w} (q, \epsilon)$ for some state q in A'. If $q \neq t$, then this run is also a run in A (by the construction of A'). However, this is a contradiction since A never empties its stack. So, $q = t$ and hence $w \in L(A')$, showing that $N(A') \subseteq L(A')$. ■

Proposition 3.18. *Let A be any PDA. Then, A is equivalent to a PDA which never empties its stack.*

Proof. Let $A = (Q, \Sigma, \Gamma, \delta, s, \perp, F)$ be a PDA. We make a new PDA A^* as follows. Let

$$A^* = (Q \cup \{s_1\}, \Sigma, \Gamma \cup \{\perp_1\}, \delta', s_1, \perp_1, F)$$

where the transitions are given by

$$\delta' = \delta \cup \{(s_1, \perp_1) \xrightarrow{\epsilon} (s, \perp\perp_1)\}$$

Intuitively, what we have done is just adding a new start state and a new initial stack symbol $\perp_1$, and we have ensured that even if the old PDA empties the stack, the new PDA cannot get rid of the new initial stack symbol $\perp_1$ at all. It is clear that $L(A^*) = L(A)$, and hence we are done. ■

Through the above propositions, we have proven the following important fact.

Proposition 3.19. *Suppose L is accepted by some PDA A, i.e $L = L(A)$ for some PDA A. Then it is accepted via empty stack by some PDA A'', i.e*

$$L = N(A'')$$

for some PDA A''.

Proof. Let A be a PDA and let $L = L(A)$. Construct a PDA A^* as in **Proposition** 3.18, i.e

$$L = L(A) = L(A^*)$$

where A^* is a PDA that does not empty its stack. Then, construct a PDA $(A^*)'$ as in **Definition** 3.16. By **Proposition** 3.17, we know that

$$L((A^*)') = N((A^*)')$$

and as in **Definition** 3.16 we know that

$$L(A^*) = L((A^*)')$$

and so

$$L = L(A) = L(A^*) = L((A^*)') = N((A^*)')$$

and the claim follows by putting $A'' = (A^*)'$. ■

We can also prove a converse to **Proposition** 3.19 and we do so now.

Proposition 3.20. *Suppose $L = N(A)$ for some PDA A. Then, there is some PDA A'' such that $L = L(A'')$.*

Proof. This construction is very similar to what we have been doing above. Let $A = (Q, \Sigma, \Gamma, \delta, s, \bot, F)$ be a PDA such that $L = N(A)$. We construct A'' as follows. Put

$$A'' = (Q \cup \{s_1\} \cup \{t\}, \Sigma, \Gamma \cup \{\bot_1\}, \delta'', s_1, \bot_1, \{t\})$$

and let

$$\delta'' = \delta \cup \{(s_1, \bot_1) \xrightarrow{\epsilon} (s, \bot\bot_1)\} \cup \{(q, \bot_1) \xrightarrow{\epsilon} (t, \epsilon) \forall q \in Q\}$$

Intuitively, we are adding a new start state and initial stack symbol $\bot_1$ to ensure that the new PDA never empties its stack. Moreover, if we reach a state of the form $(q, \bot_1)$ in the new PDA, then it means that the old PDA has emptied its stack, so we can jump directly to the new final state t. It is clear that $N(A) = L(A'')$, and hence this completes the proof. ■

Remark 3.20.1. Infact above, we have the equality $N(A) = L(A'') = N(A'')$, but this is unnecessary.

So via **Proposition** 3.19 and **Proposition** 3.20, we have proven that:

$$\begin{aligned} &\text{acceptance via final states} \\ \equiv\ &\text{acceptance via empty stack} \\ (\equiv\ &\text{acceptance that demands both}) \end{aligned}$$

and this is very convenient, since we can either work with an empty stack or final states.

3.14. **CFLs and PDAs.** Let us now look at the relationship between PDAs and CFLs.

Proposition 3.21. *Let G be a CFG. Then there is a PDA A such that $L(G) = N(A)$, i.e $L(G)$ is accepted by a PDA via the empty stack.*

Proof. The idea is simply to stimulate the CFG using a stack. Let $G = (N, \Sigma, P, S)$ be a CFG. We construct a PDA A as follows. Let

$$A = (\{q\}, \Sigma, \Gamma = N \cup \Sigma, \delta, q, \perp = S, \phi)$$

(note that there is no final state, since we will be accepting words via empty stack). The set of transitions δ will be as described below.

(1) For every production $X \to \alpha$ in G let $(q, X) \xrightarrow{\epsilon} (q, \alpha)$ be a transition in A.
(2) For every $a \in \Sigma$, add the transition $(q, a) \xrightarrow{a} (q, \epsilon)$ in A.

We claim that $N(A) = L(G)$. We will do this in two steps.

(1) We first show that $N(A) \subseteq L(G)$. To prove this, we prove a stronger statement: if $(q, S) \xrightarrow{w} (q, \gamma)$ is a run in A, then $S \xrightarrow{*} w\gamma$ is a left-most derivation in G, and we prove this by induction on the length of the run. For the base case, the length of the run is zero, i.e $w = \epsilon$ and the run is simply $(q, S) \xrightarrow{\epsilon} (q, S)$. Evidently, $S \xrightarrow{*} wS = S$ is a left-most derivation in G, and hence the base case is true. For the inductive case, suppose the run $(q, S) \xrightarrow{w} (q, \gamma)$ has length $n + 1$ for some $n \geq 0$, and suppose the claim holds true for all runs of length n. So the run can be written as $(q, S) \xrightarrow{w_1} (q, X\gamma_1) \xrightarrow{w_2} (q, \gamma)$ where $w = w_1 w_2$, $X \in \Gamma$ and $(q, S) \xrightarrow{w_1} (q, X\gamma_1)$ is a run of length n in A and $(q, X\gamma_1) \xrightarrow{w_2} (q, \gamma)$ is a transition in A. By the inductive hypothesis, we see that $S \xrightarrow{*} w_1 X\gamma_1$ is a left-most derivation in G. Now two cases are possible. First, $X \in N$ (i.e X is a non-terminal). In this case, we must have $w_2 = \epsilon$ (and so $w_1 = w$) and there is some production $X \to \alpha$ such that $\gamma = \alpha\gamma_1$. So, we have a left most derivation $S \xrightarrow{*} w_1 X\gamma_1 \to w_1\alpha\gamma_1 = w\gamma$. In the second case, $X \in \Sigma$, i.e X is a terminal. In that case, we have $w_2 = X$, and $\gamma_1 = \gamma$. So, we have a left-most derivation $S \xrightarrow{*} w_1 X\gamma_1 = w\gamma$. So by induction, the statement holds for all runs, and the proof is complete.
(2) Next we show that $L(G) \subseteq N(A)$. To prove this, we will show that if $S \xrightarrow{*} w\gamma$ is a left-most derivation in G ($w \in \Sigma^*$ and $\gamma \in (N \cup \Sigma)^*$) such that γ *does not* begin with a terminal, then there is a run $(q, S) \xrightarrow{w} (q, \gamma)$ in the PDA A, and we will do this by induction on the length of the derivation $S \xrightarrow{*} w\gamma$. For the base case, suppose the derivation length is 0, so that $w = \epsilon$ and $\gamma = S$, and clearly there is a run $(q, S) \xrightarrow{\epsilon} (q, S)$, and so the base case is true. Now suppose $S \xrightarrow{*} w\gamma$ is a left-most derivation of length $n + 1$ such that γ does not start with a terminal, and suppose the statement holds true for all derivations of length atmost n. Since we are dealing with left-most derivations, we can write this derivation as

$$S \xrightarrow{*} w_1 X\gamma_1 \xrightarrow{X \to \alpha} w_1\alpha\gamma_1 = w\gamma$$

where $X \to \alpha$ is a production in G, $w_1 \in \Sigma^*$ and $\alpha, \gamma_1 \in (N \cup \Sigma)^*$. Now we can write

$$\alpha\gamma_1 = x\gamma'$$

where $x \in \Sigma^*$ and γ' does not begin with a terminal. So it is clear that $w = w_1 x$ and $\gamma = \gamma'$. By inductive hypothesis, we have a run

$$(q, S) \xrightarrow{w_1} (q, X\gamma_1)$$

and we can extend this run as

$$(q, S) \xrightarrow{w_1} (q, X\gamma_1) \xrightarrow{\epsilon} (q, \alpha\gamma_1) = (q, x\gamma') \xrightarrow{x} (q, \gamma') = (q, \gamma)$$

and hence we have a run $(q, S) \xrightarrow{w} (q, \gamma)$. So by induction, the statement has been proven. Now if $S \xrightarrow{*} w$ is a derivation, it is clear that $(q, S) \xrightarrow{w} (q, \epsilon)$ is a derivation in A, proving that $L(G) \subseteq N(A)$.

By (1) and (2), it follows that A is a PDA accepting $L(G)$ via the empty stack. Since acceptance via empty stack is the same as acceptance via final states, it follows that $L(G)$ is accepted by some PDA. ■

Next, we will see how to convert a PDA to a CFG.

Proposition 3.22. *If $L = N(A)$ for some PDA A, then there is a CFG G such that $L = L(G)$. So, if a language is accepted by a PDA via the empty stack, then it is a CFL.*

Proof. Let $A = (Q, \Sigma, \Gamma, \delta, s, \perp, \phi)$ be a PDA with $L = N(A)$. For any $p, q \in Q$ and $X \in \Gamma$, put

$$L(p, X, q) := \{w \in \Sigma^* \mid (p, X) \xrightarrow{w} (q, \epsilon)\}$$

i.e $L(p, X, q)$ is the set of all those words which can take state p to q when the stack contains only X and which empties the stack. An immediate observation from this is that for any $\gamma \in \Gamma^*$ and any $w \in L(p, X, q)$, we have

$$(p, X\gamma) \xrightarrow{w} (q, \gamma)$$

and this is a simple fact that we will make use of. It is clear that

$$(*) \qquad \bigcup_{q \in Q} L(s, \perp, q) = N(A) = L$$

Our strategy will be to use non-terminals of the form ${}^pX^q$ in a grammar G such that

$$\{w \mid {}^pX^q \xrightarrow{*} w \text{ in } G\} = L(p, X, q)$$

So let G be a context-free grammar given by

$$G = (\{{}^pX^q \mid p, q \in Q, X \in \Gamma\} \cup S, \Sigma, P, S)$$

where S will be the start symbol of our grammar G. Keeping in mind what ${}^pX^q$ should do, the productions are as follows: first, add the productions

$$\{S \to {}^s\perp^q : \ q \in Q\}$$

Then if $(p, X) \xrightarrow{c} (q, \epsilon)$ is a transition in A ($c \in \Sigma \cup \{\epsilon\}$), add the production

$${}^pX^q \to c$$

to the grammar G. Next, suppose $(p, X) \xrightarrow{c} (p', X_1 X_2 ... X_k)$ is a transition in A. For *all* $q_1, ..., q_{k-1}$ in Q, add the production

$${}^pX^q \to c \ {}^{p'}X_1^{q_1} \ {}^{q_1}X_2^{q_2} .... {}^{q_{k-1}}X_k^q$$

to the grammar G.

Now, let $L_G({}^pX^q)$ be defined as

$$L_G({}^pX^q) := \{w \in \Sigma^* \mid {}^pX^q \xrightarrow{*} w \text{ in } G\}$$

We will now prove that

$$L_G({}^pX^q) = L(p, X, q)$$

as we mentioned earlier. We will do this in two steps.

(1) First, we show that $L_G({}^pX^q) \subseteq L(p, X, q)$, and we prove this by induction on the length of the derivation of a word. So, let $w \in L_G({}^pX^q)$. For the base case, suppose the length of the derivation ${}^pX^q \xrightarrow{*} w$ is 1. Then it must be that $w \in \Sigma$ and the derivation is ${}^pX^q \to w$, which is simply a production. It then follows that $(p, X) \xrightarrow{w} (q, \epsilon)$ is a transition in A, and hence $w \in L(p, X, q)$, proving the base case. Now, suppose the derivation ${}^pX^q \xrightarrow{*} w$ in G has length $n+1$ for some $n \geq 1$, and suppose the statement is true for all derivations of length atmost n. By the nature of productions in G, the derivation must be of the form

$${}^pX^q \to c\ {}^{p'}X_1^{q_1}\ {}^{q_1}X_2^{q_2}...{}^{q_{k-1}}X_k^q \xrightarrow{*} cw_1...w_k = w$$

where $w_1 \in L_G({}^{p'}X_1^{q_1})$ and $w_i \in L_G({}^{q_{i-1}}X_i^{q_i})$ for each $2 \leq i \leq k$, where $q_k = q$. By the inductive hypothesis, it follows that $w_1 \in L(p', X_1, q_1)$ and $w_i \in L(q_{i-1}, X_i, q_i)$ for each $2 \leq i \leq k$ (again $q_k = q$). Using this information, we can make a run $(p, X) \xrightarrow{w} (q, \epsilon)$ easily, and the run is given below.

$$(p, X) \xrightarrow{c} (p', X_1X_2...X_k) \xrightarrow{w_1} (q_1, X_2...X_k) \xrightarrow{w_2} (q_2, X_3...X_k) \xrightarrow{w_3} ... \xrightarrow{w_{k-1}} (q_{k-1}, X_k) \xrightarrow{w_k} (q, \epsilon)$$

and hence this shows that $w \in L(p, X, q)$. This completes the proof by induction, and shows that $L_G({}^pX^q) \subseteq L(p, X, q)$.

(2) Next we prove the reverse inclusion, i.e $L(p, X, q) \subseteq L_G({}^pX^q)$. Let $w \in L(p, X, q)$, i.e there is a run $(p, X) \xrightarrow{w} (q, \epsilon)$. We will prove the claim by induction on the length of the run. For the base case, suppose the length of the run is 1. In that case, $(p, X) \xrightarrow{w} (q, \epsilon)$ is a transition in A (and $w \in \Sigma \cup \{\epsilon\}$). By definition of G, it follows that ${}^pX^q \to w$ is a production in G, and hence $w \in L_G({}^pX^q)$, and so the base case is true. For the inductive case, suppose the length of the run is $n + 1$ for some $n \geq 1$, and suppose the statement is true for all runs of length atmost n. Suppose the first step in the run is as follows.

$$(p, X) \xrightarrow{c} (p', X_1...X_k) \xrightarrow{w'} (q, \epsilon)$$

where $c \in \Sigma \cup \{\epsilon\}$ and $w = cw'$. Observe that $k \geq 1$, because the run $(p', X_1...X_k) \xrightarrow{w'} (q, \epsilon)$ has length $n \geq 1$. We observe the following: after reading c, our stack contains k symbols. For the stack to become empty, it must reach height i for every $1 \leq i \leq k$ at some point in the run. So, let q_1 be the *first* state at which the stack height becomes $k - 1$, i.e let q_1 be the *first* state for which the run is of the form

$$(p, X) \xrightarrow{c} (p', X_1...X_k) \xrightarrow{w_1} (q_1, X_2...X_k) \xrightarrow{w''} (q, \epsilon)$$

where $w = cw_1w''$. We then observe that $w_1 \in L(p', X_1, q_1)$ (and this is why we chose the *first* such state). Continuing this further, we can break down this run as

$$(p, X) \xrightarrow{c} (p', X_1...X_k) \xrightarrow{w_1} (q_1, X_2...X_k) \xrightarrow{w_2} (q_2, X_3...X_k) \xrightarrow{w_3} ... \xrightarrow{w_{k-1}} (q_{k-1}, X_k) \xrightarrow{w_k} (q, \epsilon)$$

such that $w = cw_1w_2..w_{k-1}w_k$ and

$$w_1 \in L(p', X_1, q_1)$$
$$w_2 \in L(q_1, X_2, q_2)$$
$$...$$
$$w_{k-1} \in L(q_{k-2}, X_{k-1}, q_{k-1})$$
$$w_k \in L(q_{k-1}, X_k, q)$$

By inductive hypothesis, we see that $w_1 \in L_G({}^{p'}X_1^{q_1}), ..., w_k \in L_G({}^{q_{k-1}}X_k^q)$. So, we see that the derivation

$${}^pX^q \to c\ {}^{p'}X_1^{q_1}\ {}^{q_1}X_2^{q_2}...{}^{q_{k-1}}X_k^q \xrightarrow{*} cw_1...w_k = w$$

is a valid derivation in G, and this shows that $L(p, X, q) \subseteq L_G({}^pX^q)$.

So by (1) and (2), we have shown that $L_G({}^pX^q) = L(p, X, q)$. So by $(*)$, we see that $L(G) = N(A) = L$, and this completes the proof. ■

So we have proved the following powerful theorem.

Theorem 3.23. *The class of context-free languages and languages recognized by PDAs are the same.*

Proof. By **Proposition** 3.21, any CFL is recognized by some PDA via the empty stack, i.e every CFL is recognized by some PDA. Conversely, given a PDA, there is an equivalent PDA having the same language which accepts the language via the empty stack. By **Proposition** 3.22, this PDA can be converted to a CFG. This proves the theorem. ■

3.15. **Deterministic PDAs.** In this section, we will try to formulate the notion of determinism in PDAs. It turns out that PDAs without ϵ-transitions are too weak. So, we will have to formulate determinism in the presence of PDAs. Informally, this is done as follows.

(1) If a state p has an ϵ transition of the form $(p, X) \xrightarrow{\epsilon} (q, Y)$, then there *cannot* be any other transition coming out of (p, X).
(2) If there is no ϵ-transition coming out of (p, X), then for every $a \in \Sigma$, there must be a transition labelled a coming out of (p, X).

The above two conditions guarantee that any word will have a unique run in the deterministic PDA (however, it *does not* mean that *all* words will have a run). Also, in the case of PDAs, we had the power of *guessing*. In deterministic PDAs, we won't have this power anymore, and to remedy this we introduce a new symbol called the *right endmarker*, which acts as a delimiter denoting the end of string in an input. This is made more precise in the following definition.

Definition 3.17. A *Deterministic Pushdown Automaton* (DPDA) A is an octuple

$$A = (Q, \Sigma, \Gamma, \delta, \bot, \$, s, F)$$

where all symbols have the same meaning as in PDAs, and the following hold.

(1) $\$$ is a new symbol called the *right endmarker*, and

$$\delta \subseteq (Q \times \Sigma \cup \{\$, \epsilon\} \times \Gamma) \times (Q \times \Gamma^*)$$

and ofcourse δ is a finite set.
(2) A is *deterministic* in the following sense: for every $p \in Q$ and $X \in \Gamma$, exactly one of two statements below is true.

(a) The *only* transition emerging from (p, X) is an ϵ-transition $(p, X) \xrightarrow{\epsilon} (q, Y)$ (where $q \in Q, Y \in \Gamma^*$), and there is *exactly one* such ϵ-transition.
(b) There is *no* ϵ-transition emerging from (p, X), and for *every* symbol $c \in \Sigma \cup \{\$\}$, there is *exactly one* transition $(p, X) \xrightarrow{c} (q, Y)$, where $q \in Q$ and $Y \in \Gamma^*$.

We define

$$L(A) := \{w \in \Sigma^* \mid \exists q_f \in F, \gamma \in \Gamma^* \text{ such that } (s, \bot) \xrightarrow{w\$} (q_f, \gamma)\ \}$$

So in simpler words, w is accepted if and only if $w\$$ has an accepting run in the DPDA A. Any language accepted by some DPDA is called a *Deterministic Context Free Language* (DCFL).

Remark 3.23.1. By our definition, it follows that every regular language is actually a DCFL. To see this, we can look at a DFA for a regular language as a DPDA that does not use its stack at all.

3.16. **Closure of DCFLs under Complementation.** The main goal of this section will be to prove the closure of DCFLs under complementation (Unfortunately I don't have time to finish this section right now. In any case, most of the arguments done in class were taken from Kozen's book. So it is enough to look at that).

3.17. **Parikh's Theorem.** In this section, we will prove an important theorem, which will roughly mean that if we only focus on lengths of words, then there is no difference between regular languages and CFLs.

Definition 3.18. Let $\mathbb{N}$ be the set of natural numbers *including* 0. Let $k \geq 1$ be an integer, and consider the space $\mathbb{N}^k$. A set $S \subset \mathbb{N}^k$ is called a *linear set* if there are vectors $b, v_1, v_2, ..., v_n \in \mathbb{N}^k$ such that

$$S = b + \langle v_1, ..., v_n \rangle := \{b + c_1 v_1 + ... + c_n v_n \mid c_i \geq 0, c_i \in \mathbb{N}\}$$

A subset of $\mathbb{N}^k$ is said to be *semi-linear* if it is a finite union of linear subsets of $\mathbb{N}^k$.

Definition 3.19. Let $\Sigma = \{a_1, ..., a_k\}$ for some $k \geq 1$. Define the *Parikh function* $\Psi : \Sigma^* \to \mathbb{N}^k$ as

$$\Psi(w) = (|w|_{a_1}, ..., |w|_{a_k})$$

where $|w|_{a_i}$ refers to the number of a_i's in w.

Example 3.15. Let $\Sigma = \{a, b\}$, and let L_{EP} be the language of all even length palindromes over Σ. In any such palindrome, the number of a's and the number of b's must be even, and conversely, given an even number of a's and b's, we can construct an even length palindrome containing that number of a's and b's. In that case, we see that

$$\Psi(L_{\mathsf{EP}}) = \{(2i, 2j) \mid i, j \geq 0\}$$

and so

$$\Psi(L_{\mathsf{EP}}) = (0, 0) + \langle (2, 0), (0, 2) \rangle$$

which means that $\Psi(L_{\mathsf{EP}})$ is a semi-linear (infact linear) subset of $\mathbb{N}^2$. Now, let L_{P} be the set of *all* palindromes over Σ. It can be easily seen that

$$\Psi(L_{\mathsf{P}}) = (0, 0) + \langle (2, 0), (0, 2) \rangle \cup (1, 0) + \langle (2, 0), (0, 2) \rangle \cup (0, 1) + \langle (2, 0), (0, 2) \rangle$$

and hence in this case too, $\Psi(L_{\mathsf{P}})$ is a semi-linear (but not linear) subset of $\mathbb{N}^2$.

Theorem 3.24 (**Parikh's Theorem**). *If L is a CFL over $\Sigma = \{a_1, ..., a_k\}$, then $\Psi(L)$ is a semi-linear subset of $\mathbb{N}^k$.*

Before proving the theorem, I will justify the remarks made near the beginning of this section.

Theorem 3.25. *Let S be any semi-linear subset of $\mathbb{N}^k$. Then, there is a regular language R over $\Sigma = \{a_1, ..., a_k\}$ such that $\Psi(R) = S$, where Ψ is the Parikh function.*

Proof. First, let us show that any *linear* subset of $\mathbb{N}^k$ is the Parikh image of some regular language. Let the linear subset be given by

$$S = b + \langle v_1, ..., v_n \rangle$$

Take words $w, w_1, ..., w_n \in \Sigma^*$ such that $\Psi(w) = b$ and $\Psi(w_i) = v_i$ for each $1 \leq i \leq n$. Then, consider the language

$$L = w(w_1 + ... + w_n)^*$$

and it is clear that

$$\Psi(L) = S$$

Now, suppose S is a semi-linear set, i.e a finite union of linear subsets of $\mathbb{N}^k$. Corresponding to each linear set, pick a regular language, and take the union of those regular languages. Since regular languages are closed under (finite) union, the claim follows and hence the proof is complete. ■

Example 3.16. Consider the language $L = \{a^n b^n c^n \mid n \geq 0\}$, and it is easy to see that this is not a CFL by the pumping lemma. However, we have

$$\Psi(L) = (0, 0, 0) + \langle (1, 1, 1) \rangle$$

so that $\Psi(L)$ is a semi-linear subset of $\mathbb{N}^k$. This example shows that the converse of **Parikh's Theorem** 3.24 is not true.

In the proof of Parikh's Theorem, we will need some terminology which is given below.

Definition 3.20. Let G be any CFG. Then define

$$\mathcal{T}_f(G) := \{T \mid T \text{ is a derivation tree in which all non-terminals of } G \text{ appear}\}$$

Also, define

$$L_f(G) := \{y(T) \mid T \in \mathcal{T}_f(G)\}$$

and the small f usually stands for *full*, where for any derivation tree T in G, we define

$$y(T) := \text{string derived by } T$$

The letter y usually stands for *yield*. Finally, for any derivation tree T in G, define its Parikh image as

$$\Psi(T) := \Psi(y(T))$$

and for any collection B of derivation trees of G, define

$$\Psi(B) := \{\Psi(T) \mid T \in B\}$$

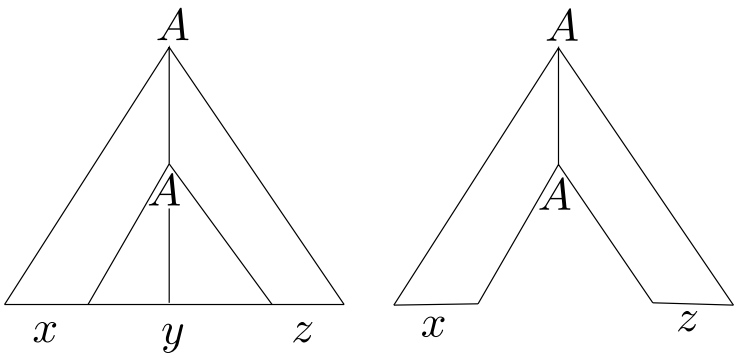


Figure 1. Tree T on the left, and its corresponding pump on the right. The subtree t generates the word y.

Definition 3.21. Let G be any CFG, and let $A \in N$ be any non-terminal. Let T be a derivation tree in G with root A such that T contains a non-root node labelled A. Let t be the subtree rooted at the non-root node labelled A. Then the tree $T - t \cup \{A\}$ is called a *pump* (where $T - t \cup \{A\}$ is the tree T with the subtree t removed but the root of t kept. See the diagram below). For any such pump p, we define its Parikh image denoted by $\Psi(p)$ as

$$\Psi(p) := \Psi(xz)$$

and for any collection P of pumps, define

$$\Psi(P) := \{\Psi(p) \mid p \in P\}$$

Remark 3.25.1. While proving the pumping lemma for CFLs (**Theorem** 3.11), we used pumps, but didn't define them explicitly.

Proof of **Parikh's Theorem** 3.24. Let $G = (N, \Sigma, P, S)$ be a CFG for L. We immediately see that

$$L = L(G) = \{y(T) \mid T \text{ is a derivation tree}\}$$

Clearly, we have that $L_f(G) \subseteq L(G)$. In most cases, this inclusion will be strict. However, we now show that

$$L(G) = L_f(G_1) \cup ... \cup L_f(G_k)$$

for a finite collection $G_1, .., G_k$ of CFGs. And this is pretty easy. Corresponding to *every* subset $X \subseteq N$ that contains the starting non-terminal S, consider the CFG G_X, where the productions in G_X are all those productions in G which only involve non-terminals in the set X. So, we see that

$$L(G) = \bigcup_{X \subseteq N, S \in X} L_f(G_X)$$

and clearly this union is finite since there are finitely many such sets X. So, it is enough to prove that $\Psi(L_f(G))$ is a semi-linear subset of $\mathbb{N}^k$ for every grammar G, because a finite union of semi-linear subsets of $\mathbb{N}^k$ is a semi-linear subset. Now, put

$$B = \{T \in T_f(G) \mid \text{No path from the root to a leaf has more than } |N| \text{ occurrences of a non-terminal}\}$$

Clearly, we see that B is a *finite* set of trees, because the height of any tree in B is bounded above by $|N|^2$. Now put

$$P = \{\text{all pumps in } G \text{ in which no path from the root to a leaf has more than } |N| \text{ occurrences of any non-terminal.}\}$$

Again, P is a finite set because the height of any tree in P is bounded above by $|N|^2$. Now, we will define a family of trees $\mathcal{T}'_f$ which is closed under inserting pumps in P as follows: Put

$$\mathcal{T}'_0 := B$$

Then, for any $i \in \mathbb{N}$, inductively define

$$\mathcal{T}'_i := \mathcal{T}'_{i-1} \cup \text{pump}(\mathcal{T}'_{i-1})$$

where $\text{pump}(\mathcal{T}'_{i-1})$ is defined for every $i \in \mathbb{N}$ as follows: for any tree in $\mathcal{T}'_{i-1}$, pick an internal node s, and pick any pump in P with root equal to the label of s, and insert the pump at the internal node. The collection of all trees that can be obtained in this way is precisely $\text{pump}(\mathcal{T}'_{i-1})$. Then, we define

$$\mathcal{T}'_f := \bigcup_{n \geq 0} \mathcal{T}'_n$$

so that $\mathcal{T}'_f$ is the *smallest* set containing B that is closed under inserting pumps at internal nodes.
We now claim that

$$(\star) \qquad \Psi(\mathcal{T}'_f) = \Psi(B) + \langle \Psi(P) \rangle$$

where

$$\Psi(B) + \langle \Psi(P) \rangle := \bigcup_{T \in B} \Psi(T) + \langle \Psi(P) \rangle$$

(and remember that $\Psi(P)$ is a finite set, since P is finite). Proof of $(\star)$ to be completed, but it is not difficult. So, $(\star)$ shows that $\Psi(\mathcal{T}'_f)$ is a semi-linear subset of $\mathbb{N}^k$, since B and P are both finite sets.
Now we easily see that $\mathcal{T}'_f \subseteq \mathcal{T}_f$. So to complete the proof, we will show that for any tree $T \in \mathcal{T}_f$ there is a tree $T' \in \mathcal{T}'_f$ such that $\Psi(T) = \Psi(T')$, and it will follow that

$$\Psi(\mathcal{T}'_f) = \Psi(\mathcal{T}_f)$$

which will complete the proof. We will prove this by contradiction. So suppose there is some $T \in \mathcal{T}_f$ such that $\Psi(T) \neq \Psi(T')$ for any $T' \in \mathcal{T}'_f$, and take the *minimal* such tree. Observe that $T \in B$ is not possible, since $B \subseteq \mathcal{T}'_f$. Hence T has a path from the root to a leaf such that some non-terminal occurs atleast $|N| + 1$ times on that path. Now, look at the *smallest* subtree of T which also contains some non-terminal which occurs atleast $|N| + 1$ times on a path from the root to a leaf in the subtree. Suppose X is the root of this subtree. So this subtree will contain the non-terminal X exactly $|N| + 1$ times on a path from the root to a leaf, and every other non-terminal Y will appear no more than $|N|$ times on a path from the root to a leaf in this subtree.
Now look at this subtree as a sequence of $|N|$ pumps, say $p_1, ..., p_{|N|}$, and the root of each pump is X. Notice that each of these pumps belong to P, because each of these pumps contains atmost $|N|$ X's on a path from the root to a leaf, and every other non-terminal appears atmost $|N|$ times on a path from the root to a leaf in the pump. Also, notice that removing any of these pumps from T will

still give us a *smaller* derivation tree that contains the non-terminal X (because deleting any pump p_i will *not* remove the last occurrence of X along the path in the subtree which contained $|N|+1$ X's). Now here is the key point: because there are $|N|$ pumps and $|N|-1$ non-terminals other than X, there is *atleast* one pump p_i whose deletion will *not* delete the $|N|-1$ non-terminals other than X (convince yourself that this is true), i.e there is some pump p_i such that $T - p_i \in \mathcal{T}_f$. Since $T - p_i$ is *smaller* than T, by the *minimality* of T it follows that

$$\Psi(T - p_i) = \Psi(T')$$

for some $T' \in \mathcal{T}'_f$. Since $\mathcal{T}'_f$ was closed under inserting pumps at internal nodes, it follows that

$$\Psi(T) = \Psi(T' \oplus p_i)$$

where $T' \oplus p_i$ denoted inserting the pump p_i at an internal node of T', and we know that $T' \oplus p_i \in \mathcal{T}'_f$. However, this contradicts our assumption about T, and completes the proof. ■

4. Turing Machines and Computability

In the final part of this course, we will be dealing with the most powerful computational tool among the ones which we will see, which are called *Turing Machines*. First, let me describe the informal working of a Turing Machine. We have an infinite one-dimensional tape consisting of cells on which symbols are written. This can be thought of as a very general version of a PDA. There is a pointer, which points at the cell which the tape is currently at. There are states as in usual automatons, and there are transitions which can change the state of the machine, the symbol of the current cell and which can move the pointer left or right. We now make this formal.

Definition 4.1. A *Deterministic Turing Machine* (TM) is a 9-tuple

$$M = (Q, \Sigma, \Gamma, \delta, s, \vdash, \sqcup, t, r)$$

where the symbols have the following meanings.

(1) Q is a (finite) set of *states* of M.
(2) Σ is a (finite) *input alphabet*.
(3) Γ is a (finite) set called the *tape alphabet*. We have $\Sigma \subseteq \Gamma$, and Γ contains two special symbols: $\vdash$, which is called the *left-end marker* and $\sqcup$, which is called the *blank* symbol. Informally, the leftmost cell of the infinite tape will always contain the symbol $\vdash$. Note that Γ may contain other symbols apart from these, i.e $\Sigma \cup \{\vdash, \sqcup\} \subseteq \Gamma$.
(4) s is the *starting state* of M.
(5) t is the *accepting state* of M.
(6) r is the *rejecting state* of M.
(7) δ is a (finite) set of *transitions* in M, which we will now describe. δ is a function $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$. So, any transition is of the form $(q, X) \to (q', Y, Z)$, where $q, q' \in Q$, $X, Y \in \Gamma$ and $Z \in \{L, R\}$, and this transition simply means that the machine M goes from state q to q', the symbol at the current cell of the tape is changed from X to Y, and the head of the tape either moves left or right depending upon whether Z is L or R. Since we are dealing with *deterministic machines* first, δ is actually a function and not just a relation.

Moreover, we also require that the left-most cell of the tape always contains the symbol $\vdash$, and that the head of the tape never goes to the left of the left-most cell. In other words, for *every* state $q \in Q$, if $(q, \vdash) \to (p, c, X)$ is a transition, then $c = \vdash$ and $X = R$, and this can be written as

$$\delta(q, \vdash) = (p, \vdash, R)$$

Definition 4.2. A *configuration* of a Turing Machine M is an element of

$$Q \times \vdash \Gamma^* (\sqcup)^\omega \times \mathbb{N}$$

and we now explain this. Here ω is the smallest infinite ordinal, which is $\mathbb{N}$, so that $(\sqcup)^\omega$ is an infinite string of $\sqcup$'s (infinitely many blank symbols), and $\mathbb{N}$ is the set of natural numbers including 0. A configuration $(q, \vdash\ y(\sqcup)^\omega, n)$ means that the current state of M is q, the infinite string $\vdash\ y(\sqcup)^\omega$ is written on the tape (where $y \in \Gamma^*$), and the current tape head is pointing at the n^{th} cell. For a machine M, the *starting configuration* on input $x \in \Sigma^*$ is $(s, \vdash\ x(\sqcup)^\omega, 0)$, which means that the head is pointing at the left-most cell.
Alternative Notation. A configuration (q, σ, i) is also represented by the string $xq\sigma'$, where $\sigma = x\sigma'$ and $|x| = i - 1$. So, the notation $xq\sigma'$ means that the head points at the $|x| + 1^{\text{th}}$ cell, the current state is q and the tape reads $x\sigma'$.

Definition 4.3. Let M be a Turing Machine. We define the *language* of M as

$$L(M) = \{w \in \Sigma^* \mid s \vdash w \xrightarrow{*} \vdash utv \text{ for some } u, v \in \Gamma^*\}$$

In the above definition, recall that s is the starting state of M, and t is the accepting state. Any language accepted by a Turing Machine is said to be *recursively enumerable* (RE or REC_ENUM).

Remark 4.0.1. Just like in DPDAs, even if we have a deterministic Turing Machine, there is no guarantee that the machine processes all of the input, i.e there can be words which have non-terminating runs which include things like infinite loops. So, there is no guarantee that if M is a Turing Machine and $w \notin L(M)$, the word w ends up at the reject state r. However, this doesn't prevent us from defining the language of a Turing Machine

Keeping the above remark in mind, we have the following definition.

Definition 4.4. If M is a Turing Machine which processes the entire input, i.e every word has a terminating run is called a *Total Turing Machine* (TTM). Informally, there is no word on which the TM reaches infinite loops. A language accepted by some TTM is said to be a *recursive language* (REC). It is clear that any TTM is a TM as well, i.e REC $\subseteq$ RE.

Remark 4.0.2. The difference between a TTM and a TM is precisely their working on words which are *not* accepted. In a TM, a word which is not accepted may have an infinite run, i.e the run for the word may never terminate. However, in a TTM, every word is guaranteed to have a finite run, and hence in a TTM any word that is *not* accepted will surely end up at the rejecting state.

Example 4.1. Let us work with the language $L = \{ww \mid w \in \Sigma^*\}$, which we know is *not* context free (which can be easily shown using the pumping lemma), where $\Sigma = \{a, b\}$. Complete this example.

Definition 4.5. Let L be any language, or equivalently let L be any *decision problem*. Then L is said to be *decidable* if some TTM recognizes L, and it is said to be *semi-decidable* if some TM recognizes L.

Remark 4.0.3. Before continuing, I will mention the power of Turing Machines. Roughly, any algorithm that can be implemented in Python can be implemented via a Turing Machine as well. So in some sense, the model of Turing Machines is the most powerful computational model that we know.

Theorem 4.1. *There exist languages which are not RE.*

Proof. We use a cardinality argument to prove this. Suppose Σ is any finite alphabet, and hence Σ^* is an infinite set. The set of all languages over Σ^* is just $\mathscr{P}(\Sigma^*)$, which is uncountable by Cantor's Theorem. However, it is easy to see that the set of all Turing Machines over Σ is a countable set. Hence, there are languages which are not recursively enumerable. ■

Remark 4.1.1. Another interesting an perhaps easier way of seeing why the set of all TMs is countable is by using the fact that TMs can be *encoded* using a suitable encoding scheme over $\{0, 1\}$. We will see this in the section on **Universal Turing Machines**.

Example 4.2. Consider the language $L = \{a^n \# a^m \mid n^2 = m^3\}$. We will show that this language is decidable by exhibiting a TTM for it. First, note that it is very easily seen (by using the Pumping Lemma) that this language is neither regular nor context free. Complete this example.

4.1. **Non-Determinism.** As usual, there is a notion of *non-determinism* in Turing Machines as well. Let us now explore this.

Definition 4.6. A *non-deterministic* TM M is a tuple

$$M = (Q, \Sigma, \Gamma, \delta, s, \sqcup, \vdash, t, r)$$

where all the symbols have the same meaning, except the set of transitions δ, which is a just a *relation* instead of being a function, i.e $\delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma \times \{L, R\})$. Given any non-deterministic TM M, its *language* $L(M)$ is defined as all those words which have *atleast one* accepting run, i.e

$$L(M) := \{w \in \Sigma^* \mid \text{There is a run } s \vdash w \xrightarrow{*} \vdash utv \text{ for some } u, v \in \Gamma^*\}$$

Theorem 4.2. *Every non-deterministic TM is equivalent to a deterministic TM. Hence, non-determinism does not add any power to TMs. Similarly, every non-deterministic TTM is equivalent to a deterministic TTM.*

Proof. The idea is simple as always. Suppose we are given an input word w and a non-deterministic TM M. The initial configuration of the TM M will be $s \vdash w$. Now, because M is non-deterministic, the computation can take several different paths. So, all of the possible paths can be represented via a *configuration tree*. A *configuration tree* is a rooted tree in which the root of the tree is the initial configuration of M, and every possible branch corresponds to the possible paths the TM can take (note that if M was deterministic, the configuration tree would be a path). Now, we make a deterministic TM M' that is equivalent to M as follows. M' will put all of the possible configurations in the configuration tree on its tape, in a *breadth first* fashion (BFS). Moreover, all the configurations will be separated by a special symbol, say #. So, M' begins by writing the

initial configuration $s \vdash w\#$ on its tape. Now suppose this configuration had 3 possible transitions, leading to configurations c_1, c_2, c_3 where $c_1, c_2, c_3 \in (Q \cup \Gamma)^*$. Then, the TM M' will write each of these configurations on its tape one by one, separated by a #, so the tape of the TM M' will look like

$$s \vdash w\#c_1\#c_2\#c_3$$

Then, the TM M' will look at the configuration c_1, and will append all possible paths that it could take from c_1 at the end of its tape. This is the breadth first fashion procedure.

Finally, a word is accepted if and only if one branch of the configuration tree *halts* and ends up at the accepting state t, i.e a word is accepted if and only if the tape ever sees the symbol t. Similarly, if M was a TTM, M' would be a TTM as well, as any word that will be rejected by M will have a halting path in the configuration tree that ends up at the rejecting state r. This completes the proof. ■

Corollary 4.2.1. *A language is decidable if and only if some non-deterministic TTM accepts it.*

Theorem 4.3. *If L is RE and L^c is RE, then L is REC.*

Proof. Let M and M' be deterministic TMs for L and L^c respectively. We make a TTM M'' as follows. Let c_0 and c_0' be the initial configurations of M and M' respectively on some input. M' writes the following on its tape.

$$c_0\#c_0'$$

i.e it writes the configurations of both M and M' on its tape separated by a #. Then suppose c_1 and c_1' are the configurations of M and M' after one step. Then M'' writes these on its tape, i.e the tape becomes

$$c_0\#c_0'\#c_1\#c_1'$$

So M'' does its computation by successively writing the configurations of M and M' on its tape. M'' accepts if it ever sees a t on its tape, and rejects if it ever sees an r. Hence, M'' is a TTM, and so L is REC. ■

Remark 4.3.1. It turns out that this is actually an if and only if statement, because as we shall see later, the complement of a REC language is also REC.

Definition 4.7. A language L is said to be co-RE if L^c is RE.

4.2. **Universal Turing Machine.** In this section, we will see our first example of a semi-decidable problem.

Suppose we are given a Turing Machine M and an input x over the input alphabet on M. The *membership problem* is the problem of determining whether $x \in L(M)$. We will state this problem as a particular language, and we will show that this language is RE but not REC. This will show that the membership problem is semi-decidable, but not decidable.

Suppose we are given some *encoding* of a Turing Machine M over the alphabet $\{0, 1\}$ (TMs can be encoded over $\{0, 1\}$. Check some online source or Kozen's book to see how). Now, suppose x is any input over the input alphabet of M which is also encoded over $\{0, 1\}$. So, consider the language

$$L = \{M\#x \mid x \in L(M)\}$$

over the alphabet $\{0, 1, \#\}$. We will show that L is a semi-decidable language.

Theorem 4.4. *L is a semi-decidable language.*

Proof. We make a TM U that will accept the language L. I will give a high level description of U, since a detailed formal description is very hard to give. U will be a *multitape* machine with three tapes (to see why multitape TMs are equivalent to single tape TMs, see PSET-12).

Given an input $M\#x$, U first checks whether M is a valid encoding of a TM and whether x is a valid encoding of a string of M's input alphabet. If either of these is invalid, then U rejects.

If the encodings are valid, then U begins by writing the encoding (description) of M on its first tape. The second tape of U will be used to blindly simulate M, so U then writes the input string x on its second tape. The final tape will contain the information about the current state M is in and the current position of the head of M. Then, U just blindly simulates M on the input x on the second tape, where U checks the first tape for which transition to use and it keeps updating the third tape containing the current state of M and the current head pointer. U accepts if M accepts, rejects if M rejects and goes in a loop if M goes in a loop. So, it follows that U is not a TTM, because there are TMs M that go in loops. It then follows that $L = L(U)$ and L is semi-decidable. ■

Note that the possibility that L is decidable still remains. We will explore this further in the next section.

4.3. **Diagonalisation.**

In this section, we will prove that the languages

$$\text{HP} := \{M\#x \mid M \text{ halts on } x\}$$
$$\text{MP} := \{M\#x \mid x \in L(M)\}$$

are *not decidable*. In particular, the language in **Theorem** 4.4 is not decidable and only semi-decidable. HP stands for *halting problem* and MP stands for *membership problem*. We will use the so called *Cantor's Diagonal Argument* which is very famous in set theory.

Theorem 4.5 (**Undecidability of HP**). *The halting problem* (HP) *is undecidable.*

Proof. First, we fix an encoding scheme over $\{0,1\}$ of TMs. For the sake of contradiction, suppose the halting problem is decidable, i.e there is some TTM K accepting the language

$$\{M\#x \mid M \text{ halts on } x\}$$

For any $x \in \{0,1\}^*$, let M_x be the TM over the input alphabet $\{0,1\}$ whose encoding is given by x (if x is *not* a valid encoding of a TM over the input alphabet $\{0,1\}$, then we let M_x to be *any* (but fixed) TM over the input alphabet $\{0,1\}$). Since $\{0,1\}^*$ is a countable set, we get an enumeration

$$M_\epsilon, M_0, M_1, M_{00}, M_{01}, \ldots.$$

of all possible TMs over the input alphabet $\{0,1\}$. Now, consider the following infinite matrix.

	ϵ	0	1	00	01	...
M_ϵ	Y	N	N	Y	N	...
M_0	Y	N	N	N	Y	...
M_1	N	N	N	N	N	...
M_{00}	N	N	Y	Y	N	...
M_{01}	N	N	N	Y	Y	...
....						

In the above matrix, we write a Y if the TM halts on the given word and we write an N if it loops on the given word.

Now, consider a TM N on the input alphabet $\{0,1\}$ as follows. For $x \in \{0,1\}^*$, N writes $M_x\#x$ on its tape, where M_x is the encoding of the TM as described earlier in the proof. Then, N simulates the TM K on the input $M_x\#x$. Note that by the definition of K, K halts and accepts if M_x halts on x, and K halts and rejects if M_x loops on x. Now, we make N so that N accepts if K rejects and N goes to a trivial loop if K accepts (in simple words, N is essentially complementing the diagonal of the above matrix).

So, N's behavior is different from every TM in the matrix above. But this is a contradiction, because the enumeration was supposed to contain *all* TMs over the input alphabet $\{0,1\}$. This completes the proof. ■

Theorem 4.6 (Undecidability of MP)**.** *The membership problem* (MP) *of TMs is undecidable.*

Proof. We will show that MP is undecidable by *reducing* HP to it, i.e we show that if MP can be decided, then HP can also be decided, but that would contradict the **Undecidability of HP** 4.5.

For the sake of contradiction, suppose MP is decidable. So, there is a TTM M' which accepts the language

$$\{M\#x \mid x \in L(M)\}$$

Now we make a new TTM K for HP as follows.

(1) Given the input $M\#x$, *construct* a new TM N as follows (by *construct*, we mean write down encoding of a machine N). The machine N, on any input y, simulates M on y and *accepts* if M accepts or rejects, i.e if M *halts* on y. This can easily be done by adding a new accept state to N and having a transition from the accept and reject states of M to this new state.
(2) After constructing the machine N, run M' on the input $N\#x$. Accept if M' accepts and reject if M' rejects.

Then, it is easily seen that K is a TTM for HP. But this is clearly a contradiction, and hence MP must be undecidable. ■

4.4. **More Undecidable Problems.** Let us begin with *ϵ-membership*. Given the encoding M of some TM, we need to determine whether $\epsilon \in L(M)$. As it turns out, this problem is undecidable.

Theorem 4.7. *ϵ-membership is undecidable.*

Proof. We will *reduce* the ϵ-membership problem to HP, and hence it will follow that ϵ-membership is undecidable. Needs to be completed. Look at PSET-13 **Problem 2.** the solution is very similar! ■

4.5. **Reductions.** In this section we will give a concrete meaning to the notion of a *reduction*.

Definition 4.8. Let Σ, Δ be two alphabets, and let $A \subseteq \Sigma^*$ and $B \subseteq \Delta^*$ be two languages. A *reduction* of A to B is a function $\sigma : \Sigma^* \to \Delta^*$ such that for all $x \in \Sigma^*$,

$$x \in A \iff \sigma(x) \in B$$

So in simple words, words in A go to words in B, and words not in A go to words not in B. Moreover, the function σ must be *total* and *effectively computable*. This means that σ must be computable by a TTM that on any input x halts with the word $\sigma(x)$ on its tape. In this case, we say that A is *reducible* to B, which is written as $A \leq B$ (equivalently one says that A is the *easier* problem and B is the *harder* problem).

Theorem 4.8. *The following hold.*

(1) *If $A \leq B$ and B is* RE*, then A is also* RE*. Equivalently, if A is not* RE*, then B is also not* RE*.*
(2) *If $A \leq B$ and B is* REC*, then A is also* REC*. Equivalently, if A is not* REC*, then B is also not* REC*.*

Proof. The proofs are rather straightforward. Since σ is a computable function, given a TM for B, we can also make it to work for A as follows: given a word $x \in \Sigma^*$, first compute $\sigma(x)$, and then run B on the word $\sigma(x)$. This is where the property

$$x \in A \iff \sigma(x) \in B$$

helps us. This completes the proof (a handwavy proof but it's not a very difficult proof anyway). ■

Example 4.3. Here we consider the membership problem MP and the ϵ-membership problem ϵ-MP. Let us call the latter EMP. First, we show that

$$\text{EMP} \leq \text{MP}$$

The computable function σ just maps a word M to the word $M\#\epsilon$, where M is the encoding of some TM, i.e

$$\sigma(M) = M\#\epsilon$$

It is easy to see that σ satisfies the required properties; $M \in$ EMP if and only if $M\#\epsilon \in$ MP.

Next, we will show that

$$\text{MP} \leq \text{EMP}$$

which is less obvious. This is very similar to the proof of **Theorem** 4.7. Our computable function σ is as follows: suppose the word is $M\#w$, where M is the encoding of a TM and w is a word over $M's$ input alphabet. Then $\sigma(M\#w) = N_{M,w}$, where $N_{M,w}$ is the TM which behaves as follows: on input y, $N_{M,w}$ ignores y, and simulates M on the word w. $N_{M,w}$ accepts if M accepts w and $N_{M,w}$ rejects if M rejects w. So we have

$$M\#w \in \text{MP} \iff N_{M,w} \in \text{EMP}$$

If the word $M\#w$ is such that M is *not* a valid encoding of a TM or if w is *not* a valid word over M's input alphabet, then we let $\sigma(M\#w)$ to be anything *fixed* that is not in EMP.

So, it follows that EMP is RE but it is undecidable, since MP is RE and undecidable. Alternatively, we can say that EMP is RE but not co-RE, since MP is RE and not co-RE.

Example 4.4. Consider the problem of *universality*, i.e given the encoding M of a TM, is $L(M) = \Sigma^*$? Let us call this problem UNIV. We will first show the relation

$$\text{MEM} \leq \text{UNIV}$$

This is done exactly as in **Example** 4.3. For a word $M\#w$, let $\sigma(M\#w) = N_{M,w}$, where $N_{M,w}$ is as in **Example** 4.3. We see that

$$M\#w \in \text{MP} \iff N_{M,w} \in \text{UNIV}$$

and hence this shows that UNIV is not co-RE, since MEM is not and also UNIV is also undecidable, since MEM is undecidable. The only question remaining is whether UNIV is RE. We will see that the answer is a *no*, and we will prove it as a theorem.

Theorem 4.9. UNIV *is not RE.*

Proof. We will prove this by reduction from $\overline{\text{MEM}}$ to UNIV, and that will prove the claim because we know that $\overline{\text{MEM}}$ is not RE (here the bar denotes the complement of MEM). The idea involved here is neat.

Suppose we are given the word $M\#w$, where M is the encoding of a TM and w is a word (i.e the encoding of a word) over M's input alphabet. Our computable function σ will be as follows. Let $\sigma(M\#w) = N_{M,w}$, where $N_{M,w}$ is the encoding of a TM which does the following: on input y, $N_{M,w}$ simulates the TM M on the word w for $|y|$ steps. If M accepts w within $|y|$ steps, then $N_{M,w}$ rejects. If M rejects w within $|y|$ steps, then $N_{M,w}$ accepts. Finally, if M does not halt on w within $|y|$ steps, then $N_{M,w}$ again accepts (thus σ is indeed a computable function because a TTM can be used to find the required encoding $N_{M,w}$. Again, as usual, M and w will be hard-wired in the encoding $N_{M,w}$). Next, if $M\#w$ is not a valid encoding, then we simply let $\sigma(M\#w)$ to be any *fixed* encoding of a TM that is in UNIV.

Observe that for valid encodings $M\#w$,

$$L(N_{M,w}) = \begin{cases} \Sigma^* & \text{, if } M \text{ rejects/does not halt on} \\ \{y \in \Sigma^* \mid M \text{ accepts } w \text{ in more than } |y| \text{ steps}\} & \text{, if } M \text{ accepts } w \end{cases}$$

So, we see that

$$M\#w \in \overline{\text{MEM}} \iff N_{M,w} = \sigma(M\#w) \in \text{UNIV}$$

This shows that $\overline{\text{MEM}} \leq$ UNIV. So, this proves that UNIV is *not* RE. ∎

Remark 4.9.1. This gives us our first example of a language that is neither RE nor co-RE! FINITENESS is another such language; look in PSET-14.

4.6. **Rice's Theorem.** In this section, we will be proving a very powerful theorem about the decidability of some problems related to some property of RE languages.

Definition 4.9. Let Σ be a fixed finite alphabet. A *property* of RE sets over Σ is a map

$$P : \{\text{RE subsets of } \Sigma^*\} \to \{T, F\}$$

where T and F represent *true* and *false* respectively. Given a property P of RE sets, we ask this problem: is the property P decidable? So, we are essentially asking that given a TM M, can it be decided what $P(L(M))$ is?

Remark 4.9.2. The above definition should be taken care of. Observe that P is a property of $L(M)$, i.e the language accepted by M and not a property of M itself!

Definition 4.10. A property P of RE sets is said to be *non-trivial* if there are RE sets $L_1, L_2 \subseteq \Sigma^*$ such that $P(L_1) \neq P(L_2)$.

Theorem 4.10 (Rice's Theorem I). *Every non-trivial property of RE sets is undecidable.*

Proof. Let P be a non-trivial property of RE sets. So, there are TMs M_1, M_2 such that $P(L(M_1)) = T$ and $P(L(M_2)) = F$. Now we know that ϕ is an RE set, and hence either $P(\phi) = T$ or $P(\phi) = F$. Without loss of generality, we assume that $P(\phi) = F$, and the proof when $P(\phi) = T$ is analogous. Also, we will be using the machine M_1 in our procedure below.

We will now reduce the halting problem HP to the set $\{M \mid P(L(M)) = T\}$. Suppose we are given the word $M\#w$. First, suppose M is a valid encoding of a TM and suppose w is a valid encoding of a word over M's input alphabet. Our computable function σ will be as follows; we let $\sigma(M\#w) = N_{M,w}$, where $N_{M,w}$ is a TM which does the following.

(1) On an input y, $N_{M,w}$ first simulates the machine M on w.
(2) If M halts on w, then $N_{M,w}$ simply simulates the machine M_1 on the word y.

Now, if $M\#w$ is not a valid encoding (i.e either M is not a valid TM or w is not a valid word over M's input), we let $\sigma(M\#w) = N_{M,w}$ to be any *fixed* TM that *does not* lie in the set $\{M \mid P(L(M)) = T\}$. For example, we might as well choose $N_{M,w}$ to be a TM accepting ϕ.

Now we can show that σ is indeed a valid reduction. Suppose $M\#w$ is a valid encoding. Then, observe that $L(N_{M,w}) = \phi$ if M *does not* halt on w, and $L(N_{M,w}) = L(M_1)$ if M halts on w. So this means that

$$M\#w \in \text{HP} \iff N_{M,w} \in \{M \mid P(L(M)) = T\}$$

and this is precisely why we chose M_1. On the other hand, if $P(\phi) = T$ was true, we would have chosen M_2 instead of M_1.

So, we have reduced the HP to the set $\{M \mid P(L(M)) = T\}$. Since HP is undecidable, it follows that the problem of determining the property P is also undecidable. This completes the proof. ∎

Definition 4.11. Let Σ be a fixed finite alphabet. A property P of RE sets is said to be *monotone* if given any RE sets A, B with $A \subset B$, it is true that $P(A) \leq P(B)$, where the order being followed is $F \leq T$. In simple words, a *monotone* property is one in which if an RE set satisfies the property, all its RE supersets also satisfy the property.

Theorem 4.11 (**Rice's Theorem II**). *Any non-monotone property of RE sets is not semidecidable.*

Proof. Let P be any non-monotone property of RE sets. We will reduce the complement of the halting problem $\overline{\mathsf{HP}}$ to the set $\{M \mid P(M) = T\} := T_P$. This will do our job since we aleady know that $\overline{\mathsf{HP}}$ is not RE.

Since P is non-monotone, there are TMs M_0, M_1 with $L(M_0) \subseteq L(M_1)$ such that $P(L(M_0)) = T$ and $P(L(M_1)) = F$. This information will be useful to us.

Since we want to reduce $\overline{\mathsf{HP}}$ to T_P, given an input $M\#w$, we want to make a TM $N_{M,w}$ such that $N_{M,w} \in T_P$ if and only if M *does not* halt on w. So, suppose $M\#w$ is a valid encoding, and we construct an encoding $N_{M,w}$ of a TM as follows.

(1) $N_{M,w}$ will have three tapes.
(2) On input y, $N_{M,w}$ will simulate M_0 on y on its first tape; it will simulate M_1 on y on its second tape, and on its third tape $N_{M,w}$ will simulate M on w. Moreover, $N_{M,w}$ will do all these simulations in *parallel*; i.e each tape operation will be performed one at a time for each of the three tapes.
(3) $N_{M,w}$ will accept the input y if either of the following occurs:
    (a) M_0 accepts y.
    (b) M *halts on* w and M_1 accepts y.

Now, either M will halt on w or it won't. If M *does not* halt on w, then we see that $L(N_{M,w}) = L(M_0)$. If M *halts* on w, we see that $L(N_{M,w}) = L(M_0) \cup L(M_1) = L(M_1)$. So we have shown that

$$M\#w \in \overline{\mathsf{HP}} \iff N_{M,w} \in T_P$$

So, we let $\sigma(M\#w) = N_{M,w}$ to be our computable function. So, this shows that problem of determining whether $M \in T_P$ is undecidable, since $\overline{\mathsf{HP}}$ is undecidable. ∎

4.7. **Post's Correspondence Problem.**

Suppose we are given n pairs

$$(u_1, v_1), (u_2, v_2), ..., (u_n, v_n)$$

where each pair is a pair of words. *Post's Correspondence Problem* or PCP asks whether there are finitely many indices $i_1, i_2, ..., i_k \in \{1, ..., n\}$ such that

$$u_{i_1} u_{i_2} ... u_{i_k} = v_{i_1} v_{i_2} ... v_{i_k}$$

We can state PCP in a different way. Given finite input alphabets Σ, Γ and two homomorphisms $h : \Sigma \to \Gamma^*$ and $g : \Sigma \to \Gamma^*$, we want to see if there is some $w \in \Sigma^* \setminus \{\epsilon\}$ such that $h(w) = g(w)$. To see why this is equivalent to PCP, we can let $\Sigma = \{(u_1, v_1), ..., (u_n, v_n)\}$ and we can let the homomorphisms h, g be such that $h(u_i, v_i) = u_i$ and $g(u_i, v_i) = v_i$. We will now state a theorem, but we will not fully prove it here.

Theorem 4.12. PCP *is RE and undecidable.*

Proof. The fact that PCP is RE is easy to see. Just enumerate every word w over Σ one by one in length-lexicographic order, and check whether $h(w) = g(w)$. We will not prove undecidability here. A good proof is given in Sipser's book. ∎

4.8. **Minsky Machines.** In this section we will see yet another computation model.

Definition 4.12. A *Minksy Machine* is a machine with finitely many states and two *counters* A and B, each counter having a value in the set $\mathbb{N} \cup \{0\}$. There are three types of transitions: A++, A-- and IFZ(A) and these are defined analogously for B. A transition of the form

$$q \xrightarrow{A++} q'$$

goes from state q to q', incrementing the counter A by 1. A transition of the form

$$q \xrightarrow{A--} q'$$

goes from q to q', decrementing the counter A by 1. Finally, a transition of the form

$$q \xrightarrow{\text{IFZ(A)}} q'$$

goes from state q to q' *only if* the current value of A is *zero* (IFZ stands for if zero). In general, we can modify this definition to hold any number of counters. A machine of that type is called a k*-counter machine*. So, a Minksy Machine is just a 2-counter machine.

Remark 4.12.1. Even though it is implicit in the definition, it is still important to note that if the current value of counter A is 0, then a transition of the form A-- cannot be taken. A similar thing holds for the counter B.

Definition 4.13. A *configuration* of a Minsky Machine M is a tuple of the form (q, m, n), where q is the current state and $m, n \in \mathbb{N} \cup \{0\}$ are the current values of counters A and B respectively. Similarly we can define configurations for k-counter machines.

4.9. **Control State Reachability Problem.** Let us now see a problem related to Minksy Machines, called the *control state reachability problem* (CSRP). The input to the problem will be a Minsky Machine M and a target state t. The question is whether there is a run in M from the starting state s to the target state t.

Proposition 4.13. *The control state reachability problem for k-counter machines is RE for any k.*

Proof. The idea is simple; we just do a BFS on the configuration graph, i.e the graph in which the nodes are configurations and there are edges between these nodes if one configuration is reachable from another. Ofcourse the graph will be infinite, but we don't need to build the whole graph. We just need to start our BFS from the root configuration, and go step by step. This proves that the problem is RE. ■

Proposition 4.14. *The control state reachability problem for 1-counter machines is decidable.*

Proof. Observe that any 1-counter machine can be simulated by a PDA; a decrement operation is simulated by popping a stack, an increment operation is simulated by pushing onto the stack, and the IFZ operation is just checking whether the stack is empty. So, the control state reachability problem for 1-counter machines is the same as that for PDAs. In the PDA, we can make the target state

final, and then just check whether the language of the PDA is non-empty. This is clearly a decidable problem, by converting the PDA to a CFG and then checking non-emptiness for the CFG. ■

Later we will show that the CSRP for 2-counter machines/Minsky Machines is undecidable.

4.10. **2-Stack PDAs.** A 2-*stack* PDA is a PDA with two stacks, i.e each configuration is of the form (q, γ, β), where $\gamma \in \Gamma_1^*$ and $\beta \in \Gamma_2^*$, where Γ_1 and Γ_2 are the stack alphabets of the two stacks respectively. We can state a CSRP for 2-stack PDAs as well; the input to this problem is a pair (A, t) where A is a 2-stack PDA and t is a target state. The question is whether t is reachable from the start state of the PDA. We will show that this problem can be reduced to the *emptiness problem* for TMs by showing that every TM can be simulated by a 2-stack PDA; thus is will follow that CSRP for 2-stack PDAs is undecidable.

Theorem 4.15. *Every TM can be simulated by a 2-stack. Thus, the control state reachability problem* CSRP *for 2-stack PDAs is undecidable.*

Proof. The idea of simulating a TM using a 2-stack PDA is simple. Consider the tape of the TM. Everything towards the left of the tape head will be stored in one stack, and everything to the right of the tape head will be stored in the second stack. If the head of the tape moves left, then we will pop the top symbol of the first stack and push it to the top of the second; if the head of the tape moves right, then the top symbol of the second stack will be popped and pushed at the top of the first stack. Thus, every TM can be simulated by a 2-stack PDA (the details of the transitions are easy to fill in).

Next, consider the control state reachability problem for 2-stack PDAs. We will reduce the non-emptiness problem of TMs to this problem, and hence it will follow that this problem is undecidable. So, let a TM M be given. Construct a 2-stack PDA A_M out of M, and let t be the accept state of the PDA A_M. So our computable function σ is $\Sigma(M) = (A_M, t)$. It is then clear that

$$M \in \{T \mid T \text{ is a TM and } L(T) \neq \phi\} \iff (A_M, t) \in \text{CSRP for 2-stack PDAs}$$

and hence σ is a valid reduction. This completes the proof. ■

4.11. **Undecidability of 4-Counter Machines.** In this section, we will show that the CSRP for 4-counter machines is an undecidable problem. The idea will be to simulate a stack using two counters, and use the fact that CSRP for 2-Stack PDAs is undecidable.

So, let us show how to simulate a stack using two counters. Without loss of generality, we assume that our stack alphabet is $\{0, 1\}$. In this way, the stack content is a binary word with the least significant digit at the top of the stack. Our first counter A will hold this binary number. Note that pushing a 0 onto the stack is the same as doubling the number in A, pushing a 1 on the stack means doubling the number in A and adding a 1. This can be easily done as follows: decrement A until it is zero, and for every decrement step, increment B twice; this achieves the effect of doubling the number. Similarly, popping a 0 or a 1 from the stack is achieved by dividing by 2. So, it follows that a stack can be simulated using two counters. Hence, a 2-Stack PDA can be simulated using a 4-Counter Machine. Hence, the CSRP for 4-Counter Machines is undecidable.

4.12. **Undecidability of Minsky Machines and Godel Numbering.** In this section, we will show that a 2-Counter Machine (equivalently a Minsky Machine) can be used to simulate a 4-Counter Machine, which will show that a 2-Counter Machine can simulate a TM, because 4-Counter Automatons can simulate TMs by simulating 2-Stack PDAs. This will inturn show that the CSRP for Minsky Machines is *undecidable*. Suppose we are given a 4-Counter Machine, and suppose the current values of the four counters are (n_1, n_2, n_3, n_4). We will encode this tuple into a single value, namely the value $2^{n_1}3^{n_2}5^{n_3}7^{n_4}$. This kind of numbering is called *Godel Numbering*. In this way, incrementing/decrementing the first counter means multiplying/dividing by 2, and a similar situation holds for the other counters as well. By carrying out multiplication/division in a second counter, we can thus simulate 4-counters using 2-counters, and this completes the proof (This is not really a proof, rather the idea of a proof. Maybe I will make this section more detailed sometime).

4.13. **Universality of CFL.** Let G be a CFG, and we want to check whether $L(G) = \Sigma^*$ or not. Let

$$\text{UCFG} = \{G \mid G \text{ is a CFG and } L(G) = \Sigma^*\}$$

It is easy to see that $\overline{\text{UCFG}}$ is RE, because we can just enumerate every word of Σ^* and check membership using the CYK algorithm. So, we see that UCFG is co-RE. Now, we will show that UCFG is undecidable, and that will show that UCFG is *not* RE. To prove this, we will use the so called *computation histories* (Unfortunately, due to time constraints, I could not write this section in my own words. However, a very nice proof of this fact using computation histories is given in Kozen's book).