Chapter 5

DFA Minimisation

5.1 Minimum States in a DFA

So, far we have dealt with DFA, NFA without ϵ, NFA with ϵ and Regular Expressions. We have also seen that all of them are equivalent and inter-convertible and represent regular languages.
Consider the language which consists of all strings which are terminated by one. The regular expression for this will be: $(0+1)^*1$. Here is a 2-state DFA for the same language:

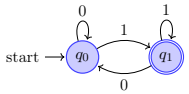


Figure 5.1: 2-state DFA for the language

The above DFA has two states q_0 and q_1. q_0 is reached when the last seen letter was 0 (also at the start). q_1 is reached when the last seen letter was 1 (also, it is an accepting state).
We can even construct a 4-state DFA for the same language. Here is an example:

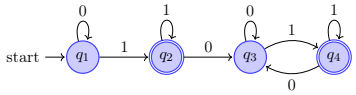


Figure 5.2: 4-state DFA for the language

The above DFA has four states q_1, q_2, q_3, q_4.

- q_1: Last letter 0 and no 1s so far
- q_2: Last letter 1 and no 101 seen so far
- q_3: Last letter 0 and more than one 1s seen so far

- q_4: Last letter 1

One can construct many more 4-state DFAs for the same language. Above is another example.

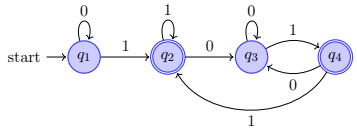


Figure 5.3: Another 4-state DFA for the same language

A natural question which comes to our mind is that can we construct another 2-state DFA for this language. One can find through trial and error, that this is not possible.
Is one state DFA possible for this language? Let us assume that this is possible. Two cases arise. If that state is accepting, then it will also accept ϵ, which is not possible. If that state is not accepting, then the language will be empty. We have arrived at a contradiction. Hence, one state DFA is not possible for this language.
Therefore, for this language, the minimum states in any DFA can be 2. We also observe that number of such 2-state DFAs is 1. So, we will now claim that for every language, the number of minimal DFAs is 1 and try to prove this. Before we prove this, we will introduce the notion of indistinguishability.

5.2 Indistinguishability

Two states of a DFA q_i and q_j are considered indistinguishable, if $\forall w \in \Sigma^*$, we start with q_i, process w and reach q_i' and we start with q_j, process w and reach q_j', then either $q_i' \in F$ and $q_j' \in F$ or $q_i' \notin F$ and $q_j' \notin F$, where F is the set of all final states of the DFA. So, we are basically changing the start states and checking whether we reach the same type of states or not through the same string.
This relation is denoted by $\equiv$. It has the following properties:

- It is **reflexive**. It is clear to see that every state is indistinguishable to itself, as it will reach a particular state on seeing w. (Due to the nature of a DFA)
- Also, it is clear to see that this relation is **symmetric**.
- This relation is also **transitive**. We can prove this by contradiction. Let us assume that $(q_i \equiv q_j) \wedge (q_j \equiv q_k)$ but $q_i \not\equiv q_k$. Then $\exists w$ such that $q_i' \in F$ and $q_k' \notin F$, where q_i' and q_k' are the states we reach from q_i and q_k respectively on seeing w. From the equivalence of q_i and q_j, we have $q_j' \in F$ but from the equivalence of q_j and q_k, we have $q_j' \notin F$, where q_j' is the state that we reach from q_j on seeing w. Hence, we have arrived at a contradiction. Therefore, this relation is transitive.
- A relation which is reflexive, symmetric and transitive, is **equivalent**. Thus the states of the DFA, on which this relation is defined can be partitioned into equivalence classes.

In the above example (Figure: 5.2), q_1 and q_3 belong to the same equivalence class, and q_2 and q_4 also belong to another equivalence class. Let us now try to construct a 2-state DFA from the above 4-state DFA example (Figure: 5.2). We will choose, one element each from both of the equivalence classes. Let us take q_1 from the first class and q_4 from the second class.

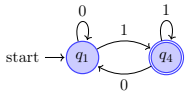
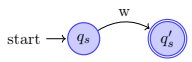


Figure 5.4: 2-state DFA constructed from the 4-state DFA

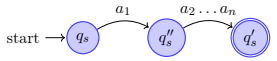
From q_1, if we see a 0, we land at q_1 itself. If we see a 1, we would have landed at q_2, but since q_2 and q_4 are equivalent, we replace q_2 by q_4. Similarly, from q_4, if we see a 1, we remain at q_4, but if we see a 0, we would have landed at q_3, but since q_1 and q_3 are equivalent, we replace q_3 by q_1. Also, q_2 and q_4 both were acceptable earlier, now q_4 is acceptable.
So, through these equivalence classes, we have minimized our 4-state DFA into a 2-state DFA, and also this 2-state DFA is structurally the same as the previous one (Figure: 7.10), thus again making us think that the claim that there is a single minimal DFA for every language might be correct.
Another interesting thing we observe is that an accepting state cannot be indistinguishable from a non-accepting state. We can take $w = \epsilon$, and observe that the states we reach from this pair of states do not satisfy the definition of indistinguishability relation. However, an initial state and a non-initial state can belong to the same equivalence class. (For example, above q_1 and q_3 belonged to the same class.)
Now, several important questions arise. How can we find the equivalence classes of this relation? When we will come to know that we cannot compress our DFA further (by compress, we mean reducing the number of states of the DFA)? How can we prove our claim that the minimal DFA will be unique?
We will try to answer all these questions subsequently. Let us start with the easiest one.

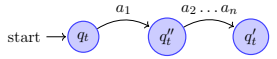
5.3 Equivalence classes of Indistinguishability relation

Now, we will develop an algorithm to find the equivalence classes of this relation.
Firstly, we should keep in mind our previous observation that an accepting state cannot be indistinguishable from a non-accepting state. That is $q_i \not\equiv q_j \ \forall q_i \in F$ and $q_j \in (Q \backslash F)$, where Q is the set of all states of the DFA.
Suppose, we find two states q_s and q_t which are distinguishable. Thus there exists a string w such that q_s leads to an accepting state but q_t leads to a non-accepting state.

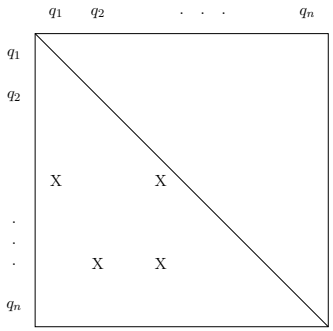


Now, w can be decomposed into $a_1 a_2 \dots a_n$, where $|w| = n$.





From, the above figures, one can observe that q''_s and q''_t are also distinguishable, where $w' = a_2 \dots a_n$ is the string which is making them distinguishable.
Thus, we can observe that for all states q_i and q_j such that $q_i \not\equiv q_j$ and for all $a \in \Sigma$, such that q_s on seeing a lands at q_i and q_t on seeing a lands at q_j then q_s will be distinguishable with q_t, where q_s and q_t are two states of the DFA.(We are basically extending $w' = a + w$.)
Therefore using a distinguishable pair, we have found another one. This is the basis of our algorithm.
We initialize our set with all the pairs, where one state belongs to the set of accepting states and other state does not belong to the set of non accepting states. And then through the above step, we keep on increasing the size of this set. (Note that this algorithm is not exponential, because there are only $\binom{n}{2}$ pairs possible, and we do need to check an already visited pair.)



We stop the process, when no more crosses can be inserted.
Now, it is quite natural for us to ask the question that whether our algorithm is correct or not i.e. can we still find a pair of distinguishable states, which are not detected even after our algorithm finishes?
Proof: Let us assume that there are two states q_s and q_t which are distinguishable but not recognized by our algorithm. By definition, there exists a string w such that q_s leads to an accepting state on seeing w and q_t reaches a non-accepting state.

Now w can be written as $a_1a_2 \dots a_n$, where $|w| = n$. And also assume that we reach q''_s and q''_t on seeing a_1 from q_s and q_t respectively. It is clear that, if our algorithm has not detected q_s and q_t as distinguishable, then it would not even have detected q''_s and q''_t as a distinguishable pair. (Because, if it would have done, then the next step would have been to make q_s and q_t as distinguishable.)

Now, we will inductively move forward our algorithm. Thus q'''_s and q'''_t would not also be detected as distinguishable after our algorithm finishes. But clearly, this is not possible, because our algorithm initializes the set of pairs of {accepting, non-accepting} states and then it's first step is to move backwards. So, it would have marked q'''_s and q'''_t as distinguishable in the first step itself.

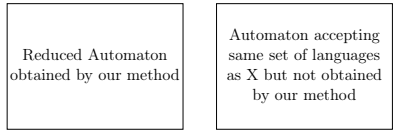
We have achieved contradiction. Therefore, we can safely conclude that our algorithm terminates and is also correct.
Now, once our algorithm detects all pairs of distinguishable states, we can choose equivalence classes of the indistinguishability relation. (Then, we can choose one representative from each class and move forward with our proof of existence of a unique minimal DFA.)
It is worth noting that distinguishability is not an equivalent relation. It is not even reflexive. It is also not transitive. But, it is indeed symmetric. And also, it proved out to be very useful for finding these equivalence classes.

5.4 Further Analysis

We have developed an algorithm which can shrink the size of a given DFA. But is that the best we can do? Can the size of the DFA be further reduced? Are there many DFAs which are minimal or just one unique? The following sections will try to answer these questions.
Note: We are assuming that there is no redundant nodes in final DFA obtained, i.e. those nodes which can not be reached from starting state by any string. If there is one such, we can always remove it to get an equivalent DFA.

5.4.1 Optimality of Acquired DFA

Let's say we have an automaton X, which is the reduced automaton obtained by picking one state from each of the equivalence classes of the set of states, and Y is an automaton obtained by some other method for the same language.



Automaton X Automaton Y

We wish to show that the number of states of Y is at least as many as X.
Let's define a new term "**Language of State**". Given a DFA A and a state s, we define L^A_s as the set of all finite strings which will end in any of the accepting state if we start from s. Readers should be able to see that when we say that two states are distinguishable, we mean that there Language is different. Similarly, when we say that the two states are indistinguishable, we are actually asserting that the language of the two states is same.

Let say we started with the automaton D. After termination the resultant automaton was A. Let S_A represent the set of distinct such languages of nodes of A. Since A has all states indistinguishable from every other state, the $|S_A|$ = no of states.
Claim: Given any DFA B, equivalent to A,

$$\forall L \in S_A(\exists S(L^B_S \equiv L)), \text{ where S represent a state of B}$$

Proof: If L $\in$S$_A$, then by definetion it must be the language of some node in A, say s. Since all the nodes of A are ir-redundant, so let's say string w is one such string through which we can reach this node starting from the starting state of A. Now run the same string w on the automaton B. Say we reach the state S. Then $L \equiv L^A_s \equiv L^B_S$, because if say string x is present in former and not in latter then, string $w.x$ will be an accepting string in A and not in B, and vice-versa. Thus $L^B_S \equiv L$.
For every language in S_A, we must have at least one node in B. Since one node has a specific language, this gives us a lower bound to number of nodes, $|S_A|$.
A achives the lower bound of states, hence it is the(?) minimum state DFA which represent the same regular language represented by D.

5.4.2 Uniqueness of Acquired DFA

So far we have discussed about proving the optimality of the automaton A, which is the output of our algorithm. We defined an important notion of "Language of State" and proved a very important result which can be summarized as follow **Given two DFAs A and B where both represent some particular regular language, then for any string *w*, the states reached in A and B by reading it will have same language,i.e.** $L^A_{s_A} \equiv L^B_{s_B}$. Now we will try to answer that if there are mutiple DFAs which are optimal or just one.
Let's consider two automatons, one is our DFA A which has been proved to be optimal and another DFA C which is a equivalent DFA and is also optimal(given). We will show that C has to be isomorphic to A.

Since C is an optimal DFA, it can not be further shrunk. That implies two things: No redundant nodes, no pair of nodes is indistinguishable. That means that the language of the nodes of C is distinct and unique. Since A and C are both optimal, both must have same number of states, and all the nodes of both of them are reachable, and all the nodes of both of them have unique languages(unique over the domain of single DFA not collectively).
Consider any node s of A, take a string w which take us to it, run it over B, say we reach the node S, then s and S have same language. Since the nodes of B are distinguishable, so S does not depend on w.

This can be used to define an injection from the nodes of A to the nodes of B. It is injective function because of distinguishability of the nodes of A as well as of B. The function is also **bijective** because the number of nodes of A and B are both finite and equal, making it both injective and surjective, thus bijective.

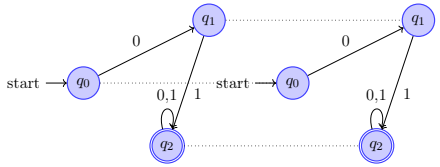


Figure 5.9: Two DFAs with corresponding states connected by dotted lines.

Claim:
As per this function, starting state of A will be mapped to starting stata of B. . This is because the languages of these two states is essentially the language of there respective DFA which is same(given).
Claim:
An accepting state will be mapped to an accepting state of B. . This is because any state which has ϵ in it's language is an accepting state by the definition of "language of state", similarly an accepting state will have ϵ in it's language. So an accepting state of A mapped to whatever state of B, that state must have ϵ in it's language making it an acceptable state of B.
Claim:
Consider Two states of A, S_1^A and S_2^A. Suppose that they are mapped to S_1^B and S_2^B respectively. Then if there is an α labelled edge going from S_1^A to S_2^A, there must be an α labelled edge going from S_1^B to S_2^B also. Consider any string w that takes us to S_1^A, we can see that $w.\alpha$ will take us to S_2^A. Now since S_1^A is mapped to S_2^A and S_1^B is mapped to S_2^B we can say that "any" string that takes us to one in A will take us to the image of it when run on B(Why? proved above:)). Thus w takes us to S_1^B and $w.\alpha$ takes us to S_2^B. Because a DFA is deterministic, there should be α labelled edge going from S_1^B to S_2^B.(Food for thought: Is mentioning determinism important here?)
Isomorphism of labelled graphs:
An isomorphism is a vertex bijection which is both edge-preserving and label-preserving.
Isomorphism of DFAs:
An isomorphism is a vertex bijection which is both edge-preserving and label-preserving, where image of starting state is starting state and image of a accepting state is an accepting state.
All these three claims shows that A and B are same automatons.

This tool can be used to check wether two different regular expressions represent the same language or different languages. Just convert them into DFAs, and then to the minimal DFAs. Check for the isomorphity of the two DFAs if they are isomorphic, then the regular expressions are equivalent otherwise not. (Isomorphism $\iff$ equivalence)

5.5 From states to words

Till now we were talking about languages of states and equivalence of two states. Now we will extend this notion for words.

5.5.1 Language of word

Consider a language L. we define Language of word w, $[w]$ as set of all strings x such that $w.x \in L$. Now we will define a relation "$\tilde{}_L$ ". If languages of two words *w1* and *w2* is same for L, then we say that w1 is related to w2. This is a reflexive, symmetric and transitive relation. (Basically, if set 1 and set 2 are same and then it is also true for the other way round that is set 2 and set 1 are same. If set 1 and set 2 are same and if set 2 and set 3 are same then set 1 and set 3 will also be same.) Thus we have defined an equivalence relation over words, where the equivalence classes depends on language L.

5.5.2 Relation between states of minimal DFA and equivalence classes for $\tilde{}_L$

If a string w ends up in a state of DFA, say s, then the language of the state is equivalent to $[w]_L$. This can be easily proved by using definition of the "language of state" and "language of word". If a string $w.x \in$ L, then if you run the $w.x$ on DFA, you first reach that state using w then x takes to some accepting state. Hence x $\in L_s^{DFA}$. Also if x $\in L_s^{DFA}$, then essentially $w.x$ will end in a accepting state because w ends in state s.
A particular equivalence class represent a particular language and a state represent a particular language. Since for a minimal DFA, all the states represent distinct language, we can define a bijection from equivalence classes to the state of minimal DFA, where a equivalence class is mapped to that state which represent the same language as that of any string in that equivalence class.

5.6 Setting up the Parallel

We defined the Nerode equivalence relation on a language L over the alphabet Σ. This relation states that $\forall w_1, w_2 \in \Sigma^*$, $w_1 \sim_L w_2$ iff $\forall x \in \Sigma^*$, $(w_1 \cdot x \in L \iff w_2 \cdot x \in L)$

If the language L is regular, then a minimized DFA $A = (Q, \Sigma, \delta, q0, F)$ can also be defined for the language L.

For this language L, w_1 and w_2 are two words in Σ^* such that $w_1 \sim_L w_2$. Let q_i and q_j be two states $\in Q$ such that

$$q_i \rightarrow \text{state reached in A after reading } w_1$$
$$q_j \rightarrow \text{state reached in A after reading } w_2$$

Since $w_1 \sim_L w_2$, it follows that for all $x \in \Sigma^*$, the state reached in DFA A after reading $w_1 \cdot x$ and the state reached in DFA A after reading $w_2 \cdot x$ will either both belong to F or both belong to $Q \setminus F$. Otherwise, one word would be accepted by L while the other would not, leading to a contradiction in the definition of the equivalence relation.

Therefore, by the definition of indistinguishability, states q_i and q_j can be concluded to be indistinguishable.However, in a minimized DFA, all distinct pairs of states are distinguishable. Consequently, the only states in A that can be indistinguishable are those where the state is compared with itself.

This demonstrates that if two words/strings belong to the same equivalence class with respect to L, then both strings will end up in the same state $q \in Q$ in the minimized DFA.

Till now, we have defined two equivalence relations $\sim_L$ over the language L and $\equiv$ over the states of a DFA characterizing a language L. We aim to define a relation between the number of equivalence classes of both these relations.

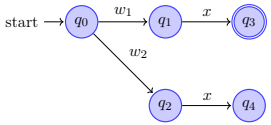
5.6.1 Can $|\sim_L| > |\equiv|$?

NO. We will in fact prove below that $|\sim_L| \leq |\equiv|$

Let there be strings w_1,w_2 s.t.

$$w_1 \in i^{th} \text{ equivalence class of } \sim_L$$
$$w_2 \in j^{th} \text{ equivalence class of } \sim_L$$
$$\text{where } i \neq j$$

Since they belong to different equivalence classes, we can say $\exists$ x $\in \Sigma^*$ s.t. $w_1 \cdot x \in L$ and $w_2 \cdot x \notin L$ (or can be vice-versa, does not matter).



- Here, q_1 represents the equivalence class of w1 while q_2 represents the equivalence class of w2.
- By the definition of indistinguishability, q1 and q2 are distinguishable states as there exists a letter in the alphabet which leads them to another pair of distinguishable states.

- Doing this for every pair of equivalence classes, we find that the states representing the distinct equivalence classes are all distinguishable from each other leading us to the following relation.

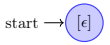
$$|\sim_L| \leq |\equiv|$$

.i.e. the number of indistinguishability equivalence classes is atleast the the number of Nerode equivalence classes

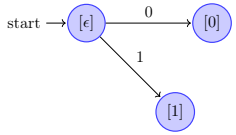
5.6.2 Can$|\sim_L| < |\equiv|$?

In order to inspect this , we are going to construct a DFA using $\sim_L \subseteq \Sigma^* \times \Sigma^*$ relation which will then accepts the language L. Let here, $\Sigma = 0, 1$

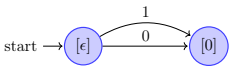
- A state denoting $[\epsilon]$ is made and is also denoted as the start state.



- Then we pick a letter from the alphabet and look the transitions from each of the existing states. If the next word already lies in the equivalence class of one of the existing states, we draw the arrow representing the particular transition otherwise a new state is made representing the equivalence class of the newly made word.

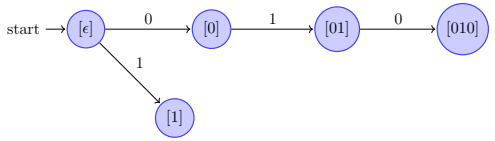


Here, 1 does not lie in the equivalence class of 0, i.e., [0].



If 1 lies in [0], then the automata will look like this.

- By repeatedly applying point 2, the automata can keep on expanding like as below



- However, this construction is guaranteed to converge because we have already proved that $|\sim_L| \leq |\equiv|$. It is known that the cardinality of equivalence classes of the indistinguishability relation is equal to the number of states in the minimal DFA representing the language L.Since the number of such states is finite, it follows that $|\equiv|$ is also finite.Given that $|\sim_L| \leq |\equiv|$, it can be deduced that $|\sim_L|$ is finite. Consequently, this guarantees the convergence of the algorithm in question.[If the above algorithm does not converge, that means new states will keep on forming contradicting the $|\sim_L| \leq |\equiv|$ identity.]

- Finally, all those states whose equivalence classes have words that are accepted by the language L are marked as accepting states.

Hence, the aforementioned algorithm successfully allows us to construct a finite state automata(DFA) which accepts only the words that are accepted by the language L. Note that here the number of states in the new automata is equal to $\sim_L$.
Now, let us assume that $|\sim_L| < |\equiv|$ holds. This implies that our newly constructed automata is the minimized DFA for the language L.
However, in the last lecture we had proved that the DFA constructed by $\equiv$ relation is minimized one and is unique. Hence, it leads to a contradiction,making our assumption wrong.

Thus proved that

$$|\sim_L| = |\equiv|$$

Note: The Nerode equivalence relation,unlike the indistinguishability relation,can be defined for any language $L \subseteq \Sigma^*$, irrespective of the fact that whether the language is regular or not.

5.7 Myhill-Nerode Theorem

L is regular if and only if $\sim_L$ has a finite number of equivalence classes.

This theorem provides an exact characterization of a regular language, unlike the Pumping Lemma. While the Pumping Lemma does not guarantee that L is regular if it holds, here, if $\sim_L$ has a finite number of equivalence classes, the language L must be regular. The proof of this theorem can be found here.

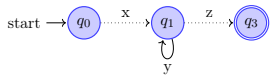
Chapter 6

Pumping Lemma for Regular Languages

If L is a regular language, then there exists a finite DFA with a minimum number of states (say p) that recognizes it. Let's consider a string w of length at least p that is accepted by this DFA. By the Pigeonhole Principle, we can deduce that when the DFA processes the string, it must revisit at least one state, thereby implying the existence of a loop.

6.1 Example

Consider string $w \in L$, where L is a regular Language. Suppose the first state in DFA which is revisited again on processing string w is q_1.



Hence for any $i \geq 0$ $xy^iz \in L$
The length of substring xy cannot exceed the size of the DFA. Hence,

$$|xy| \leq n$$

where n is the length of DFA
Also $|y| \geq 1$ as it should be possible to go back to the same state

6.2 Formal Statement of Pumping Lemma

For any regular language L, the following holds

$$\exists p > 0 \, \forall w, \, (w \in L) \cap (|w| \geq p) \, \exists x, y, z \, (w = x.y.z) \, \forall n \, (n \geq 0) \Rightarrow xy^nz \in L$$

Contrapositive:

$$\forall p > 0 \, \exists w \, (w \in L) \cap (|w| \geq p) \, \forall x, y, z \, (w = x.y.z) \exists n (n \geq 0) \cap xy^nz \notin L$$

If the above formula holds true then L is not a regular language.

- If a language L is regular then Pumping Lemma holds true for L but if Pumping Lemma holds true for Language L then it does not necessarily mean that L is regular.
- But if Pumping Lemma does not hold true for Language L then L is not regular.

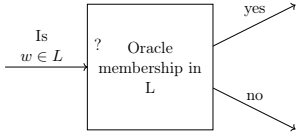
6.3 Pumping Lemma as Adversarial Game

Game between an adversary (who wants to show that the language L is not regular) and a believer (who believes that the language is regular) is as follows:

- The believer chooses an integer $p > 0$ and claims this is the count of states in the DFA that she believes recognizes the language.
- Adversary chooses a string $w \in L$ such that $|w| > p$.
- Believer then splits w into three parts $w = xyz$, where $|xy| \leq p$ and $|y| > 0$
- Adversary now chooses an integer $n \geq 0$ such that $xy^n z \notin L$ thereby winning the game. If the adversary can't do this then they lose.

6.4 If a language L is regular and the number of states in DFA for L is n, is L infinite?

Given the DFA for language L, if there is a path from the initial state to a state forming a cycle, and from that state there's a path to an accepting state, then L is infinite because the DFA can endlessly loop within this cycle, generating an infinite number of accepted strings.
Instead if only a black box is provided, which can determine whether a given string w is in L or not.



Test all strings w, such that $n < |w| \leq 2n$ for membership in L. If we find even one string which returns 'yes' on testing then L is infinite. (Since pumping lemma can be applied to such a string, and it can also be pumped to show that an infinite sequence of strings are in L).
If all strings w, ($n < |w| \leq 2n$) return 'no' on testing then we claim that there are no strings in L of length more than $2n$, and thus there are only a finite number of strings in L.
Proof: On applying the pumping lemma for such w repeatedly, we can obtain strings of length between n and $2n$, which again belong to L. This is because, for $w = xyz$, removing loop y every time decreases the length by at least 1. This is a contradiction to our assumption that there are no strings of length between n and $2n$ in L.
Hence L is infinite if it contains atleast one string of length at least $n + 1$.

6.5 Example

Consider the Language $L = \{(0+1)^{*01^k\,|\,k\ \ is\ \ prime\}}$ with alphabet $\sum = \{0, 1\}$
The application of the Pumping Lemma is independent of the initial state chosen such that length of string remains $\geq p$ (pumping length)
In the case of this language, the Pumping Lemma can indeed be applied to the substring 1^k, where k is any prime number.
Now we can break $w = 1^k$ into $w = xyz$ such that $y \neq \epsilon$ and $|xy| \leq n$. Let $|y \models m\ (m > 0)$, then $|xz \models k - m$
Consider the string a=$xy^{k-m}z$

$$|a \models |xz| + (k-m)|y| = (m+1)(k-m)$$

As length of string a is not a prime (it has factors $m + 1$,$k - m$), we have $a \notin L$
Hence Language L is not regular.
Reverse Language
If Language L is regular then its reverse language L^R is also regular.
To construct automaton for L^R from that of L swap initial and final states and reverse the edges.

Chapter 7

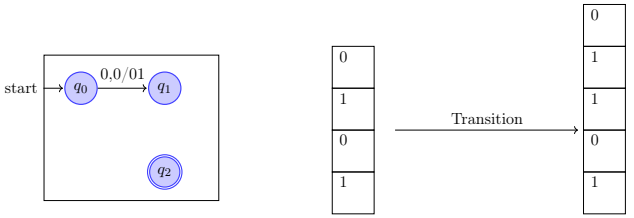
Pushdown Automata

7.1 Pushdown Automata for non-regular Languages

Thus far, the finite automata we have studied cannot be used to represent non-regular languages. One such example of a non-regular language is the language of balanced parentheses.Let it be L.

> **Proving why L is not a regular language** *It is known that intersection a language L with any regular language preserves the regularity of the original language.i.e. the new language formed after intersection will hav ethe same regularity as the original language L.*
> *Therefore, considering $L \cap (*)^*$ results in $(^n)^n$ which by applying the Pumping Lemma can be shown that that it is not a regular language, it can be thus concluded that the language of balanced parenthesis is also not a regular language.*

So if we can equip the finite state automata with more structures like a stack, it will be abl to accept the non-regular languages too.



Stack 1

Transition

Stack 2

Figure 7.1: Automaton and Stacks

7.2 Description

PushDown Automata (PDA) are essentially NFAs equipped with a stack that is maintained on an alphabet that is different from the alphabet used for state transition. PDAs have greater accepting power than regular NFAs and can accept non-regular languages as well.

7.2.1 Conventions

Just like we described a NFA L as $L(Q, \Sigma, q_0, \delta, F)$, we describe PDAs as characterized by

$$L(Q, \Sigma, \Gamma, q_0, Z_0, \delta, F)$$

where-

- Q is the set of all states in the PDA
- Σ is the alphabet accepted by the PDA
- Γ is the set of symbols that are pushed/popped from the stack associated with the PDA
- q_0 is the starting state of the PDA. There may be more than one starting state for the PDA.
- Z_0 is the starting symbol in the stack. This is so that a symbol can be popped on the first transition.
- δ is the transition function for the PDA. While the transition function for regular NFA was of the form $\delta : Q \times \Sigma \cup \{\epsilon\} \rightarrow 2^Q$, the transition function for PDAs is characterized by $\delta : Q \times \Sigma \cup \{\epsilon\} \times \Gamma \rightarrow 2^{Q \times T^*}$, where Γ^* is the set of strings made by the alphabet Γ.
- F is the set of final states of the PDA

A single transition on a PDA from state q_1 to q_2 is represented by $x, a/ba$ where $x \in \Sigma$ is the original symbol and may be ϵ, a is the top element of the stack and is popped off (it cannot be ϵ), and ba is the string that is pushed onto the stack after a is popped off. The final top element of the stack is b after this transition takes place.
This places a restriction on PDAs as compared to NFAs, that is, for any transition to occur, the top element of the stack also has to match the top element specified in the transition.

7.2.2 Example

Let us characterize the non regular language $\{0^n 1^n | n \geq 0\}$ through a PDA with a finite number of states, as shown in Figure 7.2. We define $\Gamma = \{z_0, a\}$ as our stack alphabet with our PDA starting with only z_0 in the stack. a will represent a counter for the number of zeros in our string.
Notice that if a string deviates from the form $0^m 1^n$ the automaton goes to trap. For every 0 in the string a is pushed onto the stack and hence the $l(S) = m + 1$ where $l(S)$ is the length of stack. For strings following the given structure, if $m = n$, then the string ends at an accepting state. If $m > n$, the string goes to trap and if $m < n$, the string ends the run at a non accepting state.
We reiterate that a string is accepted if and only if the run does not halt before the string is completed and the final state is an accepting state.

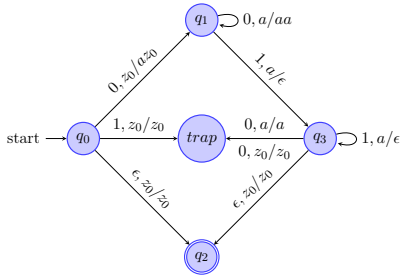


Figure 7.2: Automaton accepting $\{0^n1^n|n \geq 0\}$

7.2.3 Example

Now let us extend our previous automaton to accept the language $\{0^n1^m|n \geq m \geq 1\}$ as shown in Figure 7.3. Herein, we introduce a different notion of string acceptance, that is, if the stack becomes empty upon completion of the run, we can say that the string is accepted. This is called **acceptance through empty stack**.

7.3 Acceptance through empty stack

We introduce a new way of characterizing string acceptance in PushDown Automata wherein there is no need to specify accepting states explicitly. Here, a string is accepted if it satisfies the following constraints for atleast one possible run on the PDA-

1. String should not halt in between its run, that is, if any symbol is left in the run on the string, there must exist atleast one entry in the transition function δ corresponding to any of the possible current states, the next character and any of the possible stack tops.

2. Upon completion of the run, either the stack should be naturally empty or through a series of ϵ transitions, should be able to make its stack empty for the string to be accepting.

3. There are no constraints on the current state and the string can be accepted through any state $q \in Q$.

This gives a new method of acceptance. We represent the language accepted by this method over an automaton A as $N(A)$ and the language accepted by our original method as $L(A)$.
However, we do not yet know if the two methods of acceptance are equally powerful. We shall prove this claim by showing the equivalence of the two methods over PDAs by transforming one into the other and vice versa.

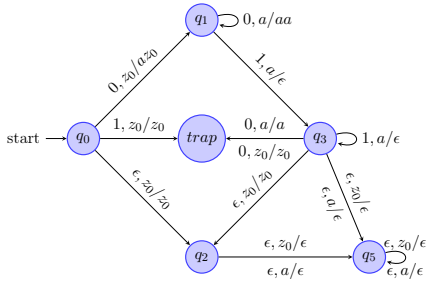


Figure 7.3: Automaton accepting $\{0^n1^m|n \geq m \geq 0\}$

7.3.1 For any automaton A, $L(A)$ is not necessarily equivalent to $N(A)$

We must observe a subtlety, if a language L is accepted through PDAs through both methods, the automata that accept these need not be the same. This is easily shown through the previous two examples.

- In Figure 1, $L(A) = \{0^n1^n|n \geq 0\}$ but $N(A) = \Phi$ as the stack never gets empty.
- In Figure 2, $N(A) = \{0^n1^m|n \geq m \geq 1\}$ but $L(A) = \Phi$ as their are no accepting states.

So we want to prove that-

1. If $L(A_1)$ is the language accepted by a PDA A_1, there exists another PDA A_2 such that $L(A_1) = N(A_2)$.

2. If $N(A_1)$ is the language accepted by a PDA A_1, there exists another PDA A_3 such that $N(A_1) = L(A_3)$.

We will show the existence of both of these by giving a construction which when applied to A_1 gives us A_2 and A_3 respectively.

7.3.2 Construction for A_2 such that $L(A_1) = N(A_2)$

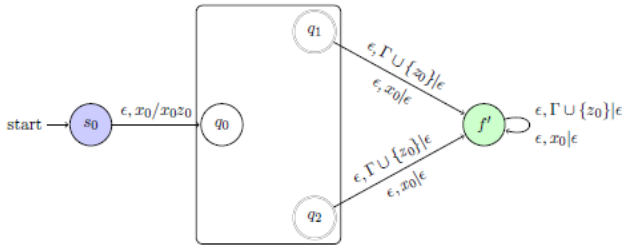
We need to modify A_1 to A_2 without making any assumptions about its structure except that it is a PDA, and make it such that it satisfies two conditions for the equivalence of the languages-

1. Rejection - $N(A_2)$ should not accept any string s that is not a part of $L(A_1)$

2. Acceptance- If a string s is a part of $L(A_1)$ it must also be a part of $N(A_2)$

We shall assume A_1 has only one starting state, and if not we can simply add extra ϵ-transitions to reach any start state from our original start state.

Before this start state, we append a new start state s_0 and have our stack start with $x_0 \notin \Gamma \cup \{z_0\}$ and connect it to start with an ϵ-edge pushing z_0x_0 to the stack. This is absolutely identical to the original stack with the exception of an extra x_0 at the bottom of the stack.
We will convert A_1 to A_2 without modifying the basic structure of A_1 such that for any accepting state in A_1, we add ϵ-transitions to a new final state f', which does not take in any input word and just pops the stack (x_0 or z_0 or $\gamma \in \Gamma$) without performing any additions onto it.



Acceptance

If string s is accepted in A_1 it will reach $f \in F$ on atleast one of its possible runs on A_1. If the stack has atleast one element upon reaching f, the string can go to f' and there the stack can get emptied through ϵ-transitions from f' to itself, thus putting s in $N(A_2)$ as well.
If the stack became empty when the string s ran on A_1 we will now have x_0 as the only element left in the stack (as no transition in the original automaton can pop x_0 because $x_0 \notin \Gamma \cup \{z_0\}$). Our automaton proceeds to f' by popping x_0 and is accepted as stack is emptied.

Hence, acceptance is proved as any string in $L(A_1)$ is also present in $N(A_2)$.

Rejection

A run of string s is rejected in $L(A_1)$ if-

1. Run successfully completes but the string is not in an accepting state
2. Run does not complete successfully because of empty stack
3. Run does not complete successfully because of no valid transitions

We show that our modifications can correctly reject all such cases.
If the run does not successfully complete on A_1 due to an empty stack, upon running on A_2, the stack will only have x_0 and as $x_0 \notin \Gamma \cup \{z_0\}$, there is no transition with x_0 as top, leading to the run halting and string being rejected.
If the run does not successfully complete on A_1 due to no valid transition, there will not be any valid transitions in A_2 as well because the stack is the same except for an extra x_0 at the bottom.

If the run successfully completes but the string is not in an accepting state, then the stack never becomes empty because x_0 will always be present.

Hence, rejection is also proved as no string rejected by $L(A_1)$ is accepted by $N(A)$.

7.4 From Empty Stack PDA to Final State PDA

We will now see how to construct a PDA with final state acceptance from a PDA with empty stack acceptance.
Consider PDA P_N = (Q, Σ, Γ, δ_N, q_0, Z_0) and N(P_N) be the set of strings that the PDA accepts by empty stack .

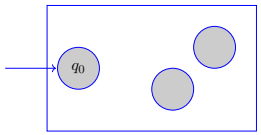
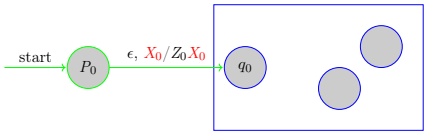


Figure 7.4: P_N - PDA with empty stack Acceptance

7.4.1 Construction

Now, Consider a construction to P that involves

- Addition of new stack variable $X_0 \notin \Gamma$ as the bottom of the stack.
- Addition of a new start state P_0 and an ϵ - transition from this new start state on reading X_0 on the stack to q_0 and Z_0 as top of stack.



- Introducing a new final state P_f and ϵ-transitions from all states in Q such that on reading X_0 on the stack transition happens to state P_f and an empty stack.

After the construction, the new automaton P_F will be $(Q \cup \{p_0, p_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta_F, p_0, X_0, \{p_f\})$ whose acceptance is by final state.

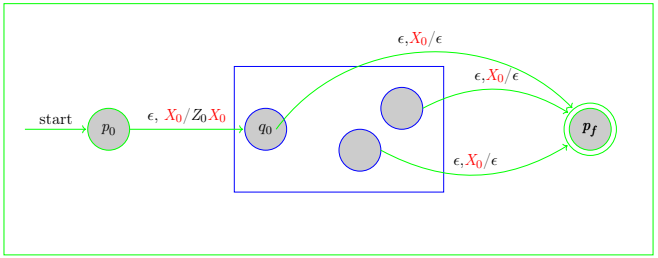


Figure 7.5: P_F (After the construction)

7.4.2 Required to show $N(P) = L(P_F)$

- First, Let us show that if a string w is accepted by P then it will also be present in $L(P_F)$

  If w is accepted by P, then the stack in P must have been emptied after consuming w, which implies that stack in P_F now has X_0 at its top. Now , taking the ϵ-transition by reading X_0 will lead us to p_f which is an accepting state of P_F.Hence w will also be present in $L(P_F)$.

- Now, Let us show that if w is in $L(P_F)$ then it will also be present in $N(P)$

  In the PDA $L(P_F)$, the only transitions that can be used to reach p_f state are ϵ transitions on reading X_0. Since w is present in $L(P_F)$, it must have taken one of these transitions and read X_0 at the top of the stack which implies that stack in P must have been emptied by w. Hence, w would also be accepted by P.

Finally one can say that,
Acceptance of PDA using final states ≡ Acceptance of PDA using empty stack.
i.e For every PDA A whose acceptance is by final states there exits a PDA A' whose acceptance is by empty stack and vice-versa.

7.5 Context Free Languages

Languages accepted by the PDAs are called Context Free Languages. These are strict superset of regular languages.

7.5.1 DPDA and NPDAs

A PDA is considered deterministic if, from a given state and upon reading a specific symbol from the top of the stack, there exists only one possible transition for a given input and a PDA is considered non-deterministic if there exists either no transition or many transitions possible for an input.

The language that can be represented using DPDAs can also be represented using the NPDAs.i.e

$$DPDA \subseteq NPDA$$

But, there are certain languages which can be represented only by NPDAs but not DPDAs.Here is an example,

$$L = \{ww^R \mid w \text{ is in } (0+1)^*\}$$

If we try to represent L using a DPDA by pushing a copy of input symbol seen on to stack, one has to make a guess every time whether input has reached end of string w or not so that popping can be done to check the palindrome nature of the string. Hence non-determinism has to be involved to represent L.
Hence,

$$DPDA \subset NPDA$$

7.5.2 Build-up and emptying of the stack of a PDA

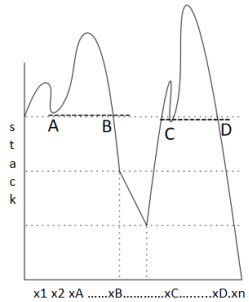


Figure 7.6: Behaviour of stack with the input being processed

In the above Figure, if we observe the part of graph where strings xA...xB and xC....xD are read, the behaviour is independent of the content of the stack before xA but just depends on the state and top of the stack when the symbol xA is being read.
Hence, a context-free language can be fragmented into some finite no.of sets (these sets need not be finite) each defined by (*top of stack*,*state of automaton*)

7.6 Acceptance by PDA

Let us begin by taking the example of the following PDA with $\Gamma = \{Z_0, X\}$ as the stacks symbols and Z_0 as the initial stack state

The language accpeted by this automata is $N(A) = \{0^n 1^n | n \geq 1\}$

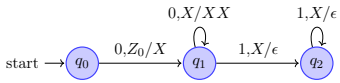


Figure 7.7: PDA A

7.6.1 Run on the PDA

Let us have a look at the run of the string 0011 on this PDA

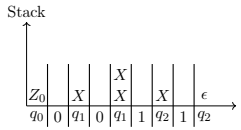
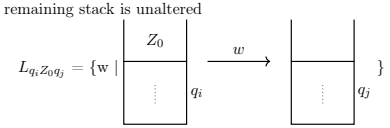


Figure 7.8: Run on 0011

We can see how the string 0011 starts with the state q_0 and Z_0 initially on the top of the stack and ends with an empty stack. Let us define a mathematical notation for the language that the string 0011 belongs to.

$L_{q_i Z_0 q_j}$ = set of all strings that start at the state q_i with Z_0 on the top of the stack and end at the state q_j with Z_0 popped off the stack and the remaining stack unaltered

Note: The stack may have risen or fallen during the transitions but finally only Z_0 is popped off the stack and the remaining stack is unaltered



For PDA A we can say:

- $011 \in L_{q1Xq_2}$
- $1 \in L_{q_2 X q_2}$
- $0011 \in L_{q_0 Z_0 q_2}$
- and many more ...

Now lets try to define a language on PDA A, N(A) such that it starts at state q_0 and an initial stack containing Z_0 and empties it.

$$N(A) = L_{q_0 Z_0 q_0} \cup L_{q_0 Z_0 q_1} \cup L_{q_0 Z_0 q_2}$$

We took the union of languages that started at q_0 emptied the stack and ended at any of the states present in the PDA A.

7.6.2 Recurrence Relations on Languages

Lets take a look at the individual transitions of PDA A

Figure 7.9: self loop on q_1

Now what can we say about the language $L_{q_1 X q_2}$

If we consider a string w such that 0 is the first letter of w then this transition is undertaken and we have added another X to the stack. So now w would have to pop two X from the stack and arrive at the state q_2 for w to belong to the language $L_{q_1 X q_2}$. This can be written as

$$L_{q_1 X q_2} \supseteq 0.L_{q_1 X q_0}.L_{q_0 X q_2} \cup 0.L_{q_1 X q_1}.L_{q_1 X q_2} \cup 0.L_{q_1 X q_2}.L_{q_2 X q_2}$$

Now if we consider the transition

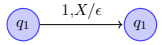


Figure 7.10: self loop on q_1

We can say

$$L_{q1 X q_2} \supseteq 1$$

We can keep doing this for all transitions from q_1 and get a super set recurrence relation for $L_{q_1 X q_2}$

7.7 More on recurrence relations

Talking about a general PDA with n states and k stack symbols n^2k languages of the form $L_{q_i S_j q_k}$ can be defined and considering individual transitions super set recurrence relations can be obtained for them. For example

$$L_1 \supseteq 1 \cup 0.L_2.L_3 \cup \ldots$$

$$\vdots$$

$$L_{n^2 k} \supseteq 0.1 \cup 0.L_1.L_3 \cup \ldots$$

7.7.1 Smallest and Largest Languages

Consider the relation as follows

$$L_1 \supseteq 0.1 \cup 0.L_1.1$$

Notice how both Σ^* and $0^n1^n n \geq 1$ are both languages that satisfy this relation but Σ^* is largest language that satisfies this relation and $0^n1^n n \geq 1$ is the smallest(can be proven smallest by induction if 0^i1^i belongs to L_1 then so does $0^{i+1}1^{i+1}$)

We will be particularly interested in finding these smallest languages satisfying the relation

NOTE: Σ^* is always a solution(largest) of the super set recurrence relation since it contains all strings

7.7.2 Context Free Grammar

Above equations imply that wherever we see a word of the language $L_{q_1Xq_2}$, we can replace it with any word from the languages on the RHS. Thus we can write:

$$L_1 \to 0L_2L_3 \mid 0L_4L_1 \mid 0L_1L_5 \mid 1$$

where $L_1 = L_{q_1Xq_2}$, $L_2 = L_{q_1Xq_0}$, $L_3 = L_{q_0Xq_2}$, $L_4 = L_{q_1Xq_1}$ and $L_5 = L_{q_2Xq_2}$.
Similarly, we can do the same for the languages $L_i \forall i \in \{1, 2, \ldots, n^2k\}$. This will form a **context free grammar**.

WIth proper reductions, we can say that the context free grammar generated for the language $L = \{0^n1^n \mid n \geq 1\}$ is:

$$S \to 0S1 \mid 01$$

We observe that the universal language $\Sigma^{*=L((0+1)^*)}$ also satisfies the context free grammar, but the language $L = \{0^n1^n \mid n \geq 1\}$ is the minimal language that satisfies the context free grammar.

Chapter 8

Context Free Grammer

8.1 Context-free Grammar

Definition: A context-free grammar (CFG) is a formal grammar whose production rules can be applied to a nonterminal symbol regardless of its context. In particular, in a context-free grammar, each production rule is of the form $V \to (V \cup T)^*$. Where V is set of Non-Terminal and T is set of Terminals.

Formally, a context-free grammar can be represented as follows -

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

where
V - is the set of **non-terminals**. These are symbols that can be replaced or expanded.
$\Sigma \cup \{\epsilon\}$ - is the set of **terminals**. These are symbols that cannot be replaced or expanded further.
P - are the set of production rules
S - is the start symbol ($S \in V$)

Again, a grammar is said to be the Context-free grammar only if every production is in the form of

$$G \to (V \cup T)^*, \quad \text{where } G \in V$$

Let's consider a context-free grammar defined as follows:

$$S \to A.S \mid \epsilon$$
$$A \to A1 \mid 0A1 \mid 01$$

Here, S and A are non-terminal symbols representing languages. ϵ is a special symbol representing an empty string and is in the language S. The symbols 0 and 1 are terminal symbols and are in the language A. The set of strings that can be generated using a context-free grammar is called a *context-free language*. This language of **A** contains strings that are of the form 01 or A1 or 0A1, by just substituting all possible strings of A in the form recursively

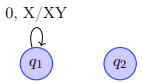
$$A \to 01 \mid 011 \mid 0011 \mid 0111 \mid 00111 \mid ...$$

Thus, Language derived from **A** can be intutively be told as

$$0^i1^j \text{ , where } j \geq i \geq 1$$

8.1.1 PDA to CFG

In the previous lecture, we talked about the language accepted by a Pushdown Automata (PDA), starting from a state q_i, with a symbol X on the top of the stack, and reaching a state q_j, with the stack remaining the same, except the fact that the top element, X, was popped off.
Such a language is denoted by $L_{q_iXq_j}$, where q_i is the start state, X is the top element of the stack, and q_j is the final state. Given certain transitions in a PDA, we could also come up with recurrence relations for such languages. For eg., if the following states and transitions were given to us, and we want to find $L_{q_1Xq_2}$



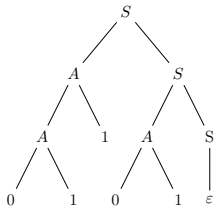
We can say that, for all states q_i in the automaton,

$$L_{q_1Xq_2} \supseteq 0 \cdot L_{q_1Xq_i} \cdot L_{q_iYq_2}$$

If we denote the languages by $L_1, L_2 \dots L_{n^2k}$, where n is the number of states and k is the number of stack symbols, then we can write the above equation as a recurrence relation for all languages.
For example, [mathescape=true] $L_1 \to 1 \mid 0 \cdot L_2 \cdot L_3 \mid 0 \cdot L_4 \cdot L_1 \mid \dots \; L_2 \to \dots \; \vdots \; L_{n^2k} \to \dots$ This set of recurrence relations is called a Context-free grammar. Hence, given a PDA, we can write the language accepted by the PDA by an empty stack as a CFG.

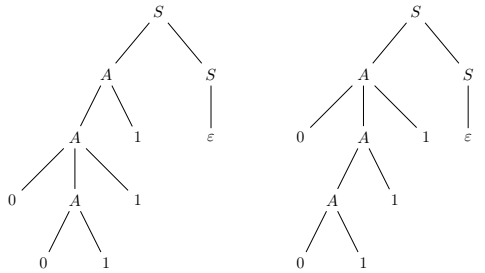
8.2 Parse Trees and Ambiguous Grammars

Suppose we have the string 01101 and we want to check whether this lies in L_S.
First, we will apply the rule $S \to A \cdot S$, since the rule $S \to \varepsilon$ is not useful yet. Now, we further see $A \cdot S$ recursively. For S, $S \to \varepsilon$ is not useful since 01101 does not lie in L_A. In this manner, we further recursively break up our string until we reach terminals and check whether the sub-parts can be further be made useful. In this manner, we can create parse/derivation trees for a string, given a grammar. The leaves in these trees are our terminals. This is shown in disgram below.



As we can see, the leaves form 01101ε, from left to right, which is exactly the string we wanted. Hence, 01101 $\in L_S$, using the production rules in the grammar.
While drawing the trees, we must ensure that the root is the start symbol, and the node at which we want to split our tree, we are using production rules with that value in the left hand side. For example, if we want to split the tree at an A, we must only use production rules of the form $A \to \dots$.

For this particular string and grammar, only one parse tree is possible. Let us now consider the string 00111 $\in L_S$.



As we can see, these are two completely different, and correct parse trees for the string in the same grammar. Such grammars, where there exist multiple parse trees for some strings, are called ambiguous grammars. Hence, the grammar we have defined is indeed ambiguous.

Are there CFLs for which every CFG representing it are ambiguous grammars? Yes, there indeed are such CFLs. Such languages are called inherently ambiguous languages. If there exists even one grammar for the CFL which is unambiguous, then the language is an unambiguous language as well. Is L_S inherently ambiguous or is it an unambiguous language?

The answer is that it is not inherently ambiguous. The intuition is that, we must force our grammar to first match up all the zeros in the strings with ones, and following that, add the terminating ones. Our current grammar puts no such restriction. We could first add zeros and ones to the start and end, add ones to the end, and then go back to adding zeros and ones to the start and end. This leads to multiple parse trees. An example of a CFG representing L_S, which is unambiguous is [mathescape=true] $S \to C \cdot S \mid \varepsilon \; C \to A \cdot B \; A \to 0 \cdot A \cdot 1 \mid 01 \; B \to 1 \cdot B \mid \varepsilon$ As we can see, A does the job of adding zeros and ones to the start and end, B does the job of creating strings of ones, and C does the job of concatenating A and B.
This is important, because every programming language is specified using CFGs, for which we can build compilers and interpreters, and it is important for these CFGs to be unambiguous since we do not want multiple interpretations of a program.

8.2.1 CFG to PDA conversion

Consider the Context-free grammar defined as follows

$$\begin{aligned} S &\rightarrow AS \mid \epsilon \\ A &\rightarrow A1 \mid 0A1 \mid 01 \end{aligned}$$

Where,
Starting Symbol: S
Production Rules: $S \rightarrow AS, \quad S \rightarrow \epsilon, \quad A \rightarrow A1, \quad A \rightarrow 0A1, \quad A \rightarrow 01$
Non-terminals: $\{S, A\}$
Terminals: $\{0, 1\}$

Now Consider the following 2 alphabets,

$$\Sigma = \{0, 1\}, \quad \Gamma = \{S, A, X_0, X_1\}$$

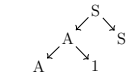
where Σ represents the alphabet of the PDA which we are going to construct and Γ represents the alphabet of the stack.
Note that we have exactly same number of symbols as total number of terminals and non-terminals in our CFG each symbol corresponding to each terminal or non-terminal.

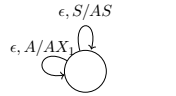
We will start constructing our PDA by adding some edge(s) to a single node, pushing or popping some letter from the stack while constructing derivation tree/ Parse tree by using each production rule.

Step 1:
We initialize our stack by pushing S as S is our starting symbol and we represent it as follows ,Where left means bottom of the stack and right is it's top
Stack = S,

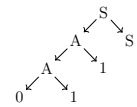
Step 2:
We used the rule: $S \rightarrow AS$ here and added the edge $\epsilon, S/AS$ into our single node as we can replace S to AS anywhere if we want to as it is defined in our CFG's production rules. Also now we pop S from our stack and push S,A (here A and S are from Γ).
Stack = S,A,

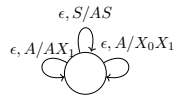
Step 3:
We used the rule: $A \rightarrow A1$ here and added the edge $\epsilon, A/AX_1$ (here X_1 is just the stack symbol corresponding to the terminal '1') into our single node as we can replace A to A1 anywhere while forming a string. Also now we pop A from our stack and push X_1,A (here A and X_1 are from Γ).
Stack = S,X_1,A,



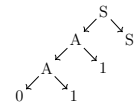


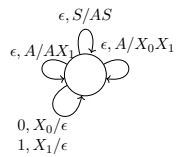
Step 4:
We used the rule: $A \rightarrow 01$ here and added the edge $\epsilon, A/X_0X_1$ (here X_1,X_0 are just the stack symbol corresponding to terminals 1 and 0) into our single node. Also now we pop A from our stack and push X_1,X_0 (here X_0 and X_1 are from Γ).
Stack = S,X_1,X_1,X_0,



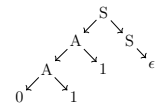


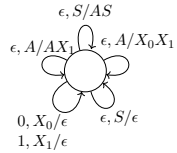
Step 5:
Now as we know that when terminals come on top of the stack and we read it from the top of the stack and pop it off and if a non terminal is seen then we apply any of the given production rules. So for this pop operation we now add the edges $0, X_0/\epsilon$ and $1, X_1/\epsilon$ into our single node. So we pop all the terminals until we see a non-terminal.
Stack = S,





Step 6:
We now use the rule: $S \rightarrow \epsilon$ here and added the edge $\epsilon, S/\epsilon$ into our single node. Also now we pop S from our stack and push ϵ that is stack becomes empty again.
Stack =





Now this stack helps in doing the DFS of the derivation tree, we first visit all the nodes on left and then all the nodes on right.
So to conclude we can say that given any arbitrary Context-free grammar we can easily construct PDA out of it by using some simple steps as described above which recognizes the exact same language as the given CFG.

8.3 Cleaning Context-free Grammar

Consider the Context-free grammar defined as follows

$$\begin{aligned} S &\rightarrow ABS \mid 0A1 \\ A &\rightarrow 1A0 \mid D \mid 01 \mid \epsilon \\ B &\rightarrow 1B \mid BB0 \\ C &\rightarrow A0 \mid 01 \\ D &\rightarrow S \mid 0AD \end{aligned}$$

Where,
Starting Symbol: S
Non-terminals: $\{S, A, B, C, D\}$
Terminals: $\{0, 1, \epsilon\}$

8.3.1 Eliminating Useless Symbols

We can eliminate the production rules for the non-terminal symbol "C" from our context-free grammar (CFG). Here's the reasoning:

- The starting symbol "S" can generate strings using symbols from "A," "B," and itself.
- While "B" depends on itself and "A" depends on itself and "D," we ultimately find that "D" relies on "A," "S," and itself.
- Therefore, knowing the strings accepted by "C" is unnecessary for generating strings from "S." In other words, even without the production rules for "C," "S" can still generate valid strings by appropriately utilizing "A," "B," and "D."

We can further simplify the grammar by identifying and removing another redundant non-terminal symbol: "B". Analyzing the production rules for "B", we observe two key characteristics:

- **Absence of Terminal Productions:** None of "B"'s production rules generate strings consisting solely of terminal symbols.
- **No Introduction of Useful Non-Terminals:** Additionally, "B"'s rules do not introduce any other non-terminal symbols that could potentially lead to terminal strings through subsequent productions in the parse tree.
- So the language generated by the non-terminal symbol "B" is empty (as it cannot produce any finite length string by terminating at some point), we can safely remove all production rules that involve "B" from the context-free grammar.

Now that we've removed the unnecessary rules, here's the updated CFG:

$$\begin{aligned} S &\rightarrow 0A1 \\ A &\rightarrow 1A0 \mid D \mid 01 \mid \epsilon \\ D &\rightarrow S \mid 0AD \end{aligned}$$

8.3.2 Eliminating ϵ-Productions

For removing epsilon just see where you can apply the rule $A \rightarrow \epsilon$.
We can add one more rule in all that places where A appears by replacing it with ϵ.
So after doing these changes we won't need the production rule $A \rightarrow \epsilon$ in our CFG.
Updated CFG:

$$\begin{aligned} S &\rightarrow 0A1 \mid 01 \\ A &\rightarrow 1A0 \mid D \mid 01 \mid 10 \\ D &\rightarrow S \mid 0AD \mid 0D \end{aligned}$$

8.3.3 Eliminating Unit Productions

Similar to the above property we can do the same thing here also.
Let's say we have a production rule like this $X \rightarrow Y$ where X, Y are non-terminals then we can add one more rule in all that places where "X" appears by replacing it with "Y".

Let's first remove the rule $D \rightarrow S$ by doing the changes described above,

$$\begin{aligned} S &\rightarrow 0A1 \mid 01 \\ A &\rightarrow 1A0 \mid D \mid S \mid 01 \mid 10 \\ D &\rightarrow 0AD \mid 0AS \mid 0D \mid 0S \end{aligned}$$

As we can see that after doing this modification to our CFG we now have two more production rules $A \rightarrow S$ and $A \rightarrow D$ so let's remove them also,

$$\begin{aligned} S &\rightarrow 0A1 \mid 0D1 \mid 0S1 \mid 01 \\ A &\rightarrow 1A0 \mid 1D0 \mid 1S0 \mid 01 \mid 10 \\ D &\rightarrow 0AD \mid 0AS \mid 0DD \mid 0DS \mid 0SD \mid 0SS \mid 0D \mid 0S \end{aligned}$$

After doing this we now have at least one non-terminal symbol on the right hand side of any rule or only terminals.

8.3.4 Reducing further to get at least two non-terminals on the right hand side or a single terminal

Just replace the terminals 0 and 1 with their corresponding symbols "X_0" and "X_1" in the CFG and add corresponding new rules.

$$\begin{aligned} S &\rightarrow X_0AX_1 \mid X_0DX_1 \mid X_0SX_1 \mid X_0X_1 \\ A &\rightarrow X_1AX_0 \mid X_1DX_0 \mid X_1SX_0 \mid X_0X_1 \mid X_1X_0 \\ D &\rightarrow X_0AD \mid X_0AS \mid X_0DD \mid X_0DS \mid X_0SD \mid X_0SS \mid X_0D \mid X_0S \\ X_0 &\rightarrow 0 \\ X_1 &\rightarrow 1 \end{aligned}$$

8.3.5 Reducing further to get exactly two non-terminals on the RHS of any production rule which has >2 non-terminals on it's RHS

Let us say our production rule has 3 non-terminals on it's right hand side then we can reduce it to exactly 2 non-teminals as follows,

$S \to X_0AX_1$ can be reduced to $S \to X_0S_1$ and $S_1 \to AX_1$

Note that the CFG after doing this modification will accept exactly same strings as accepted by it initially because now in the parse tree $S \to X_0AX_1$ will just take one step more, it first expands to $S \to X_0S_1$ and then to $S_1 \to AX_1$. However, the final string generated remains unchanged.
So it turns out that for any number of non-terminal symbols we can do this inductively to get exactly 2 non-terminals on RHS of any production rule.
Updated CFG:

$$\begin{array}{lcl} S & \to & X_0S_1 \mid X_0X_1 \\ S_1 & \to & AX_1 \mid DX_1 \mid SX_1 \\ A & \to & X_1A_1 \mid X_0X_1 \mid X_1X_0 \\ A_1 & \to & AX_0 \mid DX_0 \mid SX_0 \\ D & \to & X_0D_1 \mid X_0D \mid X_0S \\ D_1 & \to & AD \mid AS \mid DD \mid DS \mid SD \mid SS \\ X_0 & \to & 0 \\ X_1 & \to & 1 \end{array}$$

The CFG written above is said to be in it's Chomsky normal form (defined below).

8.3.6 Chomsky Normal form (CNF)

A Context-free grammar is said to be in it's Chomsky normal form if it does not contain any useless symbols, no epsilon productions ($X \to \epsilon$), no unit productions ($X \to Y$) and all production rules are either of the form $A \to BC$, where A, B, C are non-terminals or of the form $D \to u$, where D is a non-terminal and u is a terminal.

Every Context-free grammar can be expressed in it's Chomsky normal form by application of various rules as described in section 2 which accepts the same set of strings which is accepted by the original CFG.

8.4 Pumping Lemma for Context-free Languages

Let G be the Chomsky normal form of any Context-free grammar having n non-terminals. Consider derivation tree of a word "s" having length greater then 2^n as shown in the figure (8.1) below.

Now due to the restricted nature of the CNF form of CFG (can contain maximum 2 symbols in it's RHS) the derivation tree will be a binary tree.
Now a binary tree of height n can have maximum of 2^n leaves but here we are taking $|s| > 2^n$ so the height of derivation tree must be $> n$. But we are having only n non-terminals in our CFG so by pigeon hole principle there must exist some non-terminal "N" which repeats atleast twice.
We split up s into 5 parts as shown in the figure (8.1),

$$s = u.v.w.x.y$$

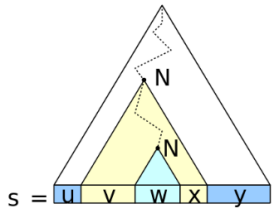


Figure 8.1: Derivation tree of a word of length $> 2^n$

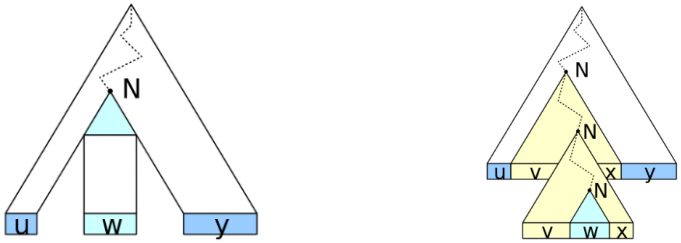


Figure 8.2: Examples of some possible derivation trees from figure (8.1)

- Using the property of derivation trees (of CFG) we can say that whatever I can generate from first "N" can also be generated by the second "N" and vice-versa.
- So after encountering a "N" in our derivation tree we can choose to generate a new "N" or terminate after some productions by choosing the "N" which doesn't produce another "N".
- So, to formally state pumping lemma for CFL we need to have some constraints like the sub-derivation tree of "N" should not have repetition of any non-terminal that is $|vwx| \leq 2^n$.
- Also as "N" is a non-terminal therefore $|w| \geq 1$.

8.4.1 Lemma:

If a language L is context-free, then there exists some integer $n \geq 1$ (number of non-terminals in G) such that every string s in L that has a length of 2^n or more symbols (i.e. with $|s| \geq 2^n$) can be

written as

$$s = uvwxy,$$

with sub strings u, v, w, x, and y, such that:

1. $|w| \geq 1$
2. $|vwx| \leq 2^n$, and
3. $u.v^i.w.x^i.y \in L$ for all $i \geq 0$.

Example: Consider a language $L = \{0^n 1^n 2^n | n \geq 0\}$ and $\Sigma = \{0, 1, 2\}$. Now we need to prove that L can not be a CFL.

Suppose it was a CFL having n distinct non-terminals, we define $k = 2^n$ and take a very large string $w = 0^{2k}1^{2k}2^{2k} \in L$.
Now as $|vwx| \leq 2^n$ or $|vwx| \leq k$ so v.w.x can either lie completely in 0^{2k}, 1^{2k} or 2^{2k} or in $0^{2k}1^{2k}$ or $1^{2k}2^{2k}$.

In any case we can can pump the string to get a word which is not in L (any case will lead to increase some but not all alphabets in that word so it won't be in L) but it should be in L, therefore L is not a CFL.

8.5 Closure Properties of CFL

In this section we will look at closure properties of Context Free Languages.

8.5.1 Union & Concatenation

Context Free Languages are closed under Union.

Let L_1, L_2 be two Context Free Languages.
Their Context Free Grammars will be like,
$S_1 = ...$, and $S_2 = ...$

To represent $L_1 \cup L_2$, we can just represent its grammar as $S = S_1|S_2$.
Hence, $L_1 \cup L_2$ will be Context Free Language.

Also to represent $L_1.L_2$, we can just represent its grammar as $S = S_1S_2$.
Hence, $L_1.L_2$ will be Context Free Language.

8.5.2 Intersection

Context Free Languages are not closed under intersection.

Lets see this with an example,
Consider languages $L_1 = \{0^n 1^n 2^k | n, k \geq 0\}$ and $L_2 = \{0^k 1^n 2^n | n, k \geq 0\}$.

Both L_1 and L_2 are Context Free Languages.
$L_1 \cap L_2 = \{0^n 1^n 2^n | n \geq 0\}$, which we saw above is not a Context Free Language.
Hence, Context Free Languages are not closed under intersection.

8.5.3 Complement

Suppose CFLs were closed under complementation. Then for any two CFLs L_1, L_2, we have $\overline{L_1}$ and $\overline{L_2}$ are CFLs. Then, since CFLs are closed under union, $\overline{L_1} \cup \overline{L_2}$ is CFL.
Then, again by hypothesis, $\overline{\overline{L_1} \cup \overline{L_2}}$ is CFL. i.e., $L_1 \cap L_2$ is a CFL i.e., CFLs are closed under intersection. which is Contradiction! Thus CFLs are not closed under Complement.
Another example let L = {x | x not of the form ww} is a CFL.
But $\overline{L} = \{ww | w \in \{a, b\}\}$ which is not a CFL. Thus CFLs are not closed under complement.

8.5.4 Substitution

Context Free Languages are closed under Substitution, this means that replacing any symbol with a Language and the result will still be a CFL.
Given grammar G: $S \to 0S0 \mid 1S1 \mid \varepsilon$ Substituting h: $0 \to aba$ and $1 \to bb$
Rules of G' such that $L(G') = L(h(L(G)))$:

$$S \to X0SX0 \mid X1SX1 \mid \varepsilon$$

$$X0 \to aba$$

$$X1 \to bb$$

Thus, CFLs are closed under substitution.

Chapter 9

Turing Machine

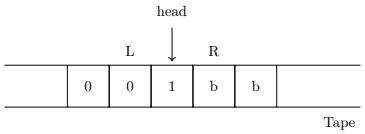


Figure 9.1: Turing Machine

Turing machines are a fundamental model of computation that can simulate the logic of any computer algorithm, regardless of complexity. They are composed of:

1. **Tape:** The machine has an infinitely long tape divided into cells, each of which can contain a symbol from a finite alphabet.
2. **Sybmols:** The tape cells contain symbols from a set. b is used to represent 'black' cell (kind of like NULL).
3. **Head:** The machine has a head that can read and write symbols on the tape and move the tape left or right one cell at a time.
4. **Transition function:** The machine uses a transition function that dictates the machine's actions based on its current state and the symbol it reads on the tape. Actions include writing a symbol, moving the tape left or right, and changing the state.
   - $0/1, L$: Read 0, replace with 1, Move to the left.
   - $0/1, S$: Read 0, replace with 1, stay there.
   - $0/1, R$: Read 0, replace with 1, Move to the right.
   - $-/0, S$: no matter what's present at head, replace with 0, stay there.
     This action is similar to pushing into a stack.
   - $0/b, L$: here b is the blank symbol kept in every cell initially(means the tape cell is free).
     This action is similar to poping from a stack.

We can minic a stack by the following interconversion of moves:

1. Pushing into the stack is equivalent to moving to the right and writing the symbols to be pushed on the tape.
2. Popping off the stack is equivalent to moving to the left and writing black symbols to the tape.

The problems which can be solved using a deterministic turing machine along with a polynomial number of cells in the tape compared to the length of the input are in P and those which can be solved similarly but using non-deterministic turing machine are in NP.

9.1 Configuration of the Turing Machine

At any point when the Turing machine is running, we need to know which cell is the tape head and which state the Turing machine is in. If the finite string we gave as **input** is $X_1X_2.....X_n$ and the **tape head** at an instant is X_i, and the turning machine is in **state 'q'**. Then, we **split the input string into two parts**, the left is from the first element of the string till the element before the tape head and the right part is from the tape head to the last element.
The **configuration** is represented as $\mathbf{X_1X_2...X_{i-1}qX_iX_{i+1}...X_n}$.

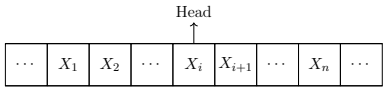


Figure 9.2: Turing Machine Tape

Let us understand more through an example:
Suppose we have been given the string **011010␣0** (where ␣ is blank), the tape head at an instant is 1 at the third position, and the turning machine is in state q_5. Therefore, the configuration is $\mathbf{01q_51010␣0}$.

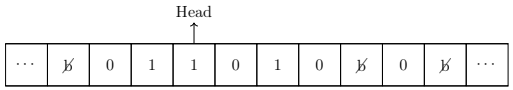


Figure 9.3: Example1 - Turing Machine Tape

Note: Here the '␣' before the first 0 and the '␣' after the last zero are to represent the blanks initialized in the tape and aren't part of the input string.

Now let us see what happens:

- The Turing Machine **scans cell '1'** (the tape head)
- The machine replaces the cell's content with 0 according to the transition **1/0, L**

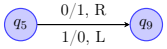


Figure 9.4: Example - Turing Machine State Transition

- The machine transitions to state $\mathbf{q_9}$
- The tape head is **shifted to the left**
- Therefore, our updated configuration is: $\mathbf{0q_91010\not b0}$.

For simplicity, let us call the left part of the input string α and the right part β and say the state is q_i, so we can represent the configuration as $\boldsymbol{\alpha q_i \beta}$

If the configuration changes from $\boldsymbol{\alpha_0 q_0 \beta_0}$ to $\boldsymbol{\alpha_1 q_1 \beta_1}$ and then to $\boldsymbol{\alpha_2 q_2 \beta_2}$.
This is depicted as

$$\boldsymbol{\alpha_0 q_0 \beta_0} \vdash \boldsymbol{\alpha_1 q_1 \beta_1} \vdash \boldsymbol{\alpha_2 q_2 \beta_2}$$

If $\boldsymbol{\alpha_0\, q_0\, \beta_0} \overset{*}{\vdash} \boldsymbol{\alpha_i\, \textbf{done}\, \beta_i}$, we have reached the end of the computation.
Note: '*' on $\vdash$ indicates multiple steps.

`Let us look at another example to understand halting:`

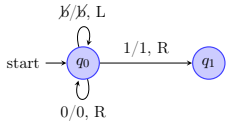


Figure 9.5: Example 2 - Turing Machine State Transition Diagram

Let us look at the computation of the string **0** where the initial configuration is $\boldsymbol{\not b\, q_0\, 0}$.

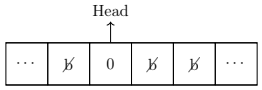


Figure 9.6: Example 2 - Turing Machine Tape (1)

Now the Turing machine scans the cell '0', replaces it with '0', moves right (to a blank cell), and stays in the same state q_0. So now the configuration becomes $\boldsymbol{0\, q_0\, \not b}$.

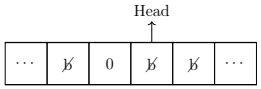


Figure 9.7: Example 2 - Turing Machine Tape (2)

This time, the Turing machine scans the cell '$\not b$', replaces it with '$\not b$', moves left (to '0'), and stays in the same state q_0, because of which the configuration changes to $\boldsymbol{\not b\, q_0\, 0}$.

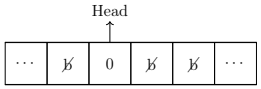


Figure 9.8: Example 2 - Turing Machine Tape (3)

As we can see, the **machine does not halt** on this input string.

Now, consider the input string **1** where the initial configuration is $\boldsymbol{\not b\, q_0\, 1}$.

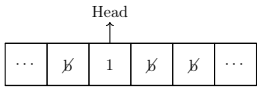


Figure 9.9: Example 3 - Turing Machine Tape (1)

The Turing machine scans the cell '1', replaces it with '1' moves right, and transitions to state q_1. Hence the configuration switches to $\boldsymbol{1\, q_1\, \not b}$.

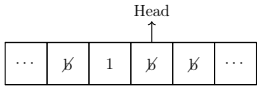


Figure 9.10: Example 3 - Turing Machine Tape (2)

As we know from the transition diagram of the Turing machine, we can't go to any other state of q_1. So the input string **1** halts on this machine whereas **0** doesn't.

We can notice that certain strings halt when processed by a Turing Machine, while others keep don't. This concept helps us define what it means for a Turing Machine to "accept" a string.
In essence, **Turing machines provide us with a way to describe different languages**.

9.2 Language described by a TM

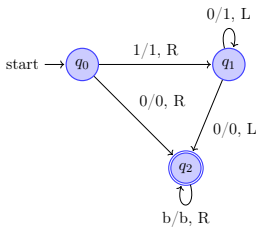
- As described above, we can use a TM to describe a set of strings based on termination. This will form the language of a TM. [1]
- We can also talk about language a TM accepts using the notion of accepting and non-accepting states. In this case, we will accept a string if the TM ever enters an accepting state during its operation on the string as input.
- However, we will primarily be interested in acceptance by halting.

9.2.1 Inter conversion between acceptance by final state and by halting:

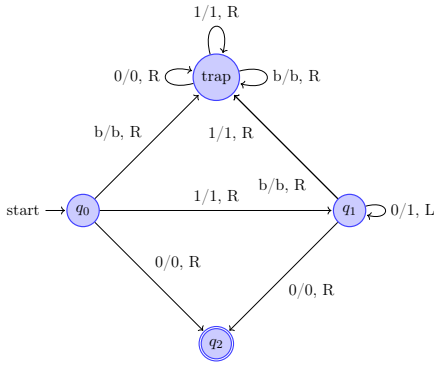
To summarize, the following are the two notions of acceptance of a string by a Turing Machine(in both cases, the input is the standard string input described above):

- **Acceptance by Halting:** All those strings are accepted for whom the Turing Machine halts its operation in finite time. We will primarily be interested in this kind of acceptance.
- **Acceptance of Final State:** All those strings are accepted for whom during the operation of the Turing Machine, a final state is reached at some point in the journey.

It is not very difficult to convert the notion of acceptance by final state to that by halting. Once you reach a final state, move on to a state whose only job is to finish parsing (similar to emptying a stack with a PDA). Moreover, we must also ensure that the TM does not halt in a non-accepting state. First, let us consider when a TM would halt on a non-accepting state:- when the TM has no outgoing transition for a particular action/input:



[1] As a side note, consider the string **0 1 0**. The behavior of termination of TM described above using this string as input depends on where you start the tail from. So, we will adopt a standard to describe the language described by a TM by assuming that the string we have taken as input lies on the tail and head is at the leftmost character of the string. If the TM halts taking this configuration as the starting configuration, then we say that the string lies in the language of the TM.



So, the process for conversion seems straightforward now: just remove the transitions going out of the final state (in this case, we removed the loop of the final state), and if some string halts on reaching some other state, make the state non-halting using trap state.

A slight note: The typical notion for acceptance by final state involves accepting the string only if after reading the entire string, you reach an accepting state. However, we have a minor modification that if you reach a final state at some point in the journey, read the rest of the string and stay in the same state. If this modification had been considered in previous things we studied, such as DFA, then it would have created a problem since that would mean all strings with their prefix as the smallest string accepted by the DFA would be accepted (there can be other strings also, but at least these would be accepted). In the case of a TM, this does not cause a problem due to 2 reasons:

- A TM can move left as well as right on the tape.
- A TM not only parses the string but also overwrites it.

So, in the case of a TM, it is not necessary that strings formed by concatenating something with an accepting prefix will be accepted.

$$B \rightarrow CB \mid C$$
$$C \rightarrow Disk$$

In this special case, because the expression that A derives is just a concatenation of variables, and *Disk* is a single variable, we actually have no need for the variables A or C. We could use the following productions instead:

$$Pc \rightarrow Model\ Price\ Processor\ Ram\ B$$
$$B \rightarrow Disk\ B \mid Disk$$

□

5.3.5 Exercises for Section 5.3

Exercise 5.3.1: Prove that if a string of parentheses is balanced, in the sense given in Example 5.19, then it is generated by the grammar $B \rightarrow BB \mid (B) \mid \epsilon$. *Hint*: Perform an induction on the length of the string.

* **Exercise 5.3.2:** Consider the set of all strings of balanced parentheses of two types, round and square. An example of where these strings come from is as follows. If we take expressions in C, which use round parentheses for grouping and for arguments of function calls, and use square brackets for array indexes, and drop out everything but the parentheses, we get all strings of balanced parentheses of these two types. For example,

```
f(a[i]*(b[i][j],c[g(x)]),d[i])
```

becomes the balanced-parenthesis string `([]([][][()])[])`. Design a grammar for all and only the strings of round and square parentheses that are balanced.

! **Exercise 5.3.3:** In Section 5.3.1, we considered the grammar

$$S \rightarrow \epsilon \mid SS \mid iS \mid iSeS$$

and claimed that we could test for membership in its language L by repeatedly doing the following, starting with a string w. The string w changes during repetitions.

1. If the current string begins with e, fail; w is not in L.
2. If the string currently has no e's (it may have i's), succeed; w is in L.
3. Otherwise, delete the first e and the i immediately to its left. Then repeat these three steps on the new string.

Prove that this process correctly identifies the strings in L.

Exercise 5.3.4: Add the following forms to the HTML grammar of Fig. 5.13:

* a) A list item must be ended by a closing tag `</LI>`.

b) An element can be an unordered list, as well as an ordered list. Unordered lists are surrounded by the tag `<UL>` and its closing `</UL>`.

! c) An element can be a table. Tables are surrounded by `<TABLE>` and its closer `</TABLE>`. Inside these tags are one or more rows, each of which is surrounded by `<TR>` and `</TR>`. The first row is the header, with one or more fields, each introduced by the `<TH>` tag (we'll assume these are not closed, although they should be). Subsequent rows have their fields introduced by the `<TD>` tag.

```
<!DOCTYPE CourseSpecs [
    <!ELEMENT COURSES (COURSE+)>
    <!ELEMENT COURSE (CNAME, PROF, STUDENT*, TA?)>
    <!ELEMENT CNAME (#PCDATA)>
    <!ELEMENT PROF (#PCDATA)>
    <!ELEMENT STUDENT (#PCDATA)>
    <!ELEMENT TA (#PCDATA)> ]>
```

Figure 5.16: A DTD for courses

Exercise 5.3.5: Convert the DTD of Fig. 5.16 to a context-free grammar.

5.4 Ambiguity in Grammars and Languages

As we have seen, applications of CFG's often rely on the grammar to provide the structure of files. For instance, we saw in Section 5.3 how grammars can be used to put structure on programs and documents. The tacit assumption was that a grammar uniquely determines a structure for each string in its language. However, we shall see that not every grammar does provide unique structures.

When a grammar fails to provide unique structures, it is sometimes possible to redesign the grammar to make the structure unique for each string in the language. Unfortunately, sometimes we cannot do so. That is, there are some CFL's that are "inherently ambiguous"; every grammar for the language puts more than one structure on some strings in the language.

5.4.1 Ambiguous Grammars

Let us return to our running example: the expression grammar of Fig. 5.2. This grammar lets us generate expressions with any sequence of $*$ and $+$ operators, and the productions $E \rightarrow E + E \mid E * E$ allow us to generate these expressions in any order we choose.

Example 5.25: For instance, consider the sentential form $E + E * E$. It has two derivations from E:

1. $E \Rightarrow E + E \Rightarrow E + E * E$
2. $E \Rightarrow E * E \Rightarrow E + E * E$

Notice that in derivation (1), the second E is replaced by $E * E$, while in derivation (2), the first E is replaced by $E + E$. Figure 5.17 shows the two parse trees, which we should note are distinct trees.

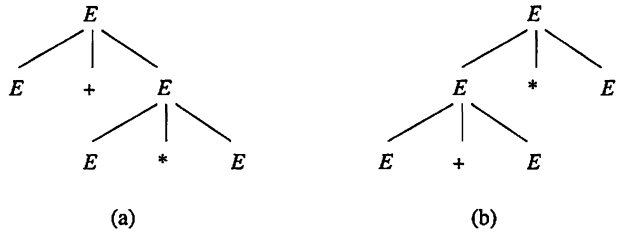


Figure 5.17: Two parse trees with the same yield

The difference between these two derivations is significant. As far as the structure of the expressions is concerned, derivation (1) says that the second and third expressions are multiplied, and the result is added to the first expression, while derivation (2) adds the first two expressions and multiplies the result by the third. In more concrete terms, the first derivation suggests that $1 + 2 * 3$ should be grouped $1 + (2 * 3) = 7$, while the second derivation suggests the same expression should be grouped $(1 + 2) * 3 = 9$. Obviously, the first of these, and not the second, matches our notion of correct grouping of arithmetic expressions.

Since the grammar of Fig. 5.2 gives two different structures to any string of terminals that is derived by replacing the three expressions in $E + E * E$ by identifiers, we see that this grammar is not a good one for providing unique structure. In particular, while it can give strings the correct grouping as arithmetic expressions, it also gives them incorrect groupings. To use this expression grammar in a compiler, we would have to modify it to provide only the correct groupings. □

On the other hand, the mere existence of different derivations for a string (as opposed to different parse trees) does not imply a defect in the grammar. The following is an example.

Example 5.26: Using the same expression grammar, we find that the string $a + b$ has many different derivations. Two examples are:

1. $E \Rightarrow E + E \Rightarrow I + E \Rightarrow a + E \Rightarrow a + I \Rightarrow a + b$
2. $E \Rightarrow E + E \Rightarrow E + I \Rightarrow I + I \Rightarrow I + b \Rightarrow a + b$

However, there is no real difference between the structures provided by these derivations; they each say that a and b are identifiers, and that their values are to be added. In fact, both of these derivations produce the same parse tree if the construction of Theorems 5.18 and 5.12 are applied. □

The two examples above suggest that it is not a multiplicity of derivations that cause ambiguity, but rather the existence of two or more parse trees. Thus, we say a CFG $G = (V, T, P, S)$ is *ambiguous* if there is at least one string w in T^* for which we can find two different parse trees, each with root labeled S and yield w. If each string has at most one parse tree in the grammar, then the grammar is *unambiguous*.

For instance, Example 5.25 almost demonstrated the ambiguity of the grammar of Fig. 5.2. We have only to show that the trees of Fig. 5.17 can be completed to have terminal yields. Figure 5.18 is an example of that completion.

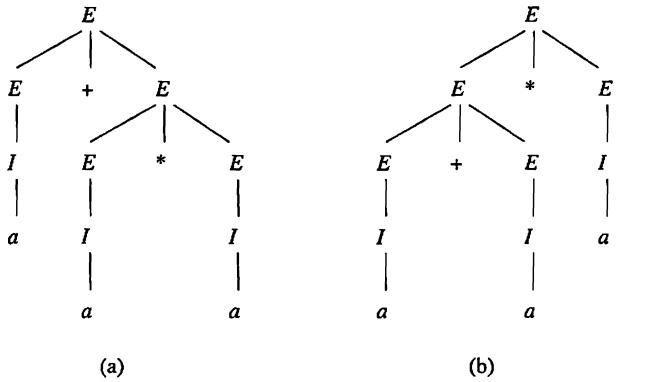


Figure 5.18: Trees with yield $a + a * a$, demonstrating the ambiguity of our expression grammar

5.4.2 Removing Ambiguity From Grammars

In an ideal world, we would be able to give you an algorithm to remove ambiguity from CFG's, much as we were able to show an algorithm in Section 4.4 to remove unnecessary states of a finite automaton. However, the surprising fact is, as we shall show in Section 9.5.2, that there is no algorithm whatsoever that can even tell us whether a CFG is ambiguous in the first place. Moreover, we

Ambiguity Resolution in YACC

If the expression grammar we have been using is ambiguous, we might wonder whether the sample YACC program of Fig. 5.11 is realistic. True, the underlying grammar is ambiguous, but much of the power of the YACC parser-generator comes from providing the user with simple mechanisms for resolving most of the common causes of ambiguity. For the expression grammar, it is sufficient to insist that:

a) $*$ takes precedence over $+$. That is, $*$'s must be grouped before adjacent $+$'s on either side. This rule tells us to use derivation (1) in Example 5.25, rather than derivation (2).

b) Both $*$ and $+$ are left-associative. That is, group sequences of expressions, all of which are connected by $*$, from the left, and do the same for sequences connected by $+$.

YACC allows us to state the precedence of operators by listing them in order, from lowest to highest precedence. Technically, the precedence of an operator applies to the use of any production of which that operator is the rightmost terminal in the body. We can also declare operators to be left- or right-associative with the keywords `%left` and `%right`. For instance, to declare that $+$ and $*$ were both left associative, with $*$ taking precedence over $+$, we would put ahead of the grammar of Fig. 5.11 the statements:

```
%left '+'
%left '*'
```

shall see in Section 5.4.4 that there are context-free languages that have nothing but ambiguous CFG's; for these languages, removal of ambiguity is impossible.

Fortunately, the situation in practice is not so grim. For the sorts of constructs that appear in common programming languages, there are well-known techniques for eliminating ambiguity. The problem with the expression grammar of Fig. 5.2 is typical, and we shall explore the elimination of its ambiguity as an important illustration.

First, let us note that there are two causes of ambiguity in the grammar of Fig. 5.2:

1. The precedence of operators is not respected. While Fig. 5.17(a) properly groups the $*$ before the $+$ operator, Fig 5.17(b) is also a valid parse tree and groups the $+$ ahead of the $*$. We need to force only the structure of Fig. 5.17(a) to be legal in an unambiguous grammar.

2. A sequence of identical operators can group either from the left or from the right. For example, if the $*$'s in Fig. 5.17 were replaced by $+$'s, we would see two different parse trees for the string $E + E + E$. Since addition and multiplication are associative, it doesn't matter whether we group from the left or the right, but to eliminate ambiguity, we must pick one. The conventional approach is to insist on grouping from the left, so the structure of Fig. 5.17(b) is the only correct grouping of two $+$-signs.

The solution to the problem of enforcing precedence is to introduce several different variables, each of which represents those expressions that share a level of "binding strength." Specifically:

1. A *factor* is an expression that cannot be broken apart by any adjacent operator, either a $*$ or a $+$. The only factors in our expression language are:
   (a) Identifiers. It is not possible to separate the letters of an identifier by attaching an operator.
   (b) Any parenthesized expression, no matter what appears inside the parentheses. It is the purpose of parentheses to prevent what is inside from becoming the operand of any operator outside the parentheses.
2. A *term* is an expression that cannot be broken by the $+$ operator. In our example, where $+$ and $*$ are the only operators, a term is a product of one or more factors. For instance, the term $a * b$ can be "broken" if we use left associativity and place $a1*$ to its left. That is, $a1 * a * b$ is grouped $(a1 * a) * b$, which breaks apart the $a * b$. However, placing an additive term, such as $a1+$, to its left or $+a1$ to its right cannot break $a * b$. The proper grouping of $a1 + a * b$ is $a1 + (a * b)$, and the proper grouping of $a * b + a1$ is $(a * b) + a1$.
3. An *expression* will henceforth refer to any possible expression, including those that can be broken by either an adjacent $*$ or an adjacent $+$. Thus, an expression for our example is a sum of one or more terms.

$$
\begin{array}{lcl}
I & \rightarrow & a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\
F & \rightarrow & I \mid (E) \\
T & \rightarrow & F \mid T * F \\
E & \rightarrow & T \mid E + T
\end{array}
$$

Figure 5.19: An unambiguous expression grammar

Example 5.27: Figure 5.19 shows an unambiguous grammar that generates the same language as the grammar of Fig. 5.2. Think of F, T, and E as the

variables whose languages are the factors, terms, and expressions, as defined above. For instance, this grammar allows only one parse tree for the string $a + a * a$; it is shown in Fig. 5.20.

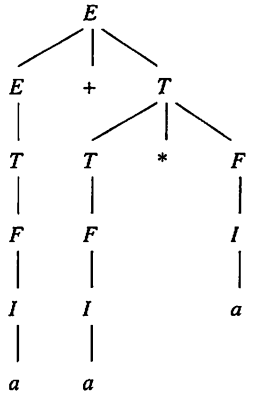


Figure 5.20: The sole parse tree for $a + a * a$

The fact that this grammar is unambiguous may be far from obvious. Here are the key observations that explain why no string in the language can have two different parse trees.

- Any string derived from T, a term, must be a sequence of one or more factors, connected by $*$'s. A factor, as we have defined it, and as follows from the productions for F in Fig. 5.19, is either a single identifier or any parenthesized expression.
- Because of the form of the two productions for T, the only parse tree for a sequence of factors is the one that breaks $f_1 * f_2 * \cdots * f_n$, for $n > 1$ into a term $f_1 * f_2 * \cdots * f_{n-1}$ and a factor f_n. The reason is that F cannot derive expressions like $f_{n-1} * f_n$ without introducing parentheses around them. Thus, it is not possible that when using the production $T \to T * F$, the F derives anything but the last of the factors. That is, the parse tree for a term can only look like Fig. 5.21.
- Likewise, an expression is a sequence of terms connected by $+$. When we use the production $E \to E + T$ to derive $t_1 + t_2 + \cdots + t_n$, the T must derive only t_n, and the E in the body derives $t_1 + t_2 + \cdots + t_{n-1}$. The reason, again, is that T cannot derive the sum of two or more terms without putting parentheses around them.

□

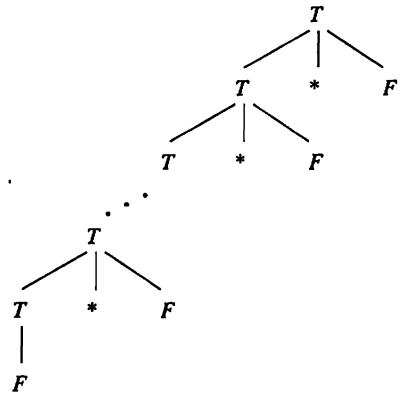


Figure 5.21: The form of all parse trees for a term

5.4.3 Leftmost Derivations as a Way to Express Ambiguity

While derivations are not necessarily unique, even if the grammar is unambiguous, it turns out that, in an unambiguous grammar, leftmost derivations will be unique, and rightmost derivations will be unique. We shall consider leftmost derivations only, and state the result for rightmost derivations.

Example 5.28: As an example, notice the two parse trees of Fig. 5.18 that each yield $E + E * E$. If we construct leftmost derivations from them we get the following leftmost derivations from trees (a) and (b), respectively:

a) $E \underset{lm}{\Rightarrow} E+E \underset{lm}{\Rightarrow} I+E \underset{lm}{\Rightarrow} a+E \underset{lm}{\Rightarrow} a+E*E \underset{lm}{\Rightarrow} a+I*E \underset{lm}{\Rightarrow} a+a*E \underset{lm}{\Rightarrow} a+a*I \underset{lm}{\Rightarrow} a+a*a$

b) $E \underset{lm}{\Rightarrow} E*E \underset{lm}{\Rightarrow} E+E*E \underset{lm}{\Rightarrow} I+E*E \underset{lm}{\Rightarrow} a+E*E \underset{lm}{\Rightarrow} a+I*E \underset{lm}{\Rightarrow} a+a*E \underset{lm}{\Rightarrow} a+a*I \underset{lm}{\Rightarrow} a+a*a$

Note that these two leftmost derivations differ. This example does not prove the theorem, but demonstrates how the differences in the trees force different steps to be taken in the leftmost derivation. □

Theorem 5.29: For each grammar $G = (V, T, P, S)$ and string w in T^*, w has two distinct parse trees if and only if w has two distinct leftmost derivations from S.

PROOF: (Only-if) If we examine the construction of a leftmost derivation from a parse tree in the proof of Theorem 5.14, we see that wherever the two parse trees first have a node at which different productions are used, the leftmost derivations constructed will also use different productions and thus be different derivations.

(If) While we have not previously given a direct construction of a parse tree from a leftmost derivation, the idea is not hard. Start constructing a tree with only the root, labeled S. Examine the derivation one step at a time. At each step, a variable will be replaced, and this variable will correspond to the leftmost node in the tree being constructed that has no children but that has a variable as its label. From the production used at this step of the leftmost derivation, determine what the children of this node should be. If there are two distinct derivations, then at the first step where the derivations differ, the nodes being constructed will get different lists of children, and this difference guarantees that the parse trees are distinct. □

5.4.4 Inherent Ambiguity

A context-free language L is said to be *inherently ambiguous* if all its grammars are ambiguous. If even one grammar for L is unambiguous, then L is an unambiguous language. We saw, for example, that the language of expressions generated by the grammar of Fig. 5.2 is actually unambiguous. Even though that grammar is ambiguous, there is another grammar for the same language that is unambiguous — the grammar of Fig. 5.19.

We shall not prove that there are inherently ambiguous languages. Rather we shall discuss one example of a language that can be proved inherently ambiguous, and we shall explain intuitively why every grammar for the language must be ambiguous. The language L in question is:

$$L = \{a^n b^n c^m d^m \mid n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}$$

That is, L consists of strings in $\mathbf{a^+b^+c^+d^+}$ such that either:

1. There are as many a's as b's and as many c's as d's, or
2. There are as many a's as d's and as many b's as c's.

L is a context-free language. The obvious grammar for L is shown in Fig. 5.22. It uses separate sets of productions to generate the two kinds of strings in L.

This grammar is ambiguous. For example, the string *aabbccdd* has the two leftmost derivations:

1. $S \underset{lm}{\Rightarrow} AB \underset{lm}{\Rightarrow} aAbB \underset{lm}{\Rightarrow} aabbB \underset{lm}{\Rightarrow} aabbcBd \underset{lm}{\Rightarrow} aabbccdd$
2. $S \underset{lm}{\Rightarrow} C \underset{lm}{\Rightarrow} aCd \underset{lm}{\Rightarrow} aaDdd \underset{lm}{\Rightarrow} aabDcdd \underset{lm}{\Rightarrow} aabbccdd$

$$\begin{array}{lcl} S & \rightarrow & AB \mid C \\ A & \rightarrow & aAb \mid ab \\ B & \rightarrow & cBd \mid cd \\ C & \rightarrow & aCd \mid aDd \\ D & \rightarrow & bDc \mid bc \end{array}$$

Figure 5.22: A grammar for an inherently ambiguous language

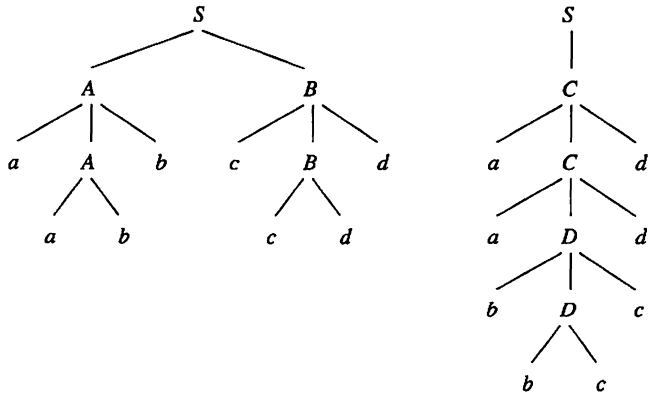


Figure 5.23: Two parse trees for *aabbccdd*

and the two parse trees shown in Fig. 5.23.

The proof that all grammars for L must be ambiguous is complex. However, the essence is as follows. We need to argue that all but a finite number of the strings whose counts of the four symbols a, b, c, and d, are all equal must be generated in two different ways: one in which the a's and b are generated to be equal and the c's and d's are generated to be equal, and a second way, where the a's and d's are generated to be equal and likewise the b's and c's.

For instance, the only way to generate strings where the a's and b's have the same number is with a variable like A in the grammar of Fig. 5.22. There are variations, of course, but these variations do not change the basic picture. For instance:

- Some small strings can be avoided, say by changing the basis production $A \rightarrow ab$ to $A \rightarrow aaabbb$, for instance.

- We could arrange that A shares its job with some other variables, e.g., by using variables A_1 and A_2, with A_1 generating the odd numbers of a's and A_2 generating the even numbers, as: $A_1 \to aA_2b \mid ab$; $A_2 \to aA_1b \mid ab$.
- We could also arrange that the numbers of a's and b's generated by A are not exactly equal, but off by some finite number. For instance, we could start with a production like $S \to AbB$ and then use $A \to aAb \mid a$ to generate one more a than b's.

However, we cannot avoid some mechanism for generating a's in a way that matches the count of b's.

Likewise, we can argue that there must be a variable like B that generates matching c's and d's. Also, variables that play the roles of C (generate matching a's and d's) and D (generate matching b's and c's) must be available in the grammar. The argument, when formalized, proves that no matter what modifications we make to the basic grammar, it will generate at least *some* of the strings of the form $a^nb^nc^nd^n$ in the two ways that the grammar of Fig. 5.22 does.

5.4.5 Exercises for Section 5.4

* **Exercise 5.4.1:** Consider the grammar

$$S \to aS \mid aSbS \mid \epsilon$$

This grammar is ambiguous. Show in particular that the string aab has two:

a) Parse trees.

b) Leftmost derivations.

c) Rightmost derivations.

! **Exercise 5.4.2:** Prove that the grammar of Exercise 5.4.1 generates all and only the strings of a's and b's such that every prefix has at least as many a's as b's.

*! **Exercise 5.4.3:** Find an unambiguous grammar for the language of Exercise 5.4.1.

!! **Exercise 5.4.4:** Some strings of a's and b's have a unique parse tree in the grammar of Exercise 5.4.1. Give an efficient test to tell whether a given string is one of these. The test "try all parse trees to see how many yield the given string" is not adequately efficient.

! **Exercise 5.4.5:** This question concerns the grammar from Exercise 5.1.2, which we reproduce here:

$$\begin{aligned} S &\to A1B \\ A &\to 0A \mid \epsilon \\ B &\to 0B \mid 1B \mid \epsilon \end{aligned}$$

a) Show that this grammar is unambiguous.

b) Find a grammar for the same language that *is* ambiguous, and demonstrate its ambiguity.

*! **Exercise 5.4.6:** Is your grammar from Exercise 5.1.5 unambiguous? If not, redesign it to be unambiguous.

Exercise 5.4.7: The following grammar generates *prefix* expressions with operands x and y and binary operators $+$, $-$, and $*$:

$$E \to +EE \mid *EE \mid -EE \mid x \mid y$$

a) Find leftmost and rightmost derivations, and a derivation tree for the string `+*-xyxy`.

! b) Prove that this grammar is unambiguous.

5.5 Summary of Chapter 5

- ✦ *Context-Free Grammars*: A CFG is a way of describing languages by recursive rules called productions. A CFG consists of a set of variables, a set of terminal symbols, and a start variable, as well as the productions. Each production consists of a head variable and a body consisting of a string of zero or more variables and/or terminals.
- ✦ *Derivations and Languages*: Beginning with the start symbol, we derive terminal strings by repeatedly replacing a variable by the body of some production with that variable in the head. The language of the CFG is the set of terminal strings we can so derive; it is called a context-free language.
- ✦ *Leftmost and Rightmost Derivations*: If we always replace the leftmost (resp. rightmost) variable in a string, then the resulting derivation is a leftmost (resp. rightmost) derivation. Every string in the language of a CFG has at least one leftmost and at least one rightmost derivation.
- ✦ *Sentential Forms*: Any step in a derivation is a string of variables and/or terminals. We call such a string a sentential form. If the derivation is leftmost (resp. rightmost), then the string is a left- (resp. right-) sentential form.

- *Parse Trees*: A parse tree is a tree that shows the essentials of a derivation. Interion nodes are labeled by variables, and leaves are labeled by terminals or ϵ. For each internal node, there must be a production such that the head of the production is the label of the node, and the labels of its children, read from left to right, form the body of that production.
- *Equivalence of Parse Trees and Derivations*: A terminal string is in the language of a grammar if and only if it is the yield of at least one parse tree. Thus, the existence of leftmost derivations, rightmost derivations, and parse trees are equivalent conditions that each define exactly the strings in the language of a CFG.
- *Ambiguous Grammars*: For some CFG's, it is possible to find a terminal string with more than one parse tree, or equivalently, more than one leftmost derivation or more than one rightmost derivation. Such a grammar is called ambiguous.
- *Eliminating Ambiguity*: For many useful grammars, such as those that describe the structure of programs in a typical programming language, it is possible to find an unambiguous grammar that generates the same language. Unfortunately, the unambiguous grammar is frequently more complex than the simplest ambiguous grammar for the language. There are also some context-free languages, usually quite contrived, that are inherently ambiguous, meaning that every grammar for that language is ambiguous.
- *Parsers*: The context-free grammar is an essential concept for the implementation of compilers and other programming-language processors. Tools such as YACC take a CFG as input and produce a parser, the component of a compiler that deduces the structure of the program being compiled.
- *Document Type Definitions*: The emerging XML standard for sharing information through Web documents has a notation, called the DTD, for describing the structure of such documents, through the nesting of semantic tags within the document. The DTD is in essence a context-free grammar whose language is a class of related documents.

5.6 References for Chapter 5

The context-free grammar was first proposed as a description method for natural languages by Chomsky [4]. A similar idea was used shortly thereafter to describe computer languages — Fortran by Backus [2] and Algol by Naur [7]. As a result, CFG's are sometimes referred to as "Backus-Naur form grammars."

Ambiguity in grammars was identified as a problem by Cantor [3] and Floyd [5] at about the same time. Inherent ambiguity was first addressed by Gross [6].

For applications of CFG's in compilers, see [1]. DTD's are defined in the standards document for XML [8].

1. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading MA, 1986.
2. J. W. Backus, "The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference," *Proc. Intl. Conf. on Information Processing* (1959), UNESCO, pp. 125–132.
3. D. C. Cantor, "On the ambiguity problem of Backus systems," *J. ACM* **9**:4 (1962), pp. 477–479.
4. N. Chomsky, "Three models for the description of language," *IRE Trans. on Information Theory* **2**:3 (1956), pp. 113–124.
5. R. W. Floyd, "On ambiguity in phrase-structure languages," *Comm. ACM* **5**:10 (1962), pp. 526–534.
6. M. Gross, "Inherent ambiguity of minimal linear grammars," *Information and Control* **7**:3 (1964), pp. 366–368.
7. P. Naur et al., "Report on the algorithmic language ALGOL 60," *Comm. ACM* **3**:5 (1960), pp. 299–314. See also *Comm. ACM* **6**:1 (1963), pp. 1–17.
8. World-Wide-Web Consortium, `http://www.w3.org/TR/REC-xml` (1998).

ID's for Finite Automata?

One might wonder why we did not introduce for finite automata a notation like the ID's we use for PDA's. Although a FA has no stack, we could use a pair (q, w), where q is the state and w the remaining input, as the ID of a finite automaton.

While we could have done so, we would not glean any more information from reachability among ID's than we obtain from the $\hat{\delta}$ notation. That is, for any finite automaton, we could show that $\hat{\delta}(q, w) = p$ if and only if $(q, wx) \vdash^* (p, x)$ for all strings x. The fact that x can be anything we wish without influencing the behavior of the FA is a theorem analogous to Theorems 6.5 and 6.6.

Theorem 6.6: If $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is a PDA, and

$$(q, xw, \alpha) \vdash_P^* (p, yw, \beta)$$

then it is also true that $(q, x, \alpha) \vdash_P^* (p, y, \beta)$. □

6.1.5 Exercises for Section 6.1

Exercise 6.1.1: Suppose the PDA $P = (\{q, p\}, \{0, 1\}, \{Z_0, X\}, \delta, q, Z_0, \{p\})$ has the following transition function:

1. $\delta(q, 0, Z_0) = \{(q, XZ_0)\}$.
2. $\delta(q, 0, X) = \{(q, XX)\}$.
3. $\delta(q, 1, X) = \{(q, X)\}$.
4. $\delta(q, \epsilon, X) = \{(p, \epsilon)\}$.
5. $\delta(p, \epsilon, X) = \{(p, \epsilon)\}$.
6. $\delta(p, 1, X) = \{(p, XX)\}$.
7. $\delta(p, 1, Z_0) = \{(p, \epsilon)\}$.

Starting from the initial ID (q, w, Z_0), show all the reachable ID's when the input w is:

* a) 01.

b) 0011.

c) 010.

6.2 The Languages of a PDA

We have assumed that a PDA accepts its input by consuming it and entering an accepting state. We call this approach "acceptance by final state." There is a second approach to defining the language of a PDA that has important applications. We may also define for any PDA the language "accepted by empty stack," that is, the set of strings that cause the PDA to empty its stack, starting from the initial ID.

These two methods are equivalent, in the sense that a language L has a PDA that accepts it by final state if and only if L has a PDA that accepts it by empty stack. However, for a given PDA P, the languages that P accepts by final state and by empty stack are usually different. We shall show in this section how to convert a PDA accepting L by final state into another PDA that accepts L by empty stack, and vice-versa.

6.2.1 Acceptance by Final State

Let $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a PDA. Then $L(P)$, the *language accepted by P by final state*, is

$$\{w \mid (q_0, w, Z_0) \vdash_P^* (q, \epsilon, \alpha)\}$$

for some state q in F and any stack string α. That is, starting in the initial ID with w waiting on the input, P consumes w from the input and enters an accepting state. The contents of the stack at that time is irrelevant.

Example 6.7: We have claimed that the PDA of Example 6.2 accepts the language L_{wwr}, the language of strings in $\{0, 1\}^*$ that have the form ww^R. Let us see why that statement is true. The proof is an if-and-only-if statement: the PDA P of Example 6.2 accepts string x by final state if and only if x is of the form ww^R.

(If) This part is easy; we have only to show the accepting computation of P. If $x = ww^R$, then observe that

$$(q_0, ww^R, Z_0) \vdash^* (q_0, w^R, w^R Z_0) \vdash (q_1, w^R, w^R Z_0) \vdash^* (q_1, \epsilon, Z_0) \vdash (q_2, \epsilon, Z_0)$$

That is, one option the PDA has is to read w from its input and store it on its stack, in reverse. Next, it goes spontaneously to state q_1 and matches w^R on the input with the same string on its stack, and finally goes spontaneously to state q_2.

(Only-if) This part is harder. First, observe that the only way to enter accepting state q_2 is to be in state q_1 and have Z_0 at the top of the stack. Also, any accepting computation of P will start in state q_0, make one transition to q_1, and never return to q_0. Thus, it is sufficient to find the conditions on x such that $(q_0, x, Z_0) \vdash^* (q_1, \epsilon, Z_0)$; these will be exactly the strings x that P accepts by final state. We shall show by induction on $|x|$ the slightly more general statement:

- If $(q_0, x, \alpha) \vdash^* (q_1, \epsilon, \alpha)$, then x is of the form ww^R.

BASIS: If $x = \epsilon$, then x is of the form ww^R (with $w = \epsilon$). Thus, the conclusion is true, so the statement is true. Note we do not have to argue that the hypothesis $(q_0, \epsilon, \alpha) \vdash^* (q_1, \epsilon, \alpha)$ is true, although it is.

INDUCTION: Suppose $x = a_1 a_2 \cdots a_n$ for some $n > 0$. There are two moves that P can make from ID (q_0, x, α):

1. $(q_0, x, \alpha) \vdash (q_1, x, \alpha)$. Now P can only pop the stack when it is in state q_1. P must pop the stack with every input symbol it reads, and $|x| > 0$. Thus, if $(q_1, x, \alpha) \vdash^* (q_1, \epsilon, \beta)$, then β will be shorter than α and cannot be equal to α.

2. $(q_0, a_1 a_2 \cdots a_n, \alpha) \vdash (q_0, a_2 \cdots a_n, a_1 \alpha)$. Now the only way a sequence of moves can end in (q_1, ϵ, α) is if the last move is a pop:

   $$(q_1, a_n, a_1 \alpha) \vdash (q_1, \epsilon, \alpha)$$

   In that case, it must be that $a_1 = a_n$. We also know that

   $$(q_0, a_2 \cdots a_n, a_1 \alpha) \vdash^* (q_1, a_n, a_1 \alpha)$$

   By Theorem 6.6, we can remove the symbol a_n from the end of the input, since it is not used. Thus,

   $$(q_0, a_2 \cdots a_{n-1}, a_1 \alpha) \vdash^* (q_1, \epsilon, a_1 \alpha)$$

   Since the input for this sequence is shorter than n, we may apply the inductive hypothesis and conclude that $a_2 \cdots a_{n-1}$ is of the form yy^R for some y. Since $x = a_1 y y^R a_n$, and we know $a_1 = a_n$, we conclude that x is of the form ww^R; specifically $w = a_1 y$.

The above is the heart of the proof that the only way to accept x is for x to be equal to ww^R for some w. Thus, we have the "only-if" part of the proof, which, with the "if" part proved earlier, tells us that P accepts exactly those strings in L_{wwr}. □

6.2.2 Acceptance by Empty Stack

For each PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, we also define

$$N(P) = \{w \mid (q_0, w, Z_0) \vdash^* (q, \epsilon, \epsilon)\}$$

for any state q. That is, $N(P)$ is the set of inputs w that P can consume and at the same time empty its stack.[2]

[2]The N in $N(P)$ stands for "null stack," a synonym for "empty stack."

Example 6.8: The PDA P of Example 6.2 never empties its stack, so $N(P) = \emptyset$. However, a small modification will allow P to accept L_{wwr} by empty stack as well as by final state. Instead of the transition $\delta(q_1, \epsilon, Z_0) = \{(q_2, Z_0)\}$, use $\delta(q_1, \epsilon, Z_0) = \{(q_2, \epsilon)\}$. Now, P pops the last symbol off its stack as it accepts, and $L(P) = N(P) = L_{wwr}$. □

Since the set of accepting states is irrelevant, we shall sometimes leave off the last (seventh) component from the specification of a PDA P, if all we care about is the language that P accepts by empty stack. Thus, we would write P as a six-tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0)$.

6.2.3 From Empty Stack to Final State

We shall show that the classes of languages that are $L(P)$ for some PDA P is the same as the class of languages that are $N(P)$ for some PDA P. This class is also exactly the context-free languages, as we shall see in Section 6.3. Our first construction shows how to take a PDA P_N that accepts a language L by empty stack and construct a PDA P_F that accepts L by final state.

Theorem 6.9: If $L = N(P_N)$ for some PDA $P_N = (Q, \Sigma, \Gamma, \delta_N, q_0, Z_0)$, then there is a PDA P_F such that $L = L(P_F)$.

PROOF: The idea behind the proof is in Fig. 6.4. We use a new symbol X_0, which must not be a symbol of Γ; X_0 is both the start symbol of P_F and a marker on the bottom of the stack that lets us know when P_N has reached an empty stack. That is, if P_F sees X_0 on top of its stack, then it knows that P_N would empty its stack on the same input.

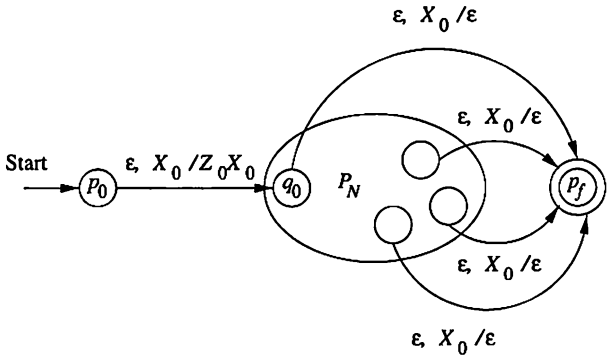


Figure 6.4: P_F simulates P_N and accepts if P_N empties its stack

We also need a new start state, p_0, whose sole function is to push Z_0, the start symbol of P_N, onto the top of the stack and enter state q_0, the start

state of P_N. Then, P_F simulates P_N, until the stack of P_N is empty, which P_F detects because it sees X_0 on the top of the stack. Finally, we need another new state, p_f, which is the accepting state of P_F; this PDA transfers to state p_f whenever it discovers that P_N would have emptied its stack.

The specification of P_F is as follows:

$$P_F = (Q \cup \{p_0, p_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta_F, p_0, X_0, \{p_f\})$$

where δ_F is defined by:

1. $\delta_F(p_0, \epsilon, X_0) = \{(q_0, Z_0X_0)\}$. In its start state, P_F makes a spontaneous transition to the start state of P_N, pushing its start symbol Z_0 onto the stack.

2. For all states q in Q, inputs a in Σ or $a = \epsilon$, and stack symbols Y in Γ, $\delta_F(q, a, Y)$ contains all the pairs in $\delta_N(q, a, Y)$.

3. In addition to rule (2), $\delta_F(q, \epsilon, X_0)$ contains (p_f, ϵ) for every state q in Q.

We must show that w is in $L(P_F)$ if and only if w is in $N(P_N)$.

(If) We are given that $(q_0, w, Z_0) \overset{*}{\underset{P_N}{\vdash}} (q, \epsilon, \epsilon)$ for some state q. Theorem 6.5 lets us insert X_0 at the bottom of the stack and conclude $(q_0, w, Z_0X_0) \overset{*}{\underset{P_N}{\vdash}} (q, \epsilon, X_0)$. Since by rule (2) above, P_F has all the moves of P_N, we may also conclude that $(q_0, w, Z_0X_0) \overset{*}{\underset{P_F}{\vdash}} (q, \epsilon, X_0)$. If we put this sequence of moves together with the initial and final moves from rules (1) and (3) above, we get:

$$(p_0, w, X_0) \underset{P_F}{\vdash} (q_0, w, Z_0X_0) \overset{*}{\underset{P_F}{\vdash}} (q, \epsilon, X_0) \underset{P_F}{\vdash} (p_f, \epsilon, \epsilon) \tag{6.1}$$

Thus, P_F accepts w by final state.

(Only-if) The converse requires only that we observe the additional transitions of rules (1) and (3) give us very limited ways to accept w by final state. We must use rule (3) at the last step, and we can only use that rule if the stack of P_F contains only X_0. No X_0's ever appear on the stack except at the bottommost position. Further, rule (1) is only used at the first step, and it *must* be used at the first step.

Thus, any computation of P_F that accepts w must look like sequence (6.1). Moreover, the middle of the computation — all but the first and last steps — must also be a computation of P_N with X_0 below the stack. The reason is that, except for the first and last steps, P_F cannot use any transition that is not also a transition of P_N, and X_0 cannot be exposed or the computation would end at the next step. We conclude that $(q_0, w, Z_0) \overset{*}{\underset{P_N}{\vdash}} (q, \epsilon, \epsilon)$. That is, w is in $N(P_N)$. □

Example 6.10: Let us design a PDA that processes sequences of `if`'s and `else`'s in a C program, where i stands for `if` and e stands for `else`. Recall from Section 5.3.1 that there is a problem whenever the number of `else`'s in any prefix exceeds the number of `if`'s, because then we cannot match each `else` against its previous `if`. Thus, we shall use a stack symbol Z to count the difference between the number of i's seen so far and the number of e's. This simple, one-state PDA, is suggested by the transition diagram of Fig. 6.5.

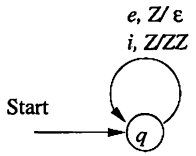


Figure 6.5: A PDA that accepts the if/else errors by empty stack

We shall push another Z whenever we see an i and pop a Z whenever we see an e. Since we start with one Z on the stack, we actually follow the rule that if the stack is Z^n, then there have been $n-1$ more i's than e's. In particular, if the stack is empty, than we have seen one more e than i, and the input read so far has just become illegal for the first time. It is these strings that our PDA accepts by empty stack. The formal specification of P_N is:

$$P_N = (\{q\}, \{i, e\}, \{Z\}, \delta_N, q, Z)$$

where δ_N is defined by:

1. $\delta_N(q, i, Z) = \{(q, ZZ)\}$. This rule pushes a Z when we see an i.
2. $\delta_N(q, e, Z) = \{(q, \epsilon)\}$. This rule pops a Z when we see an e.

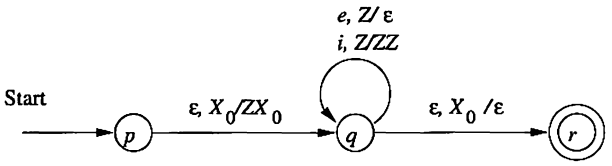


Figure 6.6: Construction of a PDA accepting by final state from the PDA of Fig. 6.5

Now, let us construct from P_N a PDA P_F that accepts the same language by final state; the transition diagram for P_F is shown in Fig. 6.6.[3] We introduce

[3]Do not be concerned that we are using new states p and r here, while the construction in Theorem 6.9 used p_0 and p_f. Names of states are arbitrary, of course.

a new start state p and an accepting state r. We shall use X_0 as the bottom-of stack marker. P_F is formally defined:

$$P_F = (\{p, q, r\}, \{i, e\}, \{Z, X_0\}, \delta_F, p, X_0, \{r\})$$

where δ_F consists of:

1. $\delta_F(p, \epsilon, X_0) = \{(q, ZX_0)\}$. This rule starts P_F simulating P_N, with X_0 as a bottom-of-stack-marker.
2. $\delta_F(q, i, Z) = \{(q, ZZ)\}$. This rule pushes a Z when we see an i; it simulates P_N.
3. $\delta_F(q, e, Z) = \{(q, \epsilon)\}$. This rule pops a Z when we see an e; it also simulates P_N.
4. $\delta_F(q, \epsilon, X_0) = \{(r, \epsilon, \epsilon)\}$. That is, P_F accepts when the simulated P_N would have emptied its stack.

□

6.2.4 From Final State to Empty Stack

Now, let us go in the opposite direction: take a PDA P_F that accepts a language L by final state and construct another PDA P_N that accepts L by empty stack. The construction is simple and is suggested in Fig. 6.7. From each accepting state of P_F, add a transition on ϵ to a new state p. When in state p, P_N pops its stack and does not consume any input. Thus, whenever P_F enters an accepting state after consuming input w, P_N will empty its stack after consuming w.

To avoid simulating a situation where P_F accidentally empties its stack without accepting, P_N must also use a marker X_0 on the bottom of its stack. The marker is P_N's start symbol, and like the construction of Theorem 6.9, P_N must start in a new state p_0, whose sole function is to push the start symbol of P_F on the stack and go to the start state of P_F. The construction is sketched in Fig. 6.7, and we give it formally in the next theorem.

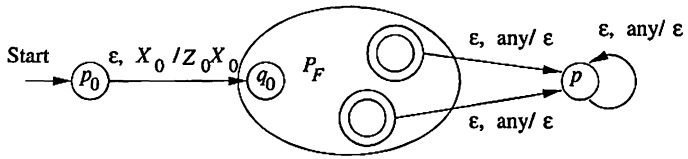


Figure 6.7: P_N simulates P_F and empties its stack when and only when P_N enters an accepting state

Theorem 6.11: Let L be $L(P_F)$ for some PDA $P_F = (Q, \Sigma, \Gamma, \delta_F, q_0, Z_0, F)$. Then there is a PDA P_N such that $L = N(P_N)$.

PROOF: The construction is as suggested in Fig. 6.7. Let

$$P_N = (Q \cup \{p_0, p\}, \Sigma, \Gamma \cup \{X_0\}, \delta_N, p_0, X_0)$$

where δ_N is defined by:

1. $\delta_N(p_0, \epsilon, X_0) = \{(q_0, Z_0X_0)\}$. We start by pushing the start symbol of P_F onto the stack and going to the start state of P_F.
2. For all states q in Q, input symbols a in Σ or $a = \epsilon$, and Y in Γ, $\delta_N(q, a, Y)$ contains every pair that is in $\delta_F(q, a, Y)$. That is, P_N simulates P_F.
3. For all accepting states q in F and stack symbols Y in Γ or $Y = X_0$, $\delta_N(q, \epsilon, Y)$ contains (p, ϵ). By this rule, whenever P_F accepts, P_N can start emptying its stack without consuming any more input.
4. For all stack symbols Y in Γ or $Y = X_0$, $\delta_N(p, \epsilon, Y) = \{(p, \epsilon)\}$. Once in state p, which only occurs when P_F has accepted, P_N pops every symbol on its stack, until the stack is empty. No further input is consumed.

Now, we must prove that w is in $N(P_N)$ if and only if w is in $L(P_F)$. The ideas are similar to the proof for Theorem 6.9. The 'if" part is a direct simulation, and the 'only-if" part requires that we examine the limited number of things that the constructed PDA P_N can do.

(If) Suppose $(q_0, w, Z_0) \overset{*}{\underset{P_F}{\vdash}} (q, \epsilon, \alpha)$ for some accepting state q and stack string α. Using the fact that every transition of P_F is a move of P_N, and invoking Theorem 6.5 to allow us to keep X_0 below the symbols of Γ on the stack, we know that $(q_0, w, Z_0X_0) \overset{*}{\underset{P_N}{\vdash}} (q, \epsilon, \alpha X_0)$. Then P_N can do the following:

$$(p_0, w, X_0) \underset{P_N}{\vdash} (q_0, w, Z_0X_0) \overset{*}{\underset{P_N}{\vdash}} (q, \epsilon, \alpha X_0) \overset{*}{\underset{P_N}{\vdash}} (p, \epsilon, \epsilon)$$

The first move is by rule (1) of the construction of P_N, while the last sequence of moves is by rules (3) and (4). Thus, w is accepted by P_N, by empty stack.

(Only-if) The only way P_N can empty its stack is by entering state p, since X_0 is sitting at the bottom of stack and X_0 is not a symbol on which P_F has any moves. The only way P_N can enter state p is if the simulated P_F enters an accepting state. The first move of P_N is surely the move given in rule (1). Thus, every accepting computation of P_N looks like

$$(p_0, w, X_0) \underset{P_N}{\vdash} (q_0, w, Z_0X_0) \overset{*}{\underset{P_N}{\vdash}} (q, \epsilon, \alpha X_0) \overset{*}{\underset{P_N}{\vdash}} (p, \epsilon, \epsilon)$$

where q is an accepting state of P_F.

Moreover, between ID's (q_0, w, Z_0X_0) and $(q, \epsilon, \alpha X_0)$, all the moves are moves of P_F. In particular, X_0 was never the top stack symbol prior to reaching ID $(q, \epsilon, \alpha X_0)$.[4] Thus, we conclude that the same computation can occur in P_F, without the X_0 on the stack; that is, $(q_0, w, Z_0) \overset{*}{\underset{P_F}{\vdash}} (q, \epsilon, \alpha)$. Now we see that P_F accepts w by final state, so w is in $L(P_F)$. □

[4] Although α could be ϵ, in which case P_F has emptied its stack at the same time it accepts.

6.2.5 Exercises for Section 6.2

Exercise 6.2.1: Design a PDA to accept each of the following languages. You may accept either by final state or by empty stack, whichever is more convenient.

* a) $\{0^n1^n \mid n \geq 1\}$.

b) The set of all strings of 0's and 1's such that no prefix has more 1's than 0's.

c) The set of all strings of 0's and 1's with an equal number of 0's and 1's.

! **Exercise 6.2.2:** Design a PDA to accept each of the following languages.

* a) $\{a^ib^jc^k \mid i = j \text{ or } j = k\}$. Note that this language is different from that of Exercise 5.1.1(b).

b) The set of all strings with twice as many 0's as 1's.

!! **Exercise 6.2.3:** Design a PDA to accept each of the following languages.

a) $\{a^ib^jc^k \mid i \neq j \text{ or } j \neq k\}$.

b) The set of all strings of a's and b's that are *not* of the form ww, that is, not equal to any string repeated.

*! **Exercise 6.2.4:** Let P be a PDA with empty-stack language $L = N(P)$, and suppose that ϵ is not in L. Describe how you would modify P so that it accepts $L \cup \{\epsilon\}$ by empty stack.

Exercise 6.2.5: PDA $P = (\{q_0, q_1, q_2, q_3, f\}, \{a, b\}, \{Z_0, A, B\}, \delta, q_0, Z_0, \{f\})$ has the following rules defining δ:

$\delta(q_0, a, Z_0) = (q_1, AAZ_0)$	$\delta(q_0, b, Z_0) = (q_2, BZ_0)$	$\delta(q_0, \epsilon, Z_0) = (f, \epsilon)$
$\delta(q_1, a, A) = (q_1, AAA)$	$\delta(q_1, b, A) = (q_1, \epsilon)$	$\delta(q_1, \epsilon, Z_0) = (q_0, Z_0)$
$\delta(q_2, a, B) = (q_3, \epsilon)$	$\delta(q_2, b, B) = (q_2, BB)$	$\delta(q_2, \epsilon, Z_0) = (q_0, Z_0)$
$\delta(q_3, \epsilon, B) = (q_2, \epsilon)$	$\delta(q_3, \epsilon, Z_0) = (q_1, AZ_0)$	

Note that, since each of the sets above has only one choice of move, we have omitted the set brackets from each of the rules.

* a) Give an execution trace (sequence of ID's) showing that string bab is in $L(P)$.

b) Give an execution trace showing that abb is in $L(P)$.

c) Give the contents of the stack after P has read b^7a^4 from its input.

! d) Informally describe $L(P)$.

Exercise 6.2.6: Consider the PDA P from Exercise 6.1.1.

a) Convert P to another PDA P_1 that accepts by empty stack the same language that P accepts by final state; i.e., $N(P_1) = L(P)$.

b) Find a PDA P_2 such that $L(P_2) = N(P)$; i.e., P_2 accepts by final state what P accepts by empty stack.

! **Exercise 6.2.7:** Show that if P is a PDA, then there is a PDA P_2 with only two stack symbols, such that $L(P_2) = L(P)$. *Hint*: Binary-code the stack alphabet of P.

*! **Exercise 6.2.8:** A PDA is called *restricted* if on any transition it can increase the height of the stack by at most one symbol. That is, for any rule $\delta(q, a, Z)$ contains (p, γ), it must be that $|\gamma| \leq 2$. Show that if P is a PDA, then there is a restricted PDA P_3 such that $L(P) = L(P_3)$.

6.3 Equivalence of PDA's and CFG's

Now, we shall demonstrate that the languages defined by PDA's are exactly the context-free languages. The plan of attack is suggested by Fig. 6.8. The goal is to prove that the following three classes of languages:

1. The context-free languages, i.e., the languages defined by CFG's.
2. The languages that are accepted by final state by some PDA.
3. The languages that are accepted by empty stack by some PDA.

are all the same class. We have already shown that (2) and (3) are the same. It turns out to be easiest next to show that (1) and (3) are the same, thus implying the equivalence of all three.

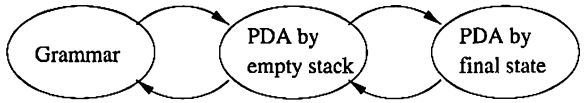


Figure 6.8: Organization of constructions showing equivalence of three ways of defining the CFL's

6.3.1 From Grammars to Pushdown Automata

Given a CFG G, we construct a PDA that simulates the leftmost derivations of G. Any left-sentential form that is not a terminal string can be written as $xA\alpha$, where A is the leftmost variable, x is whatever terminals appear to its left, and α is the string of terminals and variables that appear to the right of A.

We call $A\alpha$ the *tail* of this left-sentential form. If a left-sentential form consists of terminals only, then its tail is ϵ.

The idea behind the construction of a PDA from a grammar is to have the PDA simulate the sequence of left-sentential forms that the grammar uses to generate a given terminal string w. The tail of each sentential form $xA\alpha$ appears on the stack, with A at the top. At that time, x will be "represented" by our having consumed x from the input, leaving whatever of w follows its prefix x. That is, if $w = xy$, then y will remain on the input.

Suppose the PDA is in an ID $(q, y, A\alpha)$, representing left-sentential form $xA\alpha$. It guesses the production to use to expand A, say $A \to \beta$. The move of the PDA is to replace A on the top of the stack by β, entering ID $(q, y, \beta\alpha)$. Note that there is only one state, q, for this PDA.

Now $(q, y, \beta\alpha)$ may not be a representation of the next left-sentential form, because β may have a prefix of terminals. In fact, β may have no variables at all, and α may have a prefix of terminals. Whatever terminals appear at the beginning of $\beta\alpha$ need to be removed, to expose the next variable at the top of the stack. These terminals are compared against the next input symbols, to make sure our guesses at the leftmost derivation of input string w are correct; if not, this branch of the PDA dies.

If we succeed in this way to guess a leftmost derivation of w, then we shall eventually reach the left-sentential form w. At that point, all the symbols on the stack have either been expanded (if they are variables) or matched against the input (if they are terminals). The stack is empty, and we accept by empty stack.

The above informal construction can be made precise as follows. Let $G = (V, T, Q, S)$ be a CFG. Construct the PDA P that accepts $L(G)$ by empty stack as follows:

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

where transition function δ is defined by:

1. For each variable A,

$$\delta(q, \epsilon, A) = \{(q, \beta) \mid A \to \beta \text{ is a production of } G\}$$

2. For each terminal a, $\delta(q, a, a) = \{(q, \epsilon)\}$.

Example 6.12: Let us convert the expression grammar of Fig. 5.2 to a PDA. Recall this grammar is:

$$\begin{array}{lcl} I & \to & a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\ E & \to & I \mid E*E \mid E+E \mid (E) \end{array}$$

The set of terminals for the PDA is $\{a, b, 0, 1, (,), +, *\}$. These eight symbols and the symbols I and E form the stack alphabet. The transition function for the PDA is:

a) $\delta(q, \epsilon, I) = \{(q, a), (q, b), (q, Ia), (q, Ib), (q, I0), (q, I1)\}$.

b) $\delta(q, \epsilon, E) = \{(q, I), (q, E+E), (q, E*E), (q, (E))\}$.

c) $\delta(q, a, a) = \{(q, \epsilon)\}$; $\delta(q, b, b) = \{(q, \epsilon)\}$; $\delta(q, 0, 0) = \{(q, \epsilon)\}$; $\delta(q, 1, 1) = \{(q, \epsilon)\}$; $\delta(q, (, () = \{(q, \epsilon)\}$; $\delta(q,),)) = \{(q, \epsilon)\}$; $\delta(q, +, +) = \{(q, \epsilon)\}$; $\delta(q, *, *) = \{(q, \epsilon)\}$.

Note that (a) and (b) come from rule (1), while the eight transitions of (c) come from rule (2). Also, δ is empty except as defined by (a) through (c). □

Theorem 6.13: If PDA P is constructed from CFG G by the construction above, then $N(P) = L(G)$.

PROOF: We shall prove that w is in $N(P)$ if and only if w is in $L(G)$.

(If) Suppose w is in $L(G)$. Then w has a leftmost derivation

$$S = \gamma_1 \underset{lm}{\Rightarrow} \gamma_2 \underset{lm}{\Rightarrow} \cdots \underset{lm}{\Rightarrow} \gamma_n = w$$

We show by induction on i that $(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, y_i, \alpha_i)$, where y_i and α_i are a representation of the left-sentential form γ_i. That is, let α_i be the tail of γ_i, and let $\gamma_i = x_i\alpha_i$. Then y_i is that string such that $x_iy_i = w$; i.e., it is what remains when x_i is removed from the input.

BASIS: For $i = 1$, $\gamma_1 = S$. Thus, $x_1 = \epsilon$, and $y_1 = w$. Since $(q, w, S) \overset{*}{\vdash} (q, w, S)$ by 0 moves, the basis is proved.

INDUCTION: Now we consider the case of the second and subsequent left-sentential forms. We assume

$$(q, w, S) \overset{*}{\vdash} (q, y_i, \alpha_i)$$

and prove $(q, w, S) \overset{*}{\vdash} (q, y_{i+1}, \alpha_{i+1})$. Since α_i is a tail, it begins with a variable A. Moreover, the step of the derivation $\gamma_i \Rightarrow \gamma_{i+1}$ involves replacing A by one of its production bodies, say β. Rule (1) of the construction of P lets us replace A at the top of the stack by β, and rule (2) then allows us to match any terminals on top of the stack with the next input symbols. As a result, we reach the ID $(q, y_{i+1}, \alpha_{i+1})$, which represents the next left-sentential form γ_{i+1}.

To complete the proof, we note that $\alpha_n = \epsilon$, since the tail of γ_n (which is w) is empty. Thus, $(q, w, S) \overset{*}{\vdash} (q, \epsilon, \epsilon)$, which proves that P accepts w by empty stack.

(Only-if) We need to prove something more general: that if P executes a sequence of moves that has the net effect of popping a variable A from the top of its stack, without ever going below A on the stack, then A derives, in G, whatever input string was consumed from the input during this process. Precisely:

- If $(q, x, A) \overset{*}{\underset{P}{\vdash}} (q, \epsilon, \epsilon)$, then $A \overset{*}{\underset{G}{\Rightarrow}} x$.

The proof is an induction on the number of moves taken by P.

BASIS: One move. The only possibility is that $A \to \epsilon$ is a production of G, and this production is used in a rule of type (1) by the PDA P. In this case, $x = \epsilon$, and we know that $A \Rightarrow \epsilon$.

INDUCTION: Suppose P takes n moves, where $n > 1$. The first move must be of type (1), where A is replaced by one of its production bodies on the top of the stack. The reason is that a rule of type (2) can only be used when there is a terminal on top of the stack. Suppose the production used is $A \to Y_1Y_2 \cdots Y_k$, where each Y_i is either a terminal or variable.

The next $n-1$ moves of P must consume x from the input and have the net effect of popping each of Y_1, Y_2, and so on from the stack, one at a time. We can break x into $x_1x_2 \cdots x_k$, where x_1 is the portion of the input consumed until Y_1 is popped off the stack (i.e., the stack first is as short as $k-1$ symbols). Then x_2 is the next portion of the input that is consumed while popping Y_2 off the stack, and so on.

Figure 6.9 suggests how the input x is broken up, and the corresponding effects on the stack.. There, we suggest that β was BaC, so x is divided into three parts $x_1x_2x_3$, where $x_2 = a$. Note that in general, if Y_i is a terminal, then x_i must be that terminal.

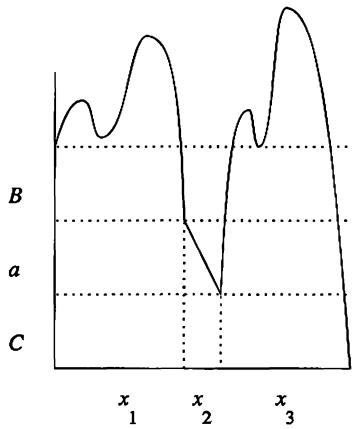


Figure 6.9: The PDA P consumes x and pops BaC from its stack

Formally, we can conclude that $(q, x_ix_{i+1} \cdots x_k, Y_i) \stackrel{*}{\vdash} (q, x_{i+1} \cdots x_k, \epsilon)$ for all $i = 1, 2, \ldots, k$. Moreover, none of these sequences can be more than $n-1$ moves, so the inductive hypothesis applies if Y_i is a variable. That is, we may conclude $Y_i \stackrel{*}{\Rightarrow} x_i$.

If Y_i is a terminal, then there must be only one move involved, and it matches the one symbol of x_i against Y_i, which are the same. Again, we can conclude

$Y_i \stackrel{*}{\Rightarrow} x_i$; this time, zero steps are used. Now we have the derivation

$$A \Rightarrow Y_1Y_2 \cdots Y_k \stackrel{*}{\Rightarrow} x_1Y_2 \cdots Y_k \stackrel{*}{\Rightarrow} \cdots \stackrel{*}{\Rightarrow} x_1x_2 \cdots x_k$$

That is, $A \stackrel{*}{\Rightarrow} x$.

To complete the proof, we let $A = S$ and $x = w$. Since we are given that w is in $N(P)$, we know that $(q, w, S) \stackrel{*}{\vdash} (q, \epsilon, \epsilon)$. By what we have just proved inductively, we have $S \stackrel{*}{\Rightarrow} w$; i.e., w is in $L(G)$. □

6.3.2 From PDA's to Grammars

Now, we complete the proofs of equivalence by showing that for every PDA P, we can find a CFG G whose language is the same language that P accepts by empty stack. The idea behind the proof is to recognize that the fundamental event in the history of a PDA's processing of a given input is the net popping of one symbol off the stack, while consuming some input. A PDA may change state as it pops stack symbols, so we should also note the state that it enters when it finally pops a level off its stack.

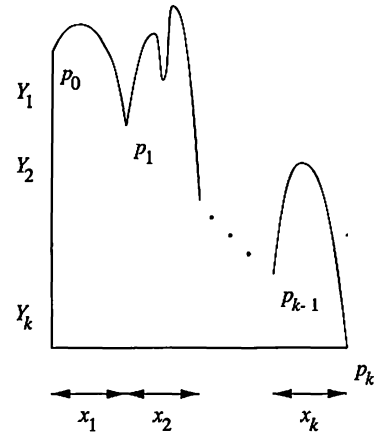


Figure 6.10: A PDA makes a sequence of moves that have the net effect of popping a symbol off the stack

Figure 6.10 suggests how we pop a sequence of symbols $Y_1, Y_2, \ldots Y_k$ off the stack. Some input x_1 is read while Y_1 is popped. We should emphasize that this "pop" is the net effect of (possibly) many moves. For example, the first move may change Y_1 to some other symbol Z. The next move may replace Z by UV, later moves have the effect of popping U, and then other moves pop V.

The net effect is that Y_1 has been replaced by nothing; i.e., it has been popped, and all the input symbols consumed so far constitute x_1.

We also show in Fig. 6.10 the net change of state. We suppose that the PDA starts out in state p_0, with Y_1 at the top of the stack. After all the moves whose net effect is to pop Y_1, the PDA is in state p_1. It then proceeds to (net) pop Y_2, while reading input string x_2 and winding up, perhaps after many moves, in state p_2 with Y_2 off the stack. The computation proceeds until each of the symbols on the stack is removed.

Our construction of an equivalent grammar uses variables each of which represents an "event" consisting of:

1. The net popping of some symbol X from the stack, and
2. A change in state from some p at the beginning to q when X has finally been replaced by ϵ on the stack.

We represent such a variable by the composite symbol $[pXq]$. Remember that this sequence of characters is our way of describing *one* variable; it is not five grammar symbols. The formal construction is given by the next theorem.

Theorem 6.14: Let $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$ be a PDA. Then there is a context-free grammar G such that $L(G) = N(P)$.

PROOF: We shall construct $G = (V, \Sigma, R, S)$, where the set of variables V consists of:

1. The special symbol S, which is the start symbol, and
2. All symbols of the form $[pXq]$, where p and q are states in Q, and X is a stack symbol, in Γ.

The productions of G are as follows:

a) For all states p, G has the production $S \to [q_0 Z_0 p]$. Recall our intuition that a symbol like $[q_0 Z_0 p]$ is intended to generate all those strings w that cause P to pop Z_0 from its stack while going from state q_0 to state p. That is, $(q_0, w, Z_0) \vdash^* (p, \epsilon, \epsilon)$. If so, then these productions say that start symbol S will generate all strings w that cause P to empty its stack, after starting in its initial ID.

b) Let $\delta(q, a, X)$ contain the pair $(r, Y_1 Y_2 \cdots Y_k)$, where:

   1. a is either a symbol in Σ or $a = \epsilon$.
   2. k can be any number, including 0, in which case the pair is (r, ϵ).

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

$$[qXr_k] \to a[rY_1r_1][r_1Y_2r_2]\cdots[r_{k-1}Y_kr_k]$$

This production says that one way to pop X and go from state q to state r_k is to read a (which may be ϵ), then use some input to pop Y_1 off the stack while going from state r to state r_1, then read some more input that pops Y_2 off the stack and goes from state r_1 to r_2, and so on.

We shall now prove that the informal interpretation of the variables $[qXp]$ is correct:

- $[qXp] \stackrel{*}{\Rightarrow} w$ if and only if $(q, w, X) \vdash^* (p, \epsilon, \epsilon)$.

(If) Suppose $(q, w, X) \vdash^* (p, \epsilon, \epsilon)$. We shall show $[qXp] \stackrel{*}{\Rightarrow} w$ by induction on the number of moves made by the PDA.

BASIS: One step. Then (p, ϵ) must be in $\delta(q, w, X)$, and w is either a single symbol or ϵ. By the construction of G, $[qXp] \to w$ is a production, so $[qXp] \Rightarrow w$.

INDUCTION: Suppose the sequence $(q, w, X) \vdash^* (p, \epsilon, \epsilon)$ takes n steps, and $n > 1$. The first move must look like

$$(q, w, X) \vdash (r_0, x, Y_1Y_2\cdots Y_k) \vdash^* (p, \epsilon, \epsilon)$$

where $w = ax$ for some a that is either ϵ or a symbol in Σ. It follows that the pair $(r_0, Y_1Y_2\cdots Y_k)$ must be in $\delta(q, a, X)$. Further, by the construction of G, there is a production $[qXr_k] \to a[r_0Y_1r_1][r_1Y_2r_2]\cdots[r_{k-1}Y_kr_k]$, where:

1. $r_k = p$, and
2. $r_1, r_2, \ldots, r_{k-1}$ are any states in Q.

In particular, we may observe, as was suggested in Fig. 6.10, that each of the symbols $Y_1, Y_2, \ldots, Y_k$ gets popped off the stack in turn, and we may choose p_i to be the state of the PDA when Y_i is popped, for $i = 1, 2, \ldots, k-1$. Let $x = w_1w_2\cdots w_k$, where w_i is the input consumed while Y_i is popped off the stack. Then we know that $(r_{i-1}, w_i, Y_i) \vdash^* (r_i, \epsilon, \epsilon)$.

As none of these sequences of moves can take as many as n moves, the inductive hypothesis applies to them. We conclude that $[r_{i-1}Y_ir_i] \stackrel{*}{\Rightarrow} w_i$. We may put these derivations together with the first production used to conclude:

$$\begin{aligned}
&[qXr_k] \Rightarrow a[r_0Y_1r_1][r_1Y_1r_2]\cdots[r_{k-1}Y_kr_k] \stackrel{*}{\Rightarrow}\\
&aw_1[r_1Y_2r_2][r_2Y_3r_3]\cdots[r_{k-1}Y_kr_k] \stackrel{*}{\Rightarrow}\\
&aw_1w_2[r_2Y_3r_3]\cdots[r_{k-1}Y_kr_k] \stackrel{*}{\Rightarrow}\\
&\cdots\\
&aw_1w_2\cdots w_k = w
\end{aligned}$$

where $r_k = p$.

(Only-if) The proof is an induction on the number of steps in the derivation.

BASIS: One step. Then $[qXp] \to w$ must be a production. The only way for this production to exist is if there is a transition of P in which X is popped and state q becomes state p. That is, (p, ϵ) must be in $\delta(q, a, X)$, and $a = w$. But then $(q, w, X) \vdash (p, \epsilon, \epsilon)$.

INDUCTION: Suppose $[qXp] \stackrel{*}{\Rightarrow} w$ by n steps, where $n > 1$. Consider the first sentential form explicitly, which must look like

$$[qXr_k] \Rightarrow a[r_0Y_1r_1][r_1Y_2r_2]\cdots[r_{k-1}Y_kr_k] \stackrel{*}{\Rightarrow} w$$

where $r_k = p$. This production must come from the fact that $(r_0, Y_1Y_2\cdots Y_k)$ is in $\delta(q, a, X)$.

We can break w into $w = aw_1w_2\cdots w_k$ such that $[r_{i-1}Y_ir_i] \stackrel{*}{\Rightarrow} w_i$ for all $i = 1, 2, \ldots, k$. By the inductive hypothesis, we know that for all i,

$$(r_{i-1}, w_i, Y_i) \stackrel{*}{\vdash} (r_i, \epsilon, \epsilon)$$

If we use Theorem 6.5 to put the correct strings beyond w_i on the input and below Y_i on the stack, we also know that

$$(r_{i-1}, w_iw_{i+1}\cdots w_k, Y_iY_{i+1}\cdots Y_k) \stackrel{*}{\vdash} (r_i, w_{i+1}\cdots w_k, Y_{i+1}\cdots Y_k)$$

If we put all these sequences together, we see that

$$(q, aw_1w_2\cdots w_k, X) \vdash (r_0, w_1w_2\cdots w_k, Y_1Y_2\cdots Y_k) \stackrel{*}{\vdash}$$
$$(r_1, w_2w_3\cdots w_k, Y_2Y_3\cdots Y_k) \stackrel{*}{\vdash} (r_2, w_3\cdots w_k, Y_3\cdots Y_k) \stackrel{*}{\vdash} \cdots \stackrel{*}{\vdash} (r_k, \epsilon, \epsilon)$$

Since $r_k = p$, we have shown that $(q, w, X) \stackrel{*}{\vdash} (p, \epsilon, \epsilon)$.

We complete the proof as follows. $S \stackrel{*}{\Rightarrow} w$ if and only if $[q_0Z_0p] \stackrel{*}{\Rightarrow} w$ for some p, because of the way the rules for start symbol S are constructed. We just proved that $[q_0Z_0p] \stackrel{*}{\Rightarrow} w$ if and only if $(q, w, Z_0) \stackrel{*}{\vdash} (p, \epsilon, \epsilon)$, i.e., if and only if P accepts x by empty stack. Thus, $L(G) = N(P)$. □

Example 6.15: Let us convert the PDA $P_N = (\{q\}, \{i, e\}, \{Z\}, \delta_N, q, Z)$ from Example 6.10 to a grammar. Recall that P_N accepts all strings that violate, for the first time, the rule that every e (else) must correspond to some preceding i (if). Since P_N has only one state and one stack symbol, the construction is particularly simple. There are only two variables in the grammar G:

a) S, the start symbol, which is in every grammar constructed by the method of Theorem 6.14, and

b) $[qZq]$, the only triple that can be assembled from the states and stack symbols of P_N.

The productions of grammar G are as follows:

1. The only production for S is $S \to [qZq]$. However, if there were n states of the PDA, then there would be n productions of this type, since the last state could be any of the n states. The first state would have to be the start state, and the stack symbol would have to be the start symbol, as in our production above.

2. From the fact that $\delta_N(q, i, Z)$ contains (q, ZZ), we get the production $[qZq] \to i[qZq][qZq]$. Again, for this simple example, there is only one production. However, if there were n states, then this one rule would produce n^2 productions, since the middle two states of the body could be any one state p, and the last states of the head and body could also be any one state. That is, if p and r were any two states of the PDA, then production $[qZp] \to i[qZr][rZp]$ would be produced.

3. From the fact that $\delta_N(q, e, Z)$ contains (q, ϵ), we have production

   $$[qZq] \to e$$

   Notice that in this case, the list of stack symbols by which Z is replaced is empty, so the only symbol in the body is the input symbol that caused the move.

We may, for convenience, replace the triple $[qZq]$ by some less complex symbol, say A. If we do, then the complete grammar consists of the productions:

$$S \to A$$
$$A \to iAA \mid e$$

In fact, if we notice that A and S derive exactly the same strings, we may identify them as one, and write the complete grammar as

$$G = (\{S\}, \{i, e\}, \{S \to iSS \mid e\}, S)$$

□

6.3.3 Exercises for Section 6.3

* **Exercise 6.3.1:** Convert the grammar

$$S \to 0S1 \mid A$$
$$A \to 1A0 \mid S \mid \epsilon$$

to a PDA that accepts the same language by empty stack.

Exercise 6.3.2: Convert the grammar

$$S \to aAA$$
$$A \to aS \mid bS \mid a$$

to a PDA that accepts the same language by empty stack.

* **Exercise 6.3.3:** Convert the PDA $P = (\{p,q\},\{0,1\},\{X,Z_0\},\delta,q,Z_0)$ to a CFG, if δ is given by:

1. $\delta(q,1,Z_0) = \{(q,XZ_0)\}$.
2. $\delta(q,1,X) = \{(q,XX)\}$.
3. $\delta(q,0,X) = \{(p,X)\}$.
4. $\delta(q,\epsilon,X) = \{(q,\epsilon)\}$.
5. $\delta(p,1,X) = \{(p,\epsilon)\}$.
6. $\delta(p,0,Z_0) = \{(q,Z_0)\}$.

Exercise 6.3.4: Convert the PDA of Exercise 6.1.1 to a context-free grammar.

Exercise 6.3.5: Below are some context-free languages. For each, devise a PDA that accepts the language by empty stack. You may, if you wish, first construct a grammar for the language, and then convert to a PDA.

a) $\{a^n b^m c^{2(n+m)} \mid n \geq 0,\ m \geq 0\}$.

b) $\{a^i b^j c^k \mid i = 2j \text{ or } j = 2k\}$.

! c) $\{0^n 1^m \mid n \leq m \leq 2n\}$.

*! **Exercise 6.3.6:** Show that if P is a PDA, then there is a one-state PDA P_1 such that $N(P_1) = N(P)$.

! **Exercise 6.3.7:** Suppose we have a PDA with s states, t stack symbols, and no rule in which a replacement stack string has length greater than u. Give a tight upper bound on the number of variables in the CFG that we construct for this PDA by the method of Section 6.3.2.

6.4 Deterministic Pushdown Automata

While PDA's are by definition allowed to be nondeterministic, the deterministic subcase is quite important. In particular, parsers generally behave like deterministic PDA's, so the class of languages that can be accepted by these automata is interesting for the insights it gives us into what constructs are suitable for use in programming languages. In this section, we shall define deterministic PDA's and investigate some of the things they can and cannot do.

6.4.1 Definition of a Deterministic PDA

Intuitively, a PDA is deterministic if there is never a choice of move in any situation. These choices are of two kinds. If $\delta(q,a,X)$ contains more than one pair, then surely the PDA is nondeterministic because we can choose among these pairs when deciding on the next move. However, even if $\delta(q,a,X)$ is always a singleton, we could still have a choice between using a real input symbol, or making a move on ϵ. Thus, we define a PDA $P = (Q,\Sigma,\Gamma,\delta,q_0,Z_0,F)$ to be *deterministic* (a deterministic PDA or DPDA), if and only if the following conditions are met:

1. $\delta(q,a,X)$ has at most one member for any q in Q, a in Σ or $a = \epsilon$, and X in Γ.
2. If $\delta(q,a,X)$ is nonempty, for some a in Σ, then $\delta(q,\epsilon,X)$ must be empty.

Example 6.16: It turns out that the language L_{wwr} of Example 6.2 is a CFL that has no DPDA. However, by putting a "center-marker" c in the middle, we can make the language recognizable by a DPDA. That is, we can recognize the language $L_{wcwr} = \{wcw^R \mid w \text{ is in } (0+1)^*\}$ by a deterministic PDA.

The strategy of the DPDA is to store 0's and 1's on its stack, until it sees the center marker c. it then goes to another state, in which it matches input symbols against stack symbols and pops the stack if they match. If it ever finds a nonmatch, it dies; its input cannot be of the form wcw^R. If it succeeds in popping its stack down to the initial symbol, which marks the bottom of the stack, then it accepts its input.

The idea is very much like the PDA that we saw in Fig. 6.2. However, that PDA is nondeterministic, because in state q_0 it always has the choice of pushing the next input symbol onto the stack or making a transition on ϵ to state q_1; i.e., it has to guess when it has reached the middle. The DPDA for L_{wcwr} is shown as a transition diagram in Fig. 6.11.

This PDA is clearly deterministic. It never has a choice of move in the same state, using the same input and stack symbol. As for choices between using a real input symbol or ϵ, the only ϵ-transition it makes is from q_1 to q_2 with Z_0 at the top of the stack. However, in state q_1, there are no other moves when Z_0 is at the stack top. □

6.4.2 Regular Languages and Deterministic PDA's

The DPDA's accept a class of languages that is between the regular languages and the CFL's. We shall first prove that the DPDA languages include all the regular languages.

Theorem 6.17: If L is a regular language, then $L = L(P)$ for some DPDA P.

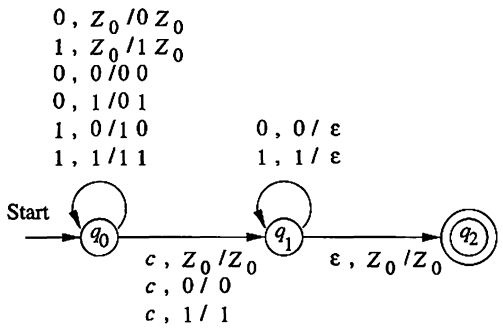


Figure 6.11: A deterministic PDA accepting L_{wcwr}

PROOF: Essentially, a DPDA can simulate a deterministic finite automaton. The PDA keeps some stack symbol Z_0 on its stack, because a PDA has to have a stack, but really the PDA ignores its stack and just uses its state. Formally, let $A = (Q, \Sigma, \delta_A, q_0, F)$ be a DFA. Construct DPDA

$$P = (Q, \Sigma, \{Z_0\}, \delta_P, q_0, Z_0, F)$$

by defining $\delta_P(q, a, Z_0) = \{(p, Z_0)\}$ for all states p and q in Q, such that $\delta_A(q, a) = p$.

We claim that $(q_0, w, Z_0) \overset{*}{\underset{P}{\vdash}} (p, \epsilon, Z_0)$ if and only if $\hat{\delta}_A(q_0, w) = p$. That is, P simulates A using its state. The proofs in both directions are easy inductions on $|w|$, and we leave them for the reader to complete. Since both A and P accept by entering one of the states of F, we conclude that their languages are the same. □

If we want the DPDA to accept by empty stack, then we find that our language-recognizing capability is rather limited. Say that a language L has the *prefix property* if there are no two different strings x and y in L such that x is a prefix of y.

Example 6.18: The language L_{wcwr} of Example 6.16 has the prefix property. That is, it is not possible for there to be two strings wcw^R and xcx^R, one of which is a prefix of the other, unless they are the same string. To see why, suppose wcw^R is a prefix of xcx^R, and $w \neq x$. Then w must be shorter than x. Therefore, the c in wcw^R comes in a position where xcx^R has a 0 or 1; it is a position in the first x. That point contradicts the assumption that wcw^R is a prefix of xcx^R.

On the other hand, there are some very simple languages that do not have the prefix property. Consider $\{0\}^*$, i.e., the set of all strings of 0's. Clearly, there are pairs of strings in this language one of which is a prefix of the other, so this language does not have the prefix property. In fact, of *any* two strings, one is a prefix of the other, although that condition is stronger than we need to establish that the prefix property does not hold. □

Note that the language $\{0\}^*$ is a regular language. Thus, it is not even true that every regular language is $N(P)$ for some DPDA P. We leave as an exercise the following relationship:

Theorem 6.19: A language L is $N(P)$ for some DPDA P if and only if L has the prefix property and L is $L(P')$ for some DPDA P'. □

6.4.3 DPDA's and Context-Free Languages

We have already seen that a DPDA can accept languages like L_{wcwr} that are not regular. To see this language is not regular, suppose it were, and use the pumping lemma. If n is the constant of the pumping lemma, then consider the string $w = 0^n c 0^n$, which is in L_{wcwr}. However, when we "pump" this string, it is the first group of 0's whose length must change, so we get in L_{wcwr} strings that have the "center" marker not in the center. Since these strings are not in L_{wcwr}, we have a contradiction and conclude that L_{wcwr} is not regular.

On the other hand, there are CFL's like L_{wwr} that cannot be $L(P)$ for any DPDA P. A formal proof is complex, but the intuition is transparent. If P is a DPDA accepting L_{wwr}, then given a sequence of 0's, it must store them on the stack, or do something equivalent to count an arbitrary number of 0's. For instance, it could store one X for every two 0's it sees, and use the state to remember whether the number was even or odd.

Suppose P has seen n 0's and then sees 110^n. It must verify that there were n 0's after the 11, and to do so it must pop its stack.[5] Now, P has seen 0^n110^n. If it sees an identical string next, it must accept, because the complete input is of the form ww^R, with $w = 0^n110^n$. However, if it sees 0^m110^m for some $m \neq n$, P must *not* accept. Since its stack is empty, it cannot remember what arbitrary integer n was, and must fail to recognize L_{wwr} correctly. Our conclusion is that:

- The languages accepted by DPDA's by final state properly include the regular languages, but are properly included in the CFL's.

6.4.4 DPDA's and Ambiguous Grammars

We can refine the power of the DPDA's by noting that the languages they accept all have unambiguous grammars. Unfortunately, the DPDA languages are not

[5] This statement is the intuitive part that requires a (hard) formal proof; could there be some other way for P to compare equal blocks of 0's?

exactly equal to the subset of the CFL's that are not inherently ambiguous. For instance, L_{wwr} has an unambiguous grammar

$$S \to 0S0 \mid 1S1 \mid \epsilon$$

even though it is not a DPDA language. The following theorems refine the bullet point above.

Theorem 6.20: If $L = N(P)$ for some DPDA P, then L has an unambiguous context-free grammar.

PROOF: We claim that the construction of Theorem 6.14 yields an unambiguous CFG G when the PDA to which it is applied is deterministic. First recall from Theorem 5.29 that it is sufficient to show that the grammar has unique leftmost derivations in order to prove that G is unambiguous.

Suppose P accepts string w by empty stack. Then it does so by a unique sequence of moves, because it is deterministic, and cannot move once its stack is empty. Knowing this sequence of moves, we can determine the one choice of production in a leftmost derivation whereby G derives w. There can never be a choice of which rule of P motivated the production to use. However, a rule of P, say $\delta(q, a, X) = \{(r, Y_1Y_2 \cdots Y_k)\}$ might cause many productions of G, with different states in the positions that reflect the states of P after popping each of $Y_1, Y_2, \ldots, Y_{k-1}$. Because P is deterministic, only one of these sequences of choices will be consistent with what P actually does, and therefore, only one of these productions will actually lead to derivation of w. □

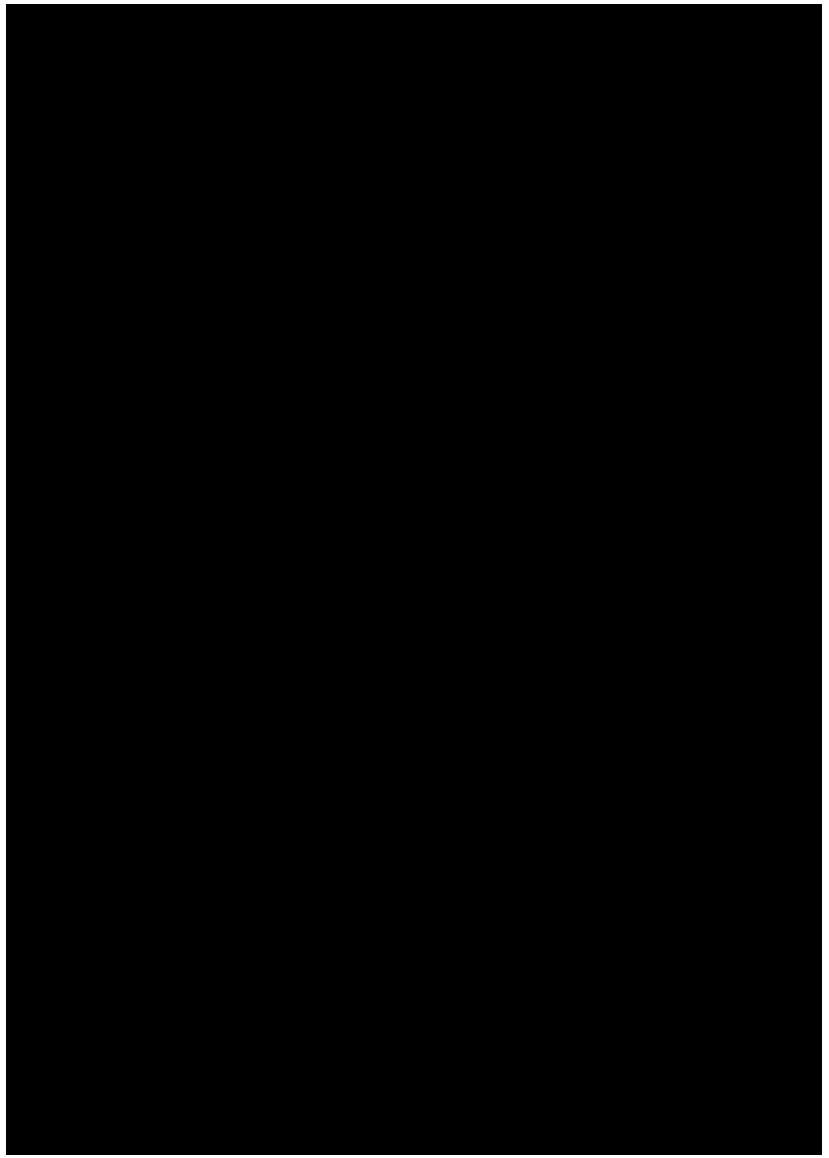
However, we can prove more: even those languages that DPDA's accept by final state have unambiguous grammars. Since we only know how to construct grammars directly from PDA's that accept by empty stack, we need to change the language involved to have the prefix property, and then modify the resulting grammar to generate the original language. We do so by use of an "endmarker" symbol.

Theorem 6.21: If $L = L(P)$ for some DPDA P, then L has an unambiguous CFG.

PROOF: let \$ be an "endmarker" symbol that does not appear in the strings of L, and let $L' = L\$$. That is, the strings of L' are the strings of L, each followed by the symbol \$. Then L' surely has the prefix property, and by Theorem 6.19, $L' = N(P')$ for some DPDA P'.[6] By Theorem 6.20, there is an unambiguous grammar G' generating the language $N(P')$, which is L'.

Now, construct from G' a grammar G such that $L(G) = L$. To do so, we have only to get rid of the endmarker \$ from strings. Thus, treat \$ as a variable

[6]The proof of Theorem 6.19 appears in Exercise 6.4.3, but we can easily see how to construct P' from P. Add a new state q that P' enters whenever P is in an accepting state and the next input is \$. In state q, P' pops all symbols off its stack. Also, P' needs its own bottom-of-stack marker to avoid accidentally emptying its stack as it simulates P.



Chapter 7

Properties of Context-Free Languages

We shall complete our study of context-free languages by learning some of their properties. Our first task is to simplify context-free grammars; these simplifications make it easier to prove facts about CFL's, since we can claim that if a language is a CFL, then it has a grammar in some special form.

We then prove a "pumping lemma" for CFL's. This theorem is in the same spirit as Theorem 4.1 for regular languages, but can be used to prove a language not to be context-free. Next, we consider the sorts of properties that we studied in Chapter 4 for the regular languages: closure properties and decision properties. We shall see that some, but not all, of the closure properties that the regular languages have are also possessed by the CFL's. Likewise, some questions about CFL's can be decided by algorithms that generalize the tests we developed for regular languages, but there are also certain questions about CFL's that we cannot answer.

7.1 Normal Forms for Context-Free Grammars

The goal of this section is to show that every CFL (without ϵ) is generated by a CFG in which all productions are of the form $A \to BC$ or $A \to a$, where A, B, and C are variables, and a is a terminal. This form is called *Chomsky Normal Form*. To get there, we need to make a number of preliminary simplifications, which are themselves useful in various ways:

1. We must eliminate *useless symbols*, those variables or terminals that do not appear in any derivation of a terminal string from the start symbol.

2. We must eliminate *ϵ-productions*, those of the form $A \to \epsilon$ for some variable A.

3. We must eliminate *unit productions*, those of the form $A \to B$ for variables A and B.

7.1.1 Eliminating Useless Symbols

We say a symbol X is *useful* for a grammar $G = (V, T, P, S)$ if there is some derivation of the form $S \stackrel{*}{\Rightarrow} \alpha X \beta \stackrel{*}{\Rightarrow} w$, where w is in T^*. Note that X may be in either V or T, and the sentential form $\alpha X \beta$ might be the first or last in the derivation. If X is not useful, we say it is *useless*. Evidently, omitting useless symbols from a grammar will not change the language generated, so we may as well detect and eliminate all useless symbols.

Our approach to eliminating useless symbols begins by identifying the two things a symbol has to be able to do to be useful:

1. We say X is *generating* if $X \stackrel{*}{\Rightarrow} w$ for some terminal string w. Note that every terminal is generating, since w can be that terminal itself, which is derived by zero steps.

2. We say X is *reachable* if there is a derivation $S \stackrel{*}{\Rightarrow} \alpha X \beta$ for some α and β.

Surely a symbol that is useful will be both generating and reachable. If we eliminate the symbols that are not generating first, and then eliminate from the remaining grammar those symbols that are not reachable, we shall, as will be proved, have only the useful symbols left.

Example 7.1: Consider the grammar:

$$S \to AB \mid a$$
$$A \to b$$

All symbols but B are generating; a and b generate themselves; S generates a, and A generates b. If we eliminate B, we must eliminate the production $S \to AB$, leaving the grammar:

$$S \to a$$
$$A \to b$$

Now, we find that only S and a are reachable from S. Eliminating A and b leaves only the production $S \to a$. That production by itself is a grammar whose language is $\{a\}$, just as is the language of the original grammar.

Note that if we start by checking for reachability first, we find that all symbols of the grammar

$$S \to AB \mid a$$
$$A \to b$$

are reachable. If we then eliminate the symbol B because it is not generating, we are left with a grammar that still has useless symbols, in particular, A and b. □

Theorem 7.2: Let $G = (V,T,P,S)$ be a CFG, and assume that $L(G) \neq \emptyset$; i.e., G generates at least one string. Let $G_1 = (V_1,T_1,P_1,S)$ be the grammar we obtain by the following steps:

1. First eliminate nongenerating symbols and all productions involving one or more of those symbols. Let $G_2 = (V_2,T_2,P_2,S)$ be this new grammar. Note that S must be generating, since we assume $L(G)$ has at least one string, so S has not been eliminated.

2. Second, eliminate all symbols that are not reachable in the grammar G_2.

Then G_1 has no useless symbols, and $L(G_1) = L(G)$.

PROOF: Suppose X is a symbol that remains; i.e., X is in $V_1 \cup T_1$. We know that $X \overset{*}{\underset{G}{\Rightarrow}} w$ for some w in T^*. Moreover, every symbol used in the derivation of w from X is also generating. Thus, $X \overset{*}{\underset{G_2}{\Rightarrow}} w$.

Since X was not eliminated in the second step, we also know that there are α and β such that $S \overset{*}{\underset{G_2}{\Rightarrow}} \alpha X \beta$. Further, every symbol used in this derivation is reachable, so $S \overset{*}{\underset{G_1}{\Rightarrow}} \alpha X \beta$.

We know that every symbol in $\alpha X \beta$ is reachable, and we also know that all these symbols are in $V_2 \cup T_2$, so each of them is generating in G_2. The derivation of some terminal string, say $\alpha X \beta \overset{*}{\underset{G_2}{\Rightarrow}} xwy$, involves only symbols that are reachable from S, because they are reached by symbols in $\alpha X \beta$. Thus, this derivation is also a derivation of G_1; that is,

$$S \overset{*}{\underset{G_1}{\Rightarrow}} \alpha X \beta \overset{*}{\underset{G_1}{\Rightarrow}} xwy$$

We conclude that X is useful in G_1. Since X is an arbitrary symbol of G_1, we conclude that G_1 has no useless symbols.

The last detail is that we must show $L(G_1) = L(G)$. As usual, to show two sets the same, we show each is contained in the other.

$L(G_1) \subseteq L(G)$: Since we have only eliminated symbols and productions from G to get G_1, it follows that $L(G_1) \subseteq L(G)$.

$L(G) \subseteq L(G_1)$: We must prove that if w is in $L(G)$, then w is in $L(G_1)$. If w is in $L(G)$, then $S \overset{*}{\underset{G}{\Rightarrow}} w$. Each symbol in this derivation is evidently both reachable and generating, so it is also a derivation of G_1. That is, $S \overset{*}{\underset{G_1}{\Rightarrow}} w$, and thus w is in $L(G_1)$. □

7.1.2 Computing the Generating and Reachable Symbols

Two points remain. How do we compute the set of generating symbols of a grammar, and how do we compute the set of reachable symbols of a grammar? For both problems, the algorithm we use tries its best to discover symbols of these types. We shall show that if the proper inductive constructions of these sets fails to discover a symbol to be generating or reachable, respectively, then the symbol is not of these types.

Let $G = (V,T,P,S)$ be a grammar. To compute the generating symbols of G, we perform the following induction.

BASIS: Every symbol of T is obviously generating; it generates itself.

INDUCTION: Suppose there is a production $A \to \alpha$, and every symbol of α is already known to be generating. Then A is generating. Note that this rule includes the case where $\alpha = \epsilon$; all variables that have ϵ as a production body are surely generating.

Example 7.3: Consider the grammar of Example 7.1. By the basis, a and b are generating. For the induction, we can use the production $A \to b$ to conclude that A is generating, and we can use the production $S \to a$ to conclude that S is generating. At that point, the induction is finished. We cannot use the production $S \to AB$, because B has not been established to be generating. Thus, the set of generating symbols is $\{a,b,A,S\}$. □

Theorem 7.4: The algorithm above finds all and only the generating symbols of G.

PROOF: For one direction, it is an easy induction on the order in which symbols are added to the set of generating symbols that each symbol added really is generating. We leave to the reader this part of the proof.

For the other direction, suppose X is a generating symbol, say $X \overset{*}{\underset{G}{\Rightarrow}} w$. We prove by induction on the length of this derivation that X is found to be generating.

BASIS: Zero steps. Then X is a terminal, and X is found in the basis.

INDUCTION: If the derivation takes n steps for $n > 0$, then X is a variable. Let the derivation be $X \Rightarrow \alpha \overset{*}{\Rightarrow} w$; that is, the first production used is $X \to \alpha$. Each symbol of α derives some terminal string that is a part of w, and that derivation must take fewer than n steps. By the inductive hypothesis, each symbol of α is found to be generating. The inductive part of the algorithm allows us to use production $X \to \alpha$ to infer that X is generating. □

Now, let us consider the inductive algorithm whereby we find the set of reachable symbols for the grammar $G = (V,T,P,S)$. Again, we can show that by trying our best to discover reachable symbols, any symbol we do not add to the reachable set is really not reachable.

BASIS: S is surely reachable.

INDUCTION: Suppose we have discovered that some variable A is reachable. Then for all productions with A in the head, all the symbols of the bodies of those productions are also reachable.

Example 7.5: Again start with the grammar of Example 7.1. By the basis, S is reachable. Since S has production bodies AB and a, we conclude that A, B, and a are reachable. B has no productions, but A has $A \to b$. We therefore conclude that b is reachable. Now, no more symbols can be added to the reachable set, which is $\{S, A, B, a, b\}$. □

Theorem 7.6: The algorithm above finds all and only the reachable symbols of G.

PROOF: This proof is another pair of simple inductions akin to Theorem 7.4. We leave these arguments as an exercise. □

7.1.3 Eliminating ϵ-Productions

Now, we shall show that ϵ-productions, while a convenience in many grammar-design problems, are not essential. Of course without a production that has an ϵ body, it is impossible to generate the empty string as a member of the language. Thus, what we actually prove is that if language L has a CFG, then $L - \{\epsilon\}$ has a CFG without ϵ-productions. If ϵ is not in L, then L itself is $L - \{\epsilon\}$, so L has a CFG with out ϵ-productions.

Our strategy is to begin by discovering which variables are "nullable." A variable A is *nullable* if $A \stackrel{*}{\Rightarrow} \epsilon$. If A is nullable, then whenever A appears in a production body, say $B \to CAD$, A might (or might not) derive ϵ. We make two versions of the production, one without A in the body ($B \to CD$), which corresponds to the case where A would have been used to derive ϵ, and the other with A still present ($B \to CAD$). However, if we use the version with A present, then we cannot allow A to derive ϵ. That proves not to be a problem, since we shall simply eliminate all productions with ϵ bodies, thus preventing any variable from deriving ϵ.

Let $G = (V, T, P, S)$ be a CFG. We can find all the nullable symbols of G by the following iterative algorithm. We shall then show that there are no nullable symbols except what the algorithm finds.

BASIS: If $A \to \epsilon$ is a production of G, then A is nullable.

INDUCTION: If there is a production $B \to C_1C_2\cdots C_k$, where each C_i is nullable, then B is nullable. Note that each C_i must be a variable to be nullable, so we only have to consider productions with all-variable bodies.

Theorem 7.7: In any grammar G, the only nullable symbols are the variables found by the algorithm above.

PROOF: For the "if" direction of the implied "A is nullable if and only if the algorithm identifies A as nullable," we simply observe that, by an easy induction on the order in which nullable symbols are discovered, that the each such symbol truly derives ϵ. For the "only-if" part, we can perform an induction on the length of the shortest derivation $A \stackrel{*}{\Rightarrow} \epsilon$.

BASIS: One step. Then $A \to \epsilon$ must be a production, and A is discovered in the basis part of the algorithm.

INDUCTION: Suppose $A \stackrel{*}{\Rightarrow} \epsilon$ by n steps, where $n > 1$. The first step must look like $A \Rightarrow C_1C_2\cdots C_k \stackrel{*}{\Rightarrow} \epsilon$, where each C_i derives ϵ by a sequence of fewer than n steps. By the inductive hypothesis, each C_i is discovered by the algorithm to be nullable. Thus, by the inductive step, A, thanks to the production $A \to C_1C_2\cdots C_k$, is found to be nullable. □

Now we give the construction of a grammar without ϵ-productions. Let $G = (V, T, P, S)$ be a CFG. Determine all the nullable symbols of G. We construct a new grammar $G_1 = (V, T, P_1, S)$, whose set of productions P_1 is determined as follows.

For each production $A \to X_1X_2\cdots X_k$ of P, where $k \geq 1$, suppose that m of the k X_i's are nullable symbols. The new grammar G_1 will have 2^m versions of this production, where the nullable X_i's, in all possible combinations are present or absent. There is one exception: if $m = k$, i.e., all symbols are nullable, then we do not include the case where all X_i's are absent. Also, note that if a production of the form $A \to \epsilon$ is in P, we do not place this production in P_1.

Example 7.8: Consider the grammar

$$S \to AB$$
$$A \to aAA \mid \epsilon$$
$$B \to bBB \mid \epsilon$$

First, let us find the nullable symbols. A and B are directly nullable because they have productions with ϵ as the body. Then, we find that S is nullable, because the production $S \to AB$ has a body consisting of nullable symbols only. Thus, all three variables are nullable.

Now, let us construct the productions of grammar G_1. First consider $S \to AB$. All symbols of the body are nullable, so there are four ways we could choose present or absent for A and B, independently. However, we are not allowed to choose to make all symbols absent, so there are only three productions:

$$S \to AB \mid A \mid B$$

Next, consider production $A \to aAA$. The second and third positions hold nullable symbols, so again there are four choices of present/absent. In this case,

all four choices are allowable, since the nonnullable symbol a will be present in any case. Our four choices yield productions:

$$A \to aAA \mid aA \mid aA \mid a$$

Note that the two middle choices happen to yield the same production, since it doesn't matter which of the A's we eliminate if we decide to eliminate one of them. Thus, the final grammar G_1 will only have three productions for A.

Similarly, the production B yields for G_1:

$$B \to bBB \mid bB \mid b$$

The two ϵ-productions of G yield nothing for G_1. Thus, the following productions:

$$\begin{aligned} S &\to AB \mid A \mid B \\ A &\to aAA \mid aA \mid a \\ B &\to bBB \mid bB \mid b \end{aligned}$$

constitute G_1. □

We conclude our study of the elimination of ϵ-productions by proving that the construction given above does not change the language, except that ϵ is no longer present if it was in the language of G. Since the construction obviously eliminates ϵ-productions, we shall have a complete proof of the claim that for every CFG G, there is a grammar G_1 with no ϵ-productions, such that

$$L(G_1) = L(G) - \{\epsilon\}$$

Theorem 7.9: If the grammar G_1 is constructed from G by the above construction for eliminating ϵ-productions, then $L(G_1) = L(G) - \{\epsilon\}$.

PROOF: We must show that if $w \neq \epsilon$, then w is in $L(G_1)$ if and only if w is in $L(G)$. As is often the case, we find it easier to prove a more general statement. In this case, we need to talk about the terminal strings that each variable generates, even though we only care what the start symbol S generates. Thus, we shall prove:

- $A \overset{*}{\underset{G_1}{\Rightarrow}} w$ if and only if $A \overset{*}{\underset{G}{\Rightarrow}} w$ and $w \neq \epsilon$.

In each case, the proof is an induction on the length of the derivation.

(Only-if) Suppose that $A \overset{*}{\underset{G_1}{\Rightarrow}} w$. Then surely $w \neq \epsilon$, because G_1 has no ϵ-productions. We must show by induction on the length of the derivation that $A \overset{*}{\underset{G}{\Rightarrow}} w$.

BASIS: One step. Then there is a production $A \to w$ in G_1. The construction of G_1 tells us that there is some production $A \to \alpha$ of G, such that α is w, with zero or more nullable variables interspersed. Then in G, $A \underset{G}{\Rightarrow} \alpha \overset{*}{\underset{G}{\Rightarrow}} w$, where the steps after the first, if any, derive ϵ from whatever variables there are in α.

INDUCTION: Suppose the derivation takes $n > 1$ steps. Then the derivation looks like $A \underset{G_1}{\Rightarrow} X_1X_2\cdots X_k \overset{*}{\underset{G_1}{\Rightarrow}} w$. The first production used must come from a production $A \to Y_1Y_2\cdots Y_m$, where the Y's are the X's, in order, with zero or more additional, nullable variables interspersed. Also, we can break w into $w_1w_2\cdots w_k$, where $X_i \overset{*}{\underset{G_1}{\Rightarrow}} w_i$ for $i = 1, 2, \ldots, k$. If X_i is a terminal, then $w_i = X_i$, and if X_i is a variable, then the derivation $X_i \overset{*}{\underset{G_1}{\Rightarrow}} w_i$ takes fewer than n steps. By the inductive hypothesis, we can conclude $X_i \overset{*}{\underset{G}{\Rightarrow}} w_i$.

Now, we construct a corresponding derivation in G as follows:

$$A \underset{G}{\Rightarrow} Y_1Y_2\cdots Y_m \overset{*}{\underset{G}{\Rightarrow}} X_1X_2\cdots X_k \overset{*}{\underset{G}{\Rightarrow}} w_1w_2\cdots w_k = w$$

The first step is application of the production $A \to Y_1Y_2\cdots Y_k$ that we know exists in G. The next group of steps represents the derivation of ϵ from each of the Y_j's that is not one of the X_i's. The final group of steps represents the derivations of the w_i's from the X_i's, which we know exist by the inductive hypothesis.

(If) Suppose $A \overset{*}{\underset{G}{\Rightarrow}} w$ and $w \neq \epsilon$. We show by induction on the length n of the derivation, that $A \overset{*}{\underset{G_1}{\Rightarrow}} w$.

BASIS: One step. Then $A \to w$ is a production of G. Since $w \neq \epsilon$, this production is also a production of G_1, and $A \overset{*}{\underset{G_1}{\Rightarrow}} w$.

INDUCTION: Suppose the derivation takes $n > 1$ steps. Then the derivation looks like $A \underset{G}{\Rightarrow} Y_1Y_2\cdots Y_m \overset{*}{\underset{G}{\Rightarrow}} w$. We can break $w = w_1w_2\cdots w_m$, such that $Y_i \overset{*}{\underset{G}{\Rightarrow}} w_i$ for $i = 1, 2, \ldots, m$. Let $X_1, X_2, \ldots, X_k$ be those of the Y_j's, in order, such that $w_j \neq \epsilon$. We must have $k \geq 1$, since $w \neq \epsilon$. Thus, $A \to X_1X_2\cdots X_k$ is a production of G_1.

We claim that $X_1X_2\cdots X_k \overset{*}{\underset{G}{\Rightarrow}} w$, since the only Y_j's that are not present among the X's were used to derive ϵ, and thus do not contribute to the derivation of w. Since each of the derivations $Y_j \overset{*}{\underset{G}{\Rightarrow}} w_j$ takes fewer than n steps, we may apply the inductive hypothesis and conclude that, if $w_j \neq \epsilon$, then $Y_j \overset{*}{\underset{G_1}{\Rightarrow}} w_j$. Thus, $A \underset{G_1}{\Rightarrow} X_1X_2\cdots X_k \overset{*}{\underset{G_1}{\Rightarrow}} w$.

Now, we complete the proof as follows. We know w is in $L(G_1)$ if and only if $S \overset{*}{\underset{G_1}{\Rightarrow}} w$. Letting $A = S$ in the above, we know that w is in $L(G_1)$ if and only if $S \overset{*}{\underset{G}{\Rightarrow}} w$ and $w \neq \epsilon$. That is, w is in $L(G_1)$ if and only if w is in $L(G)$ and $w \neq \epsilon$. □

7.1.4 Eliminating Unit Productions

A *unit production* is a production of the form $A \to B$, where both A and B are variables. These productions can be useful. For instance, in Example 5.27, we

saw how using unit productions $E \to T$ and $T \to F$ allowed us to create an unambiguous grammar for simple arithmetic expressions:

$$\begin{array}{lcl} I & \to & a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\ F & \to & I \mid (E) \\ T & \to & F \mid T * F \\ E & \to & T \mid E + T \end{array}$$

However, unit productions can complicate certain proofs, and they also introduce extra steps into derivations that technically need not be there. For instance, we could expand the T in production $E \to T$ in both possible ways, replacing it by the two productions $E \to F \mid T * F$. That change still doesn't eliminate unit productions, because we have introduced unit production $E \to F$ that was not previously part of the grammar. Further expanding $E \to F$ by the two productions for F gives us $E \to I \mid (E) \mid T * F$. We still have a unit production; it is $E \to I$. But if we further expand this I in all six possible ways, we get

$$E \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \mid (E) \mid T * F$$

Now the unit production for E is gone. Note that $E \to a$ is *not* a unit production, since the lone symbol in the body is a terminal, rather than a variable as is required for unit productions.

The technique suggested above — expand unit productions until they disappear — often works. However, it can fail if there is a cycle of unit productions, such as $A \to B$, $B \to C$, and $C \to A$. The technique that is guaranteed to work involves first finding all those pairs of variables A and B such that $A \stackrel{*}{\Rightarrow} B$ using a sequence of unit productions only. Note that it is possible for $A \stackrel{*}{\Rightarrow} B$ to be true even though no unit productions are involved. For instance, we might have productions $A \to BC$ and $C \to \epsilon$.

Once we have determined all such pairs, we can replace any sequence of derivation steps in which $A \Rightarrow B_1 \Rightarrow B_2 \Rightarrow \cdots \Rightarrow B_n \Rightarrow \alpha$ by a production that uses the nonunit production $B_n \to \alpha$ directly from A; that is, $A \to \alpha$. To begin, here is the inductive construction of the pairs (A, B) such that $A \stackrel{*}{\Rightarrow} B$ using only unit productions. Call such a pair a ***unit pair.***

BASIS: (A, A) is a unit pair for any variable A. That is, $A \stackrel{*}{\Rightarrow} A$ by zero steps.

INDUCTION: Suppose we have determined that (A, B) is a unit pair, and $B \to C$ is a production, where C is a variable. Then (A, C) is a unit pair.

Example 7.10: Consider the expression grammar of Example 5.27, which we reproduced above. The basis gives us the unit pairs (E, E), (T, T), (F, F), and (I, I). For the inductive step, we can make the following inferences:

1. (E, E) and the production $E \to T$ gives us unit pair (E, T).
2. (E, T) and the production $T \to F$ gives us unit pair (E, F).
3. (E, F) and the production $F \to I$ gives us unit pair (E, I).

4. (T, T) and the production $T \to F$ gives us unit pair (T, F).
5. (T, F) and the production $F \to I$ gives us unit pair (T, I).
6. (F, F) and the production $F \to I$ gives us unit pair (F, I).

There are no more pairs that can be inferred, and in fact these ten pairs represent all the derivations that use nothing but unit productions. □

The pattern of development should by now be familiar. There is an easy proof that our proposed algorithm does get all the pairs we want. We then use the knowledge of those pairs to remove unit productions from a grammar and show that the language of the two grammars is the same.

Theorem 7.11: The algorithm above finds exactly the unit pairs for a CFG G.

PROOF: In one direction, it is an easy induction on the order in which the pairs are discovered, that if (A, B) is found to be a unit pair, then $A \stackrel{*}{\underset{G}{\Rightarrow}} B$ using only unit productions. We leave this part of the proof to you.

In the other direction, suppose that $A \stackrel{*}{\underset{G}{\Rightarrow}} B$ using unit productions only. We can show by induction on the length of the derivation that the pair (A, B) will be found.

BASIS: Zero steps. Then $A = B$, and the pair (A, B) is added in the basis.

INDUCTION: Suppose $A \stackrel{*}{\Rightarrow} B$ using n steps, for some $n > 0$, each step being the application of a unit production. Then the derivation looks like

$$A \stackrel{*}{\Rightarrow} C \Rightarrow B$$

The derivation $A \stackrel{*}{\Rightarrow} C$ takes $n - 1$ steps, so by the inductive hypothesis, we discover the pair (A, C). Then the inductive part of the algorithm combines the pair (A, C) with the production $C \to B$ to infer the pair (A, B). □

To eliminate unit productions, we proceed as follows. Given a CFG $G = (V, T, P, S)$, construct CFG $G_1 = (V, T, P_1, S)$:

1. Find all the unit pairs of G.
2. For each unit pair (A, B), add to P_1 all the productions $A \to \alpha$, where $B \to \alpha$ is a nonunit production in P. Note that $A = B$ is possible; in that way, P_1 contains all the nonunit productions in P.

Example 7.12: Let us continue with Example 7.10, which performed step (1) of the construction above for the expression grammar of Example 5.27. Figure 7.1 summarizes step (2) of the algorithm, where we create the new set of productions by using the first member of a pair as the head and all the nonunit bodies for the second member of the pair as the production bodies.

The final step is to eliminate the unit productions from the grammar of Fig. 7.1. The resulting grammar:

Pair	Productions
(E,E)	$E \to E+T$
(E,T)	$E \to T*F$
(E,F)	$E \to (E)$
(E,I)	$E \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(T,T)	$T \to T*F$
(T,F)	$T \to (E)$
(T,I)	$T \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(F,F)	$F \to (E)$
(F,I)	$F \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(I,I)	$I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

Figure 7.1: Grammar constructed by step (2) of the unit-production-elimination algorithm

$$E \to E+T \mid T*F \mid (E) \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$
$$T \to T*F \mid (E) \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$
$$F \to (E) \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$
$$I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$

has no unit productions, yet generates the same set of expressions as the grammar of Fig. 5.19. □

Theorem 7.13: If grammar G_1 is constructed from grammar G by the algorithm described above for eliminating unit productions, then $L(G_1) = L(G)$.

PROOF: We show that w is in $L(G)$ if and only if w is in $L(G_1)$.

(If) Suppose $S \underset{G_1}{\overset{*}{\Rightarrow}} w$. Since every production of G_1 is equivalent to a sequence of zero or more unit productions of G followed by a nonunit production of G, we know that $\alpha \underset{G_1}{\Rightarrow} \beta$ implies $\alpha \underset{G}{\overset{*}{\Rightarrow}} \beta$. That is, every step of a derivation in G_1 can be replaced by one or more derivation steps in G. If we put these sequences of steps together, we conclude that $S \underset{G}{\overset{*}{\Rightarrow}} w$.

(Only-if) Suppose now that w is in $L(G)$. Then by the equivalences in Section 5.2, we know that w has a leftmost derivation, i.e., $S \underset{lm}{\Rightarrow} w$. Whenever a unit production is used in a leftmost derivation, the variable of the body becomes the leftmost variable, and so is immediately replaced. Thus, the leftmost derivation in grammar G can be broken into a sequence of steps in which zero or more unit productions are followed by a nonunit production. Note that any nonunit production that is not preceded by a unit production is a "step" by itself. Each of these steps can be performed by one production of G_1, because the construction of G_1 created exactly the productions that reflect zero or more unit productions followed by a nonunit production. Thus, $S \underset{G_1}{\overset{*}{\Rightarrow}} w$. □

We can now summarize the various simplifications described so far. We want to convert any CFG G into an equivalent CFG that has no useless symbols, ϵ-productions, or unit productions. Some care must be taken in the order of application of the constructions. A safe order is:

1. Eliminate ϵ-productions.
2. Eliminate unit productions.
3. Eliminate useless symbols.

You should notice that, just as in Section 7.1.1, where we had to order the two steps properly or the result might have useless symbols, we must order the three steps above as shown, or the result might still have some of the features we thought we were eliminating.

Theorem 7.14: If G is a CFG generating a language that contains at least one string other than ϵ, then there is another CFG G_1 such that $L(G_1) = L(G) - \{\epsilon\}$, and G_1 has no ϵ-productions, unit productions, or useless symbols.

PROOF: Start by eliminating the ϵ-productions by the method of Section 7.1.3. If we then eliminate unit productions by the method of Section 7.1.4, we do not introduce any ϵ-productions, since the bodies of the new productions are each identical to some body of an old production. Finally, we eliminate useless symbols by the method of Section 7.1.1. As this transformation only eliminates productions and symbols, never introducing a new production, the resulting grammar will still be devoid of ϵ-productions and unit productions. □

7.1.5 Chomsky Normal Form

We complete our study of grammatical simplifications by showing that every nonempty CFL without ϵ has a grammar G in which all productions are in one of two simple forms, either:

1. $A \to BC$, where A, B, and C, are each variables, or
2. $A \to a$, where A is a variable and a is a terminal.

Further, G has no useless symbols. Such a grammar is said to be in *Chomsky Normal Form*, or CNF.[1]

To put a grammar in CNF, start with one that satisfies the restrictions of Theorem 7.14; that is, the grammar has no ϵ-productions, unit productions, or useless symbols. Every production of such a grammar is either of the form $A \to a$, which is already in a form allowed by CNF, or it has a body of length 2 or more. Our tasks are to:

[1] N. Chomsky is the linguist who first proposed context-free grammars as a way to describe natural languages, and who proved that every CFG could be converted to this form. Interestingly, CNF does not appear to have important uses in natural linguistics, although we shall see it has several other uses, such as an efficient test for membership of a string in a context-free language (Section 7.4.4).

a) Arrange that all bodies of length 2 or more consist only of variables.

b) Break bodies of length 3 or more into a cascade of productions, each with a body consisting of two variables.

The construction for (a) is as follows. For every terminal a that appears in a body of length 2 or more, create a new variable, say A. This variable has only one production, $A \to a$. Now, we use A in place of a everywhere a appears in a body of length 2 or more. At this point, every production has a body that is either a single terminal or at least two variables and no terminals.

For step (b), we must break those productions $A \to B_1 B_2 \cdots B_k$, for $k \geq 3$, into a group of productions with two variables in each body. We introduce $k-2$ new variables, $C_1, C_2, \ldots, C_{k-2}$. The original production is replaced by the $k-1$ productions

$$A \to B_1 C_1, \quad C_1 \to B_2 C_2, \ldots, C_{k-3} \to B_{k-2} C_{k-2}, \quad C_{k-2} \to B_{k-1} B_k$$

Example 7.15: Let us convert the grammar of Example 7.12 to CNF. For part (a), notice that there are eight terminals, a, b, 0, 1, +, *, (, and), each of which appears in a body that is not a single terminal. Thus, we must introduce eight new variables, corresponding to these terminals, and eight productions in which the new variable is replaced by its terminal. Using the obvious initials as the new variables, we introduce:

$$A \to a \quad B \to b \quad Z \to 0 \quad O \to 1$$
$$P \to + \quad M \to * \quad L \to (\quad R \to)$$

If we introduce these productions, and replace every terminal in a body that is other than a single terminal by the corresponding variable, we get the grammar shown in Fig. 7.2.

$$\begin{array}{lcl} E & \to & EPT \mid TMF \mid LER \mid a \mid b \mid IA \mid IB \mid IZ \mid IO \\ T & \to & TMF \mid LER \mid a \mid b \mid IA \mid IB \mid IZ \mid IO \\ F & \to & LER \mid a \mid b \mid IA \mid IB \mid IZ \mid IO \\ I & \to & a \mid b \mid IA \mid IB \mid IZ \mid IO \\ A & \to & a \\ B & \to & b \\ Z & \to & 0 \\ O & \to & 1 \\ P & \to & + \\ M & \to & * \\ L & \to & (\\ R & \to &) \end{array}$$

Figure 7.2: Making all bodies either a single terminal or several variables

Now, all productions are in Chomsky Normal Form except for those with the bodies of length 3: EPT, TMF, and LER. Some of these bodies appear in more than one production, but we can deal with each body once, introducing one extra variable for each. For EPT, we introduce new variable C_1, and replace the one production, $E \to EPT$, where it appears, by $E \to EC_1$ and $C_1 \to PT$.

For TMF we introduce new variable C_2. The two productions that use this body, $E \to TMF$ and $T \to TMF$, are replaced by $E \to TC_2$, $T \to TC_2$, and $C_2 \to MF$. Then, for LER we introduce new variable C_3 and replace the three productions that use it, $E \to LER$, $T \to LER$, and $F \to LER$, by $E \to LC_3$, $T \to LC_3$, $F \to LC_3$, and $C_3 \to ER$. The final grammar, which is in CNF, is shown in Fig. 7.3. □

$$\begin{array}{lcl} E & \to & EC_1 \mid TC_2 \mid LC_3 \mid a \mid b \mid IA \mid IB \mid IZ \mid IO \\ T & \to & TC_2 \mid LC_3 \mid a \mid b \mid IA \mid IB \mid IZ \mid IO \\ F & \to & LC_3 \mid a \mid b \mid IA \mid IB \mid IZ \mid IO \\ I & \to & a \mid b \mid IA \mid IB \mid IZ \mid IO \\ A & \to & a \\ B & \to & b \\ Z & \to & 0 \\ O & \to & 1 \\ P & \to & + \\ M & \to & * \\ L & \to & (\\ R & \to &) \\ C_1 & \to & PT \\ C_2 & \to & MF \\ C_3 & \to & ER \end{array}$$

Figure 7.3: Making all bodies either a single terminal or two variables

Theorem 7.16: If G is a CFG whose language contains at least one string other than ϵ, then there is a grammar G_1 in Chomsky Normal Form, such that $L(G_1) = L(G) - \{\epsilon\}$.

PROOF: By Theorem 7.14, we can find CFG G_2 such that $L(G_2) = L(G) - \{\epsilon\}$, and such that G_2 has no useless symbols, ϵ-productions, or unit productions. The construction that converts G_2 to CNF grammar G_1 changes the productions in such a way that each production of G_1 can be simulated by one or more productions of G_2. Conversely, the introduced variables of G_2 each have only one production, so they can only be used in the manner intended. More formally, we prove that w is in $L(G_2)$ if and only if w is in $L(G_1)$.

(Only-if) If w has a derivation in G_2, it is easy to replace each production used, say $A \to X_1X_2\cdots X_k$, by a sequence of productions of G_1. That is, one step in the derivation in G_2 becomes one or more steps in the derivation of w using the productions of G_1. First, if any X_i is a terminal, we know G_1 has a corresponding variable B_i and a production $B_i \to X_i$. Then, if $k > 2$, G_1 has productions $A \to B_1C_1$, $C_1 \to B_2C_2$, and so on, where B_i is either the introduced variable for terminal X_i or X_i itself, if X_i is a variable. These productions simulate in G_1 one step of a derivation of G_2 that uses $A \to X_1X_2\cdots X_k$. We conclude that there is a derivation of w in G_1, so w is in $L(G_1)$.

(If) Suppose w is in $L(G_1)$. Then there is a parse tree in G_1, with S at the root and yield w. We convert this tree to a parse tree of G_2 that also has root S and yield w.

First, we "undo" part (b) of the CNF construction. That is, suppose there is a node labeled A, with two children labeled B_1 and C_1, where C_1 is one of the variables introduced in part (b). Then this portion of the parse tree must look like Fig. 7.4(a). That is, because these introduced variables each have only one production, there is only one way that they can appear, and all the variables introduced to handle the production $A \to B_1B_2\cdots B_k$ must appear together, as shown.

Any such cluster of nodes in the parse tree may be replaced by the production that they represent. The parse-tree transformation is suggested by Fig. 7.4(b).

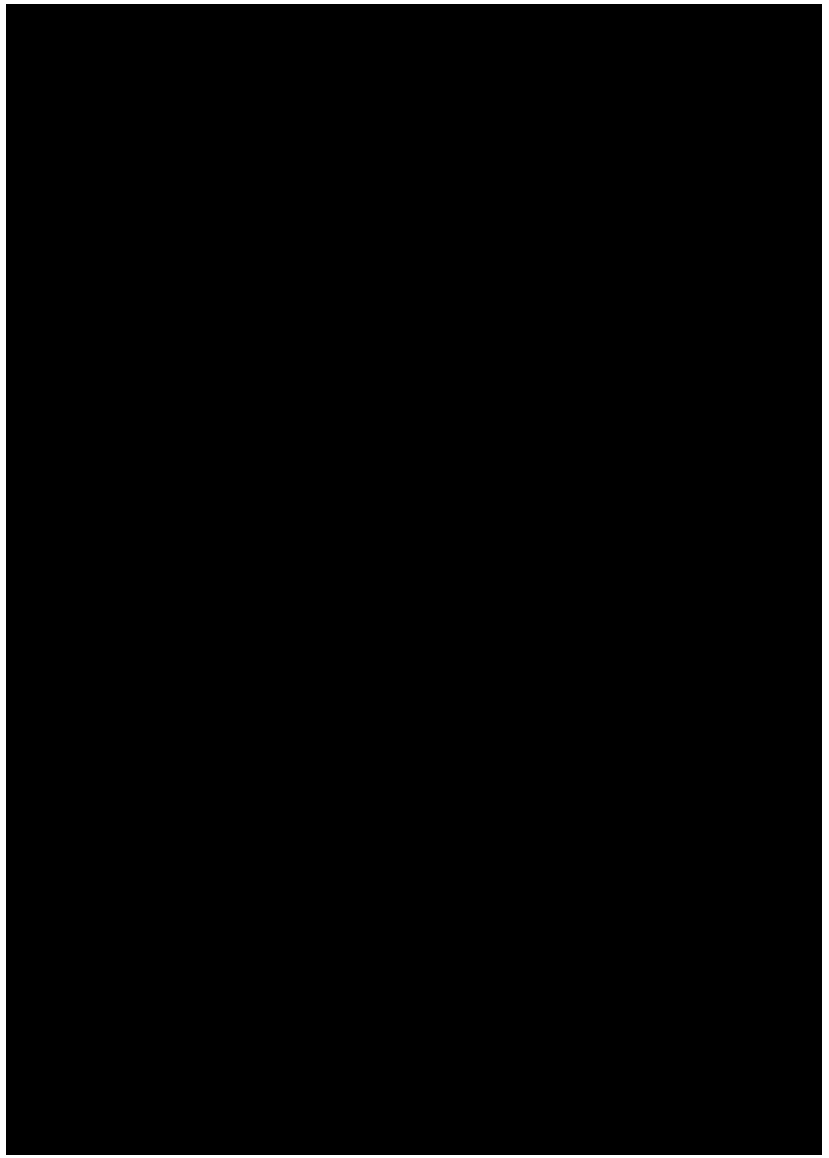
The resulting parse tree is still not necessarily a parse tree of G_2. The reason is that step (a) in the CNF construction introduced other variables that derive single terminals. However, we can identify these in the current parse tree and replace a node labeled by such a variable A and its one child labeled a, by a single node labeled a. Now, every interior node of the parse tree forms a production of G_2. Since w is the yield of a parse tree in G_2, we conclude that w is in $L(G_2)$. □

7.1.6 Exercises for Section 7.1

! **Exercise 7.1.1:** Find a grammar equivalent to

$$\begin{array}{lcl} S & \to & AB \mid CA \\ A & \to & a \\ B & \to & BC \mid AB \\ C & \to & aB \mid b \end{array}$$

with no useless symbols.



Greibach Normal Form

There is another interesting normal form for grammars that we shall not prove. Every nonempty language without ϵ is $L(G)$ for some grammar G each of whose productions are of the form $A \to a\alpha$, where a is a terminal and α is a string of zero or more variables. Converting a grammar to this form is complex, even if we simplify the task by, say, starting with a Chomsky-Normal-Form grammar. Roughly, we expand the first variable of each production, until we get a terminal. However, because there can be cycles, where we never reach a terminal, it is necessary to "short-circuit" the process, creating a production that introduces a terminal as the first symbol of the body and has variables following it to generate all the sequences of variables that might have been generated on the way to generation of that terminal.

This form, called *Greibach Normal Form*, after Sheila Greibach, who first gave a way to construct such grammars, has several interesting consequences. Since each use of a production introduces exactly one terminal into a sentential form, a string of length n has a derivation of exactly n steps. Also, if we apply the PDA construction of Theorem 6.13 to a Greibach-Normal-Form grammar, then we get a PDA with no ϵ-rules, thus showing that it is always possible to eliminate such transitions of a PDA.

* **Exercise 7.1.2:** Begin with the grammar:

$$\begin{array}{lcl} S & \to & ASB \mid \epsilon \\ A & \to & aAS \mid a \\ B & \to & SbS \mid A \mid bb \end{array}$$

a) Eliminate ϵ-productions.

b) Eliminate any unit productions in the resulting grammar.

c) Eliminate any useless symbols in the resulting grammar.

d) Put the resulting grammar into Chomsky normal form.

Exercise 7.1.3: Repeat Exercise 7.1.2 for the following grammar:

$$\begin{array}{lcl} S & \to & 0A0 \mid 1B1 \mid BB \\ A & \to & C \\ B & \to & S \mid A \\ C & \to & S \mid \epsilon \end{array}$$

Exercise 7.1.4: Repeat Exercise 7.1.2 for the following grammar:

$$\begin{array}{lcl} S & \to & AAA \mid B \\ A & \to & aA \mid B \\ B & \to & \epsilon \end{array}$$

Exercise 7.1.5: Repeat Exercise 7.1.2 for the following grammar:

$$\begin{array}{lcl} S & \to & aAa \mid bBb \mid \epsilon \\ A & \to & C \mid a \\ B & \to & C \mid b \\ C & \to & CDE \mid \epsilon \\ D & \to & A \mid B \mid ab \end{array}$$

Exercise 7.1.6: Design a CNF grammar for the set of strings of balanced parentheses. You need not start from any particular non-CNF grammar.

!! **Exercise 7.1.7:** Suppose G is a CFG with p productions, and no production body longer than n. Show that if $A \overset{*}{\underset{G}{\Rightarrow}} \epsilon$, then there is a derivation of ϵ from A of no more than $(n^p - 1)/(n - 1)$ steps. How close can you actually come to this bound?

! **Exercise 7.1.8:** Suppose we have a grammar G with n productions, none of them ϵ-productions, and we we convert this grammar to CNF.

a) Show that the CNF grammar has at most $O(n^2)$ productions.

b) Show that it is possible for the CNF grammar to have a number of productions proportional to n^2. *Hint*: Consider the construction that eliminates unit productions.

Exercise 7.1.9: Provide the inductive proofs needed to complete the following theorems:

a) The part of Theorem 7.4 where we show that discovered symbols really are generating.

b) Both directions of Theorem 7.6, where we show the correctness of the algorithm in Section 7.1.2 for detecting the reachable symbols.

c) The part of Theorem 7.11 where we show that all pairs discovered really are unit pairs.

*! **Exercise 7.1.10:** Is it possible to find, for every context-free language without ϵ, a grammar such that all its productions are either of the form $A \to BCD$ (i.e., a body consisting of three variables), or $A \to a$ (i.e., a body consisting of a single terminal)? Give either a proof or a counterexample.

Exercise 7.1.11: In this exercise, we shall show that for every context-free language L containing at least one string other than ϵ, there is a CFG in Greibach normal form that generates $L - \{\epsilon\}$. Recall that a Greibach normal form (GNF) grammar is one where every production body starts with a terminal. The construction will be done using a series of lemmas and constructions.

a) Suppose that a CFG G has a production $A \to \alpha B \beta$, and all the productions for B are $B \to \gamma_1 \mid \gamma_2 \mid \cdots \mid \gamma_n$. Then if we replace $A \to \alpha B \beta$ by all the productions we get by substituting some body of a B-production for B, that is, $A \to \alpha\gamma_1\beta \mid \alpha\gamma_2\beta \mid \cdots \mid \alpha\gamma_n\beta$, the resulting grammar generates the same language as G.

In what follows, assume that the grammar G for L is in Chomsky Normal Form, and that the variables are called $A_1, A_2, \ldots, A_k$.

*! b) Show that, by repeatedly using the transformation of part (a), we can convert G to an equivalent grammar in which every production body for A_i either starts with a terminal or starts with A_j, for some $j \geq i$. In either case, all symbols after the first in any production body are variables.

! c) Suppose G_1 is the grammar that we get by performing step (b) on G. Suppose that A_i is any variable, and let $A \to A_i\alpha_1 \mid \cdots \mid A_i\alpha_m$ be all the A_i-productions that have a body beginning with A_i. Let

$$A_i \to \beta_1 \mid \cdots \mid \beta_p$$

be all the other A_i-productions. Note that each β_j must start with either a terminal or a variable with index higher than j. Introduce a new variable B_i, and replace the first group of m productions by

$$A_i \to \beta_1 B_i \mid \cdots \mid \beta_p B_i$$
$$B_i \to \alpha_1 B_i \mid \alpha_1 \mid \cdots \mid \alpha_m B_i \mid \alpha_m$$

Prove that the resulting grammar generates the same language as G and G_1.

*! d) Let G_2 be the grammar that results from step (c). Note that all the A_i productions have bodies that begin with either a terminal or an A_j for $j > i$. Also, all the B_i productions have bodies that begin with either a terminal or some A_j. Prove that G_2 has an equivalent grammar in GNF. *Hint*: First fix the productions for A_k, then A_{k-1}, and so on, down to A_1, using part (a). Then fix the B_i productions in any order, again using part (a).

Exercise 7.1.12: Use the construction of Exercise 7.1.11 to convert the grammar

$$\begin{aligned} S &\to AA \mid 0 \\ A &\to SS \mid 1 \end{aligned}$$

to GNF.

7.2 The Pumping Lemma for Context-Free Languages

Now, we shall develop a tool for showing that certain languages are not context-free. The theorem, called the "pumping lemma for context-free languages," says that in any sufficiently long string in a CFL, it is possible to find at most two short, nearby substrings, that we can "pump" in tandem. That is, we may repeat both of the strings i times, for any integer i, and the resulting string will still be in the language.

We may contrast this theorem with the analogous pumping lemma for regular languages, Theorem 4.1, which says we can always find one small string to pump. The difference is seen when we consider a language like $L = \{0^n1^n \mid n \geq 1\}$. We can show it is not regular, by fixing n and pumping a substring of 0's, thus getting a string with more 0's than 1's. However, the CFL pumping lemma states only that we can find two small strings, so we might be forced to use a string of 0's and a string of 1's, thus generating only strings in L when we "pump." That outcome is fortunate, because L is a CFL, and thus we should not be able to use the CFL pumping lemma to construct strings not in L.

7.2.1 The Size of Parse Trees

Our first step in deriving a pumping lemma for CFL's is to examine the shape and size of parse trees. One of the uses of CNF is to turn parse trees into binary trees. These trees have some convenient properties, one of which we exploit here.

Theorem 7.17: Suppose we have a parse tree according to a Chomsky-Normal-Form grammar $G = (V, T, P, S)$, and suppose that the yield of the tree is a terminal string w. If the length of the longest path is n, then $|w| \leq 2^{n-1}$.

PROOF: The proof is a simple induction on n.

BASIS: $n = 1$. Recall that the length of a path in a tree is the number of edges, i.e., one less than the number of nodes. Thus, a tree with a maximum path length of 1 consists of only a root and one leaf labeled by a terminal. String w is this terminal, so $|w| = 1$. Since $2^{n-1} = 2^0 = 1$ in this case, we have proved the basis.

INDUCTION: Suppose the longest path has length n, and $n > 1$. The root of the tree uses a production, which must be of the form $A \to BC$, since $n > 1$; i.e., we could not start the tree using a production with a terminal. No path in the subtrees rooted at B and C can have length greater than $n - 1$, since these paths exclude the edge from the root to its child labeled B or C. Thus, by the inductive hypothesis, these two subtrees each have yields of length at most 2^{n-2}. The yield of the entire tree is the concatenation of these two yields,

and therefore has length at most $2^{n-2} + 2^{n-2} = 2^{n-1}$. Thus, the inductive step is proved. □

7.2.2 Statement of the Pumping Lemma

The pumping lemma for CFL's is quite similar to the pumping lemma for regular languages, but we break each string z in the CFL L into five parts, and we pump the second and fourth, in tandem.

Theorem 7.18: (The pumping lemma for context-free languages) Let L be a CFL. Then there exists a constant n such that if z is any string in L such that $|z|$ is at least n, then we can write $z = uvwxy$, subject to the following conditions:

1. $|vwx| \leq n$. That is, the middle portion is not too long.
2. $vx \neq \epsilon$. Since v and x are the pieces to be "pumped," this condition says that at least one of the strings we pump must not be empty.
3. For all $i \geq 0$, uv^iwx^iy is in L. That is, the two strings v and x may be "pumped" any number of times, including 0, and the resulting string will still be a member of L.

PROOF: Our first step is to find a Chomsky-Normal-Form grammar G for L. Technically, we cannot find such a grammar if L is the CFL $\emptyset$ or $\{\epsilon\}$. However, if $L = \emptyset$ then the statement of the theorem, which talks about a string z in L surely cannot be violated, since there is no such z in $\emptyset$. Also, the CNF grammar G will actually generate $L - \{\epsilon\}$, but that is again not of importance, since we shall surely pick $n > 0$, in which case z cannot be ϵ anyway.

Now, starting with a CNF grammar $G = (V, T, P, S)$ such that $L(G) = L - \{\epsilon\}$, let G have m variables. Choose $n = 2^m$. Next, suppose that z in L is of length at least n. By Theorem 7.17, any parse tree whose longest path is of length m or less must have a yield of length $2^{m-1} = n/2$ or less. Such a parse tree cannot have yield z, because z is too long. Thus, any parse tree with yield z has a path of length at least $m + 1$.

Figure 7.5 suggests the longest path in the tree for z, where k is at least m and the path is of length $k+1$. Since $k \geq m$, there are at least $m+1$ occurrences of variables $A_0, A_1, \ldots, A_k$ on the path. As there are only m different variables in V, at least two of the last $m + 1$ variables on the path (that is, A_{k-m} through A_k, inclusive) must be the same variable. Suppose $A_i = A_j$, where $k - m \leq i < j \leq k$.

Then it is possible to divide the tree as shown in Fig. 7.6. String w is the yield of the subtree rooted at A_j. Strings v and x are the strings to the left and right, respectively, of w in the yield of the larger subtree rooted at A_i. Note that, since there are no unit productions, v and x could not both be ϵ, although one could be. Finally, u and y are those portions of z that are to the left and right, respectively, of the subtree rooted at A_i.

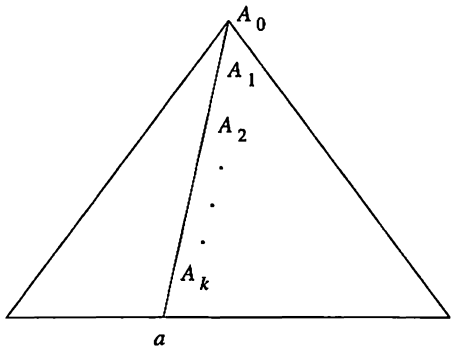


Figure 7.5: Every sufficiently long string in L must have a long path in its parse tree

If $A_i = A_j = A$, then we can construct new parse trees from the original tree, as suggested in Fig. 7.7(a). First, we may replace the subtree rooted at A_i, which has yield vwx, by the subtree rooted at A_j, which has yield w. The reason we can do so is that both of these trees have root labeled A. The resulting tree is suggested in Fig. 7.7(b); it has yield uwy and corresponds to the case $i = 0$ in the pattern of strings uv^iwx^iy.

Another option is suggested by Fig. 7.7(c). There, we have replaced the subtree rooted at A_j by the entire subtree rooted at A_i. Again, the justification is that we are substituting one tree with root labeled A for another tree with the same root label. The yield of this tree is uv^2wx^2y. Were we to then replace the subtree of Fig. 7.7(c) with yield w by the larger subtree with yield vwx, we would have a tree with yield uv^3wx^3y, and so on, for any exponent i. Thus, there are parse trees in G for all strings of the form uv^iwx^iy, and we have almost proved the pumping lemma.

The remaining detail is condition (1), which says that $|vwx| \leq n$. However, we picked A_i to be close to the bottom of the tree; that is, $k - i \leq m$. Thus, the longest path in the subtree rooted at A_i is no greater than $m + 1$. By Theorem 7.17, the subtree rooted at A_i has a yield whose length is no greater than $2^m = n$. □

7.2.3 Applications of the Pumping Lemma for CFL's

Notice that, like the earlier pumping lemma for regular languages, we use the CFl pumping lemma as an "adversary game, as follows."

1. We pick a language L that we want to show is not a CFL.

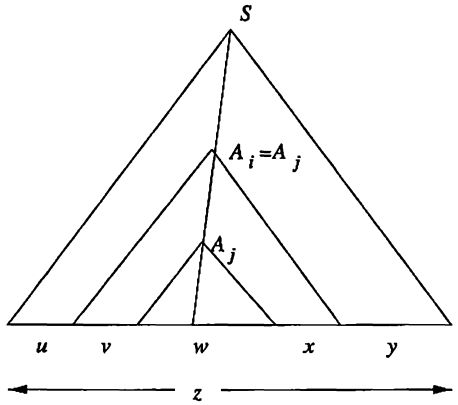


Figure 7.6: Dividing the string w so it can be pumped

2. Our "adversary" gets to pick n, which we do not know, and we therefore must plan for any possible n.

3. We get to pick z, and may use n as a parameter when we do so.

4. Our adversary gets to break z into $uvwxy$, subject only to the constraints that $|vwx| \leq n$ and $vx \neq \epsilon$.

5. We "win" the game, if we can, by picking i and showing that uv^iwx^iy is not in L.

We shall now see some examples of languages that we can prove, using the pumping lemma, not to be context-free. Our first example shows that, while context-free languages can match two groups of symbols for equality or inequality, they cannot match three such groups.

Example 7.19: Let L be the language $\{0^n1^n2^n \mid n \geq 1\}$. That is, L consists of all strings in $0^+1^+2^+$ with an equal number of each symbol, e.g., 012, 001122, and so on. Suppose L were context-free. Then there is an integer n given to us by the pumping lemma.[2] Let us pick $z = 0^n1^n2^n$.

Suppose the "adversary" breaks z as $z = uvwxy$, where $|vwx| \leq n$ and v and x are not both ϵ. Then we know that vwx cannot involve both 0's and 2's, since the last 0 and the first 2 are separated by $n+1$ positions. We shall prove that L contains some string known not to be in L, thus contradicting the assumption that L is a CFL. The cases are as follows:

[2]Remember that this n is the constant provided by the pumping lemma, and it has nothing to do with the local variable n used in the definition of L itself.

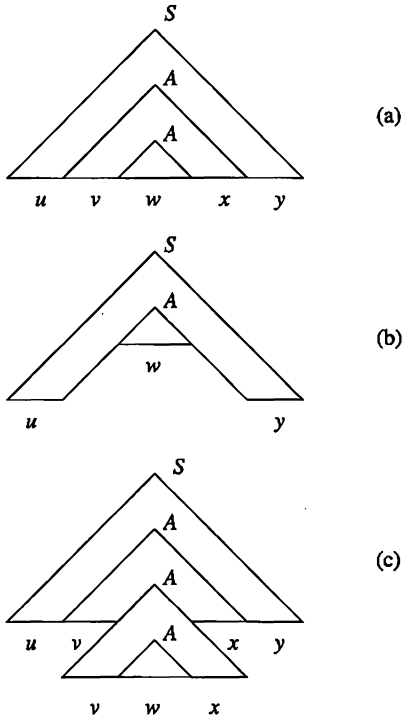


Figure 7.7: Pumping strings v and x zero times and pumping them twice

1. vwx has no 2's. Then vx consists of only 0's and 1's, and has at least one of these symbols. Then uwy, which would have to be in L by the pumping lemma, has n 2's, but has fewer than n 0's or fewer than n 1's, or both. It therefore does not belong in L, and we conclude L is not a CFL in this case.

2. vwx has no 0's. Similarly, uwy has n 0's, but fewer 1's or fewer 2's. It therefore is not in L.

Whichever case holds, we conclude that L has a string we know not to be in L. This contradiction allows us to conclude that our assumption was wrong; L is *not* a CFL. □

Another thing that CFL's cannot do is match two pairs of equal numbers of symbols, provided that the pairs interleave. The idea is made precise in the following example of a proof of non-context-freeness using the pumping lemma.

Example 7.20: Let L be the language $\{0^i1^j2^i3^j \mid i \geq 1 \text{ and } j \geq 1\}$. If L is context-free, let n be the constant for L, and pick $z = 0^n1^n2^n3^n$. We may write $z = uvwxy$ subject to the usual constraints $|vwx| \leq n$ and $vx \neq \epsilon$. Then vwx is either contained in the substring of one symbol, or it straddles two adjacent symbols.

If vwx consists of only one symbol, then uwy has n of three different symbols and fewer than n of the fourth symbol. Thus, it cannot be in L. If vwx straddles two symbols, say the 1's and 2's, then uwy is missing either some 1's or some 2's, or both. Suppose it is missing 1's. As there are n 3's, this string cannot be in L. Similarly, if it is missing 2's, then as it has n 0's, uwy cannot be in L. We have contradicted the assumption that L is a CFL and conclude that it is not. □

As a final example, we shall show that CFL's cannot match two strings of arbitrary length, if the strings are chosen from an alphabet of more than one symbol. An implication of this observation, incidentally, is that grammars are not a suitable mechanism for enforcing certain "semantic" constraints in programming languages, such as the common requirement that an identifier be declared before use. In practice, another mechanism, such as a "symbol table" is used to record declared identifiers, and we do not try to design a parser that, by itself, checks for "definition prior to use."

Example 7.21: Let $L = \{ww \mid w \text{ is in } \{0,1\}^*\}$. That is, L consists of repeating strings, such as ϵ, 0101, 00100010, or 110110. If L is context-free, then let n be its pumping-lemma constant. Consider the string $z = 0^n1^n0^n1^n$. This string is 0^n1^n repeated, so z is in L.

Following the pattern of the previous examples, we can break $z = uvwxy$, such that $|vwx| \leq n$ and $vx \neq \epsilon$. We shall show that uwy is not in L, and thus show L not to be a context-free language, by contradiction.

First, observe that, since $|vwx| \leq n$, $|uwy| \geq 3n$. Thus, if uwy is some repeating string, say tt, then t is of length at least $3n/2$. There are several cases to consider, depending where vwx is within z.

1. Suppose vwx is within the first n 0's. In particular, let vx consist of k 0's, where $k > 0$. Then uwy begins with $0^{n-k}1^n$. Since $|uwy| = 4n - k$, we know that if $uwy = tt$, then $|t| = 2n - k/2$. Thus, t does not end until after the first block of 1's; i.e., t ends in 0. But uwy ends in 1, and so it cannot equal tt.

2. Suppose vwx straddles the first block of 0's and the first block of 1's. It may be that vx consists only of 0's, if $x = \epsilon$. Then, the argument that uwy is not of the form tt is the same as case (1). If vx has at least one

we say the problem is "decidable," so TM's that always halt figure importantly into decidability theory in Chapter 9.

8.2.7 Exercises for Section 8.2

Exercise 8.2.1: Show the ID's of the Turing machine of Fig. 8.9 if the input tape contains:

* a) 00.

b) 000111.

c) 00111.

! **Exercise 8.2.2:** Design Turing machines for the following languages:

* a) The set of strings with an equal number of 0's and 1's.

b) $\{a^nb^nc^n \mid n \geq 1\}$.

c) $\{ww^R \mid w \text{ is any string of 0's and 1's}\}$.

Exercise 8.2.3: Design a Turing machine that takes as input a number N and adds 1 to it in binary. To be precise, the tape initially contains a \$ followed by N in binary. The tape head is initially scanning the \$ in state q_0. Your TM should halt with N+1, in binary, on its tape, scanning the leftmost symbol of $N+1$, in state q_f. You may destroy the \$ in creating $N+1$, if necessary. For instance, $q_0\$10011 \vdash^* \$q_f10100$, and $q_0\$11111 \vdash^* q_f100000$.

a) Give the transitions of your Turing machine, and explain the purpose of each state.

b) Show the sequence of ID's of your TM when given input \$111.

*! **Exercise 8.2.4:** In this exercise we explore the equivalence between function computation and language recognition for Turing machines. For simplicity, we shall consider only functions from nonnegative integers to nonnegative integers, but the ideas of this problem apply to any computable functions. Here are the two central definitions:

- Define the *graph* of a function f to be the set of all strings of the form $[x, f(x)]$, where x is a nonnegative integer in binary, and $f(x)$ is the value of function f with argument x, also written in binary.

- A Turing machine is said to *compute* function f if, started with any nonnegative integer x on its tape, in binary, it halts (in any state) with $f(x)$, in binary, on its tape.

Answer the following, with informal, but clear constructions.

a) Show how, given a TM that computes f, you can construct a TM that accepts the graph of f as a language.

b) Show how, given a TM that accepts the graph of f, you can construct a TM that computes f.

c) A function is said to be *partial* if it may be undefined for some arguments. If we extend the ideas of this exercise to partial functions, then we do not require that the TM computing f halts if its input x is one of the integers for which $f(x)$ is not defined. Do your constructions for parts (a) and (b) work if the function f is partial? If not, explain how you could modify the construction to make it work.

Exercise 8.2.5: Consider the Turing machine

$$M = (\{q_0, q_1, q_2, q_f\}, \{0,1\}, \{0,1,B\}, \delta, q_0, B, \{q_f\})$$

Informally but clearly describe the language $L(M)$ if δ consists of the following sets of rules:

* a) $\delta(q_0,0) = (q_1,1,R)$; $\delta(q_1,1) = (q_0,0,R)$; $\delta(q_1,B) = (q_f,B,R)$.

b) $\delta(q_0,0) = (q_0,B,R)$; $\delta(q_0,1) = (q_1,B,R)$; $\delta(q_1,1) = (q_1,B,R)$; $\delta(q_1,B) = (q_f,B,R)$.

! c) $\delta(q_0,0) = (q_1,1,R)$; $\delta(q_1,1) = (q_2,0,L)$; $\delta(q_2,1) = (q_0,1,R)$; $\delta(q_1,B) = (q_f,B,R)$.

8.3 Programming Techniques for Turing Machines

Our goal is to give you a sense of how a Turing machine can be used to compute in a manner not unlike that of a conventional computer. Eventually, we want to convince you that a TM is exactly as powerful as a conventional computer. In particular, we shall learn that the Turing machine can perform the sort of calculations on other Turing machines that we saw performed in Section 8.1.2 by a program that examined other programs. This "introspective" ability of both Turing machines and computer programs is what enables us to prove problems undecidable.

To make the ability of a TM clearer, we shall present a number of examples of how we might think of the tape and finite control of the Turing machine. None of these tricks extend the basic model of the TM; they are only notational conveniences. Later, we shall use them to simulate extended Turing-machine models that have additional features — for instance, more than one tape — by the basic TM model.

8.3.1 Storage in the State

We can use the finite control not only to represent a position in the "program" of the Turing machine, but to hold a finite amount of data. Figure 8.13 suggests this technique (as well as another idea: multiple tracks). There, we see the finite control consisting of not only a "control" state q, but three data elements A, B, and C. The technique requires no extension to the TM model; we merely think of the state as a tuple. In the case of Fig. 8.13, we should think of the state as $[q, A, B, C]$. Regarding states this way allows us to describe transitions in a more systematic way, often making the strategy behind the TM program more transparent.

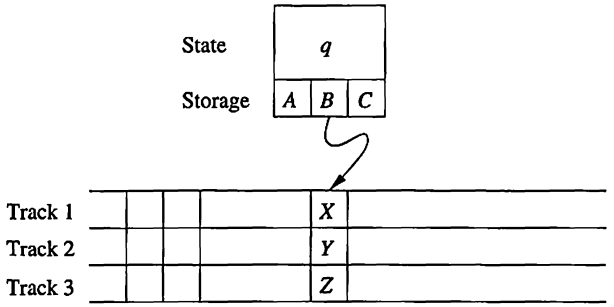


Figure 8.13: A Turing machine viewed as having finite-control storage and multiple tracks

Example 8.6: We shall design a TM

$$M = (Q, \{0,1\}, \{0,1,B\}, \delta, [q_0, B], \{[q_1, B]\})$$

that remembers in its finite control the first symbol (0 or 1) that it sees, and checks that it does not appear elsewhere on its input. Thus, M accepts the language $\mathbf{01^*+10^*}$. Accepting regular languages such as this one does not stress the ability of Turing machines, but it will serve as a simple demonstration.

The set of states Q is $\{q_0, q_1\} \times \{0,1,B\}$. That is, the states may be thought of as pairs with two components:

a) A control portion, q_0 or q_1, that remembers what the TM is doing. Control state q_0 indicates that M has not yet read its first symbol, while q_1 indicates that it *has* read the symbol, and is checking that it does not appear elsewhere, by moving right and hoping to reach a blank cell.

b) A data portion, which remembers the first symbol seen, which must be 0 or 1. The symbol B in this component means that no symbol has been read.

The transition function δ of M is as follows:

1. $\delta([q_0, B], a) = ([q_1, a], a, R)$ for $a = 0$ or $a = 1$. Initially, q_0 is the control state, and the data portion of the state is B. The symbol scanned is copied into the second component of the state, and M moves right, entering control state q_1 as it does so.

2. $\delta([q_1, a], \overline{a}) = ([q_1, a], \overline{a}, R)$ where $\overline{a}$ is the "complement" of a, that is, 0 if $a = 1$ and 1 if $a = 0$. In state q_1, M skips over each symbol 0 or 1 that is different from the one it has stored in its state, and continues moving right.

3. $\delta([q_1, a], B) = ([q_1, B], B, R)$ for $a = 0$ or $a = 1$. If M reaches the first blank, it enters the accepting state $[q_1, B]$.

Notice that M has no definition for $\delta([q_1, a], a)$ for $a = 0$ or $a = 1$. Thus, if M encounters a second occurrence of the symbol it stored initially in its finite control, it halts without having entered the accepting state. □

8.3.2 Multiple Tracks

Another useful "trick" is to think of the tape of a Turing machine as composed of several tracks. Each track can hold one symbol, and the tape alphabet of the TM consists of tuples, with one component for each "track." Thus, for instance, the cell scanned by the tape head in Fig. 8.13 contains the symbol $[X, Y, Z]$. Like the technique of storage in the finite control, using multiple tracks does not extend what the Turing machine can do. It is simply a way to view tape symbols and to imagine that they have a useful structure.

Example 8.7: A common use of multiple tracks is to treat one track as holding the data and a second track as holding a mark. We can check off each symbol as we "use" it, or we can keep track of a small number of positions within the data by marking only those positions. Examples 8.2 and 8.4 were two instances of this technique, but in neither example did we think explicitly of the tape as if it were composed of tracks. In the present example, we shall use a second track explicitly to recognize the non-context-free language

$$L_{wcw} = \{wcw \mid w \text{ is in } (0+1)^+\}$$

The Turing machine we shall design is:

$$M = (Q, \Sigma, \Gamma, \delta, [q_1, B], [B, B], \{[q_9, B]\})$$

where:

Q: The set of states is $\{q_1, q_2, \ldots, q_9\} \times \{0, 1, B\}$, that is, pairs consisting of a control state q_i and a data component: 0, 1, or blank. We again use the technique of storage in the finite control, as we allow the state to remember an input symbol 0 or 1.

Γ: The set of tape symbols is $\{B, *\} \times \{0, 1, c, B\}$. The first component, or track, can be either blank or "checked," represented by the symbols B and $*$, respectively. We use the $*$ to check off symbols of the first and second groups of 0's and 1's, eventually confirming that the string to the left of the center marker c is the same as the string to its right. The second component of the tape symbol is what we think of as the tape symbol itself. That is, we may think of the symbol $[B, X]$ as if it were the tape symbol X, for $X = 0, 1, c, B$.

Σ: The input symbols are $[B, 0]$ and $[B, 1]$, which, as just mentioned, we identify with 0 and 1, respectively.

δ: The transition function δ is defined by the following rules, in which a and b each may stand for either 0 or 1.

1. $\delta([q_1, B], [B, a]) = ([q_2, a], [*, a], R)$. In the initial state, M picks up the symbol a (which can be either 0 or 1), stores it in its finite control, goes to control state q_2, "checks off" the symbol it just scanned, and moves right. Notice that by changing the first component of the tape symbol from B to $*$, it performs the check-off.

2. $\delta([q_2, a], [B, b]) = ([q_2, a], [B, b], R)$. M moves right, looking for the symbol c. Remember that a and b can each be either 0 or 1, independently, but cannot be c.

3. $\delta([q_2, a], [B, c]) = ([q_3, a], [B, c], R)$. When M finds the c, it continues to move right, but changes to control state q_3.

4. $\delta([q_3, a], [*, b]) = ([q_3, a], [*, b], R)$. In state q_3, M continues past all checked symbols.

5. $\delta([q_3, a], [B, a]) = ([q_4, B], [*, a], L)$. If the first unchecked symbol that M finds is the same as the symbol in its finite control, it checks this symbol, because it has matched the corresponding symbol from the first block of 0's and 1's. M goes to control state q_4, dropping the symbol from its finite control, and starts moving left.

6. $\delta([q_4, B], [*, a]) = ([q_4, B], [*, a], L)$. M moves left over checked symbols.

7. $\delta([q_4, B], [B, c]) = ([q_5, B], [B, c], L)$. When M encounters the symbol c, it switches to state q_5 and continues left. In state q_5, M must make a decision, depending on whether or not the symbol immediately to the left of the c is checked or unchecked. If checked, then we have already considered the entire first block of 0's and 1's — those to the left of the c. We must make sure that all the 0's and 1's to the right of the c are also checked, and accept if no unchecked symbols remain to the right of the c. If the symbol immediately to the left of the c is unchecked, we find the leftmost unchecked symbol, pick it up, and start the cycle that began in state q_1.

8. $\delta([q_5, B], [B, a]) = ([q_6, B], [B, a], L)$. This branch covers the case where the symbol to the left of c is unchecked. M goes to state q_6 and continues left, looking for a checked symbol.

9. $\delta([q_6, B], [B, a]) = ([q_6, B], [B, a], L)$. As long as symbols are unchecked, M remains in state q_6 and proceeds left.

10. $\delta([q_6, B], [*, a]) = ([q_1, B], [*, a], R)$. When the checked symbol is found, M enters state q_1 and moves right to pick up the first unchecked symbol.

11. $\delta([q_5, B], [*, a]) = ([q_7, B], [*, a], R)$. Now, let us pick up the branch from state q_5 where we have just moved left from the c and find a checked symbol. We start moving right again, entering state q_7.

12. $\delta([q_7, B], [B, c]) = ([q_8, B], [B, c], R)$. In state q_7 we shall surely see the c. We enter state q_8 as we do so, and proceed right.

13. $\delta([q_8, B], [*, a]) = ([q_8, B], [*, a], R)$. M moves right in state q_8, skipping over any checked 0's or 1's that it finds.

14. $\delta([q_8, B], [B, B]) = ([q_9, B], [B, B], R)$. If M reaches a blank cell is state q_8 without encountering any unchecked 0 or 1, then M accepts. If M first finds an unchecked 0 or 1, then the blocks before and after the c do not match, and M halts without accepting.

□

8.3.3 Subroutines

As with programs in general, it helps to think of Turing machines as built from a collection of interacting components, or "subroutines." A Turing-machine subroutine is a set of states that perform some useful process. This set of states includes a start state and another state that temporarily has no moves, and that serves as the "return" state to pass control to whatever other set of states called the subroutine. The "call" of a subroutine occurs whenever there is a transition to its initial state. Since the TM has no mechanism for remembering a "return address," that is, a state to go to after it finishes, should our design of a TM call for one subroutine to be called from several states, we can make copies of the subroutine, using a new set of states for each copy. The "calls" are made to the start states of different copies of the subroutine, and each copy "returns" to a different state.

Example 8.8: We shall design a TM to implement the function "multiplication." That is, our TM will start with 0^m10^n1 on its tape, and will end with 0^{mn} on the tape. An outline of the strategy is:

1. The tape will, in general, have one nonblank string of the form $0^i10^n10^{kn}$ for some k.

2. In one basic step, we change a 0 in the first group to B and add n 0's to the last group, giving us a string of the form $0^{i-1}10^n10^{(k+1)n}$.

3. As a result, we copy the group of n 0's to the end m times, once each time we change a 0 in the first group to B. When the first group of 0's is completely changed to blanks, there will be mn 0's in the last group.

4. The final step is to change the leading 10^n1 to blanks, and we are done.

The heart of this algorithm is a subroutine, which we call Copy. This subroutine implements step (2) above, copying the block of n 0's to the end. More precisely, Copy converts an ID of the form $0^{m-k}1q_10^n10^{(k-1)n}$ to ID $0^{m-k}1q_50^n10^{kn}$. Figure 8.14 shows the transitions of subroutine Copy. This subroutine marks the first 0 with an X, moves right in state q_2 until it finds a blank, copies the 0 there, and moves left in state q_3 to find the marker X. It repeats this cycle until in state q_1 it finds a 1 instead of a 0. At that point, it uses state q_4 to change the X's back to 0's, and ends in state q_5.

The complete multiplication Turing machine starts in state q_0. The first thing it does is go, in several steps, from ID $q_00^m10^n$ to ID $0^{m-1}1q_10^n1$. The transitions needed are shown in the portion of Fig. 8.15 to the left of the subroutine call; these transitions involve states q_0 and q_6 only.

Then, to the right of the subroutine call in Fig. 8.15 we see states q_7 through q_{12}. The purpose of states q_7, q_8, and q_9 is to take control after Copy has just copied a block of n 0's, and is in ID $0^{m-k}1q_50^n10^{kn}$. Eventually, these states bring us to state $q_00^{m-k}10^n10^{kn}$. At that point, the cycle starts again, and Copy is called to copy the block of n 0's again.

As an exception, in state q_8 the TM may find that all m 0's have been changed to blanks (i.e., $k = m$). In that case, a transition to state q_{10} occurs. This state, with the help of state q_{11}, changes the leading 10^n1 to blanks and enters the halting state q_{12}. At this point, the TM is in ID $q_{12}0^{mn}$, and its job is done. □

8.3.4 Exercises for Section 8.3

! **Exercise 8.3.1:** Redesign your Turing machines from Exercise 8.2.2 to take advantage of the programming techniques discussed in Section 8.3.

! **Exercise 8.3.2:** A common operation in Turing-machine programs involves "shifting over." Ideally, we would like to create an extra cell at the current head position, in which we could store some character. However, we cannot edit the tape in this way. Rather, we need to move the contents of each of the cells to the right of the current head position one cell right, and then find our way back to the current head position. Show how to perform this operation. *Hint*: Leave a special symbol to mark the position to which the head must return.

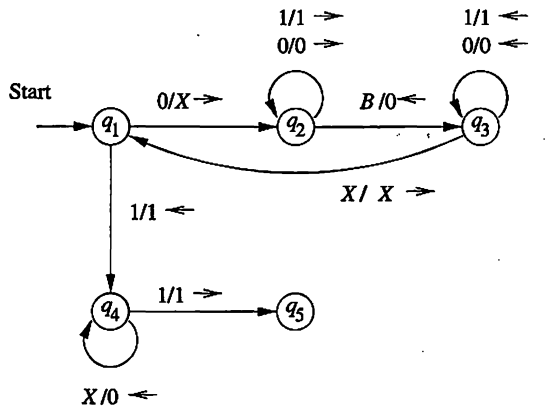


Figure 8.14: The subroutine Copy

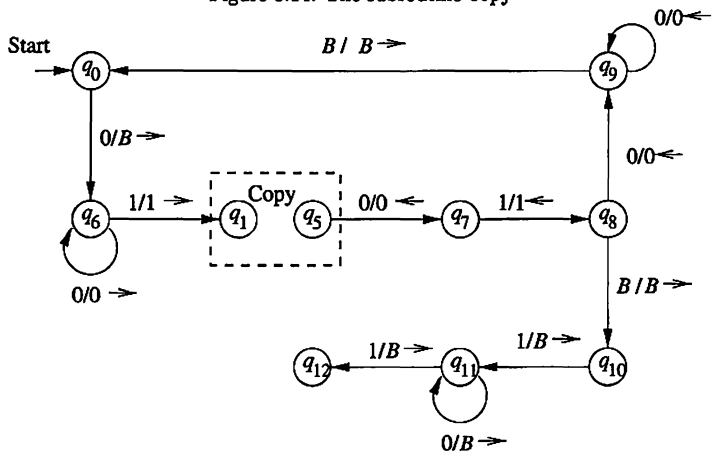


Figure 8.15: The complete multiplication program uses the subroutine Copy

* **Exercise 8.3.3:** Design a subroutine to move a TM head from its current position to the right, skipping over all 0's, until reaching a 1 or a blank. If the current position does not hold 0, then the TM should halt. You may assume that there are no tape symbols other than 0, 1, and B (blank). Then, use this subroutine to design a TM that accepts all strings of 0's and 1's that do not have two 1's in a row.

8.4 Extensions to the Basic Turing Machine

In this section we shall see certain computer models that are related to Turing machines and have the same language-recognizing power as the basic model of a TM with which we have been working. One of these, the multitape Turing machine, is important because it is much easier to see how a multitape TM can simulate real computers (or other kinds of Turing machines), compared with the single-tape model we have been studying. Yet the extra tapes add no power to the model, as far as the ability to accept languages is concerned.

We then consider the nondeterministic Turing machine, an extension of the basic model that is allowed to make any of a finite set of choices of move in a given situation. This extension also makes "programming" Turing machines easier in some circumstances, but adds no language-defining power to the basic model.

8.4.1 Multitape Turing Machines

A multitape TM is as suggested by Fig. 8.16. The device has a finite control (state), and some finite number of tapes. Each tape is divided into cells, and each cell can hold any symbol of the finite tape alphabet. As in the single-tape TM, the set of tape symbols includes a blank, and has a subset called the input symbols, of which the blank is not a member. The set of states includes an initial state and some accepting states. Initially:

1. The input, a finite sequence of input symbols, is placed on the first tape.
2. All other cells of all the tapes hold the blank.
3. The finite control is in the initial state.
4. The head of the first tape is at the left end of the input.
5. All other tape heads are at some arbitrary cell. Since tapes other than the first tape are completely blank, it does not matter where the head is placed initially; all cells of these tapes "look" the same.

A move of the multitape TM depends on the state and the symbol scanned by each of the tape heads. In one move, the multitape TM does the following:

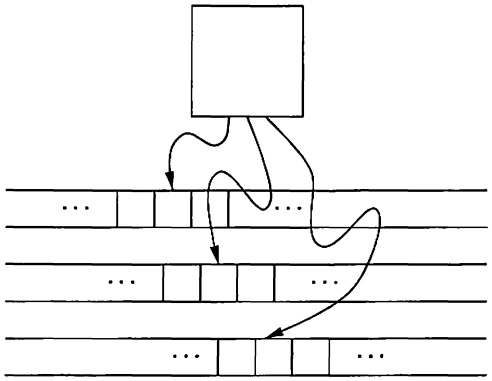


Figure 8.16: A multitape Turing machine

1. The control enters a new state, which could be the same as the previous state.
2. On each tape, a new tape symbol is written on the cell scanned. Any of these symbols may be the same as the symbol previously there.
3. Each of the tape heads makes a move, which can be either left, right, or stationary. The heads move independently, so different heads may move in different directions, and some may not move at all.

We shall not give the formal notation of transition rules, whose form is a straightforward generalization of the notation for the one-tape TM, except that directions are now indicated by a choice of L, R, or S. For the one-tape machine, we did not allow the head to remain stationary, so the S option was not present. You should be able to imagine an appropriate notation for instantaneous descriptions of the configuration of a multitape TM; we shall not give this notation formally. Multitape Turing machines, like one-tape TM's, accept by entering an accepting state.

8.4.2 Equivalence of One-Tape and Multitape TM's

Recall that the recursively enumerable languages are defined to be those accepted by a one-tape TM. Surely, multitape TM's accept all the recursively enumerable languages, since a one-tape TM *is* a multitape TM. However, are there languages that are not recursively enumerable, yet are accepted by multitape TM's? The answer is "no," and we prove this fact by showing how to simulate a multitape TM by a one-tape TM.

Theorem 8.9: Every language accepted by a multitape TM is recursively enumerable.

PROOF: The proof is suggested by Fig. 8.17. Suppose language L is accepted by a k-tape TM M. We simulate M with a one-tape TM N whose tape we think of as having $2k$ tracks. Half these tracks hold the tapes of M, and the other half of the tracks each hold only a single marker that indicates where the head for the corresponding tape of M is currently located. Figure 8.17 assumes $k = 2$. The second and fourth tracks hold the contents of the first and second tapes of M, track 1 holds the position of the head of tape 1, and track 3 holds the position of the second tape head.

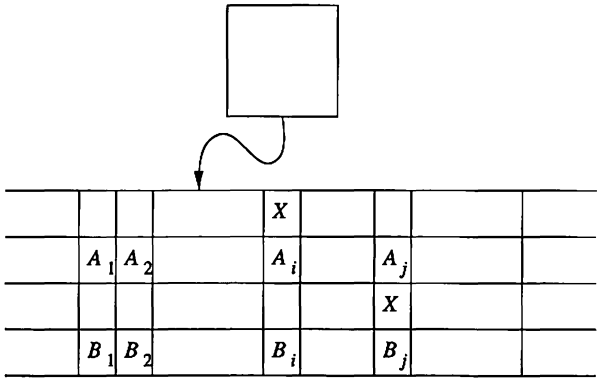


Figure 8.17: Simulation of a two-tape Turing machine by a one-tape Turing machine

To simulate a move of M, N's head must visit the k head markers. So that N not get lost, it must remember how many head markers are to its left at all times; that count is stored as a component of N's finite control. After visiting each head marker and storing the scanned symbol in a component of its finite control, N knows what tape symbols are being scanned by each of M's heads. N also knows the state of M, which it stores in N's own finite control. Thus, N knows what move M will make.

N now revisits each of the head markers on its tape, changes the symbol in the track representing the corresponding tapes of M, and moves the head markers left or right, if necessary. Finally, N changes the state of M as recorded in its own finite control. At this point, N has simulated one move of M.

We select as N's accepting states all those states that record M's state as one of the accepting states of M. Thus, whenever the simulated M accepts, N also accepts, and N does not accept otherwise. □

A Reminder About Finiteness

A common fallacy is to confuse a value that is finite at any time with a set of values that is finite. The many-tapes-to-one construction may help us appreciate the difference. In that construction, we used tracks on the tape to record the positions of the tape heads. Why could we not store these positions as integers in the finite control? Carelessly, one could argue that after n moves, the TM can have tape head positions that must be within n positions of original head positions, and so the head only has to store integers up to n.

The problem is that, while the positions are finite at any time, the complete set of positions possible at any time is infinite. If the state is to represent any head position, then there must be a data component of the state that has any integer as value. This component forces the set of states to be infinite, even if only a finite number of them can be used at any finite time. The definition of a Turing machine requires that the *set* of states be finite. Thus, it is not permissible to store a tape-head position in the finite control.

8.4.3 Running Time and the Many-Tapes-to-One Construction

Let us now introduce a concept that will become quite important later: the "time complexity" or "running time" of a Turing machine. We say the *running time* of TM M on input w is the number of steps that M makes before halting. If M doesn't halt on w, then the running time of M on w is infinite. The *time complexity* of TM M is the function $T(n)$ that is the maximum, over all inputs w of length n, of the running time of M on w. For Turing machines that do not halt on all inputs, $T(n)$ may be infinite for some or even all n. However, we shall pay special attention to TM's that do halt on all inputs, and in particular, those that have a polynomial time complexity $T(n)$; Section 10.1 initiates this study.

The construction of Theorem 8.9 seems clumsy. In fact, the constructed one-tape TM may take much more running time than the multitape TM. However, the amounts of time taken by the two Turing machines are commensurate in a weak sense: the one-tape TM takes time that is no more than the square of the time taken by the other. While "squaring" is not a very strong guarantee, it does preserve polynomial running time. We shall see in Chapter 10 that:

a) The difference between polynomial time and higher growth rates in running time is really the divide between what we can solve by computer and what is in practice not solvable.

b) Despite extensive research, the running time needed to solve many problems has not been resolved closer than to within some polynomial. Thus, the question of whether we are using a one-tape or multitape TM to solve the problem is not crucial when we examine the running time needed to solve a particular problem.

The argument that the running times of the one-tape and multitape TM's are within a square of each other is as follows.

Theorem 8.10: The time taken by the one-tape TM N of Theorem 8.9 to simulate n moves of the k-tape TM M is $O(n^2)$.

PROOF: After n moves of M, the tape head markers cannot have separated by more than $2n$ cells. Thus, if N starts at the leftmost marker, it has to move no more than $2n$ cells right, to find all the head markers. It can then make an excursion leftward, changing the contents of the simulated tapes of M, and moving head markers left or right as needed. Doing so requires no more than $2n$ moves left, plus at most $2k$ moves to reverse direction and write a marker X in the cell to the right (in the case that a tape head of M moves right).

Thus, the number of moves by N needed to simulate one of the first n moves is no more than $4n + 2k$. Since k is a constant, independent of the number of moves simulated, this number of moves is $O(n)$. To simulate n moves requires no more than n times this amount, or $O(n^2)$. □

8.4.4 Nondeterministic Turing Machines

A *nondeterministic* Turing machine (*NTM*) differs from the deterministic variety we have been studying by having a transition function δ such that for each state q and tape symbol X, $\delta(q, X)$ is a set of triples

$$\{(q_1, Y_1, D_1),\ (q_2, Y_2, D_2), \ldots, (q_k, Y_k, D_k)\}$$

where k is any finite integer. The NTM can choose, at each step, any of the triples to be the next move. It cannot, however, pick a state from one, a tape symbol from another, and the direction from yet another.

The language accepted by an NTM M is defined in the expected manner, in analogy with the other nondeterministic devices, such as NFA's and PDA's, that we have studied. That is, M accepts an input w if there is any sequence of choices of move that leads from the initial ID with w as input, to an ID with an accepting state. The existence of other choices that do *not* lead to an accepting state is irrelevant, as it is for the NFA or PDA.

The NTM's accept no languages not accepted by a deterministic TM (or *DTM* if we need to emphasize that it is deterministic). The proof involves showing that for every NTM M_N, we can construct a DTM M_D that explores the ID's that M_N can reach by any sequence of its choices. If M_D finds one that has an accepting state, then M_D enters an accepting state of its own. M_D must be systematic, putting new ID's on a queue, rather than a stack, so that after some finite time M_D has simulated all sequences of up to k moves of M_N, for $k = 1, 2, \ldots$.

Theorem 8.11: If M_N is a nondeterministic Turing machine, then there is a deterministic Turing machine M_D such that $L(M_N) = L(M_D)$.

PROOF: M_D will be designed as a multitape TM, sketched in Fig. 8.18. The first tape of M_D holds a sequence of ID's of M_N, including the state of M_N. One ID of M_N is marked as the "current" ID, whose successor ID's are in the process of being discovered. In Fig. 8.18, the third ID is marked by an x along with the inter-ID separator, which is the $*$. All ID's to the left of the current one have been explored and can be ignored subsequently.

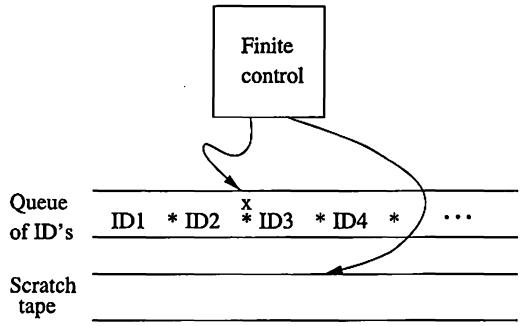


Figure 8.18: Simulation of an NTM by a DTM

To process the current ID, M_D does the following:

1. M_D examines the state and scanned symbol of the current ID. Built into the finite control of M_D is the knowledge of what choices of move M_N has for each state and symbol. If the state in the current ID is accepting, then M_D accepts and simulates M_N no further.

2. However, if the state is not accepting, and the state-symbol combination has k moves, then M_D uses its second tape to copy the ID and then make k copies of that ID at the end of the sequence of ID's on tape 1.

3. M_D modifies each of those k ID's according to a different one of the k choices of move that M_N has from its current ID.

4. M_D returns to the marked, current ID, erases the mark, and moves the mark to the next ID to the right. The cycle then repeats with step (1).

It should be clear that the simulation is accurate, in the sense that M_D will only accept if it finds that M_N can enter an accepting ID. However, we need to confirm that if M_N enters an accepting ID after a sequence of n of its own moves, then M_D will eventually make that ID the current ID and will accept.

Suppose that m is the maximum number of choices M_N has in any configuration. Then there is one initial ID of M_N, at most m ID's that M_N can reach after one move, at most m^2 ID's M_N can reach after two moves, and so on. Thus, after n moves, M_N can reach at most $1 + m + m^2 + \cdots + m^n$ ID's. This number is at most nm^n ID's.

The order in which M_D explores ID's of M_N is "breadth first"; that is, it explores all ID's reachable by 0 moves (i.e., the initial ID), then all ID's reachable by one move, then those reachable by two moves, and so on. In particular, M_D will make current, and consider the successors of, all ID's reachable by up to n moves before considering any ID's that are only reachable by more than n moves.

As a consequence, the accepting ID of M_N will be considered by M_D among the first nm^n ID's that it considers. We only care that M_D considers this ID in some finite time, and this bound is sufficient to assure us that the accepting ID is considered eventually. Thus, if M_N accepts, then so does M_D. Since we already observed that if M_D accepts it does so only because M_N accepts, we conclude that $L(M_N) = L(M_D)$. □

Notice that the constructed deterministic TM may take exponentially more time than the nondeterministic TM. It is unknown whether or not this exponential slowdown is necessary. In fact, Chapter 10 is devoted to this question and the consequences of someone discovering a better way to simulate NTM's deterministically.

8.4.5 Exercises for Section 8.4

Exercise 8.4.1: Informally but clearly describe multitape Turing machines that accept each of the languages of Exercise 8.2.2. Try to make each of your Turing machines run in time proportional to the input length.

Exercise 8.4.2: Here is the transition function of a nondeterministic TM $M = (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, B\}, \delta, q_0, B, \{q_2\})$:

δ	0	1	B
q_0	$\{(q_0, 1, R)\}$	$\{(q_1, 0, R)\}$	$\emptyset$
q_1	$\{(q_1, 0, R), (q_0, 0, L)\}$	$\{(q_1, 1, R), (q_0, 1, L)\}$	$\{(q_2, B, R)\}$
q_2	$\emptyset$	$\emptyset$	$\emptyset$

Show the ID's reachable from the initial ID if the input is:

* a) 01.

b) 011.

! **Exercise 8.4.3:** Informally but clearly describe nondeterministic Turing machines — multitape if you like — that accept the following languages. Try to

take advantage of nondeterminism to avoid iteration and save time in the nondeterministic sense. That is, prefer to have your NTM branch a lot, while each branch is short.

* a) The language of all strings of 0's and 1's that have some string of length 100 that repeats, not necessarily consecutively. Formally, this language is the set of strings of 0's and 1's of the form $wxyxz$, where $|x| = 100$, and w, y, and z are of arbitrary length.

b) The language of all strings of the form $w_1\#w_2\#\cdots\#w_n$, for any n, such that each w_i is a string of 0's and 1's, and for some j, w_j is the integer j in binary.

c) The language of all strings of the same form as (b), but for at least two values of j, we have w_j equal to j in binary.

! **Exercise 8.4.4:** Consider the nondeterministic Turing machine

$$M = (\{q_0, q_1, q_2, q_f\}, \{0,1\}, \{0,1,B\}, \delta, q_0, B, \{q_f\})$$

Informally but clearly describe the language $L(M)$ if δ consists of the following sets of rules: $\delta(q_0, 0) = \{(q_0, 1, R), (q_1, 1, R)\}$; $\delta(q_1, 1) = \{(q_2, 0, L)\}$; $\delta(q_2, 1) = \{(q_0, 1, R)\}$; $\delta(q_1, B) = \{(q_f, B, R)\}$.

* **Exercise 8.4.5:** Consider a nondeterministic TM whose tape is infinite in both directions. At some time, the tape is completely blank, except for one cell, which holds the symbol \$. The head is currently at some blank cell, and the state is q.

a) Write transitions that will enable the NTM to enter state p, scanning the \$.

! b) Suppose the TM were deterministic instead. How would you enable it to find the \$ and enter state p?

Exercise 8.4.6: Design the following 2-tape TM to accept the language of all strings of 0's and 1's with an equal number of each. The first tape contains the input, and is scanned from left to right. The second tape is used to store the excess of 0's over 1's, or vice-versa, in the part of the input seen so far. Specify the states, transitions, and the intuitive purpose of each state.

Exercise 8.4.7: In this exercises, we shall implement a stack using a special 3-tape TM.

1. The first tape will be used only to hold and read the input. The input alphabet consists of the symbol $\uparrow$, which we shall interpret as "pop the stack," and the symbols a and b, which are interpreted as "push an a (respectively b) onto the stack."

2. The second tape is used to store the stack.

3. The third tape is the output tape. Every time a symbol is popped from the stack, it must be written on the output tape, following all previously written symbols.

The Turing machine is required to start with an empty stack and implement the sequence of push and pop operations, as specified on the input, reading from left to right. If the input causes the TM to try to pop and empty stack, then it must halt in a special error state q_e. If the entire input leaves the stack empty at the end, then the input is accepted by going to the final state q_f. Describe the transition function of the TM informally but clearly. Also, give a summary of the purpose of each state you use.

Exercise 8.4.8: In Fig. 8.17 we saw an example of the general simulation of a k-tape TM by a one-tape TM.

* a) Suppose this technique is used to simulate a 5-tape TM that had a tape alphabet of seven symbols. How many tape symbols would the one-tape TM have?

* b) An alternative way to simulate k tapes by one is to use a $(k+1)$st track to hold the head positions of all k tapes, while the first k tracks simulate the k tapes in the obvious manner. Note that in the $(k+1)$st track, we must be careful to distinguish among the tape heads and to allow for the possibility that two or more heads are at the same cell. Does this method reduce the number of tape symbols needed for the one-tape TM?

c) Another way to simulate k tapes by 1 is to avoid storing the head positions altogether. Rather, a $(k+1)$st track is used only to mark one cell of the tape. At all times, each simulated tape is positioned on its track so the head is at the marked cell. If the k-tape TM moves the head of tape i, then the simulating one-tape TM slides the entire nonblank contents of the ith track one cell in the opposite direction, so the marked cell continues to hold the cell scanned by the ith tape head of the k-tape TM. Does this method help reduce the number of tape symbols of the one-tape TM? Does it have any drawbacks compared with the other methods discussed?

! **Exercise 8.4.9:** A *k-head* Turing machine has k heads reading cells of one tape. A move of this TM depends on the state and on the symbol scanned by each head. In one move, the TM can change state, write a new symbol on the cell scanned by each head, and can move each head left, right, or keep it stationary. Since several heads may be scanning the same cell, we assume the heads are numbered 1 through k, and the symbol written by the highest numbered head scanning a given cell is the one that actually gets written there. Prove that the languages accepted by k-head Turing machines are the same as those accepted by ordinary TM's.

!! **Exercise 8.4.10:** A *two-dimensional* Turing machine has the usual finite-state control but a tape that is a two-dimensional grid of cells, infinite in all directions. The input is placed on one row of the grid, with the head at the left end of the input and the control in the start state, as usual. Acceptance is by entering a final state, also as usual. Prove that the languages accepted by two-dimensional Turing machines are the same as those accepted by ordinary TM's.

8.5 Restricted Turing Machines

We have seen seeming generalizations of the Turing machine that do not add any language-recognizing power. Now, we shall consider some examples of apparent restrictions on the TM that also give exactly the same language-recognizing power. Our first restriction is minor but useful in a number of constructions to be seen later: we replace the TM tape that is infinite in both directions by a tape that is infinite only to the right. We also forbid this restricted TM to print a blank as the replacement tape symbol. The value of these restrictions is that we can assume ID's consist of only nonblank symbols, and that they always begin at the left end of the input.

We then explore certain kinds of multitape Turing machines that are generalized pushdown automata. First, we restrict the tapes of the TM to behave like stacks. Then, we further restrict the tapes to be "counters," that is, they can only represent one integer, and the TM can only distinguish a count of 0 from any nonzero count. The impact of this discussion is that there are several very simple kinds of automata that have the full power of any computer. Moreover, undecidable problems about Turing machines, which we see in Chapter 9, apply as well to these simple machines.

8.5.1 Turing Machines With Semi-infinite Tapes

While we have allowed the tape head of a Turing machine to move either left or right from its initial position, it is only necessary that the TM's head be allowed to move within the positions at and to the right of the initial head position. In fact, we can assume the tape is *semi-infinite*, that is, there are no cells to the left of the initial head position. In the next theorem, we shall give a construction that shows a TM with a semi-infinite tape can simulate one whose tape is, like our original TM model, infinite in both directions.

The trick behind the construction is to use two tracks on the semi-infinite tape. The upper track represents the cells of the original TM that are at or to the right of the initial head position. The lower track represents the positions left of the initial position, but in reverse order. The exact arrangement is suggested in Fig. 8.19. The upper track represents cells $X_0, X_1, \ldots$, where X_0 is the initial position of the head; X_1, X_2, and so on, are the cells to its right. Cells X_{-1}, X_{-2}, and so on, represent cells to the left of the initial position. Notice the * on the leftmost cell's bottom track. This symbol serves as an

endmarker and prevents the head of the semi-infinite TM from accidentally falling off the left end of the tape.

X_0	X_1	X_2	$\cdots$
*	X_{-1}	X_{-2}	$\cdots$

Figure 8.19: A semi-infinite tape can simulate a two-way infinite tape

We shall make one more restriction to our Turing machine: it never writes a blank. This simple restriction, coupled with the restriction that the tape is only semi-infinite means that the tape is at all times a prefix of nonblank symbols followed by an infinity of blanks. Further, the sequence of nonblanks always begins at the initial tape position. We shall see in Theorem 9.19, and again in Theorem 10.9, how useful it is to assume ID's have this form.

Theorem 8.12: Every language accepted by a TM M_2 is also accepted by a TM M_1 with the following restrictions:

1. M_1's head never moves left of its initial position.
2. M_1 never writes a blank.

PROOF: Condition (2) is quite easy. Create a new tape symbol B' that functions as a blank, but is not the blank B. That is:

a) If M_2 has a rule $\delta_2(q, X) = (p, B, D)$, change this rule to $\delta_2(q, X) = (p, B', D)$.

b) Then, let $\delta_2(q, B')$ be the same as $\delta_2(q, B)$, for every state q.

Condition (1) requires more effort. Let

$$M_2 = (Q_2, \Sigma, \Gamma_2, \delta_2, q_2, B, F_2)$$

be the TM M_2 as modified above, so it never writes the blank B. Construct

$$M_1 = (Q_1, \Sigma \times \{B\}, \Gamma_1, \delta_1, q_0, [B, B], F_1)$$

where:

Q_1: The states of M_1 are $\{q_0, q_1\} \cup (Q_2 \times \{U, L\})$. That is, the states of M_1 are the initial state q_0 another state q_1, and all the states of M_2 with a second data component that is either U or L (upper or lower). The second component tells us whether the upper or lower track, as in Fig. 8.19 is being scanned by M_2. Put another way, U means the head of M_2 is at or to the right of its initial position, and L means it is to the left of that position.

Γ_1: The tape symbols of M_1 are all pairs of symbols from Γ_2, that is, $\Gamma_2 \times \Gamma_2$. The input symbols of M_1 are those pairs with an input symbol of M_2 in the first component and a blank in the second component, that is, pairs of the form $[a, B]$, where a is in Σ. The blank of M_1 has blanks in both components. Additionally, for every symbol X in Γ_2, there is a pair $[X, *]$ in Γ_1. Here, * is a new symbol, not in Γ_2, and serves to mark the left end of M_1's tape.

δ_1: The transitions of M_1 are as follows:

1. $\delta_1(q_0, [a, B]) = (q_1, [a, *], R)$, for any a in Σ. The first move of M_1 puts the * marker in the lower track of the leftmost cell. The state becomes q_1, and the head moves right, because it cannot move left or remain stationary.
2. $\delta_1(q_1, [X, B]) = ([q_2, U], [X, B], L)$, for any X in Γ_2. In state q_1, M_1 establishes the initial conditions of M_2, by returning the head to its initial position and changing the state to $[q_2, U]$, i.e., the initial state of M_2, with attention focused on the upper track of M_1.
3. If $\delta_2(q, X) = (p, Y, D)$, then for every Z in Γ_2:
   (a) $\delta_1([q, U], [X, Z]) = ([p, U], [Y, Z], D)$ and
   (b) $\delta_1([q, L], [Z, X]) = ([p, L], [Z, Y], \overline{D})$,
   where $\overline{D}$ is the direction opposite D, that is, L if $D = R$ and R if $D = L$. If M_1 is not at its leftmost cell, then it simulates M_2 on the appropriate track — the upper track if the second component of state is U and the lower track if the second component is L. Note, however, that when working on the lower track, M_2 moves in the direction opposite that of M_2. That choice makes sense, because the left half of M_2's tape has been folded, in reverse, along the lower track of M_1's tape.
4. If $\delta_2(q, X) = (p, Y, R)$, then
   $$\delta_1([q, L], [X, *]) = \delta_1([q, U], [X, *]) = ([p, U], [Y, *], R)$$
   This rule covers one case of how the left endmarker * is handled. If M_2 moves right from its initial position, then regardless of whether it had previously been to the left or the right of that position (as reflected in the fact that the second component of M_1's state could be L or U), M_1 must move right and focus on the upper track. That is, M_1 will next be at the position represented by X_1 in Fig. 8.19.
5. If $\delta_2(q, X) = (p, Y, L)$, then
   $$\delta_1([q, L], [X, *]) = \delta_1([q, U], [X, *]) = ([p, L], [Y, *], R)$$
   This rule is similar to the previous, but covers the case where M_2 moves left from its initial position. M_1 must move right from its endmarker, but now focuses on the lower track, i.e., the cell indicated by X_{-1} in Fig. 8.19.

F_1: The accepting states F_1 are those states in $F_2 \times \{U, L\}$, that is all states of M_1 whose first component is an accepting state of M_2. The attention of M_1 may be focused on either the upper or lower track at the time it accepts.

The proof of the theorem is now essentially complete. We may observe by induction on the number of moves made by M_2 that M_1 will mimic the ID of M_2 on its own tape, if you take the lower track, reverse it, and follow it by the upper track. Also, we note that M_1 enters one of its accepting states exactly when M_2 does. Thus, $L(M_1) = L(M_2)$. □

8.5.2 Multistack Machines

We now consider several computing models that are based on generalizations of the pushdown automaton. First, we consider what happens when we give the PDA several stacks. We already know, from Example 8.7, that a Turing machine can accept languages that are not accepted by any PDA with one stack. It turns out that if we give the PDA two stacks, then it can accept any language that a TM can accept.

We shall then consider a class of machines called "counter machines." These machines have only the ability to store a finite number of integers ("counters"), and to make different moves depending on which if any of the counters are currently 0. The counter machine can only add or subtract one from the counter, and cannot tell two different nonzero counts from each other. In effect, a counter is like a stack on which we can place only two symbols: a bottom-of-stack marker that appears only at the bottom, and one other symbol that may be pushed and popped from the stack.

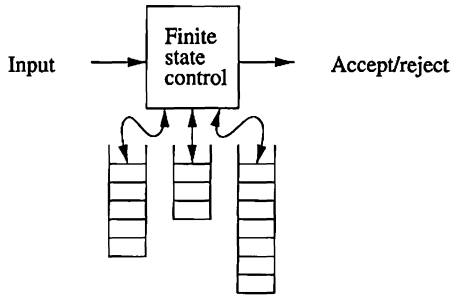


Figure 8.20: A machine with three stacks

We shall not give a formal treatment of the multistack machine, but the idea is suggested by Fig. 8.20. A k-stack machine is a deterministic PDA with k stacks. It obtains its input, like the PDA does, from an input source, rather than having the input placed on a tape or stack, as the TM does. The multistack machine has a finite control, which is in one of a finite set of states. It has a finite stack alphabet, which it uses for all its stacks. A move of the multistack machine is based on:

1. The state of the finite control.
2. The input symbol read, which is chosen from the finite input alphabet. Alternatively, the multistack machine can make a move using ϵ input, but to make the machine deterministic, there cannot be a choice of an ϵ-move or a non-ϵ-move in any situation.
3. The top stack symbol on each of its stacks.

In one move, the multistack machine can:

a) Change to a new state.

b) Replace the top symbol of each stack with a string of zero or more stack symbols. There can be (and usually is) a different replacement string for each stack.

Thus, a typical transition rule for a k-stack machine looks like:

$$\delta(q, a, X_1, X_2, \ldots, X_k) = (p, \gamma_1, \gamma_2, \ldots, \gamma_k)$$

The interpretation of this rule is that in state q, with X_i on top of the ith stack, for $i = 1, 2, \ldots, k$, the machine may consume a (either an input symbol or ϵ) from its input, go to state p, and replace X_i on top of the ith stack by string γ_i, for each $i = 1, 2, \ldots, k$. The multistack machine accepts by entering a final state.

We add one capability that simplifies input processing by this deterministic machine: we assume there is a special symbol \$, called the *endmarker*, that appears only at the end of the input and is not part of that input. The presence of the endmarker allows us to know when we have consumed all the available input. We shall see in the next theorem how the endmarker makes it easy for the multistack machine to simulate a Turing machine. Notice that the conventional TM needs no special endmarker, because the first blank serves to mark the end of the input.

Theorem 8.13: If a language L is accepted by a Turing machine, then L is accepted by a two-stack machine.

PROOF: The essential idea is that two stacks can simulate one Turing-machine tape, with one stack holding what is to the left of the head and the other stack holding what is to the right of the head, except for the infinite strings of blanks beyond the leftmost and rightmost nonblanks. In more detail, let L be $L(M)$ for some (one-tape) TM M. Our two-stack machine S will do the following:

1. S begins with a *bottom-of-stack marker* on each stack. This marker can be the start symbol for the stacks, and must not appear elsewhere on the stacks. In what follows, we shall say that a "stack is empty" when it contains only the bottom-of-stack marker.
2. Suppose that $w\$$ is on the input of S. S copies w onto its first stack, ceasing to copy when it reads the endmarker on the input.
3. S pops each symbol in turn from its first stack and pushes it onto its second stack. Now, the first stack is empty, and the second stack holds w, with the left end of w at the top.
4. S enters the (simulated) start state of M. It has an empty first stack, representing the fact that M has nothing but blanks to the left of the cell scanned by its tape head. S has a second stack holding w, representing the fact that w appears at and to the right of the cell scanned by M's head.
5. S simulates a move of M as follows.
   (a) S knows the state of M, say q, because S simulates the state of M in its own finite control.
   (b) S knows the symbol X scanned by M's tape head; it is the top of S's second stack. As an exception, if the second stack has only the bottom-of-stack marker, then M has just moved to a blank; S interprets the symbol scanned by M as the blank.
   (c) Thus, S knows the next move of M.
   (d) The next state of M is recorded in a component of S's finite control, in place of the previous state.
   (e) If M replaces X by Y and moves right, then S pushes Y onto its first stack, representing the fact that Y is now to the left of M's head. X is popped off the second stack of S. However, there are two exceptions:
      i. If the second stack has only a bottom-of-stack marker (and therefore, X is the blank), then the second stack is not changed; M has moved to yet another blank further to the right.
      ii. If Y is blank, and the first stack is empty, then that stack remains empty. The reason is that there are still only blanks to the left of M's head.

(f) If M replaces X by Y and moves left, S pops the top of the first stack, say Z, then replaces X by ZY on the second stack. This change reflects the fact that what used to be one position left of the head is now at the head. As an exception, if Z is the bottom-of-stack marker, then M must push BY onto the second stack and not pop the first stack.

6. S accepts if the new state of M is accepting. Otherwise, S simulates another move of M in the same way.

□

8.5.3 Counter Machines

A *counter machine* may be thought of in one of two ways:

1. The counter machine has the same structure as the multistack machine (Fig. 8.20), but in place of each stack is a counter. Counters hold any nonnegative integer, but we can only distinguish between zero and nonzero counters. That is, the move of the counter machine depends on its state, input symbol, and which, if any, of the counters are zero. In one move, the counter machine can:

   (a) Change state.

   (b) Add or subtract 1 from any of its counters, independently. However, a counter is not allowed to become negative, so it cannot subtract 1 from a counter that is currently 0.

2. A counter machine may also be regarded as a restricted multistack machine. The restrictions are as follows:

   (a) There are only two stack symbols, which we shall refer to as Z_0 (the *bottom-of-stack marker*), and X.

   (b) Z_0 is initially on each stack.

   (c) We may replace Z_0 only by a string of the form X^iZ_0, for some $i \geq 0$.

   (d) We may replace X only by X^i for some $i \geq 0$. That is, Z_0 appears only on the bottom of each stack, and all other stack symbols, if any, are X.

We shall use definition (1) for counter machines, but the two definitions clearly define machines of equivalent power. The reason is that stack X^iZ_0 can be identified with the count i. In definition (2), we can tell count 0 from other counts, because for count 0 we see Z_0 on top of the stack, and otherwise we see X. However, we cannot distinguish two positive counts, since both have X on top of the stack.

8.5.4 The Power of Counter Machines

There are a few observations about the languages accepted by counter machines that are obvious but worth stating:

- Every language accepted by a counter machine is recursively enumerable. The reason is that a counter machine is a special case of a stack machine, and a stack machine is a special case of a multitape Turing machine, which accepts only recursively enumerable languages by Theorem 8.9.

- Every language accepted by a one-counter machine is a CFL. Note that a counter, in point-of-view (2), is a stack, so a one-counter machine is a special case of a one-stack machine, i.e., a PDA. In fact, the languages of one-counter machines are accepted by deterministic PDA's, although the proof is surprisingly complex. The difficulty in the proof stems from the fact that the multistack and counter machines have an endmarker \$ at the end of their input. A nondeterministic PDA can guess that it has seen the last input symbol and is about to see the \$; thus it is clear that a nondeterministic PDA without the endmarker can simulate a DPDA with the endmarker. However, the hard proof, which we shall not attack, is to show that a DPDA without the endmarker can simulate a DPDA *with* the endmarker.

The surprising result about counter machines is that two counters are enough to simulate a Turing machine and therefore to accept every recursively enumerable language. It is this result we address now, first showing that three counters are enough, and then simulating three counters by two counters.

Theorem 8.14: Every recursively enumerable language is accepted by a three-counter machine.

PROOF: Begin with Theorem 8.13, which says that every recursively enumerable language is accepted by a two-stack machine. We then need to show how to simulate a stack with counters. Suppose there are $r - 1$ tape symbols used by the stack machine. We may identify the symbols with the digits 1 through $r - 1$, and think of a stack $X_1X_2 \cdots X_n$ as an integer in base r. That is, this stack (whose top is at the left end, as usual) is represented by the integer $X_nr^{n-1} + X_{n-1}r^{n-2} + \cdots + X_2r + X_1$.

We use two counters to hold the integers that represent each of the two stacks. The third counter is used to adjust the other two counters. In particular, we need the third counter when we either divide or multiply a count by r.

The operations on a stack can be broken into three kinds: pop the top symbol, change the top symbol, and push a symbol onto the stack. A move of the two-stack machine may involve several of these operations; in particular, replacing the top stack symbol X by a string of symbols must be broken down into replacing X and then pushing additional symbols onto the stack. We perform these operations on a stack that is represented by a count i, as follows.

Note that it is possible to use the finite control of the multistack machine to do each of the operations that requires counting up to r or less.

1. To pop the stack, we must replace i by i/r, throwing away any remainder, which is X_1. Starting with the third counter at 0, we repeatedly reduce the count i by r, and increase the third counter by 1. When the counter that originally held i reaches 0, we stop. Then, we repeatedly increase the original counter by 1 and decrease the third counter by 1, until the third counter becomes 0 again. At this time, the counter that used to hold i holds i/r.

2. To change X to Y on the top of a stack that is represented by count i, we increment or decrement i by a small amount, surely no more than r. If $Y > X$, as digits, increment i by $Y - X$; if $Y < X$ then decrement i by $X - Y$.

3. To push X onto a stack that initially holds i, we need to replace i by $ir + X$. We first multiply by r. To do so, repeatedly decrement the count i by 1 and increase the third counter (which starts from 0, as always), by r. When the original counter becomes 0, we have ir on the third counter. Copy the third counter to the original counter and make the third counter 0 again, as we did in item (1). Finally, we increment the original counter by X.

To complete the construction, we must initialize the counters to simulate the stacks in their initial condition: holding only the start symbol of the two-stack machine. This step is accomplished by incrementing the two counters involved to some small integer, whichever integer from 1 to $r-1$ corresponds to the start symbol. □

Theorem 8.15: Every recursively enumerable language is accepted by a two-counter machine.

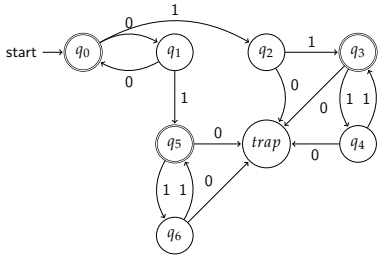
PROOF: With the previous theorem, we only have to show how to simulate three counters with two counters. The idea is to represent the three counters, say i, j, and k, by a single integer. The integer we choose is $m = 2^i 3^j 5^k$. One counter will hold this number, while the other is used to help multiply or divide m by one of the first three primes: 2, 3, and 5. To simulate the three-counter machine, we need to perform the following operations:

1. Increment i, j, and/or k. To increment i by 1, we multiply m by 2. We already saw in the proof of Theorem 8.14 how to multiply a count by any constant r, using a second counter. Likewise, we increment j by multiplying m by 3, and we increment k by multiplying m by 5.

2. Tell which, if any, of i, j, and k are 0. To tell if $i = 0$, we must determine whether m is divisible by 2. Copy m into the second counter, using the state of the counter machine to remember whether we have decremented

CS208 Tutorial 5: More on automata theory

1. Consider the DFA shown below that accepts the language $\{0^n 1^m \mid n+m \text{ is even}\}$. Assume that the trap state loops back to itself on all letters of Σ.



(a) Using the method discussed in class, find all distinguishable and indistinguishable pairs of states in the above DFA. You can record this by constructing an upper-triangular (or lower-triangular) matrix with 8 rows and 8 columns (corresponding to 8 states of the DFA), as discussed in class.

(b) Find all equivalence classes of the indistinguishability relation obtained above.

(c) Using one state from each equivalence class to represent all states of the class, construct a minimal DFA for the language represented by the above DFA.

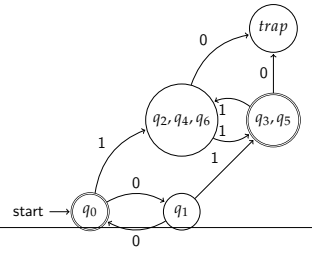
Solution:

(a) Table as shown (State 7 is trap). Each entry is a distinguishing string, if it exists

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	X	ϵ	ϵ	01	ϵ	01	ϵ	ϵ
1		X	0	ϵ	0	ϵ	0	0
2			X	ϵ		ϵ		1
3				X	ϵ		ϵ	ϵ
4					X	ϵ		1
5						X	ϵ	ϵ
6							X	1
7								X

(b) The equivalence classes are $\{q_0, q_1, (q_2, q_4, q_6), (q_3, q_5), trap\}$

(c) The minimized DFA is as shown below:



2. Consider a language $L \subseteq \Sigma^*$ for some finite alphabet Σ. As discussed in class, the *Nerode equivalence* $\sim_L$ is an equivalence relation over Σ^* such that for any $x, y \in \Sigma^*$, $x \sim_L y$ if and only if for every $z \in \Sigma^*$, $xz \in L \iff yz \in L$. The relation $\sim_L$ partitions Σ^* into equivalence classes of words. Hence, each equivalence class of $\sim_L$, viewed as a set of words, is a language by itself. Recall further from our discussion in class:

   - The Nerode equivalence is well-defined for every (regular or non-regular) language L over Σ.
   - The *Myhill-Nerode Theorem* states that L is regular if and only if the number of equivalence classes of $\sim_L$ is finite.
   - If the number of equivalence classes of $\sim_L$ equals $k \in \mathbb{N}$, then the unique (upto isomorphism) minimal DFA recognizing L has k states.

   In this problem we will explore the Nerode equivalence and some of its variants.

   (a) For each of the following languages L, describe (in any suitable form) the equivalence classes of $\sim_L$ as languages over $\{0, 1\}$.
      (a) L is the language corresponding to $(\mathbf{00} + \mathbf{11})^*$
      (b) L is the language $\{0^i 1^j \mid i \leq j\}$

   (b) Define an equivalence relation $\sim_R$ such that for any $x, y \in \Sigma^*$, $x \sim_R y$ if and only if for every $z \in \Sigma^*$, $zx \in L \iff zy \in L$. *Note the difference of $\sim_R$ from the Nerode equivalence $\sim_L$.*
      (i) Show that the number of equivalence classes of $\sim_R$ is finite if and only if L is regular.
      (ii) Let L_{rev} denote the language formed by reversing each string in L. Show that if the number of equivalence classes of $\sim_R$ is k, then the size of the unique minimal DFA recognizing L_{rev} is also k.

Solution:

(a) (a) For a regular language like the one in this question, one way to obtain the Nerode equivalence classes is to first construct the *minimal* DFA for the language, and then for each state q in the minimal DFA, list down the set of strings that bring you from the start state of the DFA to state q. Clearly, all such strings w must be in the same Nerode class, since for every string $x \in \Sigma^*$, whether $wx \in L$ or not simply depends on whether you can reach the accepting state of the minimal DFA from q on reading x. Similarly, every string w' that doesn't bring you from the start state to q can't be Nerode equivalent to w since w' must be brining you to a different state q', starting from the start state. However, since q and q' are two different states in the minimal DFA for L, there is a distinguishing string x' such that x' is accepted starting from q and not accepted starting from q' or vice versa. It follows that one of $w.x'$ and $w'.x'$ is in L and the other isn't. Hence, w and w' can't be in the same Nerode class.

We leave it as an exercise for you to construct the minimal DFA for the given language L. Once you do that, it is easy to follow the steps outlined above to find the following Nerode equivalence classes:

$S_1 = L$ (set of strings that bring you to the accepting state of the DFA), $S_2 = (00 + 11)^*0, S_3 = (00 + 11)^*1, S_4 = \Sigma^* - \{S_1 \cup S_2 \cup S_3\}\}$

(b) Since this language is not regular (why? Try using the Pumping Lemma for regular languages), we can't use the above method for finding the Nerode equivalence classes. Hence, we have to look into the specifics of the language and try to construct infinitely many Nerode equivalence classes (Myhill-Nerode theorem guarantees that there are infinitely many Nerode equivalence classes for a non-regular language).

It is easy to see that all strings w not belonging to 0^*1^* are equivalent, since no matter what string x you concatenate to w, the string $w.x$ is not in 0^*1^*, and hence not in L. So all strings not in 0^*1^* form one Nerode equivalence class, say C_1

Let us now focus on strings in 0^*1^*. Consider one such string $w = 0^i1^j$, where $j \geq i$ and $w \neq \varepsilon$ (i.e. it is not the case that $j = i = 0$). Notice that for every string $x \in 1^*$, $w.x \in L$. Similarly, for every string $x \notin 1^*$, $w.x \notin L$. Hence, all strings $w = 0^i1^j$, where $j \geq i$ and $w \neq \varepsilon$ are in the same Nerode equivalence class, say C_2 (they cannot be distinguished by any string $x \in \Sigma^*$). Moreover, C_1 is different from C_2, since ε distinguishes any string in C_1 from any string in C_2. What if $w = \varepsilon$?. The string $x = 01$ distinguishes w from every string in $C_1 \cup C_2$. Hence ε is in a class, say C_3, by itself.

What about strings of the form 0^i1^j, where $i > j$. Consider any such string $w = 0^i1^j$. The string $x = 1^{i-j}$ distinguishes w from every string in C_1. The string ε distinguishes w from every string in C_2. The string $x = 01$ distinguishes w from ε. The string $x = 1^{\min(i-j, i'-j')}$ distinguishes w from every string $w' = 0^{i'}1^{i'}$, where $i' > j'$ and $i - j \neq i' - j'$. Finally, for every string $w' = 0^{i'}1^{i'}$, where $i' > j'$ and $i - j = i' - j'$, no string x can distinguish w from w'. Therefore, all strings $0^{i+k}1^i$ belong to the same Nerode equivalence class for each $k > 0$, and the classes corresponding to k_1, k_2, where $k_1 \neq k_2$ are distinct.

Summarizing all the above cases, the Nerode equivalence classes are: $C_1 = L(0^*1^*)^c, C_2 = L \setminus \{\varepsilon\}, C_3 = \{\varepsilon\}, C_{3+k} = \{0^{i+k}1^i | i \geq 0\}$, for every $k \geq 1$.

(b) Let $\sim_N$ denote the Nerode equivalence relation for the language L_{rev} formed by reversing each string in L. Now, $x \sim_R y$ if and only if for every $z \in \Sigma^*$, $zx \in L \iff zy \in L$, ie $x^Rz^R \in L_{rev} \iff y^Rz^R \in L_{rev}$ for every $z \in \Sigma^*$, ie $x^R \sim_N y^R$.

Consider the function f mapping equivalence classes of $\sim_R$ to equivalence classes of $\sim_N$ such that for any $x \in \Sigma^*$, $f([x]_R) = [x^R]_N$. Since for any $x, y \in \Sigma^*$, $x \sim_R y$ if and only if $x^R \sim_N y^R$, f is well defined and also an injection, and since $(x^R)^R = x$ for any $x \in \Sigma^*$, $[x]_N = f([x^R]_R)$, ie f is a surjection as well. Hence there exists a bijection between the equivalence classes of $\sim_R$ and the equivalence classes of $\sim_N$, ie they have the same cardinality.

Now, by the *Myhill-Nerode Theorem*, the number of equivalence classes of $\sim_N$ is finite if and only if L_{rev} is regular. Now, for any language L, L_{rev} is regular if and only if L is regular (this can be seen by reversing the transitions in the DFA recognizing L to get an NFA recognizing L_{rev}, and also noting that $(L_{rev})_{rev} = L$).

Considering this, along with the fact the set of equivalence classes of $\sim_R$ has the same cardinality as the set of equivalence classes of $\sim_N$, we get that $\sim_R$ has a finite number of equivalence classes if and only if L is regular. Moroever, if $\sim_R$ has k equivalence classes, then so does $\sim_N$, which, by the *Myhill-Nerode Theorem* means that the unique minimal DFA recognizing L_{rev} has k states.

3. A hacker must figure out what a language L is in order to break into a top-secret system. The hacker knows that the language L is regular and that it is over the alphabet $\{0,1\}$. However, no other information about L is directly available. Instead, an oracle is available that only answers "Yes" or "No" in response to specific types of queries, labeled Q1 and Q2 below.

   Q1 **Does there exist any DFA with n states that recognizes L?**
   For every $n > 0$, the oracle truthfully responds "Yes" or "No" to this query.

   Q2 **Does word w belong to L?**
   For every $w \in \{0,1\}^*$, the oracle truthfully responds "Yes" or "No" to this query.

   We are required to help the hacker re-construct a minimal DFA for L. Towards this end, we will proceed systematically as follows.

   (a) Show that if the minimal state DFA for L has N states, then N can be determined using a sequence of $\mathcal{O}(\log_2 N)$ Q1 queries.
   Hint: Use galloping (or exponential) search.

   (b) Show that it is possible to find a word $w \in L$ or determine that $L = \emptyset$ using at most 2^N Q2 queries.
   Hint: Consider any word in L and repeatedly apply the Pumping Lemma to remove loops in the path from the initial state to an accepting state.

   (c) Once we know the minimal count of states, say N, for a DFA for L, we will construct the Nerode equivalence classes $\sim_L$ for L. Recall from our discussion in class that there are exactly N of these, and each equivalence class can be uniquely identified with a state of the minimal DFA recognizing L.
   For any two distinct equivalence classes of $\sim_L$, show the following:
      (i) There exist words $w_1, w_2 \in \Sigma^*$, where $|w_1| \leq N-1$ and $|w_2| \leq N-1$ such that w_1 belongs to the first equivalence class and w_2 to the second. We will use $[w_1]$ to denote the first equivalence class and $[w_2]$ to denote the second, in the discussion below.
      (ii) For $[w_1] \neq [w_2]$, there is a word $x \in \Sigma^*$ of length $\leq N \times (N-1) - 1$ such that $w_1 \cdot x \in L$ and $w_2 \cdot x \notin L$ or vice versa.
      (iii) For $[w_1] \neq [w_2]$, there exists an edge labeled 0 (resp. 1) from the state corresponding to $[w_1]$ to the state corresponding to $[w_2]$ iff for all $x \in \Sigma^*$, where $|x| \leq N \times (N-1) - 1$, $w_1 \cdot 0 \cdot x$ (resp. $w_1 \cdot 1 \cdot x$) and $w_2 \cdot x$ are either both in L or both not in L.
   Using all the above results, design an algorithm that helps the hacker reconstruct the minimal DFA for L. Give an upper bound on the count of Q2 queries needed for this re-construction, in terms of the count N of the states of the minimal DFA for L.

Solution:

(a) We make Q1 queries using powers of 2 $(1, 2, 4, \ldots)$ until it returns "yes". Say, it returns yes for 2^{k+1}. Then, we know the smallest DFA representing the language has size between $2^k + 1$ and 2^{k+1}. We perform binary search over this space. Total number of queries are $k + 2 + \mathcal{O}(\log(2^{k+1} - 2^k)) = 2k + 2 \in \mathcal{O}(\log_2 N)$

(b) **Claim**: A DFA whose language is non-empty having N states accepts a word of length at most $N-1$. Proof is left as an exercise to the reader (Use ideas similar to Pumping Lemma). Hence, we can make Q2 queries over all possible words having length less than or equal to $N-1$. Either we conclude that the language is empty or find a word belonging to the language in at most $2^0 + 2^1 + \ldots 2^{N-1} = 2^N - 1$ Q2 queries

(c) (i) If q_1 and q_2 are the states corresponding to these equivalence classes in the minimal DFA (and q_0 is the initial state), then a word w is in the equivalence class of q_1 iff $\hat{\delta}(q_0, w) = q_1$ (and similarly for q_2). Since equivalence classes are by definition non-empty, such a word w_1 necessarily exists. We can remove cycles in the path this word takes from q_0 to q_1 word to ensure that it's length is at most $N-1$ (Similarly for w_2).

(ii) Let q_1 be the state corresponding to $[w_1]$ and q_2 the state corresponding to $[w_2]$. Since $[w_1]$ and $[w_2]$ are distinct equivalence classes, there must exist a string x such that exactly one of w_1x and w_2x are in L. We will show that there exists such an x with length at most $N-2$.

Consider an equivalence relation $\sim_k$ over the state set Q of the minimal DFA where $q_1 \sim_k q_2$ iff for every string x of length at most k, $\hat{\delta}(q_1, x) \in F \iff \hat{\delta}(q_2, x) \in F$. Some observations:

i. $q_1 \sim_0 q_2$ iff $q_1 \in F \iff q_2 \in F$

ii. $q_1 \sim_{k+1} q_2 \implies$ q_1 $\sim_k q_2$, ie the equivalence classes of $\sim_{k+1}$ are subsets of those of $\sim_k$

By the second observation, $\sim_{k+1}$ has at least as many equivalence classes as $\sim_k$, and if the number of equivalence classes is the same, then $\sim_{k+1}$ and $\sim_k$ are identical. Note that $\sim_0$ has 2 equivalence classes. This means that in the number of equivalence classes of the sequence $\sim_0, \sim_1, \sim_2, \ldots$ keeps increasing from 2, until some k where $\sim_k = \sim_{k+1}$, after which it remains constant. Since the number of equivalence classes of $\sim_0$ is 2, and the number of equivalence classes of $\sim_k$ is at most N, k can be at most $N-2$.

Now, for distinct q_1, q_2, since the DFA is minimal, there exists some string x such that exactly one of $\hat{\delta}(q_1, x)$ and $\hat{\delta}(q_2, x)$ lies in F. Say $|x| = p$. Then $q_1 \nsim_p q_2$. If $p > N-2$, then $\sim_p = \sim k$, ie $q_1 \nsim_k q_2$, ie there is some x' of length at most k such that exactly one of $\hat{\delta}(q_1, x')$ and $\hat{\delta}(q_2, x')$ is in F. Since k is at most $N-2$, this means for any distinct q_1 and q_2 there will exist a string x of length at most $N-2$ such that exactly one of $\hat{\delta}(q_1, x)$ and $\hat{\delta}(q_2, x)$ lies in F. This means that for distinct $[w_1]$ and $[w_2]$ there will exist an x of length at most $N-2$ such that exactly one of w_1x and w_2x is in L.

Algorithm:
Find the value of N (Part a). Consider all words having length less than or equal to $N-1$. Find equivalence classes over these words by taking pairs at a time and iterating over all words having length less than or equal to $N-2$. If you find a distinguisher, they're in different equivalence classes, else they are in the same equivalence class. Each equivalence class now represents a state. Accepting state is simply found by using Q2 on one word in each equivalence class. To find the transition function, we can simply consider the shortest words in each equivalence class and use their prefixes to construct the path from the starting state. The starting state is the class which contains epsilon.

4. **Takeaway**: You can view this question as a continuation of Question 1 on Nerode equivalences and their variants. Define an equivalence relation $\sim_S$ such that for any $x, y \in \Sigma^*$, $x \sim_R y$ if and only if for every $u, v \in \Sigma^*$, $uxv \in L \iff uyv \in L$.

   (i) Show that the number of equivalence classes of $\sim_S$ is finite if and only if L is regular.

   (ii) Assuming that L is regular, if the minimal DFA recognizing L has k states, show that the number of equivalence classes of $\sim_S$ is at most k^k

Solution: Say L is regular and is recognized by minimal DFA $(Q, \Sigma, \delta, q_0, F)$.

If $x \sim_S y$, then for every state $q \in Q$ we must have $\hat{\delta}(q_0, x) = \hat{\delta}(q_0, y)$. To see this, note that since $(Q, \Sigma, \delta, q_0, F)$ is the minimal DFA recognizing L, for every $q \in Q$ there exists $u \in \Sigma^*$ such that $\hat{\delta}(q_0, u) = q$. Furthermore, if $q_1 \neq q_2$ are distinct states in Q, then there exists $v \in \Sigma^*$ such that exactly one of the following are true:

- $\hat{\delta}(q_1, v) \in F$
- $\hat{\delta}(q_2, v) \in F$

(otherwise the states q_1 and q_2 could be merged). Now, if $x \sim_S y$, but if there is some q such that $\hat{\delta}(q, x) \neq \hat{\delta}(q, y)$ (call these q_1 and q_2), then there exists $u \in \Sigma^*$ such that $\hat{\delta}(q_0, u) = q$ and there exists $v \in \Sigma^*$ such that (WLOG) $\hat{\delta}(q_1, v) \in F$ and $\hat{\delta}(q_2, v) \notin F$. This means that there exist $u, v \in \Sigma^*$ such that $\hat{\delta}(q_0, uxv) \in F$ but $\hat{\delta}(q_0, uyv) \notin F$, which means $uxv \in L$ but $uyv \notin L$, contradicting the definition of $\sim_S$.

Moreover, if $\hat{\delta}(q, x) = \hat{\delta}(q, y)$ for every $q \in Q$, then for every $u, v \in \Sigma^*$, $\hat{\delta}(q_0, uxv) = \hat{\delta}(q_0, uyv)$, ie $uxv \in L \iff uyv \in L$, ie $x \sim_S y$. Therefore, for any $x, y \in \Sigma^*$, $x \sim_S y$ if and only if for every $q \in Q$, $\hat{\delta}(q, x) = \hat{\delta}(q, y)$.

Consider the set of functions from Q to itself, denoted by Q^Q. Consider the function f mapping equivalence classes of $\sim_S$ to elements of Q^Q such that for every $q \in Q$, $f([x]_S)(q) = \hat{\delta}(q, x)$. By the previous result, f is well defined and an injection. Therefore, there exists an injection from the equivalence classes of $\sim_L$ to Q^Q, a finite set.

Therefore, if L is regular, then the number of equivalence classes of $\sim_L$ must be finite, and is at most $|Q^Q|$, where Q is the set of states of the minimal DFA recognizing L. If $|Q| = k$, then we get that the number of equivalence classes is at most k^k.

It is easier to show the other direction of the implication, ie if the number of equivalence classes of L is finite, then L must be regular. This can be done by constructing a DFA recognizing L. Consider the DFA whose set of states $Q = \{[x]_S : x \in \Sigma^*\}$ (ie the set of states is the set of equivalence classes of $\sim_S$), and transitions are of the form $[x]_S \xrightarrow{a} [xa]_S$, for any $a \in \Sigma$ (ie $\delta([x], a) = [xa]$). This transition function is well defined, since if $x \sim_S y$, then $xa \sim_S ya$ for any $a \in \Sigma$. The initial state is taken to be $q_0 = [\varepsilon]_S$ and the set of final states is $F = \{[x]_S : x \in L\}$. It can be shown that the language recognized by this automaton is precisely L (note that $\hat{\delta}(q_0, x) = [x]$ and if $x \sim_S y$ then $x \in L \iff y \in L$).

5. **Takeaway**: Let $\Sigma = \{a\}$.

   (i) Show that for every language L (regular or not) over Σ, the language $L^* = \bigcup_{i=0}^{\infty} L^i$ is regular.

   (ii) Show that for every regular language L over Σ, there exist two finite sets of words S_1 and S_2 and an integer $n > 0$ such that $L = S_1 \cup S_2 \cdot (a^n)^*$

Solution: (a) To those interested, please read here

(b) Refer to the solution of Tut 4, Question 5

6. **Takeaway**: The *star-height* of a regular expression $\mathbf{r}$, denoted $\mathsf{SH}(\mathbf{r})$, is a function from regular expressions to natural numbers. It is defined inductively as follows:

   - $\mathsf{SH}(\mathbf{0}) = \mathsf{SH}(\mathbf{1}) = \mathsf{SH}(\varepsilon) = \mathsf{SH}(\mathbf{\Phi}) = 0$.
   - $\mathsf{SH}(\mathbf{r_1} + \mathbf{r_2}) = \mathsf{SH}(\mathbf{r_1} \cdot \mathbf{r_2}) = \max(\mathsf{SH}(\mathbf{r_1}), \mathsf{SH}(\mathbf{r_2}))$
   - $\mathsf{SH}(\mathbf{r}^*) = \mathsf{SH}(\mathbf{r}) + 1$

   Give a regular expression $\mathbf{r}$ over $\Sigma = \{0, 1\}$ such that the following hold:

   - $\mathsf{SH}(\mathbf{r}) > 0$, and
   - Every regular expression with star-height $< \mathsf{SH}(\mathbf{r})$ represents a language different from that represented by $\mathbf{r}$.

   You must give a brief justification why no regular expression with lesser star-height can represent the same language.

   Solution: The answer to this specific question is really simple if you think about the definition of star height. However, the study of star heights of regular expressions and about the hierarchy of languages corresponding to increasing star heights is very interesting. For those interested in knowing more about star heights, a good starting point is here

   For this specific question, you can simply take the regular expression $\mathbf{0}^*$, which has star height 1. What are the regular expressions with star height < 1. These are $\mathbf{1}, \mathbf{0}, \varepsilon, \mathbf{\Phi}$ and combinations of these regular expressions using $+$ and $\cdot$. All of these represent languages with finitely many words, while $\mathbf{0}^*$ represents a language with infinitely many words.

   However, the study of the star height hierarchy is not just limited to star heights of 0 and 1. It extends to all star heights (see here for more details).

Tutorial 6: Context-Free Tutorial

1. Solve the following problems. Assume the alphabet to be $\Sigma = \{a, b, c\}$

   (a) Consider the language of non-palindromes (words that are not palindromes). A palindrome is a word that spells the same way forward and backwards. For example, 'abcba' is a palindrome but 'abbaa' is not). Construct a PDA and a CFG for the language.

   (b) Consider the language $\{w \mid w \neq uu \text{ for any } u \in \Sigma^*\}$. Construct a PDA and a CFG for the language.

   Solution:

   1. The NPDA keeps pushing letters on the stack up to a point which is our guess of the halfway point (state Q_0). Then we keep popping letters from the stack as long as they match the scanned letter (state Q_1), until we spot a mismatch which is when we move to state Q_2, in which we pop off letters irrespective of a match. We accept the word if we are in state Q_2 at the end, with an empty stack. Formally, the NPDA is $(\{Q_0, Q_1, Q_2\}, \Sigma = \{a, b, c\}, Q = \{a, b, c, \perp\}, \delta, Q_0, \perp, \{Q_2\})$, with $\delta =$

   $$\begin{gathered}((Q_0, x, A), (Q_0, xA))\,\forall\, x \in \Sigma, A \in Q \\ ((Q_0, x, A), (Q_1, A))\,\forall\, x \in \Sigma \cup \{\epsilon\}, A \in Q \\ ((Q_1, x, x), (Q_1, \epsilon))\,\forall\, x \in \Sigma \\ ((Q_1, x, y), (Q_2, \epsilon))\,\forall\, x, y \in \Sigma \text{ st } x \neq y \\ ((Q_2, x, A), (Q_2, \epsilon))\,\forall\, x \in \Sigma, A \in Q.\end{gathered}$$

   The CFG is:

   $$\begin{aligned} S &\to aSa|bSb|cSc \\ S &\to xS'y \text{ for } x \neq y \\ S' &\to xS'y \text{ for all } x, y \\ S' &\to a|b|c|\epsilon. \end{aligned}$$

   2. Let $w_{i..j}$ denote the word formed by the i^{th} to j^{th} position (both inclusive) of w, i.e it is a continuous substring of w. Let w_i denote the i^{th} letter of w. (We count from 1).

   Note that odd length words are immediately a part of this language, since they cannot be of the form uu for any word u. These odd words can be divided into three categories: Those which have a a at the center, those which have a b at the center, and those which have a c at the center. The reason for this division will become apparent later. The following CFG captures these words:

   $$\begin{aligned} S_{odd} &\to A \mid B \mid C \\ A &\to a \mid aAb \mid bAc \mid cAa \mid bAa \mid aAc \mid cAb \\ B &\to b \mid aBb \mid bBc \mid cBa \mid bBa \mid aBc \mid cBb \\ C &\to c \mid aCb \mid bCc \mid cCa \mid bCa \mid aCc \mid cCb \end{aligned}$$

   For even length word in this language, suppose the length of the word is $2n$. Then, there exists a position $k \leq n$ such that the letter at position k and position $n + k$ are different. (otherwise the two halves of w are the same word, and so w = uu). Now

consider the substring $w_{1..2k-1}$ and $w_{2k..2n}$. Both of these are of odd length, and the center letter of the first one is w_k and that of the second word is w_{n+k}. Thus, **every** even word of this language can be written as the concatenation of two odd words with different centers, and every concatenation of two odd words with different centers belongs to this language. Therefore, here is the complete CFG for our language:

$$S \to A \mid B \mid C \mid AB \mid BC \mid CA \mid BA \mid AC \mid CB$$
$$A \to a \mid aAb \mid bAc \mid cAa \mid bAa \mid aAc \mid cAb$$
$$B \to b \mid aBb \mid bBc \mid cBa \mid bBa \mid aBc \mid cBb$$
$$C \to c \mid aCb \mid bCc \mid cCa \mid bCa \mid aCc \mid cCb$$

In the PDA, for odd words we simply accept them. For even words, we make two non-deterministic guesses: the center of the first odd part and the center of the second odd part, and check that they have different centers. Here is a description in words:

(a) First, make a nondeterministic guess (without scanning anything) whether we expect to see an odd word or an even word.

(b) In the odd case, just transition to a 2-state automata that keeps track of whether even or odd number of letters have been seen. No need of the stack here.

(c) For the even case, keep scanning the word and pushing it on the stack. At some point, make a guess for a certain letter to be the center of the first odd word. Remember this letter (using the state, by having one state for each letter). Now, keep popping off the stack until it becomes empty. This marks the end of the first odd word. Now, once again start pushing the scanned letters onto the stack, and at some point make a nondeterministic guess for the center of the second odd word (only if it is different from the first center). Finally, pop the letters off the stack until the stack becomes empty, and then accept.

2. Determine if the following languages are context-free or not. If yes, provide a CFG and PDA for the same, else prove, using the Pumping Lemma, that they are not Context Free

   (a) $L_1 = \{w \mid w = uu \text{ for any } u \in \Sigma^*\}$

   (b) $L_2 = \{0^p | \text{p is prime}\}$

   For more on L_2, look at the takeaway problems

Solution: For both sub-parts, the languages are not context-free. To show this, we play the game between the believer and the adversary, as required by the Pumping Lemma for context-free languages. Specifically, the believer thinks the language is a CFL and chooses a constant p (> 0) that is at least as large as 2^k, where k is the number of non-terminal symbols in the supposed Chomsky Normal Form grammar for L). The subsequent reasoning for the two parts is given separately below.

1. The adversary chooses $w = 0^p1^p0^p1^p \in L$. Next, the believer decomposes w as $u.v.x.y.z$ with $|v.x.y| \leq p$ and $|x| \geq 1$. Notice that this implies $v.x.y$ must be either of the form 0^i1^j, where $i > 0, j \geq 0$ or of the form 1^i0^j, where $i > 0, j \geq 0$. In both cases, if the adversary pumps v and y twice, the string $u.v^2.x.y^2.z \notin L$ (try to reason why).

2. The adversary chooses $w = 0^m$, where m is the smallest prime larger than or equal to p. Next, the believer decomposes w as $u.v.x.y.z$ where $|v.x.y| \leq p$ and $|x| \geq 1$. Let $|v.x.y| = n$. The adversary can now pump v and y exactly $1 + m.n$ times, so that the resulting word is $0^{m+m.n}$ or $0^{m.(1+n)}$. Clearly, $m.(1 + n)$ is not prime, and hence $0^{m.(1+n)} \notin L$.

3. **Deterministic Context-Free Languages**
We know that Context-Free Languages (CFL) are accepted by Push-Down Automata (NPDA), where we had allowed non-determinism in PDA transitions. In this question, we will explore Deterministic Push-Down Automata (DPDA). Recall that a (not necessarily deterministic) PDA M can be defined as a 7-tuple:

$$M = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$$

h where Q is a finite set of states. Σ is a finite set of input symbols. Γ is a finite set of stack symbols. $q_0 \in Q$ is the start state. $Z_0 \in \Gamma$ is the starting stack symbol. $A \subseteq Q$, where A is the set of accepting, or final, states δ is a transition function, where $\delta : (Q \times (\Sigma \cup \varepsilon) \times \Gamma) \to \mathcal{P}(Q \times \Gamma^*)$ Here, $\mathcal{P}(X)$ is the power set of a set X, and ε denotes the empty string. We say that M is a deterministic PDA (or DPDA) if it satisfies both the following conditions:

For any $q \in Q, a \in \Sigma \cup \varepsilon, x \in \Gamma$, the set $\delta(q, a, x)$ has at most one element.

For any $q \in Q, x \in \Gamma$, if $\delta(q, \varepsilon, x) \neq \emptyset$, then $\delta(q, a, x) = \emptyset$ for every $a \in \Sigma$.

We call the languages accepted by DPDAs as DCFLs (Deterministic Context Free Languages). We have studied in class that PDAs can accept by *empty stack* or by *final state*, and that these provide equivalent accepting power. Interestingly, this is not so for DPDAs, so we need to make up our mind about which acceptance criterion to use. For purposes of this question, we will use acceptance by *final state*. We investigate acceptance by empty stack in a takeaway question at the end of this tutorial.

1. Consider the language of balanced parentheses. A string of parentheses is balanced if the number of opening parentheses in any proper prefix is at least as much as the number of closing parenthesis in the same prefix. Also, the total number of opening and closing parenthesis in the entire string must be equal.
   Draw a DPDA (accepting by final state) that accepts the language of balanced parentheses strings.
2. Construct a deterministic PDA for the complement of the above language. Does this give you an idea why DCFLs (recognized by DPDAs by final state) are closed under complementation?

Solution: Please see the solution posted by us on Piazza. The DPDA shown there accepts the complement of the language of balanced parentheses. If you flip the accepting/non-accepting status of the states of the DPDA, you get the DPDA accepting the language of balance parentheses.

4. **Takeaway: The Curious Case of the Unary Alphabet** Prove that any language over a unary alphabet (the alphabet has exactly one element) is context-free if and only if it is regular.

Solution: In the interests of time, we'd like to point interested students to a nice exposition on this problem here.

5. **Takeaway: Expressions in Intermediate Code**
Consider the following context-free grammar for expressions in some (familiar) programming languages, where $\langle$expr$\rangle$ is the start symbol of the grammar.

⟨*expr*⟩ ::= ⟨*term*⟩ '+' ⟨*term*⟩
  | ⟨*term*⟩

⟨*term*⟩ ::= ⟨*factor*⟩ '*' ⟨*factor*⟩
  | ⟨*factor*⟩

⟨*factor*⟩ ::= '(' ⟨*expr*⟩ ')'
  | ⟨*number*⟩

⟨*number*⟩ ::= [0-1]+

Now, though expressions can be a sum of as many terms, it is essential, during an intermediate step of compilation, that every expression must be a sum of at most two terms and every term must be a product of at most two terms. Draw a Deterministic PDA (definition provided in an earlier question) to recognise strings that are of the form stated above. Note that the alphabet is $\Sigma = \{0, 1, (,), *, +\}$.

Solution: This can be obtained directly from the CFG to PDA construction studied in class. Remember that the moves of such a PDA on a given input string effectively mimic a leftmost derivation of the string using the given grammer. Since there is a unique (deterministic) parse tree for a string in the given grammar, the leftmost derivation of the string is also unique. Hence the PDA obtained by transforming the CFG has a unique sequence of moves on a given string. Indeed, as can be seen by constructing the PDA, it is a DPDA.

6. **Takeaway: Null-stack DPDAs**
   In Problem 3, we used acceptance by final state for a DPDA. Let's see what happens if we now allow a DPDA to accept by emptying its stack (regardless of which state it is in, when the stack becomes empty). We will call such a DPDA a *null-stack DPDA*, i.e. it's a DPDA just like we had earlier, but it accepts by emptying its stack.

   1. Prove that no null-stack DPDA can accept the language of balanced parentheses. Recall that a string of parentheses is balanced if the number of opening parenthesis in any prefix of the string is at least as much as the number of closing parenthesis in the same prefix, and the total number of opening and closing parenthesis ar equal.

      [Hint:] Can a null-stack DPDA accept two strings u and $u.v$, where one is a proper prefix of the other?

      Note: This is quite damaging news in the DPDA world, since we saw in Problem 3 that the language of balanced parentheses can be accepted by a DPDA accepting by final state. The above proof should now convince you that unlike normal PDAs, acceptance by final state and acceptance by empty stack are not equally powerful in the DPDA world.

   2. A string is said to be *minimally balanced parentheses* if the number of opening parenthesis in any proper prefix is strictly more than the number of closing parenthesis in the same prefix, and the total number of opening and closing parenthesis in the entire string are equal. Thus, $(()())$ is a minimally balanced parentheses string, but $()()$ and ε are not. Any string of balanced parentheses can be written as either ε or a concatenation of a finite number of minimally balanced parentheses strings. Show that for every given value of $k > 0$, we can construct a null-stack DPDA that accepts the language of balanced parentheses strings containing exactly k minimal valid parentheses substrings. Can we construct a null-stack DPDA if we want to accept the language of balanced parentheses containing upto (instead of exactly) k minimally valid parentheses substrings?

Solution: Suppose there was a null-stack DPDA for recognizing the language of all balanced parentheses. Then both $()$ and $()()$ must be accepted by the DPDA. However, since the DPDA accepts by empty stack, and since it has a unique sequence of moves on reading an input, its stack must necessarily become empty after reading $()$. Since a DPDA with empty stack can't make any further moves (recall each move of a PDA requires looking up the symbol at the top of the stack), the null-stack DPDA will get stuck after reading the prefix $()$ of the string $()()$. Hence, it cannot accept $()()$. But then, this null-stack DPDA doesn't recognize the set of all balanced parentheses strings.

For the second part of the question, first construct a null-stack DPDA accepting a minimally balanced parentheses string. This should be straightforward, and is left as an exercise. Now take k such DPDAs, say $P_1, \ldots P_k$. Consider their stack alphabets as disjoint, say $\Gamma_1, \ldots \Gamma_k$. Let $X_{0,1}, \ldots X_{0,k}$ denote the initial symbol on the stack for $P_1, \ldots P_k$ respectively. Now create a new DPDA P whose stack alphabet is $\Gamma_1 \cup \cdots \Gamma_k \cup \{Z\}$, where Z is a new bottom of stack marker symbol that is not in $\Gamma_1 \cup \cdots \Gamma_k$. The new DPDA P uses Z as the initial symbol in its stack. It also has a new start state. The DPDA P works as follows:

From the start state of P, there is only a single transition that consumes ε, pops Z and pushes $X_{0,1}Z$ into the stack. It then transitions to the start state of P_1. From every state of P_i, we now add a transition that consumes ε and on seeing Z as the top of the stack, pops Z, pushes $X_{0,i+1}Z$ into the stack and transitions to the start state of P_{i+1}. Finally, from every state in P_k, we add a transition that consumes ε, pops Z from the stack and doesn't push anything, thereby emptying the stack.

We leave it for you to complete the reasoning that this accepts all and only strings that are concatenations of k minimal valid parentheses substrings.

Clearly, we can't construct a null-stack DPDA that accepts both $()$ and $()()$ – we've reasoned earlier about this. So we can't construct a null-stack DPDA that accepts the language of balanced parentheses containing upto (instead of exactly) k minimally valid parentheses substrings.