FUTURE VISION BIE

One Stop for All Study Materials

& Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU, Currently for CSE – Computer Science Engineering...

Join Telegram to get Instant Updates: https://bit.ly/2GKiHnJ

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

CHAPTER 13

Context-Free and Noncontext-Free Languages

The language $A^nB^n = \{a^nb^n : n \geq 0\}$ is context-free. The language $A^nB^nC^n = \{a^nb^nc^n : n \geq 0\}$ is not context free (intuitively because a PDA's stack cannot count all three of the letter regions and compare them). PalEven $= \{ww^R : w \in \{a, b\}^*\}$ is context-free. The similar language WW $= \{ww : w \in \{a, b\}^*\}$ is not context-free (again, intuitively, because a stack cannot pop the characters of w off in the same order in which they were pushed).

Given a new language L, how can we know whether or not it is context-free? In this chapter, we present a collection of techniques that can be used to answer that question.

13.1 Where Do the Context-Free Languages Fit in the Big Picture?

First, we consider the relationship between the regular languages and the context-free languages.

THEOREM 13.1 The Context-Free Languages Properly Contain the Regular Languages

Theorem: The regular languages are a proper subset of the context-free languages.

Proof: We first show that every regular language is context-free. We then show that there exists at least one context-free language that is not regular.

We show that every regular language is context-free by construction. If L is regular, then it is accepted by some DFSM $M = (K, \Sigma, \delta, s, A)$. From M we construct a PDA $M' = (K', \Sigma', \Gamma', \Delta', s', A')$ to accept L. In essence, M' will simply be M and will ignore the stack. Let M' be $(K, \Sigma, \varnothing, \Delta', s, A)$, where Δ' is constructed as follows: For every transition (q_i, c, q_j) in δ, add to Δ' the transition

$((q_i, c, \varepsilon), (q_j, \varepsilon))$. M' behaves identically to M, so $L(M) = L(M')$. So the regular languages are a subset of the context-free languages.

The regular languages are a *proper* subset of the context-free languages because there exists at least one language, A^nB^n, that is context-free but not regular.

Next, we observe that there are *many* more noncontext-free languages than there are context-free ones:

THEOREM 13.2 How Many Context-Free Languages are There?

Theorem: There is a countably infinite number of context-free languages.

Proof: Every context-free language is generated by some context-free grammar $G = (V, \Sigma, R, S)$. We can encode the elements of V as binary strings, so we can lexicographically enumerate all the syntactically legal context-free grammars. There cannot be more context-free languages than there are context-free grammars, so there is at most a countably infinite number of context-free languages. There is not a one-to-one relationship between context-free languages and context-free grammars since there is an infinite number of grammars that generate any given language. But, by Theorem 13.1, every regular language is context-free. And, by Theorem 8.1, there is a countably infinite number of regular languages. So there is at least and at most a countably infinite number of context-free languages.

But, by Theorem 2.3, there is an uncountably infinite number of languages over any nonempty alphabet Σ. So there are many more noncontext-free languages than there are regular ones.

13.2 Showing That a Language is Context-Free

We have so far seen two techniques that can be used to show that a language L is context-free:

- Exhibit a context-free grammar for it.
- Exhibit a (possibly nondeterministic) PDA for it.

There are also closure theorems for context-free languages and they can be used to show that a language is context-free if it can be described in terms of other languages whose status is already known. Unfortunately, there are fewer closure theorems for the context-free languages than there are for the regular languages. In order to be able to discuss both the closure theorems that exist, as well as the ones we'd like but don't have, we will wait and consider the issue of closure theorems in Section 13.4, after we have developed a technique for showing that a language is not context-free.

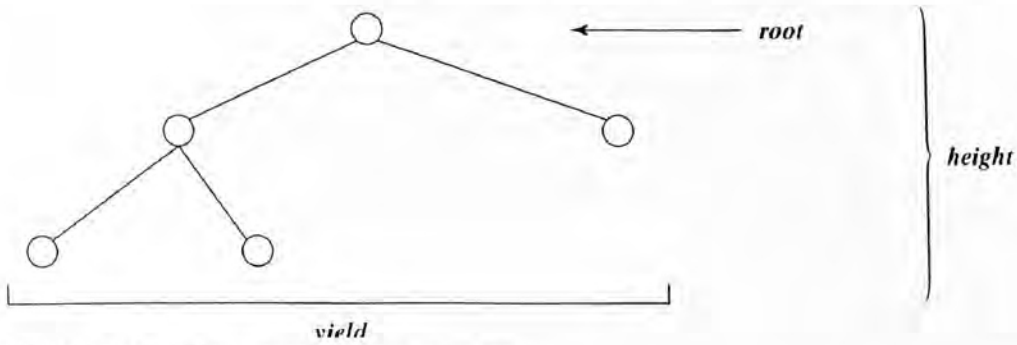


FIGURE 13.1 The structure of a parse tree.

13.3 The Pumping Theorem for Context-Free Languages

Suppose we are given a language and we want to prove that it is not context-free. Just as with regular languages, it is not sufficient simply to claim that we tried to build a grammar or a PDA and we failed. That doesn't show that there isn't some other way to approach the problem.

Instead, we will again approach this problem from the other direction. We will articulate a property that is provably true of all context-free languages. Then, if we can show that a language L does not possess this property, then we know that L is not context-free. So, just as we did when we used the Pumping Theorem for regular languages, we will construct ***proofs by contradiction***. We will say, "If L were context-free, then it would possess certain properties. But it does not possess those properties. Therefore, it is not context-free."

This time we exploit the fact that every context-free language is generated by some context-free grammar. The argument we are about to make is based on the structure of parse trees. Recall that a parse tree, derived by a grammar $G = (V, \Sigma, R, S)$, is a rooted, ordered tree in which:

- Every leaf node is labeled with an element of $\Sigma \cup \{\varepsilon\}$,
- The root node is labeled S,
- Every other node is labeled with some element of $V - \Sigma$, and
- If m is a nonleaf node labeled X and the children of m are labeled $x_1, x_2, \ldots, x_n$, then the rule $X \rightarrow x_1 x_2 \ldots, x_n$ is in R.

Consider an arbitrary parse tree, as shown in Figure 13.1 The ***height*** of a tree is the length of the longest path from the root to any leaf. The ***branching factor*** of a tree is the largest number of daughters of any node in the tree. The ***yield*** of a tree is the ordered sequence of its leaf nodes.

THEOREM 13.3 The Height of A Tree and its Branching Factor Put A Bound On its Yield

Theorem: The length of the yield of any tree T with height h and branching factor b is $\leq b^h$.

Proof: The proof is by induction on h. If h is 1, then just a single rule applies. So the longest yield is of length less than or equal to b. Assume the claim is true for $h = n$. We show that it is true for $h = n + 1$. Consider any tree with $h = n + 1$. It consists of a root, and some number of subtrees, each of which is of height $\leq n$. By the induction hypothesis, the length of the yield of each of those subtrees is $\leq b^n$. The number of subtrees of the root is $\leq b$. So the length of the yield must be $\leq b\,(b^n) = b^{n+1} = b^h$.

Let $G = (V, \Sigma, R, S)$ be a context-free grammar. Let $n = |V - \Sigma|$ be the number of nonterminal symbols in G. Let b be the branching factor of G, defined to be the length of the longest right-hand side of any rule in R.

Now consider any parse tree T generated by G. Suppose that no nonterminal appears more than once on any one path from the root of T to a nonterminal. Then the height of T is $\leq n$. So the longest string that could correspond to the yield of T has length $\leq b^n$.

Now suppose that w is a string in $L(G)$ and $|w| > b^n$. Then any parse tree that G generates for w must contain at least one path that contains at least one repeated nonterminal. Another way to think of this is that, to derive w, G must have used at least one recursive rule. So any parse tree for w must look like the one shown in Figure 13.2, where X is some repeated nonterminal. We use dotted lines to make it clear that the derivation may not be direct but may, instead, require several steps. So, for example, it is possible that the tree shown here was derived using a grammar that contained the rules $X \rightarrow$ aYb, $Y \rightarrow$ bXa, and $X \rightarrow$ ab.

Of course, it is possible that w has more than one parse tree. For the rest of this discussion we will pick some tree such that G generates no other parse tree for w that has fewer nodes. Within that tree it is possible that there are many repeated nonterminals and that some of them are repeated more than once. We will assume only that we have chosen point [1] in the tree such that X is the first repeated nonterminal on any path, coming up from the bottom, in the subtree rooted at [1]. We'll call the rule that was applied at [1] $rule_1$ and the rule that was applied at [2] $rule_2$.

We can sketch the derivation that produced this tree as:

$$S \Rightarrow^* uXz \Rightarrow^* uvXyz \Rightarrow^* uvxyz.$$

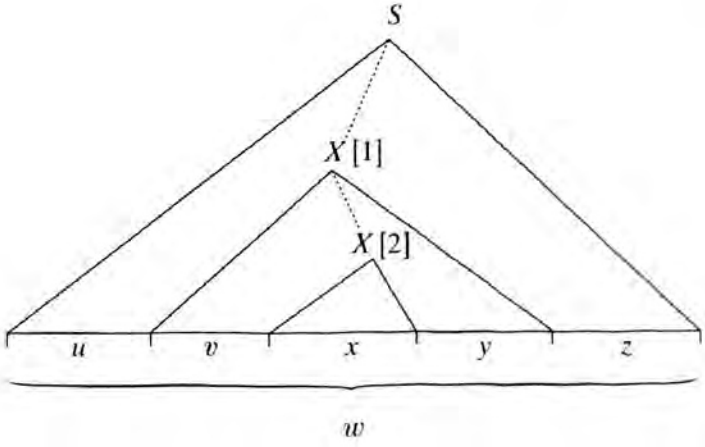


FIGURE 13.2 A parse tree whose height is greater than n.

So we have carved w up into five pieces: u, v, x, y, and z. We observe that:

- There is another derivation in G, $S \Rightarrow^* uXz \Rightarrow^* uxz$, in which, at the point labeled [1], the nonrecursive $rule_2$ is used. So uxz is also in $L(G)$.
- There are infinitely many derivations in G, such as $S \Rightarrow^* uXz \Rightarrow^* uvXyz \Rightarrow^* uvvXyyz \Rightarrow^* uvvxyyz$, in which the recursive $rule_1$ is applied one or more additional times before the nonrecursive $rule_2$ is used. Those derivations produce the strings, uv^2xy^2z, uv^3xy^3z, etc. So all of those strings are also in $L(G)$.
- It is possible that $v = \varepsilon$, as it would be, for example if $rule_1$ were $X \rightarrow Xa$. It is also possible that $y = \varepsilon$, as it would be, for example if $rule_1$ were $X \rightarrow aX$. But it is not possible that both v and y are ε. If they were, then the derivation $S \Rightarrow^* uXz \Rightarrow^* uxz$ would also yield w and it would create a parse tree with fewer nodes. But that contradicts the assumption that we started with a tree with the smallest possible number of nodes.
- The height of the subtree rooted at [1] is at most $n + 1$ (since there is one repeated nonterminal and every other nonterminal can occur no more than once). So $|vxy| \leq b^{n+1}$.

These observations are the basis for the context-free Pumping Theorem, which we state next.

THEOREM 13.4 The Pumping Theorem for Context-Free Languages

Theorem: If L is a context-free language, then:

$$\exists k \geq 1 \, (\forall \text{ strings } w \in L, \text{ where } |w| \geq k \, (\exists u, v, x, y, z \, (w = uvxyz, \; vy \neq \varepsilon, \; |vxy| \leq k, \text{ and } \forall q \geq 0 \, (uv^q xy^q z \text{ is in } L)))).$$

Proof: The proof is the argument that we gave above: If L is context-free, then it is generated by some context-free grammar $G = (V, \Sigma, R, S)$ with n nonterminal symbols and branching factor b. Let k be b^{n+1}. Any string that can be generated by G and whose parse tree contains no paths with repeated nonterminals must have length less than or equal to b^n. Assuming that $b \geq 2$, it must be the case that $b^{n+1} > b^n$. So let w be any string in $L(G)$ where $|w| \geq k$. Let T be any smallest parse tree for w (i.e., a parse tree such that no other parse tree for w has fewer nodes). T must have height at least $n + 1$. Choose some path in T of length at least $n + 1$. Let X be the bottommost repeated nonterminal along that path. Then w can be rewritten as $uvxyz$ as shown in the tree diagram of Figure 13.2. The tree rooted at [1] has height at most $n + 1$. Thus its yield, vxy, has length less than or equal to b^{n+1}, which is k. Further, $vy \neq \varepsilon$ since if vy were ε then there would be a smaller parse tree for w and we chose T so that that wasn't so. Finally, v and y can be pumped: uxz must be in L because $rule_2$ could have been used immediately at [1]. And, for any $q \geq 1$, $uv^q xy^q z$ must be in L because $rule_1$ could have been used q times before finally using $rule_2$.

So, if L is a context-free language, every "long" string in L must be pumpable. Just as with the Pumping Theorem for regular languages, the pumped region can be pumped out once or pumped in any number of times, in all cases resulting in another string that is also in L. So, if there is even one "long" string in L that is not pumpable, then L is not context-free.

Note that the value k plays two roles in the Pumping Theorem. It defines what we mean by a "long" string and it imposes an upper bound on $|vxy|$. When we set k to b^{n+1}, we guaranteed that it was large enough so that we could prove that it served both of those purposes. But we should point out that a smaller value would have sufficed as the definition for a "long" string, since any string of length greater than b^n must be pumpable.

There are a few important ways in which the context-free Pumping Theorem differs from the regular one:

- The most obvious is that two regions, v and y, must be pumped in tandem.
- We don't know anything about where the strings v and y will fall. All we know is that they are reasonably "close together", i.e., $|vxy| \leq k$.
- Either v or y could be empty, although not both.

EXAMPLE 13.1 $A^nB^nC^n$ is Not Context-Free

Let $L = A^nB^nC^n = \{a^nb^nc^n : n \geq 0\}$. We can use the Pumping Theorem to show that L is not context-free. If it were, then there would exist some k such that any string w, where $|w| \geq k$, must satisfy the conditions of the theorem. We show one string w that does not. Let $w = a^kb^kc^k$, where k is the constant from the Pumping Theorem. For w to satisfy the conditions of the Pumping Theorem, there must be some u, v, x, y, and z such that $w = uvxyz$, $vy \neq \varepsilon$, $|vxy| \leq k$, and $\forall q \geq 0$ (uv^qxy^qz is in L). We show that no such u, v, x, y, and z exist. If either v or y contains two or more different characters, then set q to 2 (i.e., pump in once) and the resulting string will have letters out of order and thus not be in $A^nB^nC^n$. (For example, if v is aabb and y is cc, then the string that results from pumping will look like aaa... aaabbaabbccc...ccc.) If both v and y each contain at most one distinct character then set q to 2. Additional copies of at most two different characters are added, leaving the third unchanged. There are no longer equal numbers of the three letters, so the resulting string is not in $A^nB^nC^n$. There is no way to divide w into $uvxyz$ such that all the conditions of the Pumping Theorem are met. So $A^nB^nC^n$ is not context-free.

As with the Pumping Theorem for regular languages, it requires some skill to design simple and effective proofs using the context-free Pumping Theorem. As before, the choices that we can make, when trying to show that a language L is not context-free are:

- We choose w, the string to be pumped. It is important to choose w so that it is in the part of L that captures the essence of why L is not context-free.
- We choose a value for q that shows that w isn't pumpable.

- We may apply closure theorems before we start, so that we show that L is not context-free by showing that some other language L' isn't. We'll have more to say about this technique later.

EXAMPLE 13.2 The Language of Strings with n^2 a's is Not Context-Free

Let $L = \{a^{n^2} : n \geq 0\}$. We can use the Pumping Theorem to show that L is not context-free. If it were, then there would exist some k such that any string w, where $|w| \geq k$, must satisfy the conditions of the theorem. We show one string w that does not. Let n (in the definition of L) be k^2. So $n^2 = k^4$ and $w = a^{k^4}$. For w to satisfy the conditions of the Pumping Theorem, there must be some u, v, x, y, and z, such that $w = uvxyz$, $vy \neq \varepsilon$, $|vxy| \leq k$, and $\forall q \geq 0$ (uv^qxy^qz is in L). We show that no such u, v, x, y, and z exist. Since w contains only a's, $vy = \mathrm{a}^p$, for some nonzero p. Set q to 2. The resulting string, which we'll call s, is a^{k^4+p}, which must be in L. But it isn't because it is too short. If a^{k^4}, which contains $(k^2)^2$ a's, is in L, then the next longer element of L contains $(k^2 + 1)^2$ a's. That's $k^4 + 2k^2 + 1$ a's. So there are no strings in L with length between k^4 and $k^4 + 2k^2 + 1$. But $|s| = k^4 + p$. So, for s to be in L, $p = |vy|$ would have to be at least $2k^2 + 1$. But $|vxy| \leq k$, so p can't be that large. Thus s is not in L. There is no way to divide w into $uvxyz$ such that all the conditions of the Pumping Theorem are met. So L is not context-free.

When using the Pumping Theorem, we focus on v and y. Once they are specified, so are u, x, and z.

To show that there exists no v, y pair that satisfies all of the conditions of the Pumping Theorem, it is sometimes necessary to enumerate a set of cases and rule them out one at a time. Sometimes the easiest way to do this is to imagine the string to be pumped as divided into a set of regions. Then we can consider all the ways in which v and y can fall across those regions.

EXAMPLE 13.3 Dividing the String w Into Regions

Let $L = \{\mathrm{a}^n\mathrm{b}^m\mathrm{a}^n : n, m \geq 0 \text{ and } n \geq m\}$. We can use the Pumping Theorem to show that L is not context-free. If it were, then there would exist some k such that any string w, where $|w| \geq k$, must satisfy the conditions of the theorem. We show one string w that does not. Let $w = \mathrm{a}^k\mathrm{b}^k\mathrm{a}^k$, where k is the constant from the Pumping Theorem. For w to satisfy the conditions of the Pumping Theorem, there must be some u, v, x, y, and z, such that $w = uvxyz$, $vy \neq \varepsilon$, $|vxy| \leq k$, and $\forall q \geq 0$ (uv^qxy^qz is in L). We show that no such u, v, x, y, and z exist. Imagine w divided into three regions as follows:

```
aaa ... aaabbb ... bbbaaa ...aaa
|   1   |   2   |   3   |
```

EXAMPLE 13.3 *(Continued)*

We consider all the cases for where v and y could fall and show that in none of them are all the conditions of the theorem met:

- If either v or y crosses regions, then set q to 2 (thus pumping in once). The resulting string will have letters out of order and so not be in L. So in all the remaining cases we assume that v and y each falls within a single region.
- (1, 1): Both v and y fall in region 1. Set q to 2. In the resulting string, the first group of a's is longer than the second group of a's. So the string is not in L.
- (2, 2): Both v and y fall in region 2. Set q to 2. In the resulting string, the b region is longer than either of the a regions. So the string is not in L.
- (3, 3): Both v and y fall in region 3. Set q to 0. The same argument as for (1, 1).
- (1, 2): Nonempty v falls in region 1 and nonempty y falls in region 2. (If either v or y is empty, it does not matter where it falls. So we can treat it as though it falls in the same region as the nonempty one. We have already considered all of those cases.) Set q to 2. In the resulting string, the first group of a's is longer than the second group of a's. So the string is not in L.
- (2, 3): Nonempty v falls in region 2 and nonempty y falls in region 3. Set q to 2. In the resulting string the second group of a's is longer than the first group of a's. So the string is not in L.
- (1, 3): Nonempty v falls in region 1 and nonempty y falls in region 3. If this were allowed by the other conditions of the Pumping Theorem, we could pump in a's and still produce strings in L. But if we pumped out, we would violate the requirement that the a regions be at least as long as the b region. More importantly, this case violates the requirement that $|vxy| \leq k$. So it need not be considered.

There is no way to divide w into $uvxyz$ such that all the conditions of the Pumping Theorem are met. So L is not context-free.

Consider the language PalEven $= \{ww^R : w \in \{a, b\}^*\}$, the language of even-length palindromes of a's and b's, which we introduced in Example 11.3. Let w be any string in PalEven. Then substrings of w are related to each other in a perfectly nested way, as shown in Figure 13.3 (a). Nested relationships of this sort can naturally be described with a context-free grammar, so languages whose strings are structured in this way are typically context-free.

But now consider the case in which the relationships are not properly nested but instead cross. For example, consider the language WcW $= \{wcw : w \in \{a, b\}^*\}$. Now let w be any string in WcW. Then substrings of w are related to each other as shown in Figure 13.3 (b). We call such dependencies, where lines cross each other, ***cross-serial dependencies***. Languages whose strings are characterized by cross serial dependencies are typically not context-free.

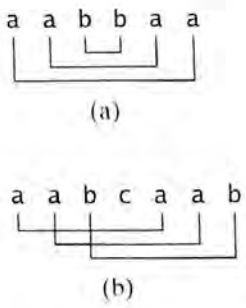


FIGURE 13.3 Nested versus cross-serial dependencies.

EXAMPLE 13.4 WcW is Not Context-Free

Let WcW = $\{wcw : w \in \{a, b\}^*\}$. WcW is not context-free. All its nonempty strings contain cross-serial dependencies.

We can use the Pumping Theorem to show that WcW is not context-free. If it were, then there would exist some k such that any string w, where $|w| \geq k$, must satisfy the conditions of the theorem. We show one string w that does not. Let $w = a^k b^k c a^k b^k$, where k is the constant from the Pumping Theorem. For w to satisfy the conditions of the Pumping Theorem, there must be some u, v, x, y, and z, such that $w = uvxyz$, $vy \neq \varepsilon$, $|vxy| \leq k$, and $\forall q \geq 0$ ($uv^q xy^q z$ is in WcW). We show that no such u, v, x, y, and z exist. Imagine w divided into five regions as follows:

aaa ... aaabbb ... bbbcaaa ... aaabbb ... bbb
| 1 | 2 |3| 4 | 5 |

Call the part before the c the left side and the part after the c the right side. We consider all the cases for where v and y could fall and show that in none of them are all the conditions of the theorem met:

- If either v or y overlaps region 3, set q to 0. The resulting string will no longer contain a c and so is not in WcW.
- If both v and y occur before region 3 or they both occur after region 3, then set q to 2. One side will be longer than the other and so the resulting string is not in WcW.
- If either v or y overlaps region 1, then set q to 2. In order to make the right side match, something would have to be pumped into region 4. But any v, y pair that did that would violate the requirement that $|vxy| \leq k$.
- If either v or y overlaps region 2, then set q to 2. In order to make the right side match, something would have to be pumped into region 5. But any v, y pair that did that would violate the requirement that $|vxy| \leq k$.

There is no way to divide w into $uvxyz$ such that all the conditions of the Pumping Theorem are met. So WcW is not context-free.

Are programming languages like C++ and Java context-free? (G.2)

The language WcW, which we just showed is not context-free, is important because of its similarity to the structure of many common programming languages. Consider a programming language that requires that variables be declared before they are used. If we consider just a single variable w, then a program that declares w and then uses it has a structure very similar to the strings in the language WcW, since the string w must occur in exactly the same form in both the declaration section and the body of the program.

13.4 Some Important Closure Properties of Context-Free Languages

It helps to be able to analyze a complex language by decomposing it into simpler pieces. Closure theorems, when they exist, enable us to do that. We'll see in this section that, while the context-free languages are closed under some common operations, we cannot prove as strong a set of closure theorems as we were able to prove for the regular languages.

13.4.1 The Closure Theorems

THEOREM 13.5 Closure Under Union, Concatenation, Kleene Star, Reverse, and Letter Substitution

Theorem: The context-free languages are closed under union, concatenation, Kleene star, reverse, and letter substitution.

Proof: We prove each of the claims separately by construction:

- The context-free languages are closed under union: If L_1 and L_2 are context-free languages, then there exist context-free grammars $G_1 = (V_1, \Sigma_1, R_1, S_1)$ and $G_2 = (V_2, \Sigma_2, R_2, S_2)$ such that $L_1 = L(G_1)$ and $L_2 = L(G_2)$. If necessary, rename the nonterminals of G_1 and G_2 so that the two sets are disjoint and so that neither includes the symbol S. We will build a new grammar G such that $L(G) = L(G_1) \cup L(G_2)$. G will contain all the rules of both G_1 and G_2. We add to G a new start symbol, S, and two new rules, $S \rightarrow S_1$ and $S \rightarrow S_2$. The two new rules allow G to generate a string iff at least one of G_1 or G_2 generates it. So $G = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S \rightarrow S_1, S \rightarrow S_2\}, S)$.
- The context-free languages are closed under concatenation: If L_1 and L_2 are context-free languages, then there exist context-free grammars $G_1 = (V_1, \Sigma_1, R_1, S_1)$ and $G_2 = (V_2, \Sigma_2, R_2, S_2)$ such that $L_1 = L(G_1)$ and $L_2 = L(G_2)$. If necessary, rename the nonterminals of G_1 and G_2 so that the two sets are disjoint and so that neither includes the symbol S. We will build a new grammar G such that $L(G) = L(G_1)\, L(G_2)$. G will contain all the rules of both G_1 and G_2. We add to G a new start symbol, S, and one new rule, $S \rightarrow S_1 S_2$. So $G = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S \rightarrow S_1 S_2\}, S)$.
- The context-free languages are closed under Kleene star: If L_1 is a context-free language, then there exists a context-free grammar $G_1 = (V_1, \Sigma_1, R_1, S_1)$

such that $L_1 = L(G_1)$. If necessary, rename the nonterminals of G_1 so that V_1 does not include the symbol S. We will build a new grammar G such that $L(G) = L(G_1)^*$. G will contain all the rules of G_1. We add to G a new start symbol, S, and two new rules, $S \rightarrow \varepsilon$ and $S \rightarrow S\,S_1$. So $G = (V_1 \cup \{S\}, \Sigma_1, R_1 \cup \{S \rightarrow \varepsilon, S \rightarrow S\,S_1\}, S)$.

- The context-free languages are closed under reverse: Recall that $L^R = \{w \in \Sigma^* : w = x^R \text{ for some } x \in L\}$. If L is a context-free language, then it is generated by some Chomsky normal form grammar $G = (V, \Sigma, R, S)$. Every rule in G is of the form $X \rightarrow BC$ or $X \rightarrow a$, where X, B, and C are elements of $V - \Sigma$ and $a \in \Sigma$. In the latter case $L(X) = \{a\}$. $\{a\}^R = \{a\}$. In the former case, $L(X) = L(B)L(C)$. By Theorem 2.4, $(L(B)L(C))^R = L(C)^R L(B)^R$. So we construct, from G, a new grammar G', such that $L(G') = L^R$. $G' = (V_G, \Sigma_G, R', S_G)$, where R' is constructed as follows:
  - For every rule in G of the form $X \rightarrow BC$, add to R' the rule $X \rightarrow CB$.
  - For every rule in G of the form $X \rightarrow a$, add to R' the rule $X \rightarrow a$.
- The context-free languages are closed under letter substitution, defined as follows: Consider any two alphabets, Σ_1 and Σ_2. Let *sub* be any function from Σ_1 to Σ_2^*. Then *letsub* is a letter substitution function from L_1 to L_2 iff *letsub* $(L_1) = \{w \in \Sigma_2^* : \exists y \in L_1 (w = y$ except that every character c of y has been replaced by *sub* $(c))\}$. We leave the proof of this as an exercise.

As with regular languages, we can use these closure theorems as a way to prove that a more complex language is context-free if it can be shown to be built from simpler ones using operations under which the context-free languages are closed.

THEOREM 13.6 Nonclosure Under Intersection, Complement, and Difference

Theorem: The context-free languages are not closed under intersection, complement, or difference.

Proof:

- The context-free languages are not closed under intersection: The proof is by counterexample. Let:

$$L_1 = \{a^n b^n c^m : n, m \geq 0\}. \qquad /^*\ \text{equal a's and b's.}$$
$$L_2 = \{a^m b^n c^n : n, m \geq 0\}. \qquad /^*\ \text{equal b's and c's.}$$

Both L_1 and L_2 are context-free since there exist straightforward context-free grammars for them.

But now consider:

$$L = L_1 \cap L_2$$
$$= \{a^n b^n c^n : n \geq 0\}.$$

If the context-free languages were closed under intersection, L would have to be context-free. But we proved, in Example 13.1, that it isn't.

- The context-free languages are not closed under complement: Given any sets L_1 and L_2,

$$L_1 \cap L_2 = \neg(\neg L_1 \cup \neg L_2).$$

The context-free languages are closed under union. So, if they were also closed under complement, they would necessarily be closed under intersection. But we just showed that they are not. Thus they are not closed under complement either. We've also seen an example that proves this claim directly. $\neg A^nB^nC^n$ is context-free. We showed a PDA that accepts it in Example 12.8. But $\neg(\neg A^nB^nC^n) = A^nB^nC^n$ is not context-free.

- The context-free languages are not closed under difference (subtraction): Given any language L,

$$\neg L = \Sigma^* - L.$$

Σ^* is context-free. So, if the context-free languages were closed under difference, the complement of any context-free language would necessarily be context-free. But we just showed that that is not so.

Recall that, in using the regular Pumping Theorem to show that some language L was not regular, we sometimes found it useful to begin by intersecting L with another regular language to create a new language L'. Since the regular languages are closed under intersection, L' would necessarily be regular if L were. We then showed that L', designed to be simpler to work with, was not regular. And so neither was L.

It would be very useful to be able to exploit this technique when using the context-free Pumping Theorem. Unfortunately, as we have just shown, the context-free languages are not closed under intersection. Fortunately, however, they are closed under intersection with the regular languages. We'll prove this result next and then, in Section 13.4.2, we'll show how it can be exploited in a proof that a language is not context-free.

THEOREM 13.7 Closure Under Intersection With the Regular Languages

Theorem: The context-free languages are closed under intersection with the regular languages.

Proof: The proof is by construction. If L_1 is context-free, then there exists some PDA $M_1 = (K_1, \Sigma, \Gamma_1, \Delta_1, s_1, A_1)$ that accepts it. If L_2 is regular then there exists a DFSM $M_2 = (K_2, \Sigma, \delta, s_2, A_2)$ that accepts it. We construct a new PDA, M_3 that accepts $L_1 \cap L_2$. M_3 will work by simulating the parallel execution of M_1 and M_2. The states of M_3 will be ordered pairs of states of M_1 and M_2. As each input character is read, M_3 will simulate both M_1 and M_2 moving appropriately to a new state. M_3 will have a single stack, which will be controlled by M_1. The only slightly tricky thing is that M_1 may contain ε-transitions. So M_3 will have to

allow M_1 to follow them while M_2 just stays in the same state and waits until the next input symbol is read.

$M_3 = (K_1 \times K_2, \Sigma, \Gamma_1, \Delta_3, (s_1, s_2), A_1 \times A_2)$, where Δ_3 is built as follows:

- For each transition $((q_1, a, \beta), (p_1, \gamma))$ in Δ_1,
  and each transition $((q_2, a), p_2)$ in δ, add to Δ_3 the transition:
  $$(((q_1, q_2), a, \beta), ((p_1, p_2), \gamma)).$$
- For each transition $((q_1, \varepsilon, \beta), (p_1, \gamma))$ in Δ_1,
  and each state q_2 in K_2, add to Δ_3 the transition:
  $$(((q_1, q_2), \varepsilon, \beta), ((p_1, q_2), \gamma)).$$

We define *intersectPDAandFSM* as follows:

intersectPDAandFSM (M_1: PDA, M_2: FSM) =
Build M_3 as defined in the proof of Theorem 13.7.

THEOREM 13.8 Closure Under Difference with the Regular Languages

Theorem: The difference $(L_1 - L_2)$ between a context-free language L_1 and a regular language L_2 is context-free.

Proof: $L_1 - L_2 = L_1 \cap \neg L_2$. If L_2 is regular, then, since the regular languages are closed under complement, $\neg L_2$ is also regular. Since L_1 is context-free, by Theorem 13.7, $L_1 \cap \neg L_2$ is context-free.

The last two theorems are important tools, both for showing that a language is context-free and for showing that a language is not context-free.

EXAMPLE 13.5 Using Closure Theorems to Prove A Language Context-Free

Consider the perhaps contrived language $L = \{a^n b^n : n \geq 0 \text{ and } n \neq 1776\}$. Another way to describe L is that it is $\{a^n b^n : n \geq 0\} - \{a^{1776} b^{1776}\}$. $A^n B^n = \{a^n b^n : n \geq 0\}$ is context-free. We have shown both a simple grammar that generates it and a simple PDA that accepts it. $\{a^{1776} b^{1776}\}$ is finite and thus regular. So, by Theorem 13.8, L is context free.

Generalizing that example a bit, from Theorem 13.8 it follows that any language that can be described as the result of subtracting a finite number of elements from some language known to be context-free must also be context-free.

13.4.2 Using the Pumping Theorem in Conjunction with the Closure Properties

Languages that impose no specific order constraints on the symbols contained in their strings are not always context-free. But it may be hard to prove that one isn't just by using the Pumping Theorem. In such a case, it is often useful to exploit Theorem 13.7, which tells us that the context-free languages are closed under intersection with the regular languages.

Recall our notational convention from Section 13.3: (n, n) means that all nonempty substrings of vy occur in region n. This may happen either because v and y are both nonempty and they both occur in region n. Or it may happen because one or the other is empty and the nonempty one occurs in region n.

Are natural languages like English or Chinese or German context-free? (L.3.3)

EXAMPLE 13.6 WW is Not Context-Free

Let WW $= \{ww : w \in \{a, b\}^*\}$. WW is similar to WcW $= \{wcw : w \in \{a, b\}^*\}$, except that there is no longer a middle marker. Because, like WcW, it contains cross-serial dependencies, it is not context-free. We could try proving that by using the Pumping Theorem alone. Here are some attempts, using various choices for w:

- Let $w = (ab)^{2k}$. If $v = \varepsilon$ and $y = ab$, pumping works fine.
- Let $w = a^k b a^k b$. If $v =$ a and is in the first group of a's and $y =$ a and is in the second group of a's, pumping works fine.
- Let $w = a^k b^k a^k b^k$. Now the constraint that $|vxy| \leq k$ prevents v and y from both being in the two a regions or the two b regions. This choice of w will lead to a successful Pumping Theorem proof. But there are four regions in w and we must consider all the ways in which v and y could overlap those regions, including all those in which either or both of v and y occur on a region boundary. While it is possible to write out all those possibilities and show, one at a time, that every one of them violates at least one condition of the Pumping Theorem, there is an easier way.

If WW were context-free, then $L' =$ WW $\cap$ a*b*a*b* would also be context-free. But it isn't, which we can show using the Pumping Theorem. If it were, then there would exist some k such that any string w, where $|w| \geq k$, must satisfy the conditions of the theorem. We show one string w that does not. Let $w = a^k b^k a^k b^k$, where k is the constant from the Pumping Theorem. For w to satisfy the conditions of the Pumping Theorem, there must be some u, v, x, y, and z, such that $w = uvxyz$, $vy \neq \varepsilon$, $|vxy| \leq k$, and $\forall q \geq 0$ ($uv^q xy^q z$ is in L'). We show that no such u, v, x, y, and z exist. Imagine w divided into four regions as follows:

```
aaa ... aaabbb ... bbbaaa ... aaabbb ... bbb
|   1   |   2   |   3   |   4   |
```

We consider all the cases for where v and y could fall and show that in none of them are all the conditions of the theorem met:

- If either v or y overlaps more than one region, set q to 2. The resulting string will not be in a*b*a*b* and so is not in L'.
- If $|vy|$ is not even then set q to 2. The resulting string will have odd length and so not be in L'. We assume in all the other cases that $|vy|$ is even.
- $(1, 1), (2, 2), (1, 2)$: Set q to 2. The boundary between the first half and the second half will shift into the first b region. So the second half will start with a b, while the first half still starts with an a. So the resulting string is not in L'.
- $(3, 3), (4, 4), (3, 4)$: Set q to 2. This time the boundary shifts into the second a region. The first half will end with an a while the second half still ends with a b. So the resulting string is not in L'.
- $(2, 3)$: Set q to 2. If $|v| \neq |y|$ then the boundary moves and, as argued above, the resulting string is not in L'. If $|v| = |y|$ then the first half contains more b's and the second half contains more a's. Since they are no longer the same, the resulting string is not in L'.
- $(1, 3), (1, 4)$, and $(2, 4)$ violate the requirement that $|vxy| \leq k$.

There is no way to divide w into $uvxyz$ such that all the conditions of the Pumping Theorem are met. So L' is not context-free. So neither is WW.

> One reason that context-free grammars are typically too weak to describe musical structures is that they cannot describe constraints such as the one that defines WW. (N.1.2)

EXAMPLE 13.7 A Simple Arithmetic Language is Not Context-Free

Let $L = \{x \# y = z : x, y, z \in \{0, 1\}^* \text{ and, if } x, y \text{ and } z \text{ are viewed as positive binary numbers without leading zeros, then } xy = z^R\}$. For example, 100#111 = 00111 $\in L$. (We do this example instead of the more natural one in which we require that $xy = z$ because it seems as though it might be more likely to be context-free. As we'll see, however, even this simpler variant is not.)

If L were context-free, then $L' = L \cap 10^*\#1^* = 0^*1^*$ would also be context-free. But it isn't, which we can show using the Pumping Theorem. If it were, then there would exist some k such that any string w, where $|w| \geq k$, must satisfy the conditions of the theorem. We show one string w that does not. Let $w = 10^k\#1^k = 0^k1^k$, where k is the constant from the Pumping Theorem. Note that $w \in L$ because $10^k \cdot 1^k = 1^k0^k$.

EXAMPLE 13.7 *(Continued)*

For w to satisfy the conditions of the Pumping Theorem, there must be some u, v, x, y, and z, such that $w = uvxyz$, $vy \neq \varepsilon$, $|vxy| \leq M$, and $\forall q \geq 0$ (uv^qxy^qz is in L). We show that no such u, v, x, y, and z exist. Imagine w divided into seven regions as follows:

```
1 000 ... 000 # 111 ... 111 = 000 ... 000111 ... 111
|1|   2     |3|   4     |5|   6     |   7     |
```

We consider all the cases for where v and y could fall and show that in none of them are all the conditions of the theorem met:

- If either v or y overlaps region 1, 3, or 5 then set q to 0. The resulting string will not be in 10*#1* = 0*1* and so is not in L'.
- If either v or y contains the boundary between 6 and 7, set q to 2. The resulting string will not be in 10*#1* = 0*1* and so is not in L'. So the only cases left to consider are those where v and y each occur within a single region.
- (2, 2), (4, 4), (2, 4): Set q to 2. Because there are no leading zeros, changing the left side of the string changes its value. But the right side doesn't change to match. So the resulting string is not in L'.
- (6, 6), (7, 7), (6, 7): Set q to 2. The right side of the equality statement changes value but the left side doesn't. So the resulting string is not in L'.
- (4, 6): Note that, because of the first argument to the multiplication, the number of 1's in the second argument must equal the number of 1's after the =. Set q to 2. The number of 1's in the second argument changed but the number of 1's in the result did not. So the resulting string is not in L'.
- (2, 6), (2, 7), and (4, 7) violate the requirement that $|vxy| \leq k$.

There is no way to divide w into $uvxyz$ such that all the conditions of the Pumping Theorem are met. So L is not context-free.

Sometimes the closure theorems can be used to reduce the proof that a new language L is not context-free to the proof that some other language L' is not context-free, where we have already proven the case for L'.

EXAMPLE 13.8 Using Intersection to Force Order Constraints

Let $L = \{w \in \{a, b, c\}^* : \#_a(w) = \#_b(w) = \#_c(w)\}$. If L were context-free, then $L' = L \cap a^*b^*c^*$ would also be context-free. But $L' = A^nB^nC^n$, which is not context-free, so neither is L.

13.5 Deterministic Context-Free Languages ✹

The regular languages are closed under complement, intersection, and difference. Why are the context-free languages different? In a nutshell, because the machines that accept them may necessarily be nondeterministic. Recall the technique that we used, in the proof of Theorem 8.4, to show that the regular languages are closed under complement: Given a (possibly nondeterministic) FSM M_1, we used the following procedure to construct a new FSM M_2 such that $L(M_2) = \neg L(M_1)$:

1. From M_1, construct an equivalent DFSM M', using the algorithm *ndfsmtodfsm*, presented in the proof of Theorem 5.3. (If M_1 is already deterministic, $M' = M_1$.)
2. M' must be stated completely, so if it is described with an implied dead state, add the dead state and all required transitions to it.
3. Begin building M_2 by setting it equal to M'. Then swap the accepting and the nonaccepting states. So $M_2 = (K_{M'}, \Sigma, \delta_{M'}, s_{M'}, K_{M'} - A_{M'})$.

We have no PDA equivalent of *ndfsmtodfsm*, so we cannot simply adapt this construction for PDAs. Our proofs that the regular languages are closed under intersection and difference relied on the fact that they were closed under complement, so we cannot adapt those proofs here either.

We have no PDA equivalent of *ndfsmtodfsm* because there provably isn't one, as we will show shortly. Recall that, in Section 12.2, we defined a PDA M to be ***deterministic*** iff:

- Δ_M contains no pairs of transitions that compete with each other, and
- if q is an accepting state of M, then there is no transition $((q, \varepsilon, \varepsilon), (p, a))$ for any p or a.

In other words, M never has a choice between two or more moves, nor does it have a choice between moving and accepting. There exist context-free languages that cannot be accepted by any deterministic PDA. But suppose that we restrict our attention to the ones that can.

What is a Deterministic Context-Free Language?

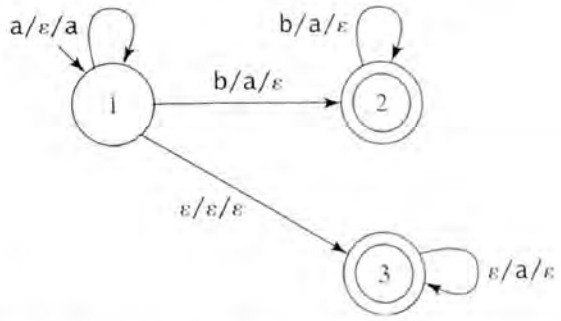
We are about to define the class of deterministic context-free languages. Because this class is useful, we would like it to be as large as possible. So let \$ be an end-of-string marker. We could use any symbol that is not in Σ_L (for example <*line feed*> or <*cr*>), but \$ is easier to read. A language L is ***deterministic context-free*** iff $L\$$ can be accepted by some deterministic PDA.

To see why we have defined the deterministic context-free languages to exploit an end-of-string marker, consider the following example of a straightforward language for which no deterministic PDA exists unless an end-of-string marker is used.

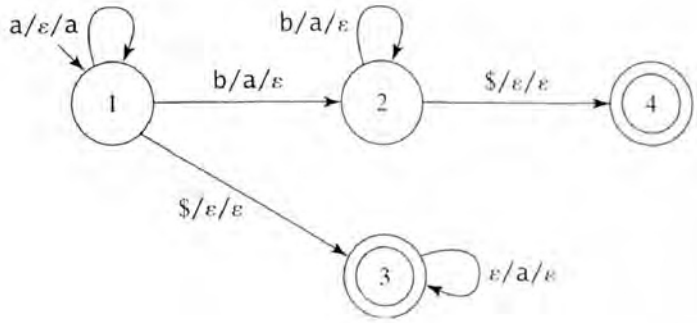
EXAMPLE 13.9 Why an End-of-String Marker is Useful

Let $L = \mathrm{a}^* \cup \{\mathrm{a}^n\mathrm{b}^n : n > 0\}$. Consider any PDA M that accepts L. When it begins reading a's, M must push them onto the stack in case there are going to be b's. But, if it runs out of input without seeing b's, it needs a way to pop those a's from

the stack before it can accept. Without an end-of-string marker, there is no way to allow that popping to happen *only* when all the input has been read. So, for example, the following PDA accepts L, but it is nondeterministic because the transition to state 3 (where the a's will be popped) can compete with both of the other transitions from state 1.



With an end-of-string marker, we can build the following deterministic PDA, which can only take the transition to state 3, the a-popping state, when it sees the $:



Before we go any farther, we have to be sure of one thing. We introduced the end-of-string marker to make it easier to build PDAs that are deterministic. We need to make sure that it doesn't make it possible to build a PDA for a language L that was not already context-free. In other words, adding the end-of-string marker cannot convert a language that was not context-free into one that is. We do that next.

THEOREM 13.9 CFLs and Deterministic CFLs

Theorem: Every deterministic context-free language (as just defined) is context-free.

Proof: If L is deterministic context-free, then $L\$$ is accepted by some deterministic PDA $M = (K, \Sigma, \Gamma, \Delta, s, A)$. From M, we construct M' such that $L(M') = L$. The idea is that, whatever M can do on reading \$, M' can do on reading ε (i.e., by simply guessing that it is at the end of the input). But, as soon as M' makes that guess, it cannot read any more input. It may perform the rest of its computation (such as popping its stack), but any path that pretends it has seen the \$ before it

has read all of its input will fail to accept. To enable M' to perform whatever stack operations M could have performed, but not to read any input, M' will be composed of two copies of M: The first copy will be identical to M, and M' will operate in that part of itself until it guesses that it is at the end of the input; the second copy will be identical to M except that it contains only the transitions that do not consume any input. The states in the first copy will be labeled as in M. Those in the second copy will have the prime symbol appended to their names. So, if M contains the transition $((q, \varepsilon, \gamma_1), (p, \gamma_2))$, M' will contain the transition $((q', \varepsilon, \gamma_1), (p', \gamma_2))$. The two copies will be connected by finding, in the first copy of M, every \$-transition from some state q to some state p. We replace each such transition with an ε-transition into the second copy. So the new transition goes from q to p'.

We can define the following procedure to construct M':

without\$(M: PDA) =

1. Initially, set M' to M.

/* Make the copy that does not read any input.

2. For every state q in M, add to M' a new state q'.
3. For every transition $((q, \varepsilon, \gamma_1), (p, \gamma_2))$ in Δ_M do:
   3.1. Add to $\Delta_{M'}$ the transition $((q', \varepsilon, \gamma_1), (p', \gamma_2))$.

/* Link up the two copies.

4. For every transition $((q, \$, \gamma_1), (p, \gamma_2))$ in Δ_M do:
   4.1. Add to $\Delta_{M'}$ the transition $((q, \varepsilon, \gamma_1), (p', \gamma_2))$.
   4.2. Remove $((q, \$, \gamma_1), (p, \gamma_2))$ from $\Delta_{M'}$.

/* Set the accepting states of M'.

5. $A_{M'} = \{q' : q \in A\}$.

Closure Properties of the Deterministic Context-Free Languages

The deterministic context-free languages are practically very significant because it is possible to build deterministic, linear time parsers for them. They also possess additional formal properties that are important, among other reasons, because they enable us to prove that not all context-free languages are deterministic context-free. The most important of these is that the deterministic context-free languages, unlike the larger class of context-free languages, are closed under complement.

THEOREM 13.10 Closure Under Complement

Theorem: The deterministic context-free languages are closed under complement.

Proof: The proof is by construction. If L is a deterministic context-free language over the alphabet Σ, then $L\$$ is accepted by some deterministic PDA $M = (K, \Sigma \cup \{\$\}, \Gamma, \Delta, s, A)$. We need to describe an algorithm that constructs

a new deterministic PDA that accepts $(\neg L)\$$. To prove Theorem 8.4 (that the regular languages are closed under complement), we defined a construction that proceeded in two steps: Given an arbitrary FSM, convert it to an equivalent DFSM, and then swap accepting and nonaccepting states. We can skip the first step here, but we must solve a new problem. A deterministic PDA may fail to accept an input string w for any one of several reasons:

1. Its computation ends before it finishes reading w.
2. Its computation ends in an accepting state but the stack is not empty.
3. Its computation loops forever, following ε-transitions, without ever halting in either an accepting or a nonaccepting state.
4. Its computation ends in a nonaccepting state.

If we simply swap accepting and nonaccepting states we will correctly fail to accept every string that M would have accepted (i.e., every string in $L\$$). But we will not necessarily accept every string in $(\neg L)\$$. To do that, we must also address issues 1 through 3 above.

An additional problem is that we don't want to accept $\neg L$ (M). That includes strings that do not end in \$. We must accept only strings that do end in \$ and that are in $(\neg L)\$$.

A construction that solves these problems is given in D.2.

What else can we say about the deterministic context-free languages? We know that they are closed under complement. What about union and intersection? We observe that $L_1 \cap L_2 = \neg(\neg L_1 \cup \neg L_2)$. So, if the deterministic context-free languages were closed under union, they would necessarily be closed under intersection also. But they are not closed under union. The context-free languages are closed under union, so the union of two deterministic context-free languages must be context-free. It may, however not be deterministic. The deterministic context-free languages are also not closed under intersection. In fact, when two deterministic context-free languages are intersected, the result may not even be context-free.

THEOREM 13.11 Nonclosure Under Union

Theorem: The deterministic context-free languages are not closed under union.

Proof: We show a counterexample:

$$\text{Let } L_1 = \{a^i b^j c^k : i, j, k \geq 0 \text{ and } i \neq j\}.$$

$$\text{Let } L_2 = \{a^i b^j c^k : i, j, k \geq 0 \text{ and } j \neq k\}.$$

$$\begin{aligned}\text{Let } L' &= L_1 \cup L_2. \\ &= \{a^i b^j c^k : i, j, k \geq 0 \text{ and } ((i \neq j) \text{ or } (j \neq k))\}.\end{aligned}$$

$$\begin{aligned}\text{Let } L'' &= \neg L'. \\ &= \{a^i b^j c^k : i, j, k \geq 0 \text{ and } i = j = k\} \cup \\ &\quad \{w \in \{a, b, c\}^* : \text{the letters are out of order}\}.\end{aligned}$$

$$\text{Let } L''' = L'' \cap a^*b^*c^*.$$
$$= \{a^n b^n c^n : n \geq 0\}.$$

L_1 and L_2 are deterministic context-free. Deterministic PDAs that accept $L_1\$$ and $L_2\$$ can be constructed using the same approach we used to build a deterministic PDA for $L = \{a^m b^n : m \neq n; m, n > 0\}$ in Example 12.7. Their union L' is context-free but it cannot be deterministic context-free. If it were, then its complement L'' would also be deterministic context-free and thus context-free. But it isn't. If it were context-free, then L''', the intersection of L'' with $a^*b^*c^*$, would also be context-free since the context-free languages are closed under intersection with the regular languages. But L''' is $A^nB^nC^n = \{a^n b^n c^n : n \geq 0\}$, which we have shown is not context-free.

THEOREM 13.12 Nonclosure Under Intersection

Theorem: The deterministic context-free languages are not closed under intersection.

Proof: We show a counterexample:

$$\text{Let } L_1 = \{a^i b^j c^k : i, j, k \geq 0 \text{ and } i = j\}.$$
$$\text{Let } L_2 = \{a^i b^j c^k : i, j, k \geq 0 \text{ and } j = k\}.$$
$$\text{Let } L' = L_1 \cap L_2.$$
$$= \{a^n b^n c^n : n \geq 0\}.$$

L_1 and L_2 are deterministic context-free. The deterministic PDA shown in Figure 13.4 accepts $L_1\$$. A similar one accepts L_2. But we have shown that their intersection L' is not context-free, much less deterministic context-free.

A Hierarchy within the Class of Context-Free Languages

The most important result of this section is the following theorem: There are context-free languages that are not deterministic context-free. Since there are context-free languages for which no deterministic PDA exists, there can exist no equivalent of *ndfsmtodfsm* for PDAs. Nondeterminism is a fact of life when working with PDAs unless we are willing to work only with languages that have been designed to be deterministic.

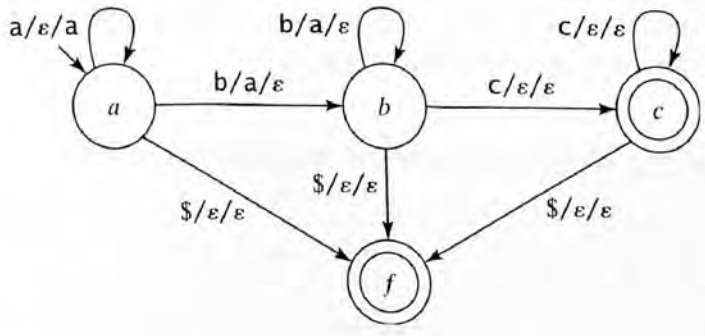


FIGURE 13.4 A deterministic PDA that accepts $\{a^i b^j c^k : i, j, k \geq 0 \text{ and } i = j\}$.

The fact that there are context-free languages that are not deterministic poses a problem for the design of efficient parsing algorithms. The best parsing algorithms we have sacrifice either generality (i.e., they cannot correctly parse all context-free languages) or efficiency (i.e., they do not run in time that is linear in the length of the input). In Chapter 15, we will describe some of these algorithms.

THEOREM 13.13 Some CFLs are not Deterministic

Theorem: The class of deterministic context-free languages is a *proper* subset of the class of context-free languages. Thus there exist nondeterministic PDAs for which no equivalent deterministic PDA exists.

Proof: By Theorem 13.9, every deterministic context-free language is context-free. So all that remains is to show that there exists at least one context-free language that is not deterministic context-free.

Consider $L = \{a^i b^j c^k : i, j, k \geq 0 \text{ and } ((i \neq j) \text{ or } (j \neq k))\}$. L is context-free. The construction of a grammar for it was an exercise in Chapter 11. But we can show that L is not deterministic context-free by the same argument that we used in the proof of Theorem 13.11. If L were deterministic context-free, then, by Theorem 13.10, its complement $L' = \{a^i b^j c^k : i, j, k \geq 0 \text{ and } i = j = k\} \cup \{w \in \{a, b, c\}^* : \text{the letters are out of order}\}$ would also be deterministic context-free and thus context-free. If L' were context-free, then $L'' = L' \cap a^*b^*c^*$ would also be context-free (since the context-free languages are closed under intersection with the regular languages). But $L'' = A^nB^nC^n = \{a^n b^n c^n : n \geq 0\}$, which is not context-free. So L is context-free but not deterministic context-free.

Since L is context-free, it is accepted by some (nondeterministic) PDA M. M is an example of a nondeterministic PDA for which no equivalent deterministic PDA exists. If such a deterministic PDA did exist and accept L, it could be converted into a deterministic PDA that accepted $L\$$. But, if that machine existed, L would be deterministic context-free and we just showed that it is not.

We get the class of deterministic context-free languages when we think about the context-free languages from the perspective of PDAs that accept them. Recall from Section 11.7.3 that, when we think about the context-free languages from the perspective of the grammars that generate them, we also get a subclass of languages that are, in some sense, "easier" than others: There are context-free languages for which unambiguous grammars exist and there are others that are inherently ambiguous, by which we mean that every corresponding grammar is ambiguous.

EXAMPLE 13.10 Inherent Ambiguity versus Nondeterminism

Recall the language $L_1 = \{a^i b^j c^k : i, j, k \geq 0 \text{ and } ((i = j) \text{ or } (j = k))\}$, which can also be described as $\{a^n b^n c^m : n, m \geq 0\} \cup \{a^n b^m c^m : n, m \geq 0\}$. L_1 is inherently ambiguous because every string that is also in $A^nB^nC^n = \{a^n b^n c^n : n \geq 0\}$ is an element of both sublanguages and so has at least two derivations in any grammar for L_1.

Now consider the slightly different language $L_2 = \{a^n b^n c^m d : n, m \geq 0\} \cup \{a^n b^m c^m e : n, m \geq 0\}$. L_2 is not inherently ambiguous. It is straightforward to write an unambiguous grammar for each of the two sublanguages and any string in L_2 is an element of only one of them (since each such string must end in d or e but not both). L_2 is not, however, deterministic. There exists no PDA that can decide which of the two sublanguages a particular string is in until it has consumed the entire string.

What is the relationship between the deterministic context-free languages and the languages that are not inherently ambiguous? The answer is shown in Figure 13.5.

The subset relations shown in the figure are proper:

- There exist deterministic context-free languages that are not regular. These languages are in the innermost donut in the figure. One example is $A^n B^n = \{a^n b^n : n \geq 0\}$.
- There exist languages that are not in the inner donut (i.e., they are not deterministic). But they are context-free and not inherently ambiguous. Two examples of languages in this second donut are:
  - PalEven $= \{ww^R : w \in \{a, b\}^*\}$. The grammar we showed for it in Example 11.3 is unambiguous.
  - $\{a^n b^n c^m d : n, m \geq 0\} \cup \{a^n b^m c^m e : n, m \geq 0\}$.
- There exist languages that are in the outer donut because they are inherently ambiguous. Two examples are:
  - $\{a^i b^j c^k : i, j, k \geq 0 \text{ and } ((i = j) \text{ or } (j = k))\}$.
  - $\{a^i b^j c^k : i, j, k \geq 0 \text{ and } ((i \neq j) \text{ or } (j \neq k))\}$.

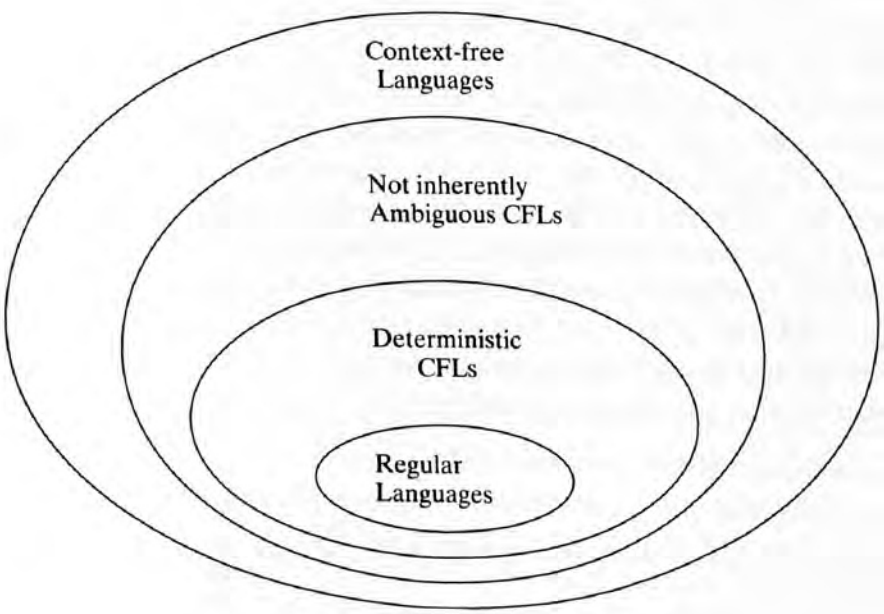


FIGURE 13.5 A hierarchy within the class of context-free languages.

To prove that the figure is properly drawn requires two additional results:

THEOREM 13.14 Every Regular Language is Deterministic Context-Free

Theorem: Every regular language is deterministic context-free.

Proof: The proof is by construction. {$} is regular. So, if L is regular, then so is L\$ (since the regular languages are closed under concatenation). So there is a DFSM M that accepts it. Using the construction that we used in the proof of Theorem 13.1 to show that every regular language is context-free, construct, from M a PDA P that accepts L\$. P will be deterministic.

THEOREM 13.15 Every Deterministic CFL has an Unambiguous Grammar

Theorem: For every deterministic context-free language there exists an unambiguous grammar.

Proof: If a language L is deterministic context-free, then there exists a deterministic PDA M that accepts L\$. We prove the theorem by construction of an unambiguous grammar G such that $L(M) = L(G)$. We construct G using approximately the same technique that we used to build a grammar from a PDA in the proof of Theorem 12.2. The algorithm *PDAtoCFG* that we presented there proceeded in two steps:

1. Invoke *convertPDAtorestricted*(M) to build M', an equivalent PDA in restricted normal form.
2. Invoke *buildgrammar* (M'), to build an equivalent grammar G.

It is straightforward to show that, if M' is deterministic, then the grammar G that *buildgrammar* constructs will be unambiguous: G produces derivations that mimic the operation of M'. Since M' is deterministic, on any input w it can follow only one path. So G will be able to produce only one leftmost derivation for w. Thus w has only one parse tree. If every string in $L(G)$ has a single parse tree, then G is unambiguous. Since M' accepts L\$, G will generate L\$. But we can build, from G, a grammar G' that generates L by substituting ε for \$ in each rule in which \$ occurs.

So it remains to show that, from any deterministic PDA M, it is possible to build an equivalent PDA M' that is in restricted normal form and is still deterministic. This can be done using the algorithm *convertPDAtodetnormalform*, which is described in the proof, presented in D.2, of Theorem 13.10 (that the deterministic context-free languages are closed under complement). If M is deterministic, then the PDA that is returned by *convertPDAtodetnormalform*(M) will be both deterministic and in restricted normal form.

So the construction that proves the theorem is:

buildunambiggrammar(M: deterministic PDA) =

1. Let G = *buildgrammar* (*convertPDAtodetnormalform* (M)).
2. Let G' be the result of substituting ε for \$ in each rule in which \$ occurs.
3. Return G'.

13.6 Ogden's Lemma ✹

The context-free Pumping Theorem is a useful tool for showing that a language is not context-free. However, there are many languages that are not context-free but that cannot be proven so just with the Pumping Theorem. In this section we consider a more powerful technique that may be useful in those cases.

Recall that the Pumping Theorem for regular languages imposed the constraint that the pumpable region y had to fall within the first k characters of any "long" string w. We exploited that fact in many of our proofs. But notice that the Pumping Theorem for context-free languages imposes no similar constraint. The two pumpable regions, v and y must be reasonably close together, but, as a group, they can fall anywhere in w. Sometimes there is a region that is pumpable, even though other regions aren't, and this can happen even in the case of long strings drawn from languages that are not context-free.

EXAMPLE 13.11 Sometimes Pumping Isn't Strong Enough

Let $L = \{a^i b^i c^j : i, j \geq 0, i \neq j\}$. We could attempt to use the context-free Pumping Theorem to show that L is not context-free. Let $w = a^k b^k c^{k+k!}$. (The reason for this choice will be clear soon.) Divide w into three regions, the a's, the b's, and the c's, which we'll call regions 1, 2, and 3, respectively. If either v or y contains two or more distinct symbols, then set q to 2. The resulting string will have letters out of order and thus not be in L. We consider the remaining possibilities:

- $(1, 1), (2, 2), (1, 3), (2, 3)$: Set q to 2. The number of a's will no longer equal the number of b's, so the resulting string is not in L.
- $(1, 2)$: If $|v| \neq |y|$ then set q to 2. The number of a's will no longer equal the number of b's, so the resulting string is not in L. If $|v| = |y|$ then set q to $(k!/|v|) + 1$. Note that $(k!/|v|)$ must be an integer since $|v| \leq k$. The string that results from pumping is $a^X b^X c^{k+k!}$, where $X = k + (q - 1) \cdot |v| = k + (k!/|v|) \cdot |v| = k + k!$. So the number of a's and of b's equals the number of c's. This string is not in L. So far, the proof is going well. But now we must consider:
- $(3, 3)$: Pumping in will result in even more c's than a's and b's, so it will produce a string that is still in L. And, while pumping out can reduce the number of c's, it can't reduce it all the way down to k because $|vxy| \leq k$. So the maximum number of c's that can be pumped out is k, which would result in a string with $k!$ c's. But, as long as $k \geq 3$, $k! > k$. So the resulting string is in L and we have failed to show that L is not context-free.

What we need is a way to prevent v and y from falling in the c region of w.

Ogden's Lemma is a generalization of the Pumping Theorem. It lets us mark some number of symbols in our chosen string w as ***distinguished***. Then at least one of v and y must contain at least one distinguished symbol. So, for example, we could

complete the proof that we started in Example 13.11 if we could force at least one of v or y to contain at least one a.

THEOREM 13.16 Ogden's Lemma

Theorem: If L is a context-free language, then:

$\exists k \geq 1$ ($\forall$ strings $w \in L$, where $|w| \geq k$, if we mark at least k symbols of w as distinguished then:

$(\exists u, v, x, y, z\ (w = uvxyz,$

vy contains at least one distinguished symbol,

vxy contains at most k distinguished symbols, and

$\forall q \geq 0\ \ (uv^q xy^q z$ is in $L)))$.

Proof: The proof is analogous to the one we did for the context-free Pumping Theorem except that we consider only paths that generate the distinguished symbols. If L is context-free, then it is generated by some context-free grammar $G = (V, \Sigma, R, S)$ with n nonterminal symbols and branching factor b. Let k be b^{n+1}. Let w be any string in $L(G)$ such that $|w| \geq k$. A parse tree T for w might look like the one shown in Figure 13.6.

Suppose that we mark at least b^{n+1} symbols as distinguished. The distinguished symbols are marked with a ✓ (Ignore the fact that there aren't enough of them in the picture. Its only role is to make it easier to visualize the process.) Call the sequence of distinguished nodes the ***distinguished subsequence*** of w. In this example, that is bje. Note that the distinguished subsequence is not necessarily a substring. The characters in it need not be contiguous. The length of the distinguished subsequence is at least b^{n+1}. We can now mark the nonleaf nodes that branched in a way that enabled the distinguished subsequence to grow to at least length b^{n+1}. Mark every nonleaf node that has at least two daughters that contain a distinguished leaf. In this example, we mark X_2, and X_1, as indicated by the symbol ♦. It is straightforward to prove by induction that T must contain at least one path that contains at least $n + 1$ marked

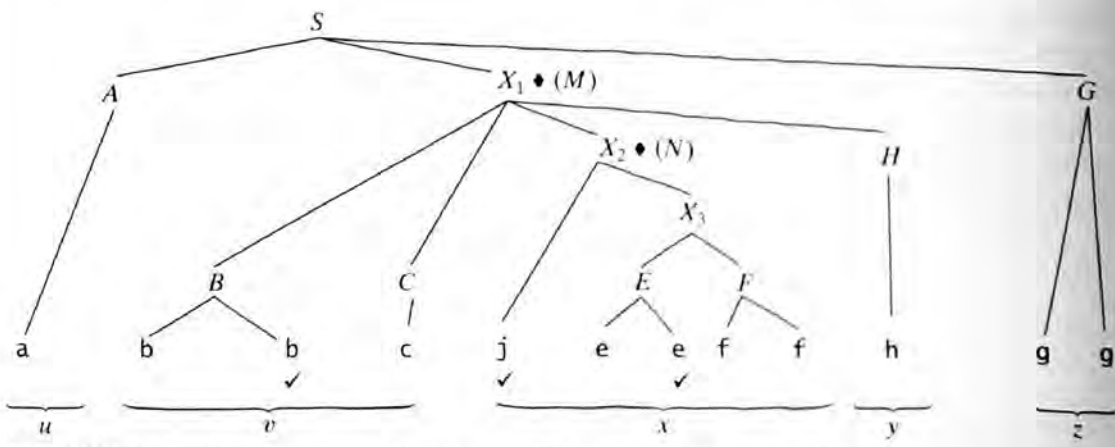


FIGURE 13.6 A parse tree with some symbols marked as distinguished.

nonleaf nodes since its yield contains b^{n+1} distinguished symbols. Choose one such path such that there is no longer one. That path must contain at least two nodes labeled with the same nonterminal symbol. Choose the two nodes that are labeled with the bottom-most pair of repeated marked nonterminals. Call the lower one N and the higher one M. In the example, M is X_1 and N is X_2. As shown in the diagram, divide w into $uvxyz$, such that x is the yield of N and vxy is the yield of M. Now observe that:

- vy contains at least one distinguished symbol because the root of the subtree with yield vxy has at least two daughters that contain distinguished symbols. One of them may be in the subtree whose yield is x, but that leaves at least one that must be in either v or y. There may be distinguished symbols in both, although, as in our example T, that is not necessary.
- vxy contains at most k (b^{n+1}) distinguished symbols because there are at most $n + 1$ marked internal nodes on a longest path in the subtree that dominates it. Only marked internal nodes create branches that lead to more than one distinguished symbol, and no internal node can create more than b branches.
- $\forall q \geq 0$ ($uv^q xy^q z$ is in L), by the same argument that we used in the proof of the context-free Pumping Theorem.

Notice that the context-free Pumping Theorem describes the special case in which all symbols of the string w are marked.

Ogden's Lemma is the tool that we need to complete the proof that we started in Example 13.11.

EXAMPLE 13.12 Ogden's Lemma May Work When Pumping Doesn't

Now we can use Ogden's Lemma to complete the proof that $L = \{a^i b^i c^j : i, j \geq 0, i \neq j\}$ is not context-free. Let $w = a^k b^k c^{k+k!}$. Mark all the a's in w as distinguished. If either v or y contains two or more distinct symbols, then set q to 2. The resulting string will have letters out of order and thus not be in L. We consider the remaining possibilities:

- (1, 1), (1, 3): Set q to 2. The number of a's will no longer equal the number of b's, so the resulting string is not in L.
- (1, 2): If $|v| \neq |y|$ then set q to 2. The number of a's will no longer equal the number of b's, so the resulting string is not in L. If $|v| = |y|$ then set q to $(k!/|v|) + 1$. Note that $(k!/|v|)$ must be an integer since $|v| \leq k$. The string that results from pumping is $a^{k+(q-1)\cdot|v|} b^{k+(q-1)\cdot|v|} c^{k+k!} = a^{k+(k!/|v|)\cdot|v|} b^{k+(k!/|v|)\cdot|v|} c^{k+k!} = a^{k+k!} b^{k+k!} c^{k+k!}$. So the number of a's and of b's equals the number of c's. This string is not in L.
- (2, 2), (2, 3), (3, 3) fail to satisfy the requirement that at least one symbol in vy be marked as distinguished.

There is no way to divide w into vxy such that all the conditions of Ogden's Lemma are met. So L is not context-free.

13.7 Parikh's Theorem ●

Suppose that we consider a language L not from the point of view of the exact strings it contains but instead by simply counting, for each string w in L, how many instances of each character in Σ w contains. So, from this perspective, the strings aaabbba and abababa are the same. If Σ is {a, b}, then both strings can be described with the pair (4, 3) since they contain 4 a's and 3 b's. We can build such descriptions by defining a family of functions ψ_Σ, with domain Σ^* and range $\{(i_1, i_2, \ldots i_k)\}$, where $k = |\Sigma|$:

$\psi_\Sigma(w) = (i_1, i_2, \ldots i_k)$ where, for all j, i_j = the number of occurrences in w of the j^{th} element of Σ.

So, if $\Sigma = \{a, b, c, d\}$, then $\psi_\Sigma(aabbbbddd) = (2, 4, 0, 3)$.

Now consider some language L, which is a set of strings over some alphabet Σ. Instead of considering L as a set of strings, we can consider it as the set of vectors that are produced by applying ψ_Σ to the strings it contains. To do this, we define another family of functions Ψ_Σ, with domain $\mathcal{P}(\Sigma^*)$ and range $\mathcal{P}\{(i_1, i_2, \ldots i_k)\}$:

$$\Psi_\Sigma(L) = \{(i_1, i_2, \ldots i_k) : \exists w \in L (\psi_\Sigma(w) = (i_1, i_2, \ldots i_k))\}.$$

If Σ is fixed, then there is a single function ψ and a single function Ψ. In that case, we will omit Σ and refer to the functions just as ψ and Ψ.

We will say that two languages L_1 and L_2, over the alphabet Σ^*, are ***letter-equivalent*** iff $\Psi_\Sigma(L_1) = \Psi_\Sigma(L_2)$. In other words, L_1 and L_2 contain the same strings if we disregard the order in which the symbols occur in the strings.

EXAMPLE 13.13 Letter Equivalence

Let $\Sigma = \{a, b\}$. Then, for example, $\psi(a) = (1, 0)$. $\psi(b) = (0, 1)$. $\psi(ab) = (1, 1)$. $\psi(aaabbbb) = (3, 4)$.

Now consider Ψ:

- Let $L_1 = A^nB^n = \{a^nb^n : n \geq 0\}$. Then $\Psi(L_1) = \{(i, i) : 0 \leq i\}$.
- Let $L_2 = (ab)^*$. Then $\Psi(L_2) = \{(i, i) : 0 \leq i\}$.
- Let $L_3 = \{a^nb^na^n : n \geq 0\}$. Then $\Psi(L_3) = \{(2i, i) : 0 \leq i\}$.
- Let $L_4 = \{a^{2n}b^n : n \geq 0\}$. Then $\Psi(L_4) = \{(2i, i) : 0 \leq i\}$.
- Let $L_5 = (aba)^*$. Then $\Psi(L_5) = \{(2i, i) : 0 \leq i\}$.

L_1 and L_2 are letter-equivalent. So are L_3, L_4 and L_5.

Just looking at the five languages we considered in Example 13.13, we can observe that it is possible for two languages with different formal properties (for example a regular language and a context-free but not regular one) to be letter equivalent to

each other. L_3 is not context-free. L_4 is context-free but not regular. is regular. But the three of them are letter equivalent to each other.

Parikh's Theorem, which we are about to state formally and then prove, tells us that that example is far from unique. In fact, given any context-free language L, there exists some regular language L' such that L and L' are letter-equivalent to each other. So A^nB^n is letter equivalent to $(ab)^*$. The language $\{a^{2n}b^n : n \geq 0\}$ is letter equivalent to $(aba)^*$ and to $(aab)^*$. And PalEven $= \{ww^R : w \in \{a, b\}^*\}$ is letter equivalent to $(aa \cup bb)^*$ since $\Psi(\text{PalEven}) = \Psi((aa \cup bb)^*) = \{(2i, 2j) : 0 \leq i \wedge 0 \leq j\}$. The proof of Parikh's Theorem is similar to the proofs we have already given for the Context-free Pumping Theorem and for Ogden's Lemma. It is based on the fact that, if L is context-free, then all the strings in L can be formed by starting with one of a finite set of "short" strings in L and then pumping in some finite number of strings (v, y pairs), all of which are chosen from a finite library of possible values for v and y.

An interesting application of Parikh's theorem is in the proof of a corollary that tells us that every context-free language over a single character alphabet must also be regular. We will add that corollary to our kit of tools for proving that a language is not context-free (by showing that, if it were, then it would also be regular but we know that it isn't).

Notice, by the way, that while we are about to prove that if L is context-free then it is letter-equivalent to some regular language, the converse of that claim is false. A language can be letter-equivalent to some regular language and not be context-free. We prove this by considering two of the languages from Example 13.13: $L_3 = \{a^nb^na^n : n \geq 0\}$ is not context-free, but it is letter-equivalent to $L_5 = (aba)^*$, which is regular.

THEOREM 13.17 Parikh's Theorem

Theorem: Every context-free language is letter-equivalent to some regular language.

Proof: The proof follows an argument similar to the one we used to prove the context-free Pumping Theorem. It is given in D.3.

An algebraic approach to thinking about what ψ and Ψ are doing is the following: We can describe the standard way of looking at strings as starting with a set S of primitive strings (ε and all the one-character strings drawn from Σ) and the single operation of concatenation, which is associative and has ε as an identity. Σ^* is then the closure of S under concatenation. ψ_Σ maps elements of Σ^* to elements of $\{(i_1, i_2, \ldots i_k)\}$, on which is defined the operation of pair wise addition, which is associative and has $(0, 0, \ldots 0)$ as an identity. But addition is also commutative, while concatenation is not. So, while, if we concatenate strings, it matters what order we do it in, if we consider the images of strings under ψ, the order in which we combine them doesn't matter. Parikh's theorem can be described as a special case of more general properties of commutative systems.

When Σ contains just a single character, the order of the characters in a string is irrelevant. So we have the following result:

THEOREM 13.18 Every CFL Over A Single-Character Alphabet is Regular

Theorem: Any context-free language over a single-character alphabet is regular.

Proof: By Parikh's Theorem, if L is context-free then L is letter-equivalent to some regular language L'. Since the order of characters has no effect on strings when all characters are the same, $L = L'$. Since L' is regular, so is L.

EXAMPLE 13.14 A^nA^n is Regular

Let $\Sigma = \{a, b\}$ and consider $L = A^nB^n = \{a^nb^n : n \geq 0\}$. A^nB^n is context-free but not regular.

$$\begin{aligned}\text{Now let: } \Sigma &= \{a\} \text{ and } L' = \{a^na^n, n \geq 0\}.\\ &= \{a^{2n} : n \geq 0\}.\\ &= \{w \in \{a\}^* : |w| \text{ is even}\}.\ L' \text{ is regular.}\end{aligned}$$

EXAMPLE 13.15 PalEven is Regular if $\Sigma = \{a\}$

Let $\Sigma = \{a, b\}$ and consider $L = \text{PalEven} = \{ww^R : w \in \{a, b\}^*\}$. PalEven is context-free but not regular.

$$\begin{aligned}\text{Now let: } \Sigma &= \{a\} \text{ and } L' = \{ww^R : w \in \{a\}^*\}\\ &= \{w \in \{a\}^* : |w| \text{ is even}\}.\ L' \text{ is regular.}\end{aligned}$$

When we are considering only a single letter alphabet, we can use Theorem 13.18 to show that a language that we already know not to be regular cannot be context-free either.

EXAMPLE 13.16 The Prime Number of a's Language is Not Context-Free

Consider again $\text{Prime}_a = \{a^n : n \text{ is prime}\}$. Prime_a is not context-free. If it were, then, by Theorem 13.18, it would also be regular. But we showed in Example 8.13 that it is not regular. So it is not context-free either.

13.8 Functions on Context-Free Languages ●

In Section 13.4, we saw that the context-free languages are closed under some important functions, including concatenation, union, and Kleene star. But their closure properties are substantially weaker than are the closure properties of the regular languages. In this section, we consider some other functions that can be applied to languages and we ask whether the context-free languages are closed under them. The proof strategies

we will use are the same as the ones we used for the regular languages and for the results we have already obtained for the context-free languages:

- To show that the context-free languages are closed under some function f, we will show an algorithm that constructs, given any context-free language L, either a grammar or a PDA that describes $f(L)$.
- To show that the context-free languages are not closed under some function f, we will exhibit a counterexample, i.e., a language L where L is context-free but $f(L)$ is not.

EXAMPLE 13.17 *Firstchars*

Consider again the function *firstchars* $(L) = \{w : \exists y \in L\ (y = cx \wedge c \in \Sigma_L \wedge x \in \Sigma_L^* \wedge w \in c^*)\}$. The context-free languages are closed under *firstchars*(L). In fact, if L is context-free then *firstchars*(L) is regular. We know that this must be true by an argument similar to the one we used in Example 8.20 to show that the regular languages are closed under *firstchars*. There must be some finite set of characters $\{c_1, c_2, \ldots, c_n\}$ that can begin strings in L (since Σ_L is finite). So there exists some regular expression of the following form that describes *firstchars*(L):

$$c_1^* \cup c_2^* \cup \cdots \cup c_n^*.$$

We can also show a constructive proof that *firstchars*(L) is context-free if L is. If L is a context-free language, then there is some context-free grammar $G = (V, \Sigma, R, S)$ that generates it. We construct a context-free grammar $G' = (V', \Sigma', R', S')$ that generates *firstchars*(L):

1. Convert G to Greibach normal form using the procedure *converttoGreibach*, defined in D.1.
2. Remove from G all unreachable nonterminals and all rules that mention them.
3. Remove from G all unproductive nonterminals and all rules that mention them.
4. Initialize V' to $\{S'\}$, Σ' to $\{\}$, and R' to $\{\}$.
5. For each remaining rule in G of the form $S \rightarrow c\ \gamma$ do:
   5.1. Add to R' the rules $S' \rightarrow C_c$, $C_c \rightarrow c\ C_c$ and $C_c \rightarrow \varepsilon$.
   5.2. Add to Σ' the symbol c.
   5.3. Add to V' the symbol C_c.
6. Return G'.

The idea behind this construction is that, if G is in Greibach normal form, then, each time a rule is applied, the next terminal symbol is generated. So, if we look at G's start symbol S and ask what terminals any of its rules can generate, we'll know exactly what terminals strings in $L(G)$ can start with.

EXAMPLE 13.18 *Maxstring*

Consider again the function $maxstring(L) = \{w : w \in L \text{ and } \forall z \in \Sigma^*(z \neq \varepsilon \rightarrow wz \notin L)\}$. The context-free languages are not closed under $maxstring(L)$. The proof is by counterexample. Consider the language $L = \{a^i b^j c^k : k \leq i \text{ or } k \leq j\}$. L is context-free but $maxstring(L)$ is not. We leave the proof of this as an exercise.

Exercises

1. For each of the following languages L, state whether L is regular, context-free but not regular, or not context-free and prove your answer.
   a. $\{xy : x, y \in \{a, b\}^* \text{ and } |x| = |y|\}$.
   b. $\{(ab)^n a^n b^n : n > 0\}$.
   c. $\{x\#y : x, y \in \{0, 1\}^* \text{ and } x \neq y\}$.
   d. $\{a^i b^n : i, n > 0 \text{ and } i = n \text{ or } i = 2n\}$.
   e. $\{wx : |w| = 2 \cdot |x| \text{ and } w \in a^+b^+ \text{ and } x \in a^+b^+\}$.
   f. $\{a^n b^m c^k : n, m, k \geq 0 \text{ and } m \leq min(n, k)\}$.
   g. $\{xyx^R : x \in \{0, 1\}^+ \text{ and } y \in \{0, 1\}^*\}$.
   h. $\{xwx^R : x, w \in \{a, b\}^+ \text{ and } |x| = |w|\}$.
   i. $\{ww^R w : w \in \{a, b\}^*\}$.
   j. $\{wxw : |w| = 2 \cdot |x| \text{ and } w \in \{a, b\}^* \text{ and } x \in \{c\}^*\}$.
   k. $\{a^i : i \geq 0\}\{b^i : i \geq 0\}\{a^i : i \geq 0\}$.
   l. $\{x \in \{a, b\}^* : |x| \text{ is even and the first half of } x \text{ has one more a than does the second half}\}$.
   m. $\{w \in \{a, b\}^* : \#_a(w) = \#_b(w) \text{ and } w \text{ does not contain either the substring aaa or abab}\}$.
   n. $\{a^n b^{2n} c^m : n, m \geq 0\} \cap \{a^n b^m c^{2m} : n, m \geq 0\}$.
   o. $\{x\,c\,y : x, y \in \{0, 1\}^* \text{ and } y \text{ is a prefix of } x\}$.
   p. $\{w : w = uu^R \text{ or } w = ua^n : n = |u|, u \in \{a, b\}^*\}$.
   q. $L(G)$, where $G = S \rightarrow aSa$
      $S \rightarrow SS$
      $S \rightarrow \varepsilon$
   r. $\{w \in (A\text{-}Z, a\text{-}z, ., \text{blank})^+ : \text{there exists at least one duplicated, capitalized word in } w\}$. For example, the string, `The history of China can be viewed from the perspective of an outsider or of someone living in China.` $\in L$.
   s. $\neg L_0$, where $L_0 = \{ww : w \in \{a, b\}^*\}$.
   t. L^*, where $L = \{0^*1^i0^*1^i0^* : i \geq 0\}$.
   u. $\neg A^n B^n$.
   v. $\{ba^i b : j = n^2 \text{ for some } n \geq 0\}$. For example, baaaab $\in L$.
   w. $\{w \in \{a, b, c, d\}^* : \#_b(w) \geq \#_c(w) \geq \#_d(w) \geq 0\}$.

2. Let $L = \{w \in \{a, b\}^* :$ the first, middle, and last characters of w are identical$\}$.
   a. Show a context-free grammar for L.
   b. Show a natural PDA that accepts L.
   c. Prove that L is not regular.
3. Let $L = \{a^n b^m c^n d^m : n, m \geq 1\}$. L is interesting because of its similarity to a useful fragment of a typical programming language in which one must declare procedures before they can be invoked. The procedure declarations include a list of the formal parameters. So now imagine that the characters in a^n correspond to the formal parameter list in the declaration of procedure 1. The characters in b^m correspond to the formal parameter list in the declaration of procedure 2. Then the characters in c^n and d^m correspond to the parameter lists in an invocation of procedure 1 and procedure 2 respectively, with the requirement that the number of parameters in the invocations match the number of parameters in the declarations. Show that L is not context-free.
4. Without using the Pumping Theorem, prove that $L = \{w \in \{a, b, c\}^* : \#_a(w) = \#_b(w) = \#_c(w)$ and $\#_a(w) > 50\}$ is not context-free.
5. Give an example of a context-free language L ($\neq \Sigma^*$) that contains a subset L_1 that is not context-free. Prove that L is context free. Describe L_1 and prove that it is not context-free.
6. Let $L_1 = L_2 \cap L_3$.
   a. Show values for L_1, L_2, and L_3, such that L_1 is context-free but neither L_2 nor L_3 is.
   b. Show values for L_1, L_2, and L_3, such that L_2 is context-free but neither L_1 nor L_3 is.
7. Give an example of a context-free language L, other than one of the ones in the book, where $\neg L$ is not context-free.
8. Theorem 13.7 tells us that the context-free languages are closed under intersection with the regular languages. Prove that the context-free languages are also closed under union with the regular languages.
9. Complete the proof that the context-free languages are not closed under *maxstring* by showing that $L = \{a^i b^j c^k : k \leq i$ or $k \leq j\}$ is context-free but *maxstring*(L) is not context-free.
10. Use the Pumping Theorem to complete the proof, started in L.3.3, that English is not context-free if we make the assumption that subjects and verbs must match in a "respectively" construction.
11. In N.1.2, we give an example of a simple musical structure that cannot be described with a context-free grammar. Describe another one, based on some musical genre with which you are familiar. Define a sublanguage that captures exactly that phenomenon. In other words, ignore everything else about the music you are considering and describe a set of strings that meets the one requirement you are studying. Prove that your language is not context-free.
12. Define the leftmost maximal P subsequence m of a string w as follows:
    - P must be a nonempty set of characters.
    - A string S is a P subsequence of w iff S is a substring of w and S is composed entirely of characters in P. For example 1, 0, 10, 01, 11, 011, 101, 111, 1111, and 1011 are $\{0, 1\}$ subsequences of 2312101121111.

- Let S be the set of all P subsequences of w such that, for each element t of S, there is no P subsequence of w longer than t. In the example above, $S = \{1111, 1011\}$.
- Then m is the leftmost (within w) element of S. In the example above, $m = 1011$.

a. Let $L = \{w \in \{0\text{-}9\}^* :$ if y is the leftmost maximal $\{0, 1\}$ subsequence of w then $|y|$ is even$\}$. Is L regular (but not context free), context free or neither? Prove your answer.

b. Let $L = \{w \in \{a, b, c\}^* :$ the leftmost maximal $\{a, b\}$ subsequence of w starts with a$\}$. Is L regular (but not context free), context free or neither? Prove your answer.

13. Are the context-free languages closed under each of the following functions? Prove your answer.

a. $chop(L) = \{w : \exists x \in L\ (x = x_1 c x_2 \wedge x_1 \in \Sigma_L{}^* \wedge x_2 \in \Sigma_L{}^* \wedge c \in \Sigma_L \wedge |x_1| = |x_2| \wedge w = x_1 x_2)\}$

b. $mix(L) = \{w : \exists x, y, z: (x \in L, x = yz, |y| = |z|, w = yz^R)\}$

c. $pref(L) = \{w : \exists x \in \Sigma^*(wx \in L)\}$

d. $middle(L) = \{x : \exists y, z \in \Sigma^*(yxz \in L)\}$

e. Letter substitution

f. $shuffle(L) = \{w : \exists x \in L\ (w$ is some permutation of $x)\}$

g. $copyreverse\ (L) = \{w : \exists x \in L\ (w = xx^R)\}$

14. Let $alt\ (L) = \{x : \exists y, n\ (y \in L, |y| = n, n > 0, y = a_1 \cdots a_n, \forall i \le n\ (a_i \in \Sigma)$, and $x = a_1 a_3 a_5 \cdots a_k$, where $k =$ (if n is even then $n - 1$ else n))$\}$.

a. Consider $L =$ $a^n b^n$. Clearly describe $L_1 = alt(L)$.

b. Are the context free languages closed under the function *alt*? Prove your answer.

15. Let $L_1 = \{a^n b^m : n \ge m\}$. Let $R_1 = \{(a \cup b)^* :$ there is an odd number of a's and an even number of b's$\}$. Use the construction that is described in the proof of Theorem 13.7 to build a PDA that accepts $L_1 \cap R_1$.

16. Let T be a set of languages defined as follows:

$$T = \{L : L \text{ is a context-free language over the alphabet } \{a, b, c\} \text{ and, if } x \in L, \text{ then } |x| \equiv_3 0\}.$$

Let P be the following function on languages:

$$P(L) = \{w : \exists x \in \{a, b, c\} \text{ and } \exists y \in L \text{ and } y = xw\}.$$

Is the set T closed under P? Prove your answer.

17. Show that the following languages are deterministic context-free:

a. $\{w : w \in \{a, b\}^*$ and each prefix of w has at least as many a's as b's$\}$

b. $\{a^n b^n : n \ge 0\} \cup \{a^n c^n : n \ge 0\}$

18. Show that $L = \{a^n b^n : n \ge 0\} \cup \{a^n b^{2n} : n \ge 0\}$ is not deterministic context-free.

19. Are the deterministic context-free languages closed under reverse? Prove your answer.

20. Prove that each of the following languages is not context-free. (Hint: Use Ogden's Lemma.)

a. $\{a^i b^j c^k : i \geq 0, j \geq 0, k \geq 0, \text{and } i \neq j \neq k\}$

b. $\{a^i b^j c^k d^n : i \geq 0, j \geq 0, k \geq 0, n \geq 0, \text{and } (i = 0 \text{ or } j = k = n)\}$

21. Let $\Psi(L)$ be as defined in Section 13.7, in our discussion of Parikh's Theorem. For each of the following languages L, first state what $\Psi(L)$ is. Then give a regular language that is letter-equivalent to L.

a. Bal $= \{w \in \{), (\}^* : \text{the parentheses are balanced}\}$

b. Pal $= \{w \in \{a, b\}^* : w \text{ is a palindrome}\}$

c. $\{x^R \# y : x, y \in \{0, 1\}^* \text{ and } x \text{ is a substring of } y\}$

22. For each of the following claims, state whether it is *True* or *False*. Prove your answer.

a. If L_1 and L_2 are two context-free languages, $L_1 - L_2$ must also be context-free.

b. If L_1 and L_2 are two context-free languages and $L_1 = L_2 L_3$, then L_3 must also be context-free.

c. If L is context free and R is regular, $R - L$ must be context-free.

d. If L_1 and L_2 are context-free languages and $L_1 \subseteq L \subseteq L_2$, then L must be context-free.

e. If L_1 is a context-free language and $L_2 \subseteq L_1$, then L_2 must be context-free.

f. If L_1 is a context-free language and $L_2 \subseteq L_1$, it is possible that L_2 is regular.

g. A context-free grammar in Chomsky normal form is always unambiguous.

CHAPTER 14

Algorithms and Decision Procedures for Context-Free Languages

Many questions that we could answer when asked about regular languages are unanswerable for context-free ones. But a few important questions can be answered and we have already presented a useful collection of algorithms that can operate on context-free grammars and PDAs. We'll present a few more here.

14.1 The Decidable Questions

Fortunately, the most important questions (i.e., the ones that must be answerable if context-free grammars are to be of any practical use) are decidable.

14.1.1 Membership

We begin with the most fundamental question, "Given a language L and a string w, is w in L?" Fortunately this question can be answered for every context-free language. By Theorem 12.1, for every context-free language L, there exists a PDA M such that M accepts L. But we must be careful. As we showed in Section 12.4, PDAs are not guaranteed to halt. So the mere existence of a PDA that accepts L does not guarantee the existence of a procedure that decides it (i.e., always halts and says yes or no appropriately).

It turns out that there are two alternative approaches to solving this problem, both of which work:

- Use a grammar: Using facts about every derivation that is produced by a grammar in Chomsky normal form, we can construct an algorithm that explores a finite number of derivation paths and finds one that derives a particular string w iff such a path exists.

- Use a PDA: While not all PDAs halt, it is possible, for any context-free language L, to craft a PDA M that is guaranteed to halt on all inputs and that accepts all strings in L and rejects all strings that are not in L.

Using a Grammar to Decide

We begin by considering the first alternative. We show a straightforward algorithm for deciding whether a string w is in a language L:

decideCFLusingGrammar(L: CFL, w: string) =

1. If L is specified as a PDA, use *PDAtoCFG*, presented in the proof of Theorem 12.2, to construct a grammar G such that $L(G) = L(M)$.
2. If L is specified as a grammar G, simply use G.
3. If $w = \varepsilon$ then if S_G is nullable (as defined in the description of *removeEps* in Section 11.7.4) then accept, otherwise reject.
4. If $w \neq \varepsilon$ then:
   - 4.1. From G, construct G' such that $L(G') = L(G) - \{\varepsilon\}$ and G' is in Chomsky normal form.
   - 4.2. If G derives w, it does so in $2 \cdot |w| - 1$ steps. Try all derivations in G of that number of steps. If one of them derives w, accept. Otherwise reject.

The running time of *decideCFLusingGrammar* can be analyzed as follows: We assume that the time required to build G' is constant, since it does not depend on w. Let $n = |w|$. Let g be the search-branching factor of G', defined to be the maximum number of rules that share a left-hand side. Then the number of derivations of length $2n - 1$ is bounded by g^{2n-1}, and it takes at most $2n - 1$ steps to check each one. So the worst-case running time of *decideCFLusingGrammar* is $O(n2^n)$. In Section 15.3.1, we will present techniques that are substantially more efficient. We will describe the CKY algorithm, which, given a grammar G in Chomsky normal form, decides the membership question for G in time that is $O(n^3)$. We will then describe an algorithm that can decide the question in time that is linear in n if the grammar that is provided meets certain requirements.

THEOREM 14.1 Decidability of Context-Free Languages

Theorem: Given a context-free language L (represented as either a context-free grammar or a PDA) and a string w, there exists a decision procedure that answers the question, "Is $w \in L$?"

Proof: The following algorithm, *decideCFL*, uses *decideCFLusingGrammar* to answer the question:

decideCFL(L: CFL, w: string) =

1. If *decideCFLusingGrammar*(L, w) accepts, return *True* else return *False*.

Using a PDA to Decide ●

It is also possible to solve the membership problem using PDAs. We take a two-step approach. We first show that, for every context-free language L, it is possible to build a PDA that accepts $L - \{\varepsilon\}$ and that has no ε-transitions. Then we show that every PDA with no ε-transitions is guaranteed to halt.

THEOREM 14.2 Elimination of ε-Transitions

Theorem: Given any context-free grammar $G = (V, \Sigma, R, S)$, there exists a PDA M such that $L(M) = L(G) - \{\varepsilon\}$ and M contains no transitions of the form $((q_1, \varepsilon, \alpha), (q_2, \beta))$. In other words, every transition reads exactly one input character.

Proof: The proof is by a construction that begins by converting G to Greibach normal form. Recall that, in any grammar in Greibach normal form, all rules are of the form $X \rightarrow a\,A$, where $a \in \Sigma$ and $A \in (V - \Sigma)^*$. Now consider again the algorithm *cfgtoPDAtopdown*, which builds, from any context-free grammar G, a PDA M that, on input w, simulates G deriving w, starting from S. $M = (\{p, q\}, \Sigma, V, \Delta, p, \{q\})$, where Δ contains:

1. The start-up transition $((p, \varepsilon, \varepsilon), (q, S))$, which pushes the start symbol onto the stack and goes to state q.
2. For each rule $X \rightarrow s_1 s_2 \ldots s_n$ in R, the transition $((q, \varepsilon, X), (q, s_1 s_2 \ldots s_n))$, which replaces X by $s_1 s_2 \ldots s_n$. If $n = 0$ (i.e., the right-hand side of the rule is ε), then the transition $((q, \varepsilon, X), (q, \varepsilon))$.
3. For each character $c \in \Sigma$, the transition $((q, c, c), (q, \varepsilon))$, which compares an expected character from the stack against the next input character and continues if they match.

The start-up transition, plus all the transitions generated in step 2, are ε-transitions. But now suppose that G is in Greibach normal form. If G contains the rule $X \rightarrow c s_2 \ldots s_n$ (where $c \in \Sigma$ and s_2 through s_n are elements of V-Σ), it is not necessary to push c onto the stack, only to pop it with a rule from step 3. Instead, we collapse the push and the pop into a single transition. So we create a transition that can be taken only if the next input character is c. In that case, the string $s_2 \ldots s_n$ is pushed onto the stack.

Now we need only find a way to get rid of the start-up transition, whose job is to push S onto the stack so that the derivation process can begin. Since G is in Greibach normal form, any rules with S on the left-hand side must have the form $S \rightarrow c s_2 \ldots s_n$. So instead of reading no input and just pushing S, M will skip pushing S and instead, if the first input character is c, read it and push the string $s_2 \ldots s_n$.

Since terminal symbols are no longer pushed onto the stack, we no longer need the transitions created in step 3 of the original algorithm.

So $M = (\{p, q\}, \Sigma, V, \Delta, p, \{q\})$, where Δ contains:

1. The start-up transitions: For each rule $S \rightarrow cs_2 \ldots s_n$, the transition $((p, c, \varepsilon), (q, s_2 \ldots s_n))$.
2. For each rule $X \rightarrow cs_2 \ldots s_n$ (where $c \in \Sigma$ and s_2 through s_n are elements of $V - \Sigma$), the transition $((q, c, X), (q, s_2 \ldots s_n))$.

The following algorithm builds the required PDA:

cfgtoPDAnoeps(*G*: context-free grammar) =

1. Convert G to Greibach normal form, producing G'.
2. From G' build the PDA M described above.

THEOREM 14.3 Halting Behavior of PDAs Without ε-Transitions

Theorem: Let M be a PDA that contains no transitions of the form $((q_1, \varepsilon, s_1), (q_2, s_2))$, i.e., no ε-transitions. Consider the operation of M on input $w \in \Sigma^*$. M must halt and either accept or reject w. Let $n = |w|$. We make three additional claims:

a. Each individual computation of M must halt within n steps.

b. The total number of computations pursued by M must be less than or equal to b^n, where b is the maximum number of competing transitions from any state in M.

c. The total number of steps that will be executed by all computations of M is bounded by nb^n.

Proof:

a. Since each computation of M must consume one character of w at each step and M will halt when it runs out of input, each computation must halt within n steps.

b. M may split into at most b branches at each step in a computation. The number of steps in a computation is less than or equal to n. So the total number of computations must be less than or equal to b^n.

c. Since the maximum number of computations is b^n and the maximum length of each is n, the maximum number of steps that can be executed before all computations of M halt is nb^n.

So a second way to answer the question, "Given a context-free language L and a string w, is w in L?" is to execute the following algorithm:

decideCFLusingPDA(L: CFL, w: string) =

1. If L is specified as a PDA, use *PDAtoCFG*, as presented in the proof of Theorem 12.2, to construct a grammar G such that $L(G) = L(M)$.
2. If L is specified as a grammar G, simply use G.

3. If $w = \varepsilon$ then if S_G is nullable (as defined in the description of *removeEps* in Section 11.7.4) then accept, otherwise reject.
4. If $w \neq \varepsilon$ then:
   - **4.1.** From G, construct G' such that $L(G') = L(G) - \{\varepsilon\}$ and G' is in Greibach normal form.
   - **4.2.** From G' construct, using *cfgtoPDAnoeps*, the algorithm described in the proof of Theorem 14.2, a PDA M' such that $L(M') = L(G')$ and M' has no ε-transitions.
   - **4.3.** By Theorem 14.3, all paths of M' are guaranteed to halt within a finite number of steps. So run M' on w. Accept if M' accepts and reject otherwise.

The running time of *decideCFLusingPDA* can be analyzed as follows: We will take as a constant the time required to build M', since that can be done once. It need not be repeated for each string that is to be analyzed. Given M', the time required to analyze a string w is then the time required to simulate all paths of M' on w. Let $n = |w|$. From Theorem 14.3, we know that the total number of steps that will be executed by all paths of M is bounded by nb^n, where b is the maximum number of competing transitions from any state in M'. But is that number of steps required? If one state has a large number of competing transitions but the others do not, then the average branching factor will be less than b, so fewer steps will be necessary. But if b is greater than 1, the number of steps still grows exponentially with n. The exact number of steps also depends on how the simulation is done. A straightforward depth-first search of the tree of possibilities will explore b^n steps, which is less than nb^n because it does not start each path over at the beginning. But it still requires time that is $\mathcal{O}(b^n)$. In Section 15.2.3, we present an alternative approach to top-down parsing that runs in time that is linear in n if the grammar that is provided meets certain requirements.

14.1.2 Emptiness and Finiteness

While many interesting questions are not decidable for context-free languages, two others, in addition to membership are: emptiness and finiteness.

THEOREM 14.4 Decidability of Emptiness and Finiteness

Theorem: Given a context-free language L, there exists a decision procedure that answers each of the following questions:

1. Given a context-free language L, is $L = \emptyset$?
2. Given a context-free language L, is L infinite?

Since we have proven that there exists a grammar that generates L iff there exists a PDA that accepts it, these questions will have the same answers whether we ask them about grammars or about PDAs.

Proof:

1. Let $G = (V, \Sigma, R, S)$ be a context-free grammar that generates L. $L(G) = \varnothing$ iff S is unproductive (i.e., not able to generate any terminal strings). The following algorithm exploits the procedure *removeunproductive*, defined in Section 11.4, to remove all unproductive nonterminals from G. It answers the question, "Given a context-free language L, is $L = \varnothing$?"

   decideCFLempty(G: context-free grammar) =

   1. Let G' = *removeunproductive* (G).
   2. If S is not present in G' then return *True* else return *False*.

2. Let $G = (V, \Sigma, R, S)$ be a context-free grammar that generates L. We use an argument similar to the one that we used to prove the context-free Pumping Theorem. Let n be the number of nonterminals in G. Let b be the branching factor of G. The longest string that G can generate without creating a parse tree with repeated nonterminals along some path is of length b^n. If G generates no strings of length greater than b^n, then $L(G)$ is finite. If G generates even one string w of length greater than b^n, then, by the same argument we used to prove the Pumping Theorem, it generates an infinite number of strings since $w = uvxyz$, $|vy| > 0$, and $\forall q \geq 0$ (uv^qxy^qz is in L). So we could try to test to see whether L is infinite by invoking *decideCFL*(L, w) on all strings in Σ^* of length greater than b^n. If it returns *True* for any such string, then L is infinite. If it returns *False* on all such strings, then L is finite.

   But, assuming Σ is not empty, there is an infinite number of such strings. Fortunately, it is necessary to try only a finite number of them. Suppose that G generates even one string of length greater than $b^{n+1} + b^n$. Let t be the shortest such string. By the Pumping Theorem, $t = uvxyz$, $|vy| > 0$, and uxz (the result of pumping vy out once) $\in L$. Note that $|uxz| < |t|$ since some nonempty vy was pumped out of t to create it. Since, by assumption, t is the shortest string in L of length greater than $b^{n+1} + b^n$, $|uxz|$ must be less than or equal to $b^{n+1} + b^n$. But the Pumping Theorem also tells us that $|vxy| \leq k$ (i.e., b^{n+1}), so no more than b^{n+1} strings could have been pumped out of t. Thus we have that $b^n < |uxz| \leq b^{n+1} + b^n$. So, if L contains any strings of length greater than b^n, it must contain at least one string of length less than or equal to $b^{n+1} + b^n$. We can now define *decideCFLinfinite* to answer the question, "Given a context-free language L, is L infinite?":

   decideCFLinfinite(G: context-free grammar) =

   1. Lexicographically enumerate all strings in Σ^* of length greater than b^n and less than or equal to $b^{n+1} + b^n$.
   2. If, for any such string w, *decideCFL*(L, w) returns *True* then return *True*. L is infinite.
   3. If, for all such strings w, *decideCFL*(L, w) returns *False* then return *False*. L is not infinite.

14.1.3 Equality of Deterministic Context-Free languages

THEOREM 14.5 Decidability of Equivalence for Deterministic Context-Free Languages

Theorem: Given two *deterministic* context-free languages L_1 and L_2, there exists a decision procedure to determine whether $L_1 = L_2$.

Proof: This claim was not proved until 1997 and the proof [Sénizergues 2001] is beyond the scope of this book, but see 💻.

14.2 The Undecidable Questions

Unfortunately, we will prove in Chapter 22 that there exists no decision procedure for many other questions that we might like to be able to ask about context-free languages, including:

- Given a context-free language L, is $L = \Sigma^*$?
- Given a context-free language L, is the complement of L context-free?
- Given a context-free language L, is L regular?
- Given two context-free languages L_1 and L_2, is $L_1 = L_2$? (Theorem 14.5 tells us that this question is decidable for the restricted case of two deterministic context-free languages. But it is undecidable in the more general case.)
- Given two context-free languages L_1 and L_2, is $L_1 \subseteq L_2$?
- Given two context-free languages L_1 and L_2, is $L_1 \cap L_2 = \emptyset$?
- Given a context-free language L, is L inherently ambiguous?
- Given a context-free grammar G, is G ambiguous?

14.3 Summary of Algorithms and Decision Procedures for Context-Free Languages

Although we have presented fewer algorithms and decision procedures for context-free languages than we did for regular languages, there are many important ones, which we summarize here:

- Algorithms that transform grammars:
  - *removeunproductive*(G: context-free grammar): Construct a grammar G' that contains no unproductive nonterminals and such that $L(G') = L(G)$.
  - *removeunreachable*(G: context-free grammar): Construct a grammar G' that contains no unreachable nonterminals and such that $L(G') = L(G)$.

- *removeEps*(G: context-free grammar): Construct a grammar G' that contains no rules of the form $X \rightarrow \varepsilon$ and such that $L(G') = L(G) - \{\varepsilon\}$.
- *atmostoneEps*(G: context-free grammar): Construct a grammar G' that contains no rules of the form $X \rightarrow \varepsilon$ except possibly $S^* \rightarrow \varepsilon$, in which case there are no rules whose right-hand side contains S^*, and such that $L(G') = L(G)$.
- *converttoChomsky*(G: context-free grammar): Construct a grammar G' in Chomsky normal form, where $L(G') = L(G) - \{\varepsilon\}$.
- *converttoGreibach*(G: context-free grammar): Construct a grammar G' in Greibach normal form, where $L(G') = L(G) - \{\varepsilon\}$.
- *removeUnits*(G: context-free grammar): Construct a grammar G' that contains no unit productions, where $L(G') = L(G)$.

- Algorithms that convert between context-free grammars and PDAs:
  - *cfgtoPDAtopdown*(G: context-free grammar): Construct a PDA M such that $L(M) = L(G)$ and M operates top-down to simulate a left-most derivation in G.
  - *cfgtoPDAbottomup*(G: context-free grammar): Construct a PDA M such that $L(M) = L(G)$ and M operates bottom up to simulate, backwards, a right-most derivation in G.
  - *cfgtoPDAnoeps*(G: context-free grammar): Construct a PDA M such that M contains no transitions of the form $((q_1, \varepsilon, s_1), (q_2, s_2))$ and $L(M) = L(G) - \{\varepsilon\}$.

- Algorithms that transform PDAs:
  - *convertPDAtorestricted*(M: PDA): Construct a PDA M' in restricted normal form where $L(M') = L(M)$.

- Algorithms that compute functions of languages defined as context-free grammars:
  - Given two grammars G_1 and G_2, construct a new grammar G_3 such that $L(G_3) = L(G_1) \cup L(G_2)$.
  - Given two grammars G_1 and G_2, construct a new grammar G_3 such that $L(G_3) = L(G_1)L(G_2)$.
  - Given a grammar G, construct a new grammar G' such that $L(G') = (L(G))^*$.
  - Given a grammar G, construct a new grammar G' such that $L(G') = (L(G))^R$.
  - Given a grammar G, construct a new grammar G' that accepts *letsub*($L(G)$), where *letsub* is a letter substitution function.

- Miscellaneous algorithms for PDAs:
  - *intersectPDAandFSM* (M_1: PDA, M_2: FSM): Construct a PDA M_3 such that $L(M_3) = L(M_1) \cap L(M_2)$.
  - *without$*($M$: PDA): If M accepts $L\$$, construct a PDA M' such that $L(M') = L$.
  - *complementdetPDA*(M: DPDA): If M accepts $L\$$, construct a PDA M' such that $L(M') = (\neg L)\$$.

- Decision procedures that answer questions about context-free languages:
  - *decideCFLusingPDA*(L: CFL, w : string): Decide whether w is in L.
  - *decideCFLusingGrammar*(L: CFL, w: string): Decide whether w is in L.
  - *decideCFL*(L: CFL, w: string): Decide whether w is in L.
  - *decideCFLempty*(G: context-free grammar): Decide whether $L(G) = \emptyset$.
  - *decideCFLinfinite*(G: context-free grammar): Decide whether $L(G)$ is infinite.

Exercises

1. Give a decision procedure to answer each of the following questions:

a. Given a regular expression α and a PDA M, is the language accepted by M a subset of the language generated by α?

b. Given a context-free grammar G and two strings s_1 and s_2, does G generate s_1s_2?

c. Given a context-free grammar G, does G generate at least three strings?

d. Given a context-free grammar G, does G generate any even length strings?

e. Given a regular grammar G, is $L(G)$ context-free?

9 Turing Machines and Linear Bounded Automata

In the early 1930s, mathematicians were trying to define effective computation. Alan Turing in 1936, Alanzo Church in 1933, S.C. Kleene in 1935, Schonfinkel in 1965 gave various models using the concept of Turing machines, λ-calculus, combinatory logic, post-systems and μ-recursive functions. It is interesting to note that these were formulated much before the electro-mechanical/electronic computers were devised. Although these formalisms, describing effective computations, are dissimilar, they turn to be equivalent.

Among these formalisms, the Turing's formulation is accepted as a model of algorithm or computation. The Church–Turing thesis states that any algorithmic procedure that can be carried out by human beings/computer can be carried out by a Turing machine. It has been universally accepted by computer scientists that the Turing machine provides an ideal theoretical model of a computer.

Turing machines are useful in several ways. As an automaton, the Turing machine is the most general model. It accepts type-0 languages. It can also be used for computing functions. It turns out to be a mathematical model of partial recursive functions. Turing machines are also used for determining the undecidability of certain languages and measuring the space and time complexity of problems. These are the topics of discussion in this chapter and some of the subsequent chapters.

For formalizing computability, Turing assumed that, while computing, a person writes symbols on a one-dimensional paper (instead of a two-dimensional paper as is usually done) which can be viewed as a tape divided into cells.

One scans the cells one at a time and usually performs one of the three simple operations, namely (i) writing a new symbol in the cell being currently

scanned, (ii) moving to the cell left of the present cell, and (iii) moving to the cell right of the present cell. With these observations in mind, Turing proposed his 'computing machine.'

9.1 TURING MACHINE MODEL

The Turing machine can be thought of as finite control connected to a R/W (read/write) head. It has one tape which is divided into a number of cells. The block diagram of the basic model for the Turing machine is given in Fig. 9.1.

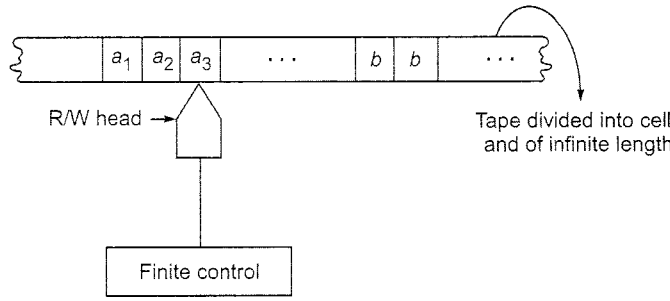


Fig. 9.1 Turing machine model.

Each cell can store only one symbol. The input to and the output from the finite state automaton are effected by the R/W head which can examine one cell at a time. In one move, the machine examines the present symbol under the R/W head on the tape and the present state of an automaton to determine

(i) a new symbol to be written on the tape in the cell under the R/W head,
(ii) a motion of the R/W head along the tape: either the head moves one cell left (L), or one cell right (R),
(iii) the next state of the automaton, and
(iv) whether to halt or not.

The above model can be rigorously defined as follows:

Definition 9.1 A Turing machine M is a 7-tuple, namely $(Q, \Sigma, \Gamma, \delta, q_0, b, F)$, where

1. Q is a finite nonempty set of states,
2. Γ is a finite nonempty set of tape symbols,
3. $b \in \Gamma$ is the blank,
4. Σ is a nonempty set of input symbols and is a subset of Γ and $b \notin \Sigma$,
5. δ is the transition function mapping (q, x) onto (q', y, D) where D denotes the direction of movement of R/W head; $D = L$ or R according as the movement is to the left or right.
6. $q_0 \in Q$ is the initial state, and
7. $F \subseteq Q$ is the set of final states.

Notes: (1) The acceptability of a string is decided by the reachability from the initial state to some final state. So the final states are also called the accepting states.

(2) δ may not be defined for some elements of $Q \times \Gamma$.

9.2 REPRESENTATION OF TURING MACHINES

We can describe a Turing machine employing (i) instantaneous descriptions using move-relations, (ii) transition table, and (iii) transition diagram (transition graph).

9.2.1 REPRESENTATION BY INSTANTANEOUS DESCRIPTIONS

'Snapshots' of a Turing machine in action can be used to describe a Turing machine. These give 'instantaneous descriptions' of a Turing machine. We have defined instantaneous descriptions of a pda in terms of the current state, the input string to be processed, and the topmost symbol of the pushdown store. But the input string to be processed is not sufficient to be defined as the ID of a Turing machine, for the R/W head can move to the left as well. So an ID of a Turing machine is defined in terms of the entire input string and the current state.

Definition 9.2 An ID of a Turing machine M is a string $\alpha\beta\gamma$, where β is the present state of M, the entire input string is split as $\alpha\gamma$, the first symbol of γ is the current symbol a under the R/W head and γ has all the subsequent symbols of the input string, and the string α is the substring of the input string formed by all the symbols to the left of a.

EXAMPLE 9.1

A snapshot of Turing machine is shown in Fig. 9.2. Obtain the instantaneous description.

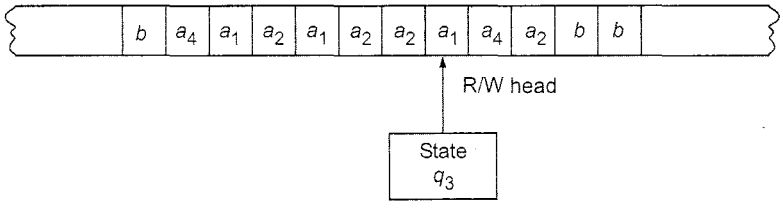


Fig. 9.2 A snapshot of Turing machine.

Solution

The present symbol under the R/W head is a_1. The present state is q_3. So a_1 is written to the right of q_3. The nonblank symbols to the left of a_1 form the string $a_4a_1a_2a_1a_2a_2$, which is written to the left of q_3. The sequence of nonblank symbols to the right of a_1 is a_4a_2. Thus the ID is as given in Fig. 9.3.

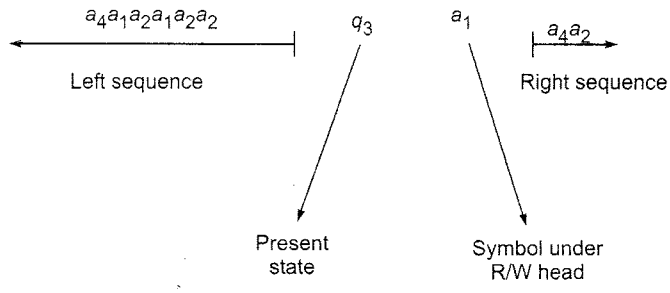


Fig. 9.3 Representation of ID.

Notes: (1) For constructing the ID, we simply insert the current state in the input string to the left of the symbol under the R/W head.

(2) We observe that the blank symbol may occur as part of the left or right substring.

Moves in a TM

As in the case of pushdown automata, $\delta(q, x)$ induces a change in ID of the Turing machine. We call this change in ID a move.

Suppose $\delta(q, x_i) = (p, y, L)$. The input string to be processed is $x_1x_2 \ldots x_n$, and the present symbol under the R/W head is x_i. So the ID before processing x_i is

$$x_1x_2 \ldots x_{i-1}qx_i \ldots x_n$$

After processing x_i, the resulting ID is

$$x_1 \ldots x_{i-2}\, px_{i-1}yx_{i+1} \ldots x_n$$

This change of ID is represented by

$$x_1x_2 \ldots x_{i-1}\, q\, x_i \ldots x_n \vdash x_i \ldots x_{i-2}\, px_{i-1}\, yx_{i+1} \ldots x_n$$

If $i = 1$, the resulting ID is $p\, y\, x_2x_3 \ldots x_n$.

If $\delta(q, x_i) = (p, y, R)$, then the change of ID is represented by

$$x_1x_2 \ldots x_{i-1}q\, x_i \ldots x_n \vdash x_1x_2 \ldots x_{i-1}\, y\, px_{i+1} \ldots x_n$$

If $i = n$, the resulting ID is $x_1x_2 \ldots x_{n-1}\, y\, p\, b$.

We can denote an ID by I_j for some j. $I_j \vdash I_k$ defines a relation among IDs. So the symbol $\vdash^*$ denotes the reflexive–transitive closure of the relation $\vdash$. In particular, $I_j \vdash^* I_j$. Also, if $I_1 \vdash^* I_n$, then we can split this as $I_1 \vdash I_2 \vdash I_3 \vdash \ldots \vdash I_n$ for some IDs, $I_2, \ldots, I_{n-1}$.

Note: The description of moves by IDs is very much useful to represent the processing of input strings.

9.2.2 Representation by Transition Table

We give the definition of δ in the form of a table called the transition table. If $\delta(q, a) = (\gamma, \alpha, \beta)$, we write $\alpha\beta\gamma$ under the α-column and in the q-row. So if

we get $\alpha\beta\gamma$ in the table, it means that α is written in the current cell, β gives the movement of the head (L or R) and γ denotes the new state into which the Turing machine enters.

Consider, for example, a Turing machine with five states $q_1, \ldots, q_5$, where q_1 is the initial state and q_5 is the (only) final state. The tape symbols are 0, 1 and b. The transition table given in Table 9.1 describes δ.

TABLE 9.1 Transition Table of a Turing Machine

Present state	*Tape symbol*		
	b	0	1
$\rightarrow q_1$	$1Lq_2$	$0Rq_1$	
q_2	bRq_3	$0Lq_2$	$1Lq_2$
q_3		bRq_4	bRq_5
q_4	$0Rq_5$	$0Rq_4$	$1Rq_4$
(q_5)	$0Lq_2$		

As in Chapter 3, the initial state is marked with $\rightarrow$ and the final state with $\circ$.

EXAMPLE 9.2

Consider the TM description given in Table 9.1. Draw the computation sequence of the input string 00.

Solution

We describe the computation sequence in terms of the contents of the tape and the current state. If the string in the tape is $a_1a_2 \ldots a_j a_{j+1} \ldots a_m$ and the TM in state q is to read a_{j+1}, then we write $a_1a_2 \ldots a_j q\, a_{j+1} \ldots a_m$.

For the input string $00b$, we get the following sequence:

$$q_100b \vdash 0q_10b \vdash 00q_1b \vdash 0q_201 \vdash q_2001$$
$$\vdash q_2b001 \vdash bq_3001 \vdash bbq_401 \vdash bb0q_41 \vdash bb0_1q_4b$$
$$\vdash bb010q_5 \vdash bb01q_200 \vdash bb0q_2100 \vdash bbq_20100$$
$$\vdash bq_2b0100 \vdash bbq_30100 \vdash bbbq_4100 \vdash bbb_1q_400$$
$$\vdash bbb10q_40 \vdash bbb100q_4b \vdash bbb1000q_5b$$
$$\vdash bbb100q_200 \vdash bbb10q_2000 \vdash bbb1q_20000$$
$$\vdash bbbq_210000 \vdash bbq_2b10000 \vdash bbbq_310000 \vdash bbbbq_50000$$

9.2.3 Representation by Transition Diagram

We can use the transition systems introduced in Chapter 3 to represent Turing machines. The states are represented by vertices. Directed edges are used to

represent transition of states. The labels are triples of the form (α, β, γ), where $\alpha, \beta \in \Gamma$ and $\gamma \in \{L, R\}$. When there is a directed edge from q_i to q_j with label (α, β, γ), it means that

$$\delta(q_i, \alpha) = (q_j, \beta, \gamma)$$

During the processing of an input string, suppose the Turing machine enters q_i and the R/W head scans the (present) symbol α. As a result, the symbol β is written in the cell under the R/W head. The R/W head moves to the left or to the right, depending on γ, and the new state is q_j.

Every edge in the transition system can be represented by a 5-tuple $(q_i, \alpha, \beta, \gamma, q_j)$. So each Turing machine can be described by the sequence of 5-tuples representing all the directed edges. The initial state is indicated by $\rightarrow$ and any final state is marked with $\circ$.

EXAMPLE 9.3

M is a Turing machine represented by the transition system in Fig. 9.4. Obtain the computation sequence of M for processing the input string 0011.

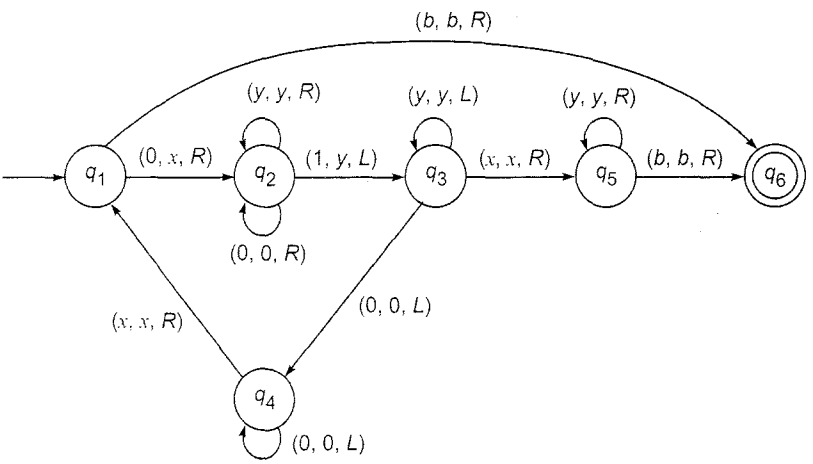


Fig. 9.4 Transition system for M.

Solution

The initial tape input is $b0011b$. Let us assume that M is in state q_1 and the R/W head scans 0 (the first 0). We can represent this as in Fig. 9.5. The figure can be represented by

$$\begin{array}{c} \downarrow \\ b0011b \\ q_1 \end{array}$$

From Fig. 9.4 we see that there is a directed edge from q_1 to q_2 with the label $(0, x, R)$. So the current symbol 0 is replaced by x and the head moves right. The new state is q_2. Thus, we get

$$\begin{array}{c} \downarrow \\ bx011b \\ q_2 \end{array}$$

The change brought about by processing the symbol 0 can be represented as

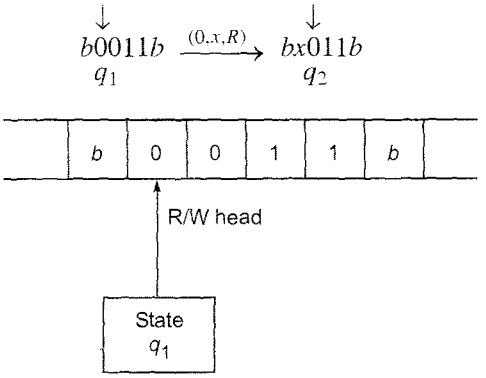


Fig. 9.5 TM processing 0011.

The entire computation sequence reads as follows:

$$\underset{q_1}{\downarrow}\, b0011b \xrightarrow{(0,x,R)} \underset{q_2}{\downarrow}\, bx011b \xrightarrow{(0,0,R)} \underset{q_2}{\downarrow}\, bx011b$$

$$\xrightarrow{(1,y,L)} \underset{q_3}{\downarrow}\, bx0y1b \xrightarrow{(0,0,L)} \underset{q_4}{\downarrow}\, bx0y1b \xrightarrow{(x,x,R)} \underset{q_1}{\downarrow}\, bx0y1b$$

$$\xrightarrow{(0,x,R)} \underset{q_2}{\downarrow}\, bxxy1b \xrightarrow{(y,y,R)} \underset{q_2}{\downarrow}\, bxxy1b \xrightarrow{(1,y,L)} \underset{q_3}{\downarrow}\, bxxyyb$$

$$\xrightarrow{(y,y,L)} \underset{q_3}{\downarrow}\, bxxyyb \xrightarrow{(x,x,R)} \underset{q_5}{\downarrow}\, bxxyyb \xrightarrow{(y,y,R)} \underset{q_5}{\downarrow}\, bxxyyb$$

$$\xrightarrow{(y,y,R)} \underset{q_5}{\downarrow}\, bxxyyb \xrightarrow{(b,b,R)} \underset{q_6}{\downarrow}\, bxxyybb$$

9.3 LANGUAGE ACCEPTABILITY BY TURING MACHINES

Let us consider the Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, b, F)$. A string w in Σ^* is said to be accepted by M if $q_0 w \overset{*}{\vdash} \alpha_1 p \alpha_2$ for some $p \in F$ and $\alpha_1, \alpha_2 \in \Gamma^*$.

M does not accept w if the machine M either halts in a nonaccepting state or does not halt.

It may be noted that though there are other equivalent definitions of acceptance by the Turing machine, we will be not discussing them in this text.

EXAMPLE 9.4

Consider the Turing machine M described by the transition table given in Table 9.2. Describe the processing of (a) 011, (b) 0011, (c) 001 using IDs. Which of the above strings are accepted by M?

TABLE 9.2 Transition Table for Example 9.4

Present state	Tape symbol				
	0	1	x	y	b
$\rightarrow q_1$	xRq_2				bRq_5
q_2	$0Rq_2$	yLq_3		yRq_2	
q_3	$0Lq_4$		xRq_5	yLq_3	
q_4	$0Lq_4$		xRq_1		
q_5				$yxRq_5$	bRq_6
(q_6)					

Solution

(a) $q_1011 \vdash xq_211 \vdash q_3xy1 \vdash xq_5y1 \vdash xyq_51$

As $\delta(q_5, 1)$ is not defined, M halts; so the input string 011 is not accepted.

(b) $q_10011 \vdash xq_2011 \vdash x0q_211 \vdash xq_30y1 \vdash q_4x0yl \vdash xq_10y1.$

$\vdash xxq_2y1 \vdash xxyq_21 \vdash xxq_3yy \vdash xq_3xyy \vdash xxq_5yy$

$\vdash xxyq_5y \vdash xxyyq_5b \vdash xxyybq_6$

M halts. As q_6 is an accepting state, the input string 0011 is accepted by M.

(c) $q_1001 \vdash xq_201 \vdash x0q_21 \vdash xq_30y \vdash q_4x0y$

$\vdash xq_10y \vdash xxq_2y \vdash xxyq_2$

M halts. As q_2 is not an accepting state, 001 is not accepted by M.

9.4 DESIGN OF TURING MACHINES

We now give the basic guidelines for designing a Turing machine.

(i) The fundamental objective in scanning a symbol by the R/W head is to 'know' what to do in the future. The machine must remember the past symbols scanned. The Turing machine can remember this by going to the next unique state.

(ii) The number of states must be minimized. This can be achieved by changing the states only when there is a change in the written symbol or when there is a change in the movement of the R/W head. We shall explain the design by a simple example.

EXAMPLE 9.5

Design a Turing machine to recognize all strings consisting of an even number of 1's.

Solution

The construction is made by defining moves in the following manner:

(a) q_1 is the initial state. M enters the state q_2 on scanning 1 and writes b.

(b) If M is in state q_2 and scans 1, it enters q_1 and writes b.

(c) q_1 is the only accepting state.

So M accepts a string if it exhausts all the input symbols and finally is in state q_1. Symbolically,

$$M = (\{q_1, q_2\}, \{1, b\}, \{1, b\}, \delta, q, b, \{q_1\})$$

where δ is defined by Table 9.3.

TABLE 9.3 Transition Table for Example 9.5

Present state	1
$\rightarrow(q_1)$	bq_2R
q_2	bq_1R

Let us obtain the computation sequence of 11. Thus, $q_1 11 \vdash bq_2 1 \vdash bbq_1$. As q_1 is an accepting state, 11 is accepted. $q_1 111 \vdash bq_2 11 \vdash bbq_1 1 \vdash bbbq_2$. M halts and as q_2 is not an accepting state, 111 is not accepted by M.

EXAMPLE 9.6

Design a Turing machine over $\{1, b\}$ which can compute a concatenation function over $\Sigma = \{1\}$. If a pair of words (w_1, w_2) is the input, the output has to be $w_1 w_2$.

Solution

Let us assume that the two words w_1 and w_2 are written initially on the input tape separated by the symbol b. For example, if $w_1 = 11$, $w_2 = 111$, then the input and output tapes are as shown in Fig. 9.6.

Fig. 9.6 Input and output tapes.

We observe that the main task is to remove the symbol b. This can be done in the following manner:

(a) The separating symbol b is found and replaced by 1.

(b) The rightmost 1 is found and replaced by a blank b.
(c) The R/W head returns to the starting position.

A computation is illustrated in Table 9.4.

TABLE 9.4 Computation for $11b111$

$q_0 11b111 \vdash 1q_0 1b111 \vdash 11q_0 b111 \vdash 111q_1 111$
$\vdash 1111q_1 11 \vdash 11111q_1 1 \vdash 111111q_1 b \vdash 11111q_2 1b$
$\vdash 1111q_3 1bb \vdash 111q_3 11bb \vdash 11q_3 111bb \vdash 1q_3 1111bb$
$\vdash q_3 11111bb \vdash q_3 b11111bb \vdash bq_f 11111bb$

From the above computation sequence for the input string $11b111$, we can construct the transition table given in Table 9.5.

For the input string $1b1$, the computation sequence is given as

$$q_0 1b1 \vdash 1q_0 b1 \vdash 11q_1 1 \vdash 111q_1 b \vdash 11q_2 b \vdash 1q_3 1bb$$
$$\vdash q_3 11bb \vdash q_3 b11bb \vdash bq_f 11bb.$$

TABLE 9.5 Transition Table for Example 9.6

Present state	*Tape symbol*	
	1	b
$\rightarrow q_0$	$1Rq_0$	$1Rq_1$
q_1	$1Rq_1$	bLq_2
q_2	bLq_3	—
q_3	$1Lq_3$	bRq_f
(q_f)	—	—

EXAMPLE 9.7

Design a TM that accepts

$$\{0^n 1^n | n \geq 1\}.$$

Solution

We require the following moves:

(a) If the leftmost symbol in the given input string w is 0, replace it by x and move right till we encounter a leftmost 1 in w. Change it to y and move backwards.
(b) Repeat (a) with the leftmost 0. If we move back and forth and no 0 or 1 remains, move to a final state.
(c) For strings not in the form $0^n 1^n$, the resulting state has to be nonfinal.

Keeping these ideas in our mind, we construct a TM M as follows:

where

$$M = (Q, \Sigma, \Gamma, \delta, q_0, b, F)$$
$$Q = \{q_0, q_1, q_2, q_3, q_f\}$$
$$F = \{q_f\}$$
$$\Sigma = \{0, 1\}$$
$$\Gamma = \{0, 1, x, y, b\}$$

The transition diagram is given in Fig. 9.7. M accepts $\{0^n1^n \mid n \geq 1\}$. The moves for 0011 and 010 are given below just to familiarize the moves of M to the reader.

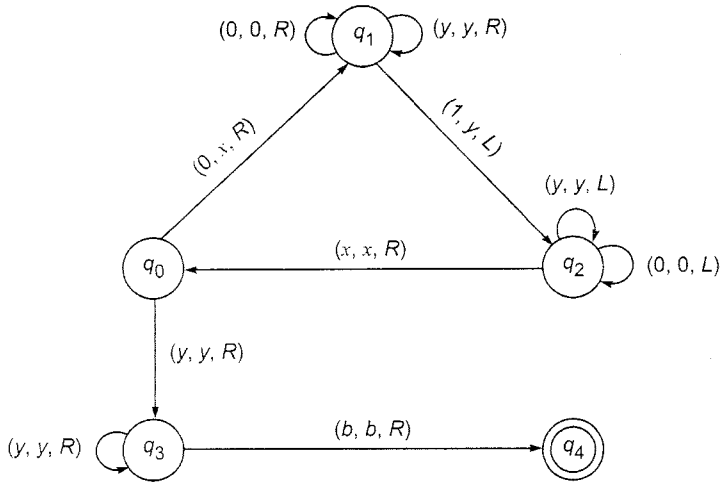


Fig. 9.7 Transition diagram for Example 9.7.

$$q_00011 \vdash xq_1011 \vdash x0q_111 \vdash xq_20y1$$
$$\vdash q_2x0y1 \vdash xq_00y1 \vdash xxq_1y1 \vdash xxyq_11$$
$$\vdash xxq_2yy \vdash xq_2xyy \vdash xxq_0yy \vdash xxyq_3y$$
$$\vdash xxyyq_3 = xxyyq_3b \vdash xxyybq_4b$$

Hence 0011 is accepted by M.

$$q_0010 \vdash xq_110 \vdash q_2xy0 \vdash xq_0y0 \vdash xyq_50$$

As $\delta(q_3, 0)$ is not defined, M halts. So 010 is not accepted by M.

EXAMPLE 9.8

Design a Turing machine M to recognize the language

$$\{1^n2^n3^n \mid n \geq 1\}.$$

Solution

Before designing the required Turing machine M, let us evolve a procedure for processing the input string 112233. After processing, we require the ID to be of the form $bbbbbbq_7$. The processing is done by using five steps:

Step 1 q_1 is the initial state. The R/W head scans the leftmost 1, replaces 1 by b, and moves to the right. M enters q_2.

Step 2 On scanning the leftmost 2, the R/W head replaces 2 by b and moves to the right. M enters q_3.

Step 3 On scanning the leftmost 3, the R/W head replaces 3 by b, and moves to the right. M enters q_4.

Step 4 After scanning the rightmost 3, the R/W heads moves to the left until it finds the leftmost 1. As a result, the leftmost 1, 2 and 3 are replaced by b.

Step 5 Steps 1–4 are repeated until all 1's, 2's and 3's are replaced by blanks.

The change of IDs due to processing of 112233 is given as

$$q_1112233 \vdash bq_212233 \vdash blq_2233 \vdash b1bq_3233 \vdash b1b2q_333$$

$$\vdash b1b2bq_43 \vdash b1b_2q_5b3 \vdash b1bq_52b3 \vdash b1q_5b2b3 \vdash bq_51b2b3$$

$$\vdash q_6b1b2b3 \vdash bq_11b2b3 \vdash bbq_2b2b3 \vdash bbbq_22b3$$

$$\vdash bbbbq_3b3 \vdash bbbbbq_33 \vdash bbbbbbq_4b \vdash bbbbbq_7bb$$

Thus,

$$q_1112233 \vdash^* q_7bbbbbb$$

As q_7 is an accepting state, the input string 112233 is accepted.

Now we can construct the transition table for M. It is given in Table 9.6.

TABLE 9.6 Transition Table for Example 9.7

Present state	Input tape symbol			
	1	2	3	b
$\rightarrow q_1$	bRq_2			bRq_1
q_2	$1Rq_2$	bRq_3		bRq_2
q_3		$2Rq_3$	bRq_4	bRq_3
q_4			$3Lq_5$	bLq_7
q_5	$1Lq_6$	$2Lq_5$		bLq_5
q_6	$1Lq_6$			bRq_1
(q_7)				

It can be seen from the table that strings other than those of the form $0^n1^n2^n$ are not accepted. It is advisable to compute the computation sequence for strings like 1223, 1123, 1233 and then see that these strings are rejected by M.

9.5 DESCRIPTION OF TURING MACHINES

In the examples discussed so far, the transition function δ was described as a partial function (function $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is not defined for all (q, x)) by spelling out the current state, the input symbol, the resulting state, the tape symbol replacing the input symbol and the movement of R/W head to the left or right. We can call this a formal description of a TM. Just as we have the machine language and higher level languages for a computer, we can have a higher level of description, called the *implementation description*. In this case we describe the movement of the head, the symbol stored etc. in English. For example, a single instruction like 'move to right till the end of the input string' requires several moves. A single instruction in the implementation description is equivalent to several moves of a standard TM (Hereafter a standard TM refers to the TM defined in Definition 9.1). At a higher level we can give instructions in English language even without specifying the state or transition function. This is called a *high-level description*.

In the remaining sections of this chapter and later chapters, we give implementation description or high-level description.

9.6 TECHNIQUES FOR TM CONSTRUCTION

In this section we give some high-level conceptual tools to make the construction of TMs easier. The Turing machine defined in Section 9.1 is called the standard Turing machine.

9.6.1 TURING MACHINE WITH STATIONARY HEAD

In the definition of a TM we defined $\delta(q, a)$ as (q', y, D) where $D = L$ or R. So the head moves to the left or right after reading an input symbol. Suppose, we want to include the option that the head can continue to be in the same cell for some input symbol. Then we define $\delta(q, a)$ as (q', y, S). This means that the TM, on reading the input symbol a, changes the state to q' and writes y in the current cell in place of a and continues to remain in the same cell. In terms of IDs,

$$wqax \vdash wq'yx$$

Of course, this move can be simulated by the standard TM with two moves, namely

$$wqax \vdash wyq''x \vdash wq'yx$$

That is, $\delta(q, a) = (q', y, S)$ is replaced by $\delta(q, a) = (q'', y, R)$ and $\delta(q'', X) = (q', y, L)$ for any tape symbol X.

Thus in this model $\delta(q, a) = (q', y, D)$ where $D = L, R$ or S.

9.6.2 Storage in the State

We are using a state, whether it is of a FA or pda or TM, to 'remember' things. We can use a state to store a symbol as well. So the state becomes a pair (q, a) where q is the state (in the usual sense) and a is the tape symbol stored in (q, a). So the new set of states becomes $Q \times \Gamma$.

EXAMPLE 9.9

Construct a TM that accepts the language **0 1* + 1 0*.**

Solution

We have to construct a TM that remembers the first symbol and checks that it does not appear afterwards in the input string. So we require two states, q_0, q_1. The tape symbols are 0, 1 and b. So the TM, having the 'storage facility in state', is

$$M = (\{q_0, q_1\} \times \{0, 1, b\}, \{0, 1\}, \{0, 1, b\}, \delta, [q_0, b], \{[q_1, b]\})$$

We describe δ by its implementation description.

1. In the initial state, M is in q_0 and has b in its data portion. On seeing the first symbol of the input sting w, M moves right, enters the state q_1 and the first symbol, say a, it has seen.
2. M is now in $[q_1, a]$. (i) If its next symbol is b, M enters $[q_1, b]$, an accepting state. (ii) If the next symbol is a, M halts without reaching the final state (i.e. δ is not defined). (iii) If the next symbol is $\bar{a}$ ($\bar{a} = 0$ if $a = 1$ and $\bar{a} = 1$ if $a = 0$), M moves right without changing state.
3. Step 2 is repeated until M reaches $[q_1, b]$ or halts (δ is not defined for an input symbol in w).

9.6.3 Multiple Track Turing Machine

In the case of TM defined earlier, a single tape was used. In a multiple track TM, a single tape is assumed to be divided into several tracks. Now the tape alphabet is required to consist of k-tuples of tape symbols, k being the number of tracks. Hence the only difference between the standard TM and the TM with multiple tracks is the set of tape symbols. In the case of the standard Turing machine, tape symbols are elements of Γ; in the case of TM with multiple track, it is Γ^k. The moves are defined in a similar way.

9.6.4 Subroutines

We know that subroutines are used in computer languages, when some task has to be done repeatedly. We can implement this facility for TMs as well.

First, a TM program for the subroutine is written. This will have an initial state and a 'return' state. After reaching the return state, there is a temporary halt. For using a subroutine, new states are introduced. When there is a need for calling the subroutine, moves are effected to enter the initial state for the subroutine (when the return state of the subroutine is reached) and to return to the main program of TM.

We use this concept to design a TM for performing multiplication of two positive integers.

EXAMPLE 9.10

Design a TM which can multiply two positive integers.

Solution

The input (m, n), m, n being given, the positive integers are represented by 0^m10^n. M starts with 0^m10^n in its tape. At the end of the computation, 0^{mn}(mn in unary representation) surrounded by b's is obtained as the ouput.

The major steps in the construction are as follows:

1. 0^m10^n1 is placed on the tape (the output will be written after the rightmost 1).
2. The leftmost 0 is erased.
3. A block of n 0's is copied onto the right end.
4. Steps 2 and 3 are repeated m times and 10^m10^{mn} is obtained on the tape.
5. The prefix 10^m1 of 10^m10^{mn} is erased, leaving the product mn as the output.

For every 0 in 0^m, 0^n is added onto the right end. This requires repetition of step 3. We define a subroutine called COPY for step 3.

For the subroutine COPY, the initial state is q_1 and the final state is q_5. δ is given by the transition table (see Table 9.7).

TABLE 9.7 Transition Table for Subroutine COPY

State	Tape symbol			
	0	1	2	b
q_1	$q_2 2R$	$q_4 1L$	—	—
q_2	$q_2 0R$	$q_2 1R$	—	$q_3 0L$
q_3	$q_3 0L$	$q_3 1L$	$q_1 2R$	—
q_4	—	$q_5 1R$	$q_4 0L$	—
q_5	—	—	—	—

The Turing machine M has the initial state q_0. The initial ID for M is $q_00^m10^n1$. On seeing 0, the following moves take place (q_6 is a state of M). $q_00^m10^n1 \vdash bq_60^{m-1}10^n1 \overset{*}{\vdash} b0^{m-1}q_610^n1 \vdash b0^{m-1}1q_10^n1$. q_1 is the initial state

of COPY. The TM M_1 performs the subroutine COPY. The following moves take place for M_1: $q_1 0^n 1 \vdash 2q_2 0^{n-1} 1 \vdash^* 20^{n-1} 1 q_3 b \vdash 20^{n-1} q_3 10 \vdash^* 2q_1 0^{n-1} 10$. After exhausting 0's, q_1 encounters 1. M_1 moves to state q_4. All 2's are converted back to 0's and M_1 halts in q_5. The TM M picks up the computation by starting from q_5. The q_0 and q_6 are the states of M. Additional states are created to check whether each 0 in 0^m gives rise to 0^m at the end of the rightmost 1 in the input string. Once this is over, M erases $10^n 1$ and finds 0^{nm} in the input tape.

M can be defined by

$$M = (\{q_0, q_1, \ldots, q_{12}\}, \{0, 1\}, \{0, 1, 2, b\}, \delta, q_0, b, \{q_{12}\})$$

where δ is defined by Table 9.8.

TABLE 9.8 Transition Table for Example 9.10

	0	1	2	b
q_0	$q_6 bR$	—	—	—
q_6	$q_6 0R$	$q_1 1R$	—	—
q_5	$q_7 0L$	—	—	—
q_7	—	$q_8 1L$	—	—
q_8	$q_9 0L$	—	—	$q_{10} bR$
q_9	$q_9 0L$	—	—	$q_0 bR$
q_{10}	—	$q_{11} bR$	—	—
q_{11}	$q_{11} bR$	$q_{12} bR$	—	—

Thus M performs multiplication of two numbers in unary representation.

9.7 VARIANTS OF TURING MACHINES

The Turing machine we have introduced has a single tape. $\delta(q, a)$ is either a single triple (p, y, D), where $D = R$ or L, or is not defined. We introduce two new models of TM:

(i) a TM with more than one tape

(ii) a TM where $\delta(q, a) = \{(p_1, y_1, D_1), (p_2, y_2, D_2), \ldots, (p_r, y_r, D_r)\}$. The first model is called a multitape TM and the second a nondeterministic TM.

9.7.1 MULTITAPE TURING MACHINES

A multitape TM has a finite set Q of states, an initial state q_0, a subset F of Q called the set of final states, a set P of tape symbols, a new symbol b, not in P called the blank symbol. (We assume that $\Sigma \subseteq \Gamma$ and $b \notin \Sigma$.)

There are k tapes, each divided into cells. The first tape holds the input string w. Initially, all the other tapes hold the blank symbol.

Initially the head of the first tape (input tape) is at the left end of the input w. All the other heads can be placed at any cell initially.

δ is a partial function from $Q \times \Gamma^k$ into $Q \times \Gamma^k \times \{L, R, S\}^k$. We use implementation description to define δ. Figure 9.8 represents a multitape TM. A move depends on the current state and k tape symbols under k tape heads.

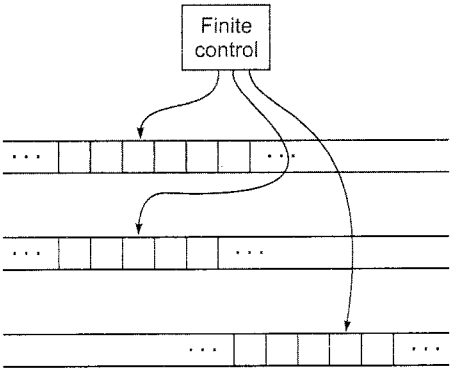


Fig. 9.8 Multitape Turing machine.

In a typical move:

(i) M enters a new state.
(ii) On each tape, a new symbol is written in the cell under the head.
(iii) Each tape head moves to the left or right or remains stationary. The heads move independently: some move to the left, some to the right and the remaining heads do not move.

The initial ID has the initial state q_0, the input string w in the first tape (input tape), empty strings of b's in the remaining $k - 1$ tapes. An accepting ID has a final state, some strings in each of the k tapes.

Theorem 9.1 Every language accepted by a multitape TM is acceptable by some single-tape TM (that is, the standard TM).

Proof Suppose a language L is accepted by a k-tape TM M. We simulate M with a single-tape TM with $2k$ tracks. The second, fourth, . . ., $(2k)$th tracks hold the contents of the k-tapes. The first, third, . . ., $(2k - 1)$th tracks hold a head marker (a symbol say X) to indicate the position of the respective tape head. We give an 'implementation description' of the simulation of M with a single-tape TM M_1. We give it for the case $k = 2$. The construction can be extended to the general case.

Figure 9.9 can be used to visualize the simulation. The symbols A_2 and B_5 are the current symbols to be scanned and so the headmarker X is above the two symbols.

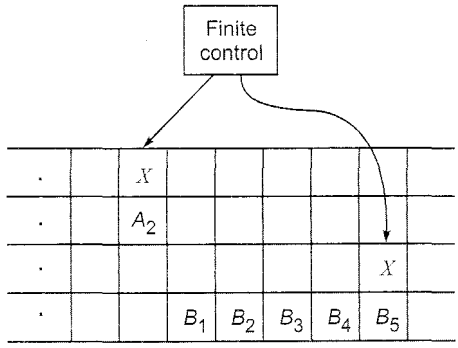


Fig. 9.9 Simulation of multitape TM.

Initially the contents of tapes 1 and 2 of M are stored in the second and fourth tracks of M_1. The headmarkers of the first and third tracks are at the cells containing the first symbol.

To simulate a move of M, the $2k$-track TM M_1 has to visit the two headmarkers and store the scanned symbols in its control. Keeping track of the headmarkers visited and those to be visited is achieved by keeping a count and storing it in the finite control of M_1. Note that the finite control of M_1 has also the information about the states of M and its moves. After visiting both head markers, M_1 knows the tape symbols being scanned by the two heads of M.

Now M_1 revisits each of the headmarkers:

(i) It changes the tape symbol in the corresponding track of M_1 based on the information regarding the move of M corresponding to the state (of M) and the tape symbol in the corresponding tape M.
(ii) It moves the headmarkers to the left or right.
(iii) M_1 changes the state of M in its control.

This is the simulation of a single move of M. At the end of this, M_1 is ready to implement its next move based on the revised positions of its headmarkers and the changed state available in its control.

M_1 accepts a string w if the new state of M, as recorded in its control at the end of the processing of w, is a final state of M.

Definition 9.3 Let M be a TM and w an input string. The running time of M on input w, is the number of steps that M takes before halting. If M does not halt on an input string w, then the running time of M on w is infinite.

Note: Some TMs may not halt on all inputs of length n. But we are interested in computing the running time, only when the TM halts.

Definition 9.4 The time complexity of TM M is the function $T(n)$, n being the input size, where $T(n)$ is defined as the maximum of the running time of M over all inputs w of size n.

Theorem 9.2 If M_1 is the single-tape TM simulating multitape TM M, then the time taken by M_1 to simulate n moves of M is $O(n^2)$.

Proof Let M be a k-tape TM. After n moves of M, the head markers of M_1 will be separated by $2n$ cells or less. (At the worst, one tape movement can be to the left by n cells and another can be to the right by n cells. In this case the tape headmarkers are separated by $2n$ cells. In the other cases, the 'gap' between them is less). To simulate a move of M, the TM M_1 must visit all the k headmarkers. If M starts with the leftmost headmarker, M_1 will go through all the headmarkers by moving right by at most $2n$ cells. To simulate the change in each tape, M_1 has to move left by at most $2n$ cells; to simulate changes in k tapes, it requires at most two moves in the reverse direction for each tape.

Thus the total number of moves by M_1 for simulating one move of M is atmost $4n + 2k$. ($2n$ moves to right for locating all headmarkers, $2n + 2k$ moves to the left for simulating the change in the content of k tapes.) So the number of moves of M_1 for simulating n moves of M is $n(4n + 2k)$. As the constant k is independent of n, the time taken by M_1 is $O(n^2)$.

9.7.2 Nondeterministic Turing Machines

In the case of standard Turing machines (hereafter we refer to this machine as deterministic TM), $\delta(q_1, a)$ was defined (for some elements of $Q \times \Gamma$) as an element of $Q \times \Gamma \times \{L, R\}$. Now we extend the definition of δ. In a nondeterministic TM, $\delta(q_1, a)$ is defined as a subset of $Q \times \Gamma \times \{L, R\}$.

Definition 9.5 A nondeterministic Turing machine is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, b, F)$ where

1. Q is a finite nonempty set of states
2. Γ is a finite nonempty set of tape symbols
3. $b \in \Gamma$ is called the blank symbol
4. Σ is a nonempty subset of Γ, called the set of input symbols. We assume that $b \notin \Sigma$.
5. q_0 is the initial state
6. $F \subseteq Q$ is the set of final states
7. δ is a partial function from $Q \times \Gamma$ into the power set of $Q \times \Gamma \times \{L, R\}$.

Note: If $q \in Q$ and $x \in \Gamma$ and $\delta(q, x) = \{(q_1, y_1, D_1), (q_2, y_2, D_2), \ldots, (q_n, y_n, D_n)\}$ then the NTM can chose any one of the actions defined by (q_i, y_i, D_i) for $i = 1, 2, \ldots, n$.

We can also express this in terms of $\vdash$ relation. If $\delta(q, x) = \{(q_i, y_i, D_i) | i = 1, 2, \ldots, n\}$ then the ID $zqxw$ can change to any one of the n IDs specified by the n-element set $\delta(q, x)$.

Suppose $\delta(q, x) = \{(q_1, y_1, L), (q_2, y_2, R), (q_3, y_3, L)\}$. Then

$$z_1 z_2 \cdots z_k q x z_{k+1} \cdots z_n \vdash z_1 z_2 \cdots z_{k-1} q_1 z_k y_1 z_{k+1} \cdots z_n$$

or

$$z_1 z_2 \cdots z_k q x z_{k+1} \cdots z_n \vdash z_1 z_2 \cdots z_k y_2 q_2 z_{k+1} \cdots z_n$$

or

$$z_1 z_2 \cdots z_k q x z_{k+1} \cdots z_n \vdash z_1 z_2 \cdots z_{k-1} q_3 z_k y_3 z_{k+1} \cdots z_n.$$

So on reading the input symbol, the NTM M whose current ID is $z_1z_2 \ldots z_kqxz_{k+1} \ldots z_n$ can change to any one of the three IDs given earlier.

Remark When $\delta(q, x) = \{(q_i, y_i, D_i) \mid i = 1, 2, \ldots, n\}$ then NTM chooses any one of the n triples totally (that is, it cannot take a state from one triple, another tape symbol from a second triple and a third $D(L$ or $R)$ from a third triple, etc.

Definition 9.6 $w \in \Sigma^*$ is accepted by a nondeterministic TM M if $q_0w \overset{*}{\vdash} xq_fy$ for some final state q_f.

The set of all strings accepted by M is denoted by $T(M)$.

Note: As in the case of NDFA, an ID of the form xqy (for some $q \notin F$) may be reached as the result of applying the input string w. But w is accepted by M as long as there is some sequence of moves leading to an ID with an accepting state. It does not matter that there are other sequences of moves leading to an ID with a nonfinal state or TM halts without processing the entire input string.

Theorem 9.3 If M is a nondeterministic TM, there is a deterministic TM M_1 such that $T(M) = T(M_1)$.

Proof We construct M_1 as a multitape TM. Each symbol in the input string leads to a change in ID. M_1 should be able to reach all IDs and stop when an ID containing a final state is reached. So the first tape is used to store IDs of M as a sequence and also the state of M. These IDs are separated by the symbol $*$ (included as a tape symbol). The current ID is known by marking an x along with the ID-separator $*$ (The symbol $*$ marked with x is a new tape symbol.) All IDs to the left of the current one have been explored already and so can be ignored subsequently. Note that the current ID is decided by the current input symbol of w.

Figure 9.10 illustrates the deterministic TM M_1.

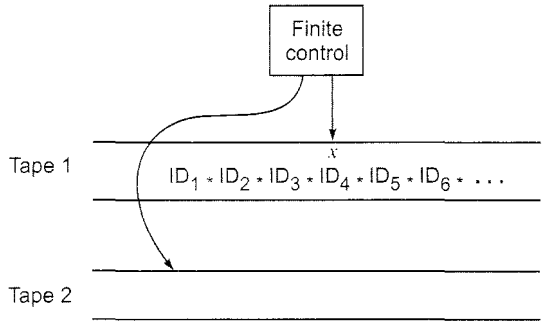


Fig. 9.10 The deterministic TM simulating M.

To process the current ID, M_1 performs the following steps.

1. M_1 examines the state and the scanned symbol of the current ID. Using the knowledge of moves of M stored in the finite control of M_1, M_1 checks whether the state in the current ID is an accepting state of M. In this case M_1 accepts and stops simulating M.

2. If the state q say in the current ID $xqay$ is not an accepting state of M_1 and $\delta(q, a)$ has k triples, M_1 copies the ID $xqay$ in the second tape and makes k copies of this ID at the end of the sequence of IDs in tape 2.
3. M_1 modifies these k IDs in tape 2 according to the k choices given by $\delta(q, a)$.
4. M_1 returns to the marked current ID, erases the mark x and marks the next ID-separator * with x (to the * which is to the left of the next ID to be processed). Then M_1 goes back to step 1.

M_1 stops when an accepting state of M is reached in step 1.

Now M_1 accepts an input string w only when it is able to find that M has entered an accepting state, after a finite number of moves. This is clear from the simulated sequence of moves of M_1 (ending in step 1)

We have to prove that M_1 will eventually reach an accepting ID (that is, an ID having an accepting state of M) if M enters an accepting ID after n moves. Note each move of M is simulated by several moves of M_1.

Let m be the maximum number of choices that M has for various (q, a)'s. (It is possible to find m since we have only finite number of pairs in $Q \times \Gamma$.) So for each initial ID of M, there are at most m IDs that M can reach after one move, at most m^2 IDs that M can reach after two moves, and so on. So corresponding to n moves of M, there are at most $1 + m + m^2 + \cdots + m^n$ moves of M_1. Hence the number of IDs to be explored by M_1 is at most nm^n.

We assume that M_1 explores these IDs. These IDs have a tree structure having the initial ID as its root. We can apply breadth-first search of the nodes of the tree (that is, the nodes at level 1 are searched, then the nodes at level 2, and so on.) If M reaches an accepting ID after n moves, then M_1 has to search atmost nm^n IDs before reaching an accepting ID. So, if M accepts w, then M_1 also accepts w (eventually). Hence $T(M) = T(M_1)$.

9.8 THE MODEL OF LINEAR BOUNDED AUTOMATON

This model is important because (a) the set of context-sensitive languages is accepted by the model, and (b) the infinite storage is restricted in size but not in accessibility to the storage in comparison with the Turing machine model. It is called the *linear bounded automaton* (LBA) because a linear function is used to restrict (to bound) the length of the tape.

In this section we define the model of linear bounded automaton and develop the relation between the linear bounded automata and context-sensitive languages. It should be noted that the study of context-sensitive languages is important from practical point of view because many compiler languages lie between context-sensitive and context-free languages.

A linear bounded automaton is a nondeterministic Turing machine which has a single tape whose length is not infinite but bounded by a linear function

of the length of the input string. The models can be described formally by the following set format:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, b, ¢, \$, F)$$

All the symbols have the same meaning as in the basic model of Turing machines with the difference that the input alphabet Σ contains two special symbols ¢ and \$. ¢ is called the left-end marker which is entered in the left-most cell of the input tape and prevents the R/W head from getting off the left end of the tape. \$ is called the right-end marker which is entered in the right-most cell of the input tape and prevents the R/W head from getting off the right end of the tape. Both the endmarkers should not appear on any other cell within the input tape, and the R/W head should not print any other symbol over both the endmarkers.

Let us consider the input string w with $|w| = n - 2$. The input string w can be recognized by an LBA if it can also be recognized by a Turing machine using no more than kn cells of input tape, where k is a constant specified in the description of LBA. The value of k does not depend on the input string but is purely a property of the machine. Whenever we process any string in LBA, we shall assume that the input string is enclosed within the endmarkers ¢ and \$. The above model of LBA can be represented by the block diagram of Fig. 9.11. There are two tapes: one is called the input tape, and the other, working tape. On the input tape the head never prints and never moves to the left. On the working tape the head can modify the contents in any way, without any restriction.

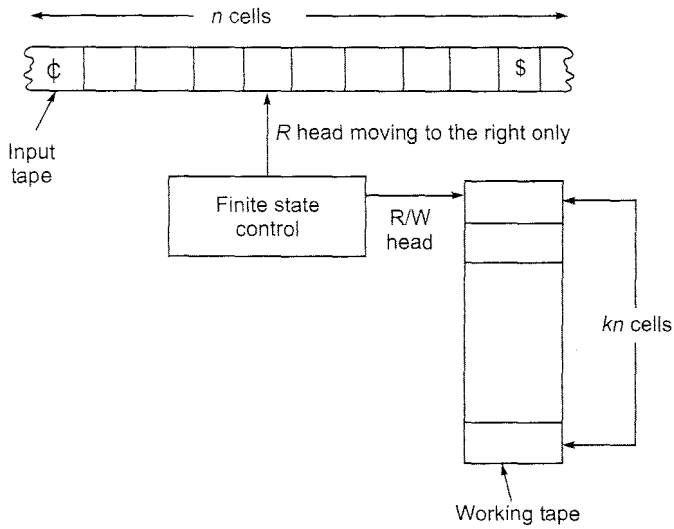


Fig. 9.11 Model of linear bounded automaton.

In the case of LBA, an ID is denoted by (q, w, k), where $q \in Q$, $w \in \Gamma$ and k is some integer between 1 and n. The transition of IDs is similar except

that k changes to $k - 1$ if the R/W head moves to the left and to $k + 1$ if the head moves to the right.

The language accepted by LBA is defined as the set

$$\{w \in (\Sigma - \{¢, \$\})^* | (q_0, ¢w\$, 1) \vdash^* (q, \alpha, i)$$

for some $q \in F$ and for some integer i between 1 and $n\}$.

Note: As a null string can be represented either by the absence of input string or by a completely blank tape, an LBA may accept the null string.

9.8.1 Relation Between LBA and Context-sensitive Languages

The set of strings accepted by nondeterministic LBA is the set of strings generated by the context-sensitive grammars, excluding the null strings. Now we give an important result:

If L is a context-sensitive language, then L is accepted by a linear bounded automaton. The converse is also true.

The construction and the proof are similar to those for Turing machines with some modifications.

9.9 TURING MACHINES AND TYPE 0 GRAMMARS

In this section we construct a type 0 grammar generating the set accepted by a given Turing machine M. The productions are constructed in two steps. In step 1 we construct productions which transform the string $[q_1¢\,w\$]$ into the string $[q_2b]$, where q_1 is the initial state, q_2 is an accepting state, ¢ is the left-endmarker, and \$ is the right-endmarker. The grammar obtained by applying step 1 is called the *transformational grammar*. In step 2 we obtain inverse production rules by reversing the productions of the transformational grammar to get the required type 0 grammar G. The construction is in such a way that w is accepted by M if and only if w is in $L(G)$.

9.9.1 Construction of a Grammar Corresponding to TM

For understanding the construction, we have to note that a transition of ID corresponds to a production. We enclose IDs within brackets. So acceptance of w by M corresponds to the transformation of initial ID $[q_1\,¢\,w\,\$]$ into $[q_2b]$. Also, the 'length' of ID may change if the R/W head reaches the left-end or the right-end, i.e. when the left-hand side or the right-hand side bracket is reached. So we get productions corresponding to transition of IDs with (i) no change in length, and (ii) change in length. We assume that the transition table is given.

We now describe the construction which involves two steps:

Step 1 (i) *No change in length of* IDs: (a) *Right move.* a_kRq_l corresponding to q_i-row and a_j-column leads to the production

$$q_ia_j \rightarrow a_kq_l$$

(b) *Left move.* a_kLq_l corresponding to q_i-row and a_j-column yields several productions

$$a_mq_ia_j \rightarrow q_la_ma_k \qquad \text{for all } a_m \in \Gamma$$

(ii) *Change in length of* IDs: (a) *Left-end.* a_kLq_l corresponding to q_i-row and a_j-column gives

$$[q_ia_j \rightarrow [q_lba_k$$

When b occurs next to the left-bracket, it can be deleted. This is achieved by including the production $[b \rightarrow [$.

(b) *Right-end.* When b occurs to the left of], it can be deleted. This is achieved by the production

$$a_jb] \rightarrow a_j] \qquad \text{for all } a_j \in \Gamma$$

When the R/W head moves to the right of], the length increases. Corresponding to this we have a production

$$q_i] \rightarrow q_ib] \qquad \text{for all } q_i \in Q$$

(iii) *Introduction of endmarkers.* For introducing endmarkers for the input string, the following productions are included:

$$a_i \rightarrow [q_1 ¢ a_i \qquad \text{for } a_i \in \Gamma,\ a_i \neq b$$

$$a_i \rightarrow a_i\$] \qquad \text{for all } a_i \in \Gamma,\ a_i \neq b$$

For removing the brackets from $[q_2b]$, we include the production

$$[q_2b] \rightarrow S$$

Recall that q_1 and q_2 are the initial and final states, respectively.

Step 2 To get the required grammar, reverse the arrows of the productions obtained in step 1. The productions we get can be called *inverse productions.* The new grammar is called the *generative grammar.* We illustrate the construction with an example.

EXAMPLE 9.11

Consider the TM described by the transition table given in Table 9.9. Obtain the inverse production rules.

Solution

In this example, q_1 is both initial and final.

Step 1 (i) *Productions corresponding to right moves*

$$q_1¢ \rightarrow ¢q_1, \qquad q_11 \rightarrow bq_2, \qquad q_21 \rightarrow bq_1 \tag{9.1}$$

(ii) (a) *Productions corresponding to left-end*

$$[b \to [\tag{9.2}$$

(b) *Productions corresponding to rig't-end*

$$bb] \to b], \quad 1b] \to 1], \quad q_1] \to q_1 b], \quad q_2] \to q_2 b] \tag{9.3}$$

(iii) $$1 \to [q_1 ¢1, \quad 1 \to 1\$], \quad [q_1 b] \to S \tag{9.4}$$

TABLE 9.9 Transition Table for Example 9.11

Present state	¢	*b*	1
$\to (q_1)$	$¢Rq_1$		bRq_2
q_2			bRq_1

Step 2 The inverse productions are obtained by reversing the arrows of the productions (9.1)–(9.4).

$$¢q_1 \to q_1 ¢, \quad bq_2 \to q_1 1, \quad bq_1 \to q_2 1$$
$$[\to [b, \; b] \to bb], \quad 1] \to 1b]$$
$$q_1 b \to q_1], \quad q_2 b \to q_2], \quad [q_1 ¢1 \to 1$$
$$1\$] \to 1, \quad S \to [q_1 b]$$

Thus we have shown that there exists a type 0 grammar corresponding to a Turing machine. The converse is also true (we are not proving this), i.e. given a type 0 grammar G, there exists a Turing machine accepting $L(G)$. Actually, the class of recursively enumerable sets, the type 0 languages, and the class of sets accepted by TM are one and the same. We have shown that there exists a recursively enumerable set which is not a context-sensitive language (see Theorem 4.4). As a recursive set is recursively enumerable, Theorem 4.4 gives a type 0 language which is not type 1. Hence, $\mathcal{L}_{\text{csl}} \subset \mathcal{L}_0$ (cf Property 4, Section 4.3) is established.

9.10 LINEAR BOUNDED AUTOMATA AND LANGUAGES

A linear bounded automaton M accepts a string w if, after starting at the initial state with R/W head reading the left-endmarker, M halts over the right-endmarker in a final state. Otherwise, w is rejected.

The production rules for the generative grammar are constructed as in the case of Turing machines. The following additional productions are needed in the case of LBA.

$$a_i q_f \$ \to q_f \$ \qquad \text{for all } a_i \in \Gamma$$
$$¢q_f S \to ¢q_f, \qquad ¢q_f \to q_f$$

EXAMPLE 9.12

Find the grammar generating the set accepted by a linear bounded automaton M whose transition table is given in Table 9.10.

TABLE 9.10 Transition Table for Example 9.12

Present state	Tape input symbol			
	¢	$	0	1
$\rightarrow q_1$	$¢Rq_1$		$1Lq_2$	$0Rq_2$
q_2	$¢Rq_4$		$1Rq_3$	$1Lq_1$
q_3		$\$Lq_1$	$1Rq_3$	$1Rq_3$
(q_4)		Halt	$0Lq_4$	$0Rq_4$

Solution

Step 1 (A) (i) *Productions corresponding to right moves.* The seven right moves in Table 9.10 give the following productions:

$$\begin{aligned} &q_1¢ \rightarrow ¢q_1, \quad && q_30 \rightarrow 1q_3 \\ &q_11 \rightarrow 0q_2, && q_31 \rightarrow 1q_3 \\ &q_2¢ \rightarrow ¢q_4, && q_41 \rightarrow 0q_4 \\ &q_20 \rightarrow 1q_3 \end{aligned} \tag{9.5}$$

(ii) *Productions corresponding to left moves.* There are four left moves in Table 9.10. Each left move yields four productions (corresponding to the four tape symbols). These are:

(a) $1Lq_2$ corresponding to q_1-row and 0-column gives

$$¢q_10 \rightarrow q_2¢1,\ \$q_10 \rightarrow q_2\$1,\ 0q_10 \rightarrow q_201,\ 1q_10 \rightarrow q_211 \tag{9.6}$$

(b) $1Lq_1$ corresponding to q_1-row and 1-column yields

$$¢q_21 \rightarrow q_1¢1,\ \$q_21 \rightarrow q_1\$1,\ 0q_21 \rightarrow q_101,\ 1q_21 \rightarrow q_111 \tag{9.7}$$

(c) $\$Lq_1$ corresponding to q_3-row and \$-column gives

$$¢q_3\$ \rightarrow q_1¢\,\$,\ \$q_3\$ \rightarrow q_1\$\$,\ 0q_3\$ \rightarrow q_10\$,\ 1q_3\$ \rightarrow q_11\$ \tag{9.8}$$

(d) $0Lq_4$ corresponding to q_4-row and 0-column yields

$$¢q_40 \rightarrow q_4¢0,\ \$q_40 \rightarrow q_4\$0,\ 0q_40 \rightarrow q_400,\ 1q_40 \rightarrow q_410 \tag{9.9}$$

(B) There are no productions corresponding to change in length.

(C) The productions for introducing the endmarkers are

$$\begin{aligned} ¢ &\rightarrow [q_1¢¢, && ¢ \rightarrow ¢\$] \\ \$ &\rightarrow [q_1¢\$, && \$ \rightarrow \$\$] \\ 0 &\rightarrow [q_1¢0, && 0 \rightarrow 0\$] \\ 1 &\rightarrow [q_1¢1, && 1 \rightarrow 1\$] \end{aligned} \tag{9.10}$$

$$[q_4] \rightarrow S \tag{9.11}$$

(D) The LBA productions are

$$\begin{aligned} &¢q_4\$ \to q_4\$, \qquad && ¢q_4\$ \to ¢q_4 \\ &\$q_4\$ \to q_4\$, && ¢q_4 \to q_4 \\ &0q_4\$ \to q_4\$, \\ &1q_4\$ \to q_4\$ \end{aligned} \tag{9.12}$$

Step 2 The productions of the generative grammar are obtained by reversing the arrows of productions given by (9.5)–(9.12).

9.11 SUPPLEMENTARY EXAMPLES

EXAMPLE 9.13

Design a TM that copies strings of 1's.

Solution

We design a TM so that we have ww after copying $w \in \{1\}^*$. Define M by

$$M = (\{q_0, q_1, q_2, q_3\}, \{1\}, \{1, b\}, \delta, q_0, b, \{q_3\})$$

where δ is defined by Table 9.11.

TABLE 9.11 Transition Table for Example 9.13

Present state	*Tape symbol*		
	1	b	a
q_0	q_0aR	q_1bL	—
q_1	$q_1 1L$	q_3bR	$q_2 1R$
q_2	$q_2 1R$	$q_1 1L$	—
q_3	—	—	—

The procedure is simple.

M replaces every 1 by the symbol a. Then M replaces the rightmost a by 1. It goes to the right end of the string and writes a 1 there. Thus M has added a 1 for the rightmost 1 in the input string w. This process can be repeated.

M reaches q_1 after replacing all 1's by a's and reading the blank at the end of the input string. After replacing a by 1, M reaches q_2. M reaches q_3 at the end of the process and halts. If $w = 1^n$, than we have 1^{2n} at the end of the computation. A sample computation is given below.

$$\begin{aligned} q_0 11 &\vdash aq_0 1 \vdash aaq_0 b \vdash aq_1 a \\ &\vdash a1q_2 b \vdash aq_1 11 \vdash q_1 a11 \\ &\vdash 1q_2 11 \vdash 11q_2 1 \vdash 111q_2 b \\ &\vdash 11q_2 11 \vdash 1q_1 111 \\ &\vdash q_1 1111 \vdash q_1 b1111 \vdash q_3 1111 \end{aligned}$$

EXAMPLE 9.14

Construct a TM to accept the set L of all strings over $\{0,1\}$ ending with 010.

Solution

L is certainly a regular set and hence a deterministic automaton is sufficient to recognize L. Figure 9.12 gives a DFA accepting L.

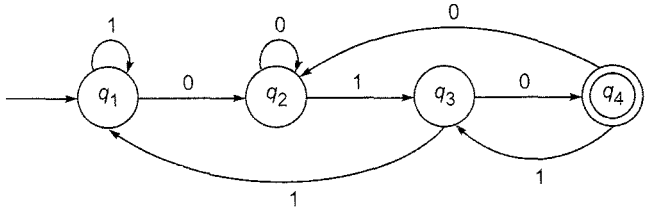


Fig. 9.12 DFA for Example 9.14.

Converting this DFA to a TM is simple. In a DFA M, the move is always to the right. So the TM's move will always be to the right. Also M reads the input symbol and changes state. So the TM M_1 does the same; it reads an input symbol, does not change the symbol and changes state. At the end of the computation, the TM sees the first blank b and changes to its final state. The initial ID of M_1 is q_0w. By defining $\delta(q_0, b) = (q_1, b, R)$, M_1 reaches the initial state of M. M_1 can be described by Fig. 9.13.

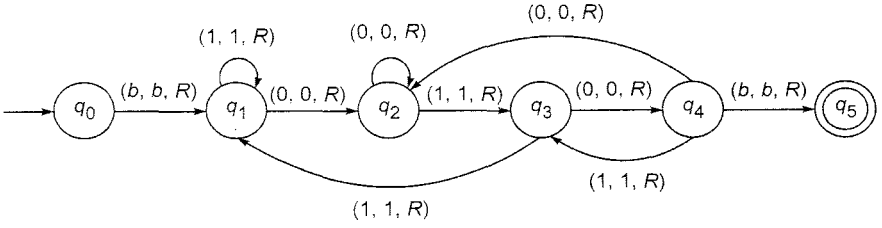


Fig. 9.13 TM for Example 9.14.

Note: q_5 is the unique final state of M_1. By comparing Figs. 9.12 and 9.13 it is easy to see that strings of L are accepted by M_1.

EXAMPLE 9.15

Design a TM that reads a string in $\{0, 1\}^*$ and erases the rightmost symbol.

Solution

The required TM M is given by

$$M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \{0, 1, b\}, \delta, q_0, b, \{q_4\})$$

where δ is defined by

$$\delta(q_0, 0) = (q_1, 0, R) \qquad \delta(q_0, 1) = (q_1, 1, R) \qquad (R_1)$$

$$\delta(q_1, 0) = (q_1, 0, R) \qquad \delta(q_1, 1) = (q_1, 1, R) \qquad (R_2)$$

$$\delta(q_1, b) = (q_2, b, L) \qquad (R_3)$$

$$\delta(q_2, 0) = (q_3, b, L) \qquad \delta(q_2, 1) = (q_3, b, L) \qquad (R_4)$$

$$\delta(q_3, 0) = (q_3, 0, L) \qquad \delta(q_3, 1) = (q_3, 1, L) \qquad (R_5)$$

$$\delta(q_3, b) = (q_4, b, R) \qquad (R_6)$$

Let w be the input string. By (R_1) and (R_2), M reads the entire input string w. At the end, M is in state q_1. On seeing the blank to the right of w, M reaches the state q_2 and moves left. The rightmost string in w is erased (by (R_4)) and the state becomes q_3. Afterwards M moves to the left until it reaches the left-end of w. On seeing the blank b to the right of w, M changes its state to q_4, which is the final state of M. From the construction it is clear that the rightmost symbol of w is erased.

EXAMPLE 9.16

Construct a TM that accepts $L = \{0^{2^n} \mid n \geq 0\}$.

Solution

Let w be an input string in $\{0\}^*$. The TM accepting L functions as follows:

1. It writes b (blank symbol) on the leftmost 0 of the input string w. This is done to mark the left-end of w.
2. M reads the symbols of w from left to right and replaces the alternate 0's with x's.
3. If the tape contains a single 0 in step 2, M accepts w.
4. If the tape contains more than one 0 and the number of 0's is odd in step 2, M rejects w.
5. M returns the head to the left-end of the tape (marked by blank b in step 1).
6. M goes to step 2.

Each iteration of step 2 reduces w to half its size. Also whether the number of 0's seen is even or odd is known after step 2. If that number is odd and greater than 1, w cannot be 0^{2^n} (step 4). In this case M rejects w. If the number of 0's seen is 1 (step 3), M accepts w (In this case 0^{2^n} is reduced to 0 in successive stages of step 2).

We define M by

$$M = (\{q_0, q_1, q_2, q_3, q_4, q_f, q_t\}, \{0\}, \{0, x, b\}, \delta, q_0, b, \{q_f\})$$

where δ is defined by Table 9.12.

TABLE 9.12 Transition Table for Example 9.16

Present state	Tape symbol		
	0	b	x
q_0	bRq_1	bRq_t	xRq_t
q_1	xRq_2	bRq_f	xRq_1
q_2	$0Rq_3$	bRq_4	xRq_2
q_3	xRq_2	bRq_6	xRq_3
q_4	$0Lq_4$	bRq_1	xLq_4
q_f	—	—	—
q_t	—	—	—

From the construction, it is apparent that the states are used to know whether the number of 0's read is odd or even.

We can see how M processes 0000.

$$\begin{aligned} q_0 0000 &\vdash bq_1 000 \vdash bxq_2 00 \vdash bxq_3 0 \vdash bx0xq_2 b \\ &\vdash bx0q_4 xb \vdash bxq_4 0xb \vdash bq_4 x0xb \vdash q_4 bx0xb \\ &\vdash bq_1 x0xb \vdash bxq_1 0xb \vdash bxxq_2 xb \vdash bxxxq_2 b \\ &\vdash bxxq_4 xb \vdash bxq_4 xxb \vdash bq_4 xxxb \vdash q_4 bxxxb \\ &\vdash bq_1 xxxb \vdash bxq_1 xxb \vdash bxxq_1 xb \vdash bxxxq_1 b \\ &\vdash bxxxbq_f. \end{aligned}$$

Hence M accepts w.

Also note that M always halts. If M reaches q_f, the input string w is accepted by M. If M reaches q_t, w is not accepted by M; in this case M halts in the trap state.

EXAMPLE 9.17

Let $M = (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, b\}, \delta, q_0, \{q_2\})$

where δ is given by

$$\delta(q_0, 0) = (q_1, 1, R) \qquad (R_1)$$

$$\delta(q_1, 1) = (q_0, 0, R) \qquad (R_2)$$

$$\delta(q_1, b) = (q_2, b, R) \qquad (R_3)$$

Find $T(M)$.

Solution

Let $w \in T(M)$. As $\delta(q_0, 1)$ is not defined, w cannot start with 1. From (R_1) and (R_2), we can conclude that M starts from q_0 and comes back to q_0 after reaching 01.

So, $q_0(01)^n \vdash^* (10)^n q_0$. Also, $q_0 0b \vdash 1q_1 b \vdash 1bq_2$.

So, $(01)^n0 \in T(M)$. Also, $(01)^n0$ is the only string that makes M move from q_0 to q_2. Hence, $T(M) = \{(01)^n0 \mid n \geq 0\}$.

SELF-TEST

Choose the correct answer to Questions 1–10:

1. For the standard TM:
(a) $\Sigma = \Gamma$
(b) $\Gamma \subseteq \Sigma$
(c) $\Sigma \subseteq \Gamma$
(d) Σ is a proper subset of Γ.

2. In a standard TM, $\delta(q, a)$, $q \in Q$, $a \in \Gamma$ is
(a) defined for all $(q, a) \in Q \times \Gamma$
(b) defined for some, not necessarily for all $(q, a) \in Q \times \Gamma$
(c) defined for no element (q, a) of $Q \times \Gamma$
(d) a set of triples with more than one element.

3. If $\delta(q, x_i) = (p, y, L)$, then
(a) $x_1x_2 \ldots x_{i-1}qx_i \ldots x_n \vdash x_1x_2 \ldots x_{i-2}px_{i-1}yx_{i+1} \ldots x_n$
(b) $x_1x_2 \ldots x_{i-1}qx_i \ldots x_n \vdash x_1x_2 \ldots x_{i-1}ypx_{i+1} \ldots x_n$
(c) $x_1x_2 \ldots x_{i-1}qx_i \ldots x_n \vdash x_1 \ldots x_{i-3}px_{i-2}x_{i-1}yx_{i+1} \ldots x_n$
(d) $x_1x_2 \ldots x_{i-1}qx_i \ldots x_n \vdash x_1 \ldots x_{i+1}pyx_{i+2} \ldots x_n$

4. If $\delta(q, x_i) = (p, y, R)$, then
(a) $x_1x_2 \ldots x_{i-1}qx_i \ldots x_n \vdash x_1x_2 \ldots x_{i-1}ypx_{i+1} \ldots x_n$
(b) $x_1x_2 \ldots x_{i-1}qx_i \ldots x_n \vdash x_1x_2 \ldots x_ipx_{i+1} \ldots x_n$
(c) $x_1x_2 \ldots x_{i-1}qx_i \ldots x_n \vdash x_1x_2 \ldots x_{i-1}px_ix_{i+1} \ldots x_n$
(d) $x_1x_2 \ldots x_{i-1}qx_i \ldots x_n \vdash x_1x_2 \ldots x_{i-1}ypx_{i+1} \ldots x_n$

5. If $\delta(q, x_1) = (p, y, L)$, then
(a) $qx_1x_2 \ldots x_n \vdash pyx_2 \ldots x_n$
(b) $qx_1x_2 \ldots x_n \vdash ypx_2 \ldots x_n$
(c) $qx_1x_2 \ldots x_n \vdash pbx_1 \ldots x_n$
(d) $qx_1x_2 \ldots x_n \vdash pbx_2 \ldots x_n$

6. If $\delta(q, x_n) = (p, y, R)$, then
(a) $x_1 \ldots x_{n-1}qx_n \vdash pyx_2x_3 \ldots x_n$
(b) $x_1 \ldots x_{n-1}qx_n \vdash^* pyx_2x_3 \ldots x_n$
(c) $x_1 \ldots x_{n-1}qx_n \vdash x_1x_2 \ldots x_{n-1}ypb$
(d) $x_1 \ldots x_{n-1}qx_n \vdash^* x_1x_2 \ldots x_{n-1}ypb$

7. For the TM given in Example 9.6:
(a) $q_01b11 \vdash^* bq_f11bb1$
(b) $q_01b11 \vdash bq_f11bb1$
(c) $q_01b11 \vdash 1q_0b111$
(d) $q_01b11 \vdash q_2b11bb1$

8. For the TM given in Example 9.4:
   (a) 011 is accepted by M
   (b) 001 is accepted by M
   (c) 00 is accepted by M
   (d) 0011 is accepted by M.
9. For the TM given in Example 9.5:
   (a) 1 is accepted by M
   (b) 11 is accepted by M
   (c) 111 is accepted by M
   (d) 11111 is accepted by M
10. In a standard TM $(Q, \Sigma, \Gamma, \delta, q_0, b, F)$ the blank symbol b is
   (a) in $\Sigma - \Gamma$
   (b) in $\Gamma - \Sigma$
   (c) $\Gamma \cap \Sigma$
   (d) none of these

EXERCISES

9.1 Draw the transition diagram of the Turing machine given in Table 9.1.

9.2 Represent the transition function of the Turing machine given in Example 9.2 as a set of quintuples.

9.3 Construct the computation sequence for the input $1b11$ for the Turing machine given in Example 9.5.

9.4 Construct the computation sequence for strings 1213, 2133, 312 for the Turing machine given in Example 9.8.

9.5 Explain how a Turing machine can be considered as a computer of integer functions (i.e. as one that can compute integer functions; we shall discuss more about this in Chapter 11).

9.6 Design a Turing machine that converts a binary string into its equivalent unary string.

9.7 Construct a Turing machine that enumerates $\{0^n1^n \mid n \geq 1\}$.

9.8 Construct a Turing machine that can accept the set of all even palindromes over $\{0, 1\}$.

9.9 Construct a Turing machine that can accept the strings over $\{0, 1\}$ containing even number of 1's.

9.10 Design a Turing machine to recognize the language $\{a^nb^nc^m \mid n, m \geq 1\}$.

9.11 Design a Turing machine that can compute proper subtraction, i.e. $m \dot{-} n$, where m and n are positive integers. $m \dot{-} n$ is defined as $m - n$ if $m > n$ and 0 if $m \leq n$.