15CS54
Automata Theory and Computability
(CBCS Scheme)

Module-4: Properties of CFL, Turing Machine

Contents

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

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.

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, Σ, R, S). *Upper-bound:* 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 every regular language is context free (*lower-bound)*. Also, 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.

2. Closure Theorems for Context-Free Languages

Unfortunately, there are fewer closure theorems than regular languages.

The context-free languages are closed under:

- Union
- Concatenation
- Kleene star
- Reverse
- Letter substitution

TechJourney.in

Proof: The context-free languages are closed under Union

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 $G = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S \rightarrow S_1, S \rightarrow S_2\}, S)$.

Proof: The context-free languages are closed under Concatenation

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 $G = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S \rightarrow S_1 S_2\}, S)$.

Proof: The context-free languages are closed under Kleene Star

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 $G = (V_1 \cup \{S\}, \Sigma_1, R_1 \cup \{S \rightarrow \varepsilon, S \rightarrow S S_1\}, S)$.

Proof: The context-free languages are closed under Reverse

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 not closed under intersection, complement or difference.

Proof: The context-free languages are <u>not closed</u> under intersection:

The proof is by counterexample. Let:

$L_1 = \{a^n b^n c^m : n, m \geq 0\}$ /* equal a's and b's. $L_2 = \{a^m b^n c^n : n, m \geq 0\}$ /* 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: $L = L_1 \cap L_2 = \{a^n b^n c^n : n \geq 0\}$ which not a CFL. (we prove this using pumping theorem ; discussed in section 3)

Proof: The context-free languages are <u>not closed</u> under complement:

Closure under complement implies closure under intersection, since:

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

The context-free languages are closed under union, so if they were closed under complement, they would be closed under intersection (which they are not).

Proof: The context-free languages are <u>not closed</u> 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.

lourney.in

Closure under intersection/difference 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:
  $(((q_1, q_2), \varepsilon, \beta), ((p_1, q_2), \gamma))$.

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.

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.

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

1. Exhibit a context-free grammar for L.
2. Exhibit a PDA for L.

The third method is

3. Use the closure properties of context-free languages.

3. The Pumping Theorem for Context-Free Languages

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 $\leq b\,(b^n) = b^{n+1} = b^h$.

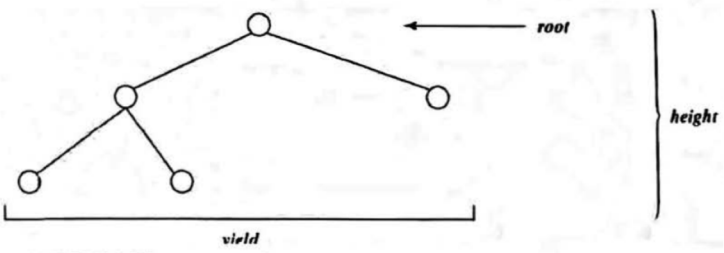


FIGURE 13.1 The structure of a parse tree.

Pumping Theorem

Statement: If L is a context-free language, then $\exists k \geq 1$, such that $\forall$ strings $w \in L$, where $|w| \geq k$, $\exists\ u, v, x, y, z$, such that: $w = uvxyz$, $vy \neq \varepsilon$, $|vxy| \leq k$, and $\forall q \geq 0$, uv^qxy^qz is in L.

Proof: L is generated by some CFG G = (V, Σ, R, S) with n nonterminal symbols and branching factor b. The longest string that can be generated by G with no repeated nonterminals in the resulting parse tree has length bn.

Let k be b^{n+1}. 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 $|w| \geq k$.

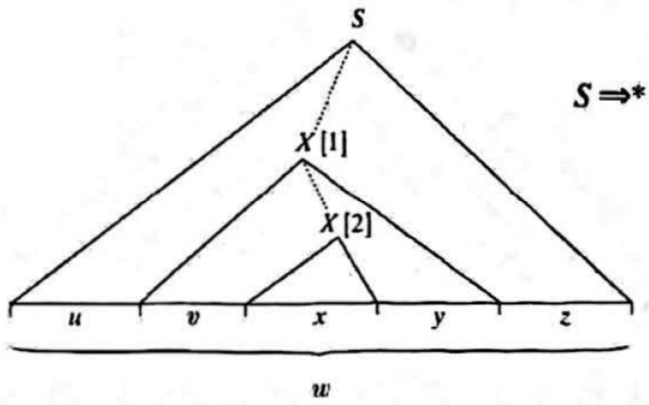
Let T be any smallest parse tree for w. T must have height at least $n + 1$. Choose some path in T of length at least $n + 1$.

Let X be the bottom-most repeated nonterminal along that path. Then w can be rewritten as $uvxyz$. 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.

$vy \neq \varepsilon$ 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.

uxz must be in L because $rule_2$ could have been used immediately at [1]. For any $q \geq 1$, uv^qxy^qz 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 he pumpable. If there is even one long string in L that is not pumpable then L is not context-free.



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

Regular vs CF Pumping Theorems

Similarities:

- We choose w, the string to be pumped.
- We choose a value for q that shows that w isn't pumpable.
- We may apply closure theorems before we start.

Differences:

- Two regions, v and y, must be pumped in tandem.
- We don't know anything about where in 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.

Prove that L = { $a^n b^n c^n$, n≥0} not context free

Let L = { $a^n b^n c^n$, $n \geq 0$}

We use the pumping theorem to show that L is not CFL. If it were, then there would exist some *k* such that any string *w,* where |w| ≥ *k,* must satisfy the conditions of the theorem. We show one string *w* that does not.

Let $w = a^k b^k c^k$, 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 ≠ ε, |vxy|≤ k and ∀ $q \geq 0$, $uv^q xy^q z$ 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 he in L (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 almost 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 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.

The Language of Strings with n^2 a's is not CFL

Proof: Let L = $\{a^{n^2} : n \geq 0\}$.

We use the pumping theorem to show that L is not CFL. If it were, then there would exist some *k* such that any string *w,* where |w| ≥ *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* ≠ *ε. lvxy*|≤ *k.* and ∀ $q \geq 0$, $uv^q xy^q z$ is in L. We show that no such *u, v, x, y,* and *z* exist.

Since *w* contains only a's, *vy* = a^p, for some nonzero *p*. Set *q* to 2. The resulting string, s = 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 $(k^2 + 1)^2 = k^4 + 2k^2 + 1$ 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.

TechJourney.in

Dividing the String w Into Regions

Prove that L = {$a^n b^m a^n$: m, n ≥ 0, m≥n} is not context free.

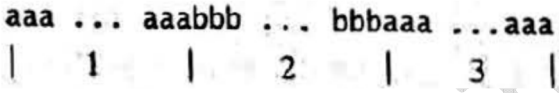
Let L = {$a^n b^m a^n$: m,n ≥ 0, m≥n}.

We use the pumping theorem to show that L is not CFL. If it were, then there would exist some *k* such that any string *w,* where |w| ≥ *k,* must satisfy the conditions of the theorem. We show one string *w* that does not.

Let w = $a^k b^k c^k$

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 ≠ ε. lvxy|≤ k. and ∀ q ≥ 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 three regions as follows:



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* in region 2. 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* 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* 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| ≤ *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.

Techjourney.in

Prove L = {wcw, w is in {a,b}* } if not CFL

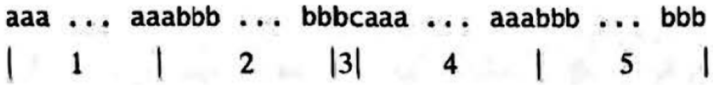
Proof: Let L = {wcw; w belongs to {a,b}* }.

We use the pumping theorem to show that L is not CFL. If it were, then there would exist some *k* such that any string *w,* where |w| ≥ *k,* must satisfy the conditions of the theorem. We show one string *w* that does not.

Let w = $a^k b^k c a^k b^k$

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 ≠ ε. lvxy|≤ k. and ∀ q ≥ 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 three regions as follows:



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 L.
- 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 L.
- If either v or y overlaps region 1, then set q to 2. 1n 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| ≤ 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| ≤ 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.

Using the pumping theorem in conjunction with the closure Properties

Prove WW = {*ww, w* ∈ {a. b }*} is not context free

If WW were context-free, then *L′ = WW* ∩ *a*b*a*b** would also be context free. But it isn't, (a*b*a*b* is a regular language).

We use the pumping theorem to show that L' is not CFL. If it were, then there would exist some *k* such that any string *w,* where |w| ≥ *k,* must satisfy the conditions of the theorem. We show one string *w* that does not.

Let w = $a^k b^k c a^k b^k$

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 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.

Techjourney.in

A Simple Arithmetic Language is Not Context-Free

Proof:

Let $L = \{x \# y = z : x, y, z \in \{0,1\}^*$ and, if x, y and z 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 $10^k \cdot 1^k = 1^k0^k$.

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.

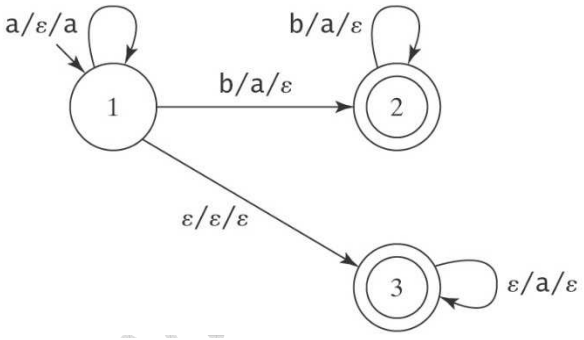
4. Deterministic Context-Free Languages

A PDA M is **deterministic** iff: i) ΔM contains no pairs of transitions that compete with each other, and ii) Whenever M is in an accepting configuration it has no available moves.

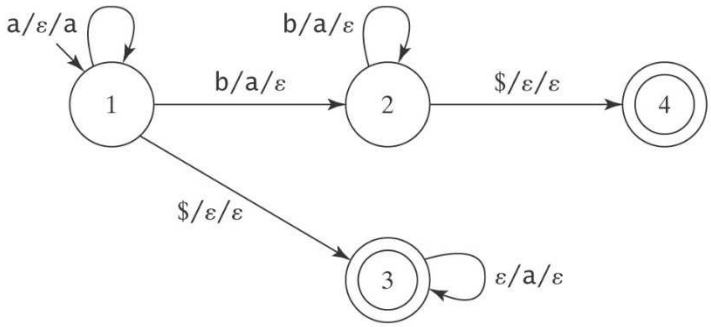
A language L is deterministic context-free iff **L$** can be accepted by some deterministic PDA.

Let L = a* U { a"b": 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 $:



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

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.

Nonclosure Under Union

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

Proof: We show a counterexample:

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

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

$$\text{Let } L' = L_1 \cup L_2.$$

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

$$\text{Let } L'' = \neg L'.$$

$$= \{a^i b^j c^k : i, j, k \geq 0 \text{ and } i = j = k\} \cup \{w \in \{a, b, c\}^* : \text{the letters are out of order}\}.$$

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

$$= \{a^n b^n c^n : n \geq 0\}.$$

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.

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.

6. A Hierarchy within the Class of Context-Free language

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.

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 example.

Let $L = \{a^i b^j c^k, i \neq j \text{ or } j \neq k\}$. L is CF. If L is DCFL then so is:

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

But then so is: $L'' = L' \cap a^*b^*c^* = \{a^n b^n c^n, n \geq 0\}$.

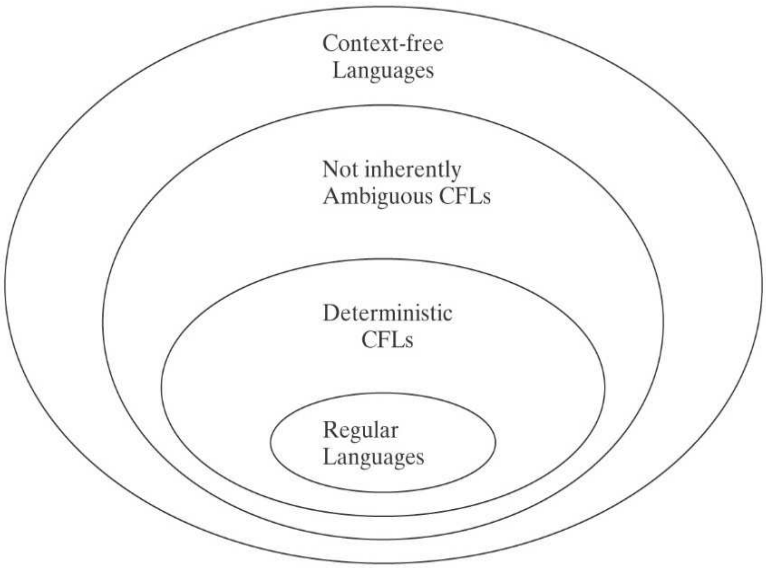
But it isn't. So, L is context-free but not deterministic context-free.

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 every string that is also in $\{ a^n b^n c^n : n \geq 0\}$ 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 $L_2 = \{ a^n b^n c^m d : n, m \geq 0\} \cup \{ a^n b^m c^m d : n, m \geq 0\}$. 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 ore 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.



Techjourney.in

- There exist deterministic context-free languages that are not regular. These languages are in the innermost donut in the figure. One example is $A^nB^n = \{a^nb^n : n \geq 0\}$.
- 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.
  - $\{a^nb^nc^md : n, m \geq 0\} \cup \{a^nb^mc^me : n, m \geq 0\}$.
- There exist languages that are in the outer donut because they are inherently ambiguous. Two examples are:
  - $\{a^ib^jc^k : i, j, k \geq 0 \text{ and } ((i = j) \text{ or } (j = k))\}$.
  - $\{a^ib^jc^k : i, j, k \geq 0 \text{ and } ((i \neq j) \text{ or } (j \neq k))\}$.

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.

6. Decidable questions

6.1 Membership

Given a context free language L and a string w, there exists a decision procedure that answers the questions, "is w in L?". There are two approaches:

- Find a context-free grammar to generate it
- Find a PDA to accept it

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:

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.

decideCFLusingGrammar(L: CFL, w: string) =

1. If given a PDA, build G so that $L(G) = L(M)$.
2. If $w = \varepsilon$ then if S_G is nullable then accept, else reject.
3. If $w \neq \varepsilon$ then:
   3.1 Construct G' in Chomsky normal form such that $L(G') = L(G) - \{\varepsilon\}$.
   3.2 If G derives w, it does so in $2 \cdot |w| - 1$ steps. Try all derivations in G of $2|w| - 1$ steps. If one of them derives w, accept. Otherwise reject.

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 - { ε } and that has no ε-transitions. Then we show that every PDA with no ε-transitions is guaranteed to halt.

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.

cfgtoPDAnoeps(G: context-free grammar) =

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

Step 2 can be implemented as follows.

$M = (\{p, q\}, \Sigma, V, \Delta, p, \{q\})$, 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 $((q, c, X), (q, s_2 \ldots s_n))$.

TechJourney.in

Halting Behavior of PDAs without ε-Transitions

A PDA Without ε-Transitions must halt.

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.

The final algorithm can be listed as follows.

decideCFLusingPDA(L: CFL, w: string) =

1. If L is specified as a PDA, use *PDAtoCFG* 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 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 a PDA M such that $L(M) = L(G')$ and M' has no ε-transitions.
   4.3 All paths of M are guaranteed to halt within a finite number of steps. So, run M on w. Accept if it accepts and reject otherwise.

6.2. 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. $L(G) = \emptyset$ 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 $L = \emptyset$?"

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.

7. The Undecidable Questions

- 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 a context free language L_1 and L_2,, is $L_1 = L_2$?
- Given a context free language L_1 and L_2, is $L_1 \subseteq L_2$?
- Given a context free language 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?

8. Turing Machine

Turing formulated a model of algorithm or computation, that is widely accepted. 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.

For formalizing computability, Turing assumed that, while computing, a person writes symbols on a one-dimensional tape which is 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 light of the present cell. With these observations in mind, Turing proposed his 'computing machine.'

8.1 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 below.

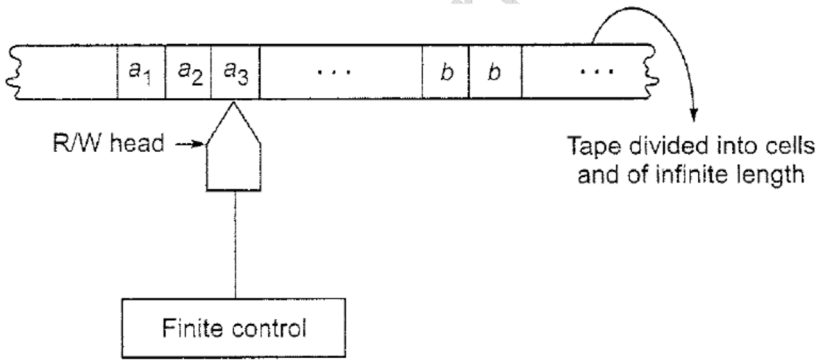


Fig. 9.1 Turing machine model.

Each cell can store only one symbol. The input to and the output from the finite state automaton are affected 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: A Turing machine M is a 7-tuple, namely **(Q, Σ, Γ, δ, q_0, b, F),** where

- Q is a finite nonempty set of states.
- Γ is a finite nonempty set of tape symbols,
- b is the blank.
- Σis a nonempty set of input symbols and is a subset of Γ rand b ∉ Σ.
- 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.
- $q_0 \in Q$ is the initial state, and
- $F \subseteq Q$ is the set of final states.

8.2. Representation of Turing Machines

We can describe a Turing machine employing

(i) Transition diagram (transition graph).
(ii) Instantaneous descriptions using move-relations.
(iii) Transition table

8.2.1. 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. An ID of a Turing machine is defined in terms of the entire input string and the current state.

Definition: 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: A snapshot of Turing machine is shown in. Obtain the instantaneous description.

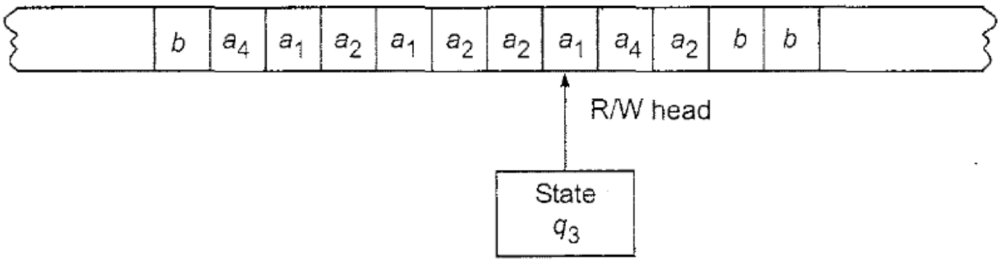


Fig. 9.2 A snapshot of Turing machine.

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 al 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 figure given below.

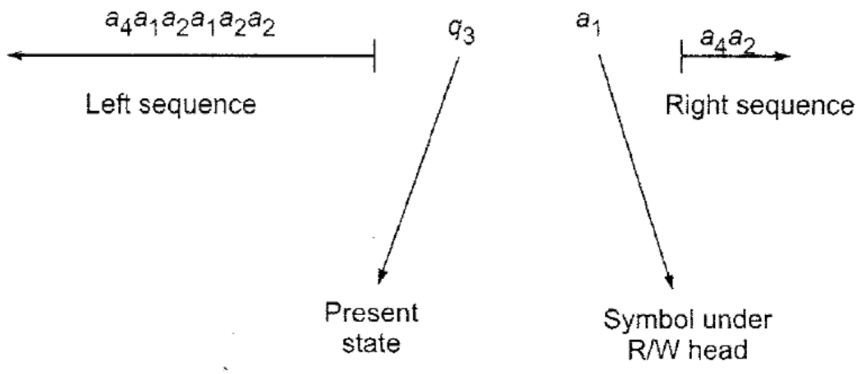


Fig. 9.3 Representation of ID.

Note: (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.

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.

8.2.2. 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 αβγ under the α-column and in the *q-row*. So if we get αβγ 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)	$0Lq_2$		

8.2.3. Representation by Transition Diagram

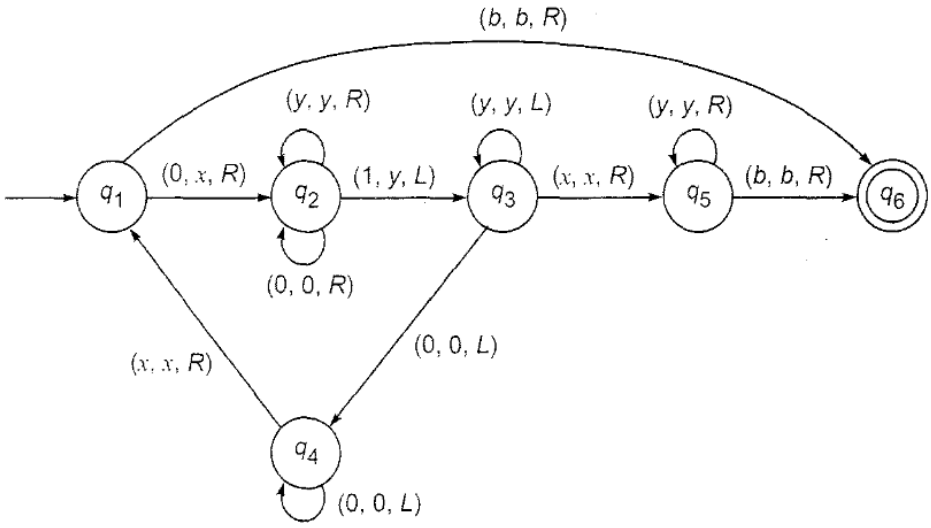
The states are represented by vertices. Directed edges are used to represent transition of states. The labels are triples of the form (α, β, γ), where $\alpha, \beta, \in \Gamma$ and $\gamma \in \{L, R\}$. When there is a directed edge from q_i to q_j with label (α, β, γ), it means that

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

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 $\bigcirc$.

Example:



8.3. 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 $q_0w \vdash^* \alpha_1 p \alpha_2$ 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.

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

Present state	Tape symbol				
	0	1	x	y	b
→ 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) $q_10011 \vdash xq_2011 \vdash x0q_211 \vdash xq_30y1 \vdash q_4x0y1 \vdash xq_10y1.$

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

$\vdash xxyq_5y \vdash xxyyq_5b \vdash xxyybq_6$

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.

8.4 Design of Turing Machines

The basic guidelines for designing a Turing machine is given below.

1. 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.
2. 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-1:

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.

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

Transition diagram

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-2:

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 Figure.

Fig. 9.6 Input and output tapes.

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:

1. The separating symbol b is found and replaced by 1.
2. The rightmost 1 is found and replaced by a blank b.
3. The R/W head returns to the starting position.

We can construct the transition table as follows

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)	—	—

The transition diagram is given here.

A computation for **11b111** is illustrated below.

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

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

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

Example-3: Design TM that accepts $\{0^n1^n \mid n \geq 1\}$

Solution:

We require the following moves:

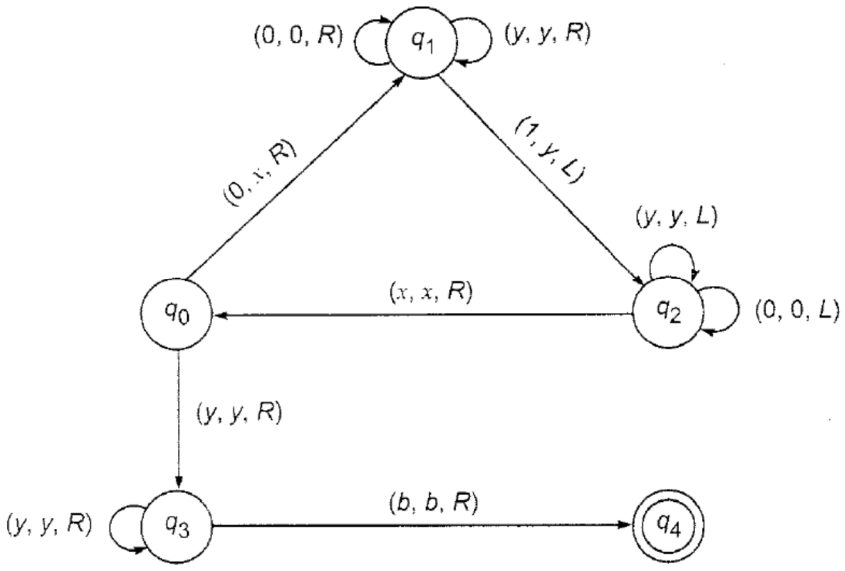
(a) If the leftmost symbol in the given input string w is 0, replace it by x and move right till we encounter a leftmost 1 in w. Change it to y and move backwards.
(b) Repeat (a) with the leftmost 0. If we move back and forth and no 0 or 1 remains, move to a final state.
(c) For strings not in the form 0^n1^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.

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

Hence 0011 is accepted by M.

$$q_0010 \vdash xq_110 \vdash q_2xy0 \vdash xq_0y0 \vdash xyq_30$$

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

Example-3: Design TM that accepts $\{1^n2^n3^n \mid n \geq 1\}$

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.

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)				

Transition diagram

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

$$q_1112233 \vdash bq_212233 \vdash blq_22233 \vdash b1bq_3233 \vdash b1b2q_333$$

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

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

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

Thus,

$$q_1112233 \vdash^* q_7bbbbbb$$

It can be seen from the table that strings other than those of the form $0^n1^n2^n$ are not accepted.

Exercise: Compute the computation sequence for strings like 1223, 1123, 1233 and then see that these strings are rejected by *M*.

Techjourney.in

8.5 Description of Turing Machines

In the examples discussed so far, the transition function δ was described as a partial function δ: Q x Γ→ Q x Γ x {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. 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.

8.6 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 earlier is called the standard Turing machine.

8.6.1. 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.

8.6.2. Storage in the State

We are using a state, whether it is of a FSM 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 x Γ.

Example: 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 $\overline{a}$ ($\overline{a} = 0$ if $a = 1$ and $\overline{a} = 1$ 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).

8.6.3. 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.

8.6.4. 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 affected 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: 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 $q_0 0^m 10^n 1$. On seeing 0, the following moves take place (q_6 is a state of M). $q_0 0^m 10^n 1 \vdash bq_6 0^{m-1} 10^n 1 \vdash^* b0^{m-1} q_6 10^n 1 \vdash b0^{m-1} 1q_1 0^n 1$. q_1 is the initial state of COPY. The TM M_1 performs the subroutine COPY. The following moves take place for M_1: $q_1 0^n 1 \vdash 2q_2 0^{n-1} 1 \vdash^* 20^{n-1} 1q_3 b \vdash 20^{n-1} q_3 10 \vdash^* 2q_1 0^{n-1} 10$. 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^{mn} 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_8bR	—	—	—
q_6	q_60R	q_11R	—	—
q_5	q_70L	—	—	—
q_7	—	q_81L	—	—
q_8	q_90L	—	—	$q_{10}bR$
q_9	q_90L	—	—	q_0bR
q_{10}	—	$q_{11}bR$	—	—
q_{11}	$q_{11}bR$	$q_{12}bR$	—	—

Transition Diagram

Thus, M performs multiplication of two numbers in unary representation.