Solutions Manual

for

Languages and Machines:

An Introduction to the Theory of Computer Science

Third Edition

Thomas A. Sudkamp

Preface

This solution manual was written to accompany the third edition of *Languages and Machines: An Introduction to the Theory of Computer Science.* It contains complete solutions to approximately 200 exercises from the text, including the "starred" exercises.

Acquiring a thorough background in and mastery of the foundations of computer science is not a spectator sport, active participation is required. This participation takes several forms; putting forth the effort to understand the theorems and their proofs, following the techniques used to solve the problems in the examples, and solving problems on your own. It is the ability to do the latter that exhibits proficiency with the material. These solutions have been provided as a tool to assist students in developing their problem solving skills.

The exercises in this manual were selected to complement the presentation in the text. The solutions illustrate general techniques for solving the particular type of problem or extend the theoretical development of a topic. Understanding the strategies employed in these solutions should provide valuable insights that can be used in solving many of the other exercises in the text.

Thomas Sudkamp Dayton, Ohio

Contents

Chapter 1

Mathematical Preliminaries

5. To prove that $\overline{X} \cap \overline{Y} = \overline{(X \cup Y)}$ requires establishing both of the inclusions $\overline{X} \cap \overline{Y} \subseteq \overline{(X \cup Y)}$ and $\overline{(X \cup Y)} \subseteq \overline{X} \cap \overline{Y}$.

   i) Let $a \in \overline{X} \cap \overline{Y}$. By the definition of intersection, $a \in \overline{X}$ and $a \in \overline{Y}$. Thus $a \notin X$ and $a \notin Y$ or, in terms of union, $a \notin (X \cup Y)$. It follows that $a \in \overline{(X \cup Y)}$.

   ii) Now assume that $a \in \overline{(X \cup Y)}$. Then $a \notin X \cup Y$. This implies that $a \notin X$ and $a \notin Y$. Consequently, $a \in \overline{X} \cap \overline{Y}$.

   Part (i) shows that $\overline{X} \cap \overline{Y}$ is a subset of $\overline{(X \cup Y)}$, while (ii) establishes the inclusion $\overline{(X \cup Y)} \subseteq \overline{X} \cap \overline{Y}$.

   A completely analogous argument can be used to establish the equality of the sets $\overline{(X \cap Y)}$ and $\overline{X} \cup \overline{Y}$.

6. a) The function $f(n) = 2n$ is total and one-to-one. However, it is not onto since the range is the set of even numbers.

   b) The function
   $$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ n - 1 & \text{otherwise} \end{cases}$$
   is total and onto. It is not one-to-one since $f(0) = f(1) = 0$.

   c) The function
   $$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ 0 & \text{if } n = 1 \\ n & \text{otherwise} \end{cases}$$
   is total, one-to-one, and onto but not the identity.

   d) The function
   $$f(n) = \begin{cases} n/2 & \text{if } n \text{ is even} \\ \uparrow & \text{otherwise} \end{cases}$$
   maps the even natural numbers onto the entire set of natural numbers. It is not a total function from **N** to **N**, since it is not defined for odd natural numbers.

13. To prove that $\equiv_p$ is an equivalence relation, we must show that it is reflexive, symmetric, and transitive. This is shown using the same argument given in Example 1.3.1, which explicitly considers the case when $p = 2$.

    i) *Reflexivity:* For every natural number n, $n \bmod p = n \bmod p$.

ii) *Symmetry:* If $n \bmod p = m \bmod p$, then $m \bmod p = n \bmod p$.

iii) *Transitivity:* If $n \bmod p = m \bmod p$ and $m \bmod p = k \bmod p$, then $n \bmod p = k \bmod p$.

The equivalence classes of $\equiv_p$ are the sets consisting of natural numbers that are equal mod p:

$$\begin{aligned}
[0]_{\equiv_p} &= \{0, p, 2p, 3p, \ldots\}\\
[1]_{\equiv_p} &= \{1, p+1, 2p+1, 3p+1, \ldots\}\\
[2]_{\equiv_p} &= \{2, p+2, 2p+2, 3p+2, \ldots\}\\
&\vdots\\
[p-1]_{\equiv_p} &= \{p-1, 2p-1, 3p-1, 4p-1, \ldots\}.
\end{aligned}$$

15. i) *Reflexivity:* To demonstrate reflexivity, we must show that every ordered pair $[m, n]$ is related to itself. The requirement for $[m, n]$ to be related to $[m, n]$ by $\equiv$ is $m+n = n+m$, which follows from the commutativity of addition.

ii) *Symmetry:* If $[m, n] \equiv [j, k]$, then $m+k = n+j$. Again, by commutativity, $j+n = k+m$ and $[j, k] \equiv [m, n]$.

iii) *Transitivity:* $[m, n] \equiv [j, k]$ and $[j, k] \equiv [s, t]$ imply $m + k = n + j$ and $j + t = k + s$. Adding the second equality to the first, we obtain

$$m + k + j + t = n + j + k + s.$$

Subtracting $j + k$ from each side yields $m + t = n + s$, showing that $[m, n] \equiv [s, t]$ as desired.

18. The set of non-negative rational numbers is defined by

$$\{n/m \mid n \in \mathbf{N},\ m \in \mathbf{N} - \{0\}\}$$

A rational number n/m can be represented by the ordered pair $[n, m]$. This representation defines a one-to-one correspondence between the rational numbers and the set $\mathbf{N} \times (\mathbf{N} - \{0\})$. The latter set is known to be countable by Theorem 1.4.4.

22. Diagonalization is used to prove that there are an uncountable number of monotone increasing functions. Assume that the set of monotone increasing functions is countable. Then these functions can be listed in a sequence $f_0, f_1, f_2, \ldots, f_n, \ldots$.

Now we will show that there is at least one (in fact there are infinitely many) monotone increasing function that is not in the listing. Define a function f as follows:

$$\begin{aligned}
f(0) &= f_0(0)\ +\ 1\\
f(i) &= f_i(i)\ +\ f(i-1)
\end{aligned}$$

for $i > 0$. Since $f_i(i) > 0$, it follows that $f(i) > f(i-1)$ for all i and f is monotone increasing.

Clearly $f(i) \neq f_i(i)$ for any i, contradicting the assumption that $f_0, f_1, \ldots, f_n, \ldots$ exhaustively enumerates the monotone increasing functions. Thus the assumption that the monotone increasing functions comprise a countable set leads to a contradiction and we conclude that the set is uncountable.

25. To show that the set of real numbers in the interval $[0, a]$ is uncountable, we first observe that every real number in $(0, 1]$ can be expressed by an infinite decimal $.x_0x_1x_2\ldots x_n\ldots$. With such a representation, the number $\frac{1}{2}$ is represented by both $.50000\ldots$ and $.49999\ldots$. To obtain a

unique representation, we consider only decimal expansions that do not end with an infinite sequence of zeros.

Assume the set of real numbers in $(0, 1]$ is countable. This implies that there is a sequence

$$r_0,\ r_1,\ r_2,\ \ldots\ ,\ r_n,\ \ldots$$

that contains all of the real numbers in the interval $(0, 1]$. Let the decimal expansion of r_n be denoted $.x_{n0}x_{n1}x_{n2}\ldots$. The enumeration given above is used to construct an infinite two-dimensional array, the i^{th} row of which consists of the expansion of r_i.

$$\begin{array}{lllll} r_0 = & x_{00} & x_{01} & x_{02} & \ldots \\ r_1 = & x_{10} & x_{11} & x_{12} & \ldots \\ r_2 = & x_{20} & x_{21} & x_{22} & \ldots \\ \vdots & \vdots & \vdots & \vdots & \end{array}$$

With the unique decimal representation, two numbers are distinct if they differ at any position in the decimal expansion. A real number $r = x_0x_1\ldots$ is defined using the diagonal elements in the array formed by the x_{ii}'s as follows:

$$x_i = \begin{cases} 2 & \text{if } x_{ii} = 1 \\ 1 & \text{otherwise.} \end{cases}$$

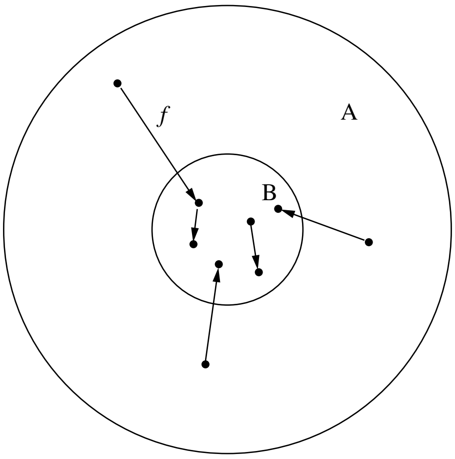
Clearly $r \neq r_i$ for any i since the i^{th} position of r, x_i, is not identical to the i^{th} position of r_i. Therefore the assumption that the enumeration contains all real numbers in $(0, 1]$ fails, and we conclude that the set is uncountable.

28. Before proving the Schröder-Bernstein Theorem, we consider the special case where $card(\mathrm{B}) \leq card(\mathrm{A})$, $card(\mathrm{A}) \leq card(\mathrm{B})$, and $\mathrm{B} \subseteq \mathrm{A}$.

The relationship $card(\mathrm{B}) \leq card(\mathrm{A})$ follows immediately from the inclusion $\mathrm{B} \subseteq \mathrm{A}$ since the identity function $id : \mathrm{B} \to \mathrm{A}$ is a one-to-one function from B into A.

By definition, $card(\mathrm{A}) \leq card(\mathrm{B})$ means there is a one-to-one function f from A into B. We will use f to construct a one-to-one function $h : \mathrm{A} \to \mathrm{B}$ from A onto B. The function h demonstrates that A and B have the same cardinality.

The diagram



illustrates the mapping f. The function f is defined for all elements in A (which includes all elements of B) and the values of f, indicated by the heads of the arrows, must all be in B.

For each $x \in$ A $-$ B, we define the set

$$ch(x) = \{x, f(x), f(f(x))), \ldots, f^i(x), \ldots\}.$$

Every element in $ch(x)$ is in B, except for x itself which is in A $-$ B. Let

$$\mathrm{C} = \bigcup_{x \in \mathrm{\ A-B}} ch(x).$$

Now define the function $h : \mathrm{A} \rightarrow \mathrm{B}$ as follows:

$$h(z) = \begin{cases} f(z), & \text{if } z \in \mathrm{C}; \\ z, & \text{otherwise.} \end{cases}$$

To show that h is a one-to-one, we must prove that $h(x) = h(y)$ implies that $x = y$. There are four cases to consider.

Case 1: $x, y \notin$ C. Then $x = h(x) = h(y) = y$.

Case 2: $x \in$ C and $y \notin$ C. Since $x \in$ C, $h(x) = f(x)$ is in C. But $h(x) = h(y) = y$. This implies that $y \in$ C, which is a contradiction. Thus $h(x)$ cannot equal $h(y)$ in this case.

Case 3: $x \notin$ C and $y \in$ C. Same argument as case 2.

Case 4: $x, y \in$ C. Let f^i denote the composition of f with itself i times, and f^0 denote the identity function. The proof uses the fact that the composition of one-to-one functions is one-to-one. Although you will be familiar with functional composition from previous mathematical studies, a description and formal definition are given in Section 9.4 of the text if you need a reminder.

Since x and y are both in C, $x = f^m(s)$ and $y = f^n(t)$ for some $s, t \in$ A $-$ B. Then

$$h(x) = f(f^m(s)) = h(y) = f(f^n(t)).$$

If $m = n$, then $s = t$ and $x = y$ and we are done. Assume that $m > n$. Then $f^{m-n}(s) = t$. Applying the function f^n to both sides we get $f^m(s) = f^n(t)$, or equivalently, $x = y$. A similar argument shows $x = y$ when $m < n$.

We now show that h maps A onto B. For each $x \in$ B but not in C, $h(x) = x$ and x is covered by the mapping. If $x \in$ B and $x \in$ C, then $x = f(t)$ for some $t \in$ C because each element in C is either in A $-$ B or obtained by the result of applying f to an element in C. Consequently, each element of B is 'hit' by the mapping h. Because h is a one-to-one function from A to B, we conclude that $card$(A) $= card$(B).

To prove the Schröder-Bernstein Theorem in its generality, we must show that $card$(X) $\leq$ $card$(Y) and $card$(Y) $\leq card$(X) implies $card$(X) $= card$(Y) for arbitrary sets X and Y. By the assumption, there are one-to-one functions $f : \mathrm{X} \rightarrow \mathrm{Y}$ and $g : \mathrm{Y} \rightarrow \mathrm{X}$. Let

$$Im(\mathrm{Y}) = \{x \in \mathrm{X} \mid x = g(y) \text{ for some } y \text{ in } \mathrm{Y}\}$$

be the image of Y in X under g. Now

- Im(Y) is a subset of X,
- Im(Y) has the same cardinality as Y (g is one-to-one and onto), and
- the composition $f \circ g$ is a one-to-one mapping from X into Im(Y).

By the preceding result, $card(\mathrm{X}) = card(Im(\mathrm{Y}))$. It follows that $card(\mathrm{X}) = card(\mathrm{Y})$ by Exercise 27.

31. Let L be the set of the points in $\mathbf{N} \times \mathbf{N}$ on the line defined by $n = 3 \cdot m$. L can be defined recursively by

    Basis: $[0,\ 0] \in \mathrm{L}$.

    Recursive step: If $[m,\ n] \in \mathrm{L}$, then $[s(m),\ s(s(s(n)))] \in \mathrm{L}$.

    Closure: $[m,\ n] \in \mathrm{L}$ only if it can be obtained from $[0, 0]$ using finitely many applications of the recursive step.

33. The product of two natural numbers can be defined recursively using addition and the successor operator s.

    Basis: if $n = 0$ then $m \cdot n = 0$

    Recursive step: $m \cdot s(n) \ = \ m \ + \ (m \cdot n)$

    Closure: $m \cdot n \ = \ k$ only if this equality can be obtained from $m \cdot 0 = 0$ using finitely many applications of the recursive step.

37. The set $\mathcal{F}$ of finite subsets of the natural numbers can be defined recursively as follows:

    Basis: $\emptyset, \{0\} \in \mathcal{F}$

    Recursive step: If $\{n\} \in \mathcal{F}$, then $\{s(n)\} \in \mathcal{F}$.
    If X, $\mathrm{Y} \in \mathcal{F}$, then $\mathrm{X} \cup \mathrm{Y} \in \mathcal{F}$.

    Closure: A set X is in $\mathcal{F}$ only if it can be obtained from the basis elements by a finite number of applications of the recursive step.

    The first rule in the recursive step generates all sets containing a single natural number. The second rule combines previously generated sets to obtain sets of larger cardinality.

39. We prove, by induction on n, that

    $$\sum_{i=0}^{n} 2^i \ = \ 2^{n+1} - 1$$

    Basis: The basis is $n = 0$. We explicitly show that the equality holds for this case.

    $$\sum_{i=0}^{0} 2^i \ = \ 2^0 \ = \ 1 \ = \ 2^1 - 1$$

    Inductive hypothesis: Assume, for all values $k = 1, 2, \ldots, n$, that

    $$\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$$

    Inductive step: We need to show that

    $$\sum_{i=0}^{n+1} 2^i \ = \ 2^{(n+1)+1} - 1$$

    To utilize the inductive hypothesis, the summation is decomposed into the sum of the first n powers of 2 and 2^{n+1}.

$$\begin{aligned}\sum_{i=0}^{n+1} 2^i &= \sum_{i=0}^{n} 2^i + 2^{n+1} \\ &= 2^{n+1} - 1 + 2^{n+1} && \text{(inductive hypothesis)} \\ &= 2 \cdot 2^{n+1} - 1 \\ &= 2^{(n+1)+1} - 1\end{aligned}$$

43. The set R of nodes reachable from a given node x in a directed graph is defined recursively using the adjacency relation A.

Basis: $x \in \mathrm{R}$.

Recursive step: If $y \in \mathrm{R}$ and $[y, z] \in \mathrm{A}$, then $z \in \mathrm{R}$.

Closure: $y \in \mathrm{R}$ only if y can be obtained from x by finitely many applications of the recursive step.

46. a) The depth of the tree is 4.

b) The set of ancestors of x_{11} is $\{x_{11}, x_7, x_2, x_1\}$. Recall that by our definition is node is an ancestor of itself, which is certainly not the case in family trees.

c) The minimal common ancestor of x_{14} and x_{11} is x_2; of x_{15} and x_{11} is x_1.

d) The subtree generated by x_2 is comprised of the arcs $[x_2, x_5]$, $[x_2, x_6]$, $[x_2, x_7]$, $[x_5, x_{10}]$, $[x_7, x_{11}]$, and $[x_{10}, x_{14}]$.

e) The frontier is the set $\{x_{14}, x_6, x_{11}, x_3, x_8, x_{12}, x_{15}, x_{16}\}$.

48. Induction on the depth of the tree is used to prove that a complete binary tree T of depth n has $2^{n+1} - 1$ nodes. Let $\mathit{nodes}(\mathrm{T})$ and $\mathit{leaves}(\mathrm{T})$ denote the number of nodes and leaves in a tree T.

Basis: The basis consists of trees of depth zero; that is, trees consisting solely of the root. For any such tree T, $\mathit{nodes}(\mathrm{T}) = 1 = 2^1 - 1$.

Inductive hypothesis: Assume that every complete binary tree T of depth k, $k = 0, \ldots, n$, satisfies $\mathit{nodes}(\mathrm{T}) = 2^{k+1} - 1$.

Inductive step: Let T be a complete binary tree of depth $n + 1$, where $n \geq 0$. We need to show that $\mathit{nodes}(\mathrm{T}) = 2^{(n+1)+1} - 1$. T is obtained by adding two children to each leaf of a complete binary tree T' of depth n. Since T' is complete binary, it is also strictly binary and

$$\mathit{leaves}(\mathrm{T}') = (\mathit{nodes}(\mathrm{T}') + 1)/2$$

by Exercise 47. Thus

$$\begin{aligned}\mathit{nodes}(\mathrm{T}) &= \mathit{nodes}(\mathrm{T}') + 2 \cdot \mathit{leaves}(\mathrm{T}') \\ &= \mathit{nodes}(\mathrm{T}') + 2 \cdot [(\mathit{nodes}(\mathrm{T}') + 1)/2] && \text{(Exercise 47)} \\ &= 2 \cdot \mathit{nodes}(\mathrm{T}') + 1 \\ &= 2 \cdot (2^{n+1} - 1) + 1 && \text{(inductive hypothesis)} \\ &= 2^{(n+1)+1} - 1\end{aligned}$$

Chapter 2

Languages

3. We prove, by induction on the length of the string, that $w = (w^R)^R$ for every string $w \in \Sigma^*$.

   Basis: The basis consists of the null string. In this case, $(\lambda^R)^R = (\lambda)^R = \lambda$ as desired.

   Inductive hypothesis: Assume that $(w^R)^R = w$ for all strings $w \in \Sigma^*$ of length n or less.

   Inductive step: Let w be a string of length $n + 1$. Then $w = ua$ and

$$\begin{aligned}
(w^R)^R &= ((ua)^R)^R && \\
&= (a^R u^R)^R && \text{(Theorem 2.1.6)} \\
&= (au^R)^R && \\
&= (u^R)^R a^R && \text{(Theorem 2.1.6)} \\
&= ua^R && \text{(inductive hypothesis)} \\
&= ua && \\
&= w &&
\end{aligned}$$

8. Let L denote the set of strings over $\Sigma = \{a, b\}$ that contain twice as many a's as b's. The set L can be defined recursively as follows:

   Basis: $\lambda \in$ L

   Recursive step: If $u \in$ L and u can be written $xyzw$ where $x, y, z, w \in \Sigma^*$ then

   i) $xayazbw \in$ L,
   ii) $xaybzaw \in$ X, and
   iii) $xbyazaw \in$ X.

   Closure: A string u is in L only if it can be obtained from λ using a finite number of applications of the recursive step.

   Clearly, every string generated by the preceding definition has twice as many a's as b's. A proof by induction on the length of strings demonstrates that every string with twice as many a's as b's is generated by the recursive definition. Note that every string in L has length divisible by three.

   Basis: The basis consists of the null string, which satisfies the relation between the number of a's and b's.

   Inductive hypothesis: Assume that all strings with twice as many a's as b's of length k, $0 \leq k \leq n$, are generated by the recursive definition.

Inductive step: Let u be a string of length $n+3$ with twice as many a's as b's. Since u contains at least three elements, x can be written $xayazbw$, $xaybzaw$, or $xbyazaw$ for some $x, y, w, z \in \Sigma^*$. It follows that $xyzw$ has twice as many a's as b's and, by the inductive hypothesis, is generated by the recursive definition. Thus, one additional application of the recursive step produces u from $xyzw$.

12. Let P denote the set of palindromes defined by the recursive definition and let $\text{W} = \{w \in \Sigma^* \mid w = w^R\}$. Establishing that P = W requires demonstrating that each of the sets is a subset of the other.

We begin by proving that P $\subseteq$ W. The proof is by induction on the number of applications of the recursive step in the definition of palindrome required to generate the string.

Basis: The basis consists of strings of P that are generated with no applications of the recursive step. This set consists of λ and a, for every $a \in \Sigma$. Clearly, $w = w^R$ for every such string.

Inductive hypothesis: Assume that every string generated by n or fewer applications of the recursive step is in W.

Inductive step: Let u be a string generated by $n+1$ applications of the recursive step. Then $u = awa$ for some string w and symbol $a \in \Sigma$, where w is generated by n applications of the recursive step. Thus,

$$\begin{aligned} u^R &= (awa)^R \\ &= a^R w^R a^R && \text{(Theorem 2.1.6)} \\ &= aw^R a \\ &= awa && \text{(inductive hypothesis)} \\ &= u \end{aligned}$$

and $u \in$ W.

We now show that W $\subseteq$ P. The proof is by induction on the length of the strings in W.

Basis: If $length(u) = 0$, then $w = \lambda$ and $\lambda \in$ P by the basis of the recursive definition. Similarly, strings of length one in W are also in P.

Inductive hypothesis: Assume that every string $w \in$ W with length n or less is in P.

Inductive step: Let $w \in$ W be a string of length $n+1$, $n \geq 1$. Then w can be written ua where $length(u) = n$. Taking the reversal,

$$w = w^R = (ua)^R = au^R$$

Since w begins and ends with with same symbol, it may be written $w = ava$. Again, using reversals we get

$$\begin{aligned} w^R &= (ava)^R \\ &= a^R v^R a^R \\ &= av^R a \end{aligned}$$

Since $w = w^R$, we conclude that $ava = av^R a$ and $v = v^R$. By the inductive hypothesis, $v \in$ P. It follows, from the recursive step in the definition of P, that $w = ava$ is also in P.

13. The language L_2 consists of all strings over $\{a, b\}$ of length four. L_3, obtained by applying the Kleene star operation to L_2, contains all strings with length divisible by four. The null string, with length zero, is in L_3.

    The language $\mathrm{L}_1 \cap \mathrm{L}_3$ contains all strings that are in both L_1 and L_3. By being in L_1, each string must consist solely of a's and have length divisible by three. Since L_3 requires length divisible by four, a string in $\mathrm{L}_1 \cap \mathrm{L}_3$ must consist only of a's and have length divisible by 12. That is, $\mathrm{L}_1 \cap \mathrm{L}_3 = \boldsymbol{(a^{12})^*}$.

15. Since the null string is not allowed in the language, each string must contain at least one a or one b or one c. However, it need not contain one of every symbol. The $+$'s in the regular expression $\boldsymbol{a^+b^*c^* \cup a^*b^+c^* \cup a^*b^*c^+}$ ensure the presence of at least one element in each string in the language.

21. The set of strings over $\{a, b\}$ that contain the substrings aa and bb is represented by

    $$\boldsymbol{(a \cup b)^*aa(a \cup b)^*bb(a \cup b)^* \cup (a \cup b)^*bb(a \cup b)^*aa(a \cup b)^*}.$$

    The two expressions joined by the $\cup$ indicate that the aa may precede or follow the bb.

23. The leading a and trailing cc are explicitly placed in the expression

    $$\boldsymbol{a(a \cup c)^*b(a \cup c)^*b(a \cup c)^*cc}.$$

    Any number of a's and c's, represented by the expression $\boldsymbol{(a \cup c)^*}$, may surround the two b's.

24. At first glance it may seem that the desired language is given by the regular expression

    $$\boldsymbol{(a \cup b)^*ab(a \cup b)^*ba(a \cup b)^* \cup (a \cup b)^*ab(a \cup b)^*ba(a \cup b)^*}.$$

    Clearly every string in this language has substrings ab and ba. Every string described by the preceding expression has length four or more. Have we missed some strings with the desired property? Consider aba, bab, and $aaaba$. A regular expression for the set of strings containing substrings ab and ba can be obtained by adding

    $$\boldsymbol{(a \cup b)^*aba(a \cup b)^* \cup (a \cup b)^*bab(a \cup b)^*}$$

    to the expression above.

26. The language consisting of strings over $\{a, b\}$ containing exactly three a's is defined by the regular expression $\boldsymbol{b^*ab^*ab^*ab^*}$. Applying the Kleene star to the preceding expression produces strings in which the number of a's is a multiple of three. However, $\boldsymbol{(b^*ab^*ab^*ab^*)^*}$ does not contain strings consisting of b's alone; the null string is the only string that does have at least three a's. To obtain strings with no a's, $\boldsymbol{b^*}$ is concatenated to the end of the preceding expression. Strings consisting solely of b's are obtained from $\boldsymbol{(b^*ab^*ab^*ab^*)^*b^*}$ by concatenating λ from $\boldsymbol{(b^*ab^*ab^*ab^*)^*}$ with b's from $\boldsymbol{b^*}$. The regular expression $\boldsymbol{(b^*ab^*ab^*a)b^*}$ defines the same language.

28. In the regular expression

    $$\boldsymbol{(ab \cup ba \cup aba \cup b)^*},$$

    a's followed by a b are produced by the first component, a's preceded by a b by the second component, and two a's "covered" by a single b by the third component. The inclusion of $\boldsymbol{b}$ within the scope of the * operator allows any number additional b's to occur anywhere in the string.

    Another regular expression for this language is $\boldsymbol{((a \cup \lambda)b(a \cup \lambda))^*}$.

31. Every aa in the expression $\boldsymbol{(b \cup ab \cup aab)^*}$ is followed by at least one b, which precludes the occurrence of aaa. However, every string in this expression ends with b while strings in the language may end in a or aa as well as b. Thus the expression

$$\boldsymbol{(b \cup ab \cup aab)^*(\lambda \cup a \cup aa)}$$

represents the set of all strings over $\{a, b\}$ that do not contain the substring aaa.

32. A string can begin with any number of b's. Once a's are generated, it is necessary to ensure that an occurrence of ab is either followed by another b or ends the string. The regular expression $\boldsymbol{b^*(a \cup abb^+)^*(\lambda \cup ab)}$ describes the desired language.

34. To obtain an expression for the strings over $\{a, b, c\}$ with an odd number of occurrences of the substring ab we first construct an expression $\boldsymbol{w}$ defining strings over $\{a, b, c\}$ that do not contain the substring ab,

$$\boldsymbol{w = b^*(a \cup cb^*)^*.}$$

Using $\boldsymbol{w}$, the desired set can be written $\boldsymbol{(wabwab)^*wabw}$.

37. The regular expression $\boldsymbol{(b^*ab^*a)^*b^* \cup (a^*ba^*b)^*a^*ba^*}$ defines strings over $\{a, b\}$ with an even number of a's or an odd number of b's. This expression is obtained by combining an expression for each of the component subsets with the union operation.

38. Exercises 37 and 38 illustrate the significant difference between union and intersection in describing patterns with regular expressions. There is nothing intuitive about a regular expression for this language. The exercise is given, along with the hint, to indicate the need for an algorithmic approach for producing regular expressions. A technique accomplish this will follow from the ability to reduce finite state machines to regular expressions developed in Chapter 6. Just for fun, a solution is

$$\boldsymbol{\Big([aa \cup ab(bb)^*ba] \cup [(b \cup a(bb)^*ba)(a(bb)^*a)^*(b \cup ab(bb)^*a)]\Big)^*}.$$

39. a)

$$\begin{aligned} \boldsymbol{(ba)^+(a^*b^* \cup a^*)} &= \boldsymbol{(ba)^+(a^*)(b^* \cup \lambda)} && \text{(identity 5)} \\ &= \boldsymbol{(ba)^*baa^*(b^* \cup \lambda)} && \text{(identity 4)} \\ &= \boldsymbol{(ba)^*ba^+(b^* \cup \lambda)} \end{aligned}$$

c) Except where noted, each step employs an identity from rule 12 in Table 2.1.

$$\begin{aligned} \boldsymbol{(a \cup b)^*} &= \boldsymbol{(b \cup a)^*} && \text{(identity 5)} \\ &= \boldsymbol{b^*(b \cup a)^*} && \text{(identity 5)} \\ &= \boldsymbol{b^*(a^*b^*)^*} \\ &= \boldsymbol{(b^*a^*)^*b^*} && \text{(identity 11)} \\ &= \boldsymbol{(b \cup a)^*b^*} \\ &= \boldsymbol{(a \cup b)^*b^*} && \text{(identity 5)} \end{aligned}$$

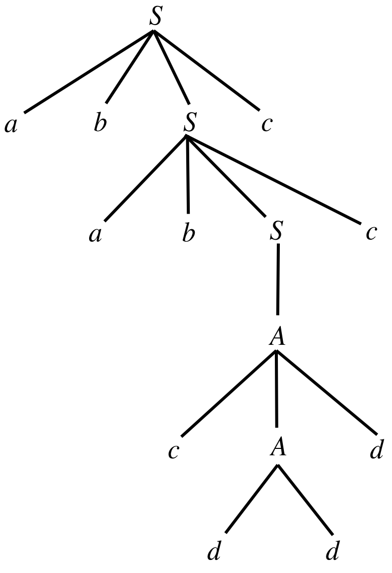
Chapter 3

Context-Free Grammars

1. a)

Derivation	Rule
$S \Rightarrow abSc$	$S \rightarrow abSc$
$\Rightarrow ababScc$	$S \rightarrow abSc$
$\Rightarrow ababAcc$	$S \rightarrow A$
$\Rightarrow ababcAdcc$	$A \rightarrow cAd$
$\Rightarrow ababccddcc$	$A \rightarrow cd$

b) The derivation tree corresponding to the preceding derivation is



c) L(G)= $\{(ab)^n c^m d^m c^n \mid n \geq 0, m > 0\}$

4. a)

$$\begin{aligned} S &\Rightarrow AB \\ &\Rightarrow aAB \\ &\Rightarrow aaB \\ &\Rightarrow aaAB \\ &\Rightarrow aaaB \\ &\Rightarrow aaab \end{aligned}$$

b)

$$S \Rightarrow AB$$

$$\begin{aligned}&\Rightarrow AAB\\&\Rightarrow AAb\\&\Rightarrow Aab\\&\Rightarrow aAab\\&\Rightarrow aaab\end{aligned}$$

c) There are 20 derivations that generate DT. There is no trick to obtaining this answer, you simply must count the possible derivations represented by the tree.

6. a) The S rules $S \to aaSB$ and $S \to \lambda$ generate an equal number of leading aa's and trailing B's. Each B is transformed into one or more b's. Since at least one application of the rule $S \to aaSB$ is necessary to generate b's, strings consisting solely of b's are not in the language. The language of the grammar is $\{(aa)^i b^j \mid i > 0, j \geq i\} \cup \{\lambda\}$.

   c) Repeated applications of the recursive rule $S \to abSdc$ produce a string of the form $(ab)^i S(dc)^i$, $i \geq 0$. The recursion is terminated by an application of the rule $S \to A$. The A rules produce strings of the form $(cd)^j(ba)^j$, $j \geq 0$. The language generated by the rules is $\{(ab)^i(cd)^j(ba)^j(dc)^i \mid i \geq 0, j \geq 0\}$.

8. The language consisting of the set of strings $\{a^n b^m c^{2n+m} \mid m, n > 0\}$ is generated by

$$\begin{aligned}&S \to aScc \mid aAcc\\&A \to bAc \mid bc\end{aligned}$$

For each leading a generated by the S rules, two c's are produced at the end of the string. The rule A rules generate an equal number of b's and c's.

12. A recursive definition of the set of strings over $\{a, b\}$ that contain the same number of a's and b's is given in Example 2.2.3. The recursive definition provides the insight for constructing the rules of the context-free grammar

$$\begin{aligned}&S \to \lambda\\&S \to SaSbS \mid SbSaS \mid SS\end{aligned}$$

that generates the language. The basis of the recursive definition consists of the null string. The first S rule produces this string. The recursive step consists of inserting an a and b in a previously generated string or by concatenating two previously generated strings. These operations are captured by the second set of S rules.

13. An a can be specifically placed in the middle position of a string using the rules

$$A \to aAa \mid aAb \mid bAa \mid bAb \mid a$$

The termination of the application of recursive A rules by the rule $A \to a$ inserts the symbol a into the middle of the string. Using this strategy, the grammar

$$\begin{aligned}&S \to aAa \mid aAb \mid bBa \mid bBb \mid a \mid b\\&A \to aAa \mid aAb \mid bAa \mid bAb \mid a\\&B \to aBa \mid aBb \mid bBa \mid bBb \mid b\end{aligned}$$

generates all strings over $\{a, b\}$ with the same symbol in the beginning and middle positions. If the derivation begins with an S rule that begins with an a, the A rules ensure that an a occurs in the middle of the string. Similarly, the S and B rules combine to produce strings with a b in the first and middle positions.

16. The language $(\boldsymbol{a} \cup \boldsymbol{b})^*\boldsymbol{aa}(\boldsymbol{a} \cup \boldsymbol{b})^*\boldsymbol{bb}(\boldsymbol{a} \cup \boldsymbol{b})^*$ is generated by

$$\begin{aligned} \mathrm{G}_1\colon S_1 &\to aS_1 \mid bS_1 \mid aA \\ A &\to aB \\ B &\to aB \mid bB \mid bC \\ C &\to bD \\ D &\to aD \mid bD \mid \lambda \end{aligned}$$

G$_2$ generates the strings $(\boldsymbol{a} \cup \boldsymbol{b})^*\boldsymbol{bb}(\boldsymbol{a} \cup \boldsymbol{b})^*\boldsymbol{aa}(\boldsymbol{a} \cup \boldsymbol{b})^*$

$$\begin{aligned} \mathrm{G}_2\colon S_2 &\to aS_2 \mid bS_2 \mid bE \\ E &\to bF \\ F &\to aF \mid bF \mid aG \\ G &\to aH \\ H &\to aH \mid bH \mid \lambda \end{aligned}$$

A grammar G that generates

$$(\boldsymbol{a} \cup \boldsymbol{b})^*\boldsymbol{aa}(\boldsymbol{a} \cup \boldsymbol{b})^*\boldsymbol{bb}(\boldsymbol{a} \cup \boldsymbol{b})^* \cup (\boldsymbol{a} \cup \boldsymbol{b})^*\boldsymbol{bb}(\boldsymbol{a} \cup \boldsymbol{b})^*\boldsymbol{aa}(\boldsymbol{a} \cup \boldsymbol{b})^*$$

can be obtained from G$_1$ and G$_2$. The rules of G consist of the rules of G$_1$ and G$_2$ augmented with $S \to S_1 \mid S_2$ where S is the start symbol of the composite grammar. The alternative in these productions corresponds to the $\cup$ in the definition of the language.

While the grammar described above generates the desired language, it is not regular. The rules $S \to S_1 \mid S_2$ do not have the form required for rules of a regular grammar. A regular grammar can be obtained by explicitly replacing S_1 and S_2 in the S rules with the right-hand sides of the S_1 and S_2 rules. The S rules of the new grammar are

$$\begin{aligned} S &\to aS_1 \mid bS_1 \mid aA \\ S &\to aS_2 \mid bS_2 \mid bE \end{aligned}$$

The strategy used to modify the rules $S \to S_1 \mid S_2$ is an instance of a more general rule modification technique known as removing chain rules, which will be studied in detail in Chapter 4.

20. The language $((\boldsymbol{a} \cup \boldsymbol{\lambda})\boldsymbol{b}(\boldsymbol{a} \cup \boldsymbol{\lambda}))^*$ is generated by the grammar

$$\begin{aligned} S &\to aA \mid bB \mid \lambda \\ A &\to bB \\ B &\to aS \mid bB \mid \lambda \end{aligned}$$

This language consists of all strings over $\{a, b\}$ in which every a is preceded or followed by a b. An a generated by the rule $S \to aA$ is followed by a b. An a generated by $B \to aS$ is preceded by a b.

24. The variables of the grammar indicate whether an even or odd number of ab's has been generated and the progress toward the next ab. The interpretation of the variables is

Variable	Parity	Progress toward ab
S	even	none
A	even	a
B	odd	none
C	odd	a

The rules of the grammar are

$$\begin{aligned}&S \to aA \mid bS\\&A \to aA \mid bB\\&B \to aC \mid bB \mid \lambda\\&C \to aC \mid bS \mid \lambda\end{aligned}$$

A derivation may terminate with a λ-rule when a B or a C is present in the sentential form since this indicates that an odd number of ab's have been generated.

25. The objective is to construct a grammar that generates the set of strings over $\{a, b\}$ containing an even number of a's or an odd number of b's. In the grammar,

$$\begin{aligned}&S \to aO_a \mid bE_a \mid bO_b \mid aE_b \mid \lambda\\&E_a \to aO_a \mid bE_a \mid \lambda\\&O_a \to aE_a \mid bO_a\\&O_b \to aO_b \mid bE_b \mid \lambda\\&E_b \to aE_b \mid bO_b\end{aligned}$$

the E_a and O_a rules generate strings with an even number of a's. The derivation of a string with a positive even number of a's is initiated by the application of an S rule that generates either E_a or O_a. The derivation then alternates between occurrences of E_a and O_a until it is terminated with an application of the rule $E_a \to \lambda$.

In a like manner, the O_b and E_b rules generate strings with an odd number of b's. The string aab can be generated by two derivations; one beginning with the application of the rule $S \to aO_a$ and the other beginning with $S \to aE_b$.

30. Let G be the grammar $S \to aSbS \mid aS \mid \lambda$. We prove that every prefix of sentential form of G has at least as many a's as b's. We will refer to this condition as the prefix property. The proof is by induction of the length of derivations of G.

Basis: The strings $aSbS$, aS and λ are the only sentential forms produced by derivations of length one. The prefix property is seen to hold for these strings by inspection.

Inductive hypothesis: Assume that every sentential form that can be obtained by a derivation of length n or less satisfies the prefix property.

Inductive step: Let w be a sentential form of G that can be derived using $n + 1$ rule applications. The derivation of w can be written

$$S \stackrel{n}{\Rightarrow} uSv \Rightarrow w$$

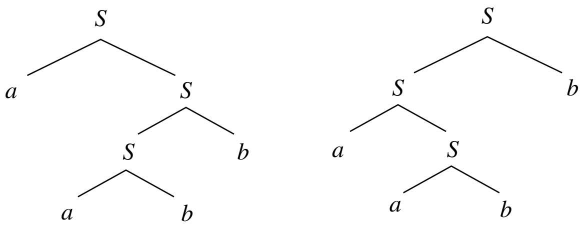
where w is obtained by applying an S rule to uSv. By the inductive hypothesis uSv satisfies the prefix property. A prefix of w consists of a prefix of uSv with the S replaced by λ, aS or $aSbS$. Thus w also satisfies the prefix property.

32. a) The language of G is $(\boldsymbol{a}^+\boldsymbol{b}^+)^+$.

    b) The rules $S \to aS$ and $S \to Sb$ allow the generation of leading a's or trailing b's in any order. Two leftmost derivations for the string $aabb$ are

$$\begin{aligned}S &\Rightarrow aS & S &\Rightarrow Sb\\&\Rightarrow aSb & &\Rightarrow aSb\\&\Rightarrow aabb & &\Rightarrow aabb\end{aligned}$$

c) The derivation trees corresponding to the derivations of (b) are



d) The unambiguous regular grammar

$$\begin{aligned} S &\rightarrow aS \mid aA \\ A &\rightarrow bA \mid bS \mid b \end{aligned}$$

is equivalent to G and generates strings unambiguously producing the terminals in a left-to-right manner. The S rules generate the a's and the A rules generate the b's. The process is repeated by the application of the rule $A \rightarrow bS$ or terminated by an application of $A \rightarrow b$.

33. a) The string consisting of ten a's can be produced by two distinct derivations; one consisting of five applications of the rule $S \rightarrow aaS$ followed by the λ-rule and the other consisting of two applications of $S \rightarrow aaaaaS$ and the λ-rule.

To construct an unambiguous grammar that generates L(G), it is necessary to determine precisely which strings are in L(G). The rules $S \rightarrow \lambda$ and $S \rightarrow aaS$ generate all strings with an even number of a's. Starting a derivation with a single application of $S \rightarrow aaaaaS$ and completing it with applications of $S \rightarrow aaS$ and $S \rightarrow \lambda$ produces strings of length 5, 7, 9, ... The language of G consists of all strings of a's except for a and aaa. This language is generated by the unambiguous grammar

$$\begin{aligned} S &\rightarrow \lambda \mid aa \mid aaaA \\ A &\rightarrow aA \mid a \end{aligned}$$

d) L(G) is the set $\{\lambda\} \cup \{a^i b^j \mid i > 1, j > 0\}$. The null string is generated directly by the rule $S \rightarrow \lambda$. The rule $S \rightarrow AaSbB$ generates one a and one b. Applications of $S \rightarrow AaSbB$, $A \rightarrow a$, $A \rightarrow aA$, and $B \rightarrow bB$ generate additional a's and b's. The rule $A \rightarrow a$ guarantees that there are two or more a's in every string nonnull in L(G).

To show that G is ambiguous we must find a string $w \in$ L(G) that can be generated by two distinct leftmost derivations. The prefix of a's can be generated either by applications of the rule $S \rightarrow AaSbB$ or by the A rules. Consider the derivation of the string $aaaabb$ in which two a's are generated by applications of the S rule.

Derivation	Rule
$S \Rightarrow AaSbB$	$S \rightarrow AaSbB$
$\Rightarrow aaSbB$	$A \rightarrow a$
$\Rightarrow aaAaSbBbB$	$S \rightarrow AaSbB$
$\Rightarrow aaaaSbBbB$	$A \rightarrow a$
$\Rightarrow aaaabBbB$	$S \rightarrow \lambda$
$\Rightarrow aaaabbB$	$B \rightarrow \lambda$
$\Rightarrow aaaabb$	$B \rightarrow \lambda$

The string can also be derived using the rules $A \rightarrow aA$ and $A \rightarrow a$ to generate the a's.

Derivation	Rule

$S \Rightarrow AaSbB$	$S \rightarrow AaSbB$
$\Rightarrow aAaSbB$	$A \rightarrow aA$
$\Rightarrow aaAaSbB$	$A \rightarrow aA$
$\Rightarrow aaaaSbB$	$A \rightarrow a$
$\Rightarrow aaaabB$	$S \rightarrow \lambda$
$\Rightarrow aaaabbB$	$B \rightarrow bB$
$\Rightarrow aaaabb$	$B \rightarrow \lambda$

The two distinct leftmost derivations of $aaaabb$ demonstrate the ambiguity of G.

The regular grammar

$$\begin{aligned} &S \rightarrow aA \mid \lambda \\ &A \rightarrow aA \mid aB \\ &B \rightarrow bB \mid b \end{aligned}$$

generates strings in L(G) in a left-to-right manner. The rules $S \rightarrow aA$ and $A \rightarrow aB$ guarantee the generation of at least two a's in every nonnull string. The rule $B \rightarrow b$, whose application completes a derivation, ensures the presence of at least one b.

e) The variable A generates strings of the form $(ab)^i$ and B generates $a^i b^i$ for all $i \geq 0$. The only strings in common are λ and ab. Each of these strings can be generated by two distinct leftmost derivations.

To produce an unambiguous grammar, it is necessary to ensure that these strings are generated by only one of A and B. Using the rules

$$\begin{aligned} &S \rightarrow A \mid B \\ &A \rightarrow abA \mid abab \\ &B \rightarrow aBb \mid \lambda, \end{aligned}$$

λ and ab are derivable only from the variable B.

34. a) The grammar G was produced in Exercise 33 d) to generate the language $\{\lambda\} \cup \{a^i b^j \mid i > 0, j > 1\}$ unambiguously. A regular expression for this language is $\boldsymbol{a^+ b^+ b \cup \lambda}$.

b) To show that G is unambiguous it is necessary to show that there is a unique leftmost derivation of every string in L(G). The sole derivation of λ is $S \Rightarrow \lambda$. Every other string in L(G) has the form $a^i b^j$ where $i > 0$ and $j > 1$. The derivation of such a string has the form

Derivation	Rule
$S \Longrightarrow aA$	$S \rightarrow aA$
$\overset{i-1}{\Longrightarrow} a^i A$	$A \rightarrow aA$
$\Longrightarrow a^i bB$	$A \rightarrow bB$
$\overset{j-2}{\Longrightarrow} a^i bb^{j-2} B$	$B \rightarrow bB$
$\Longrightarrow a^i bb^{j-2} b$	$B \rightarrow b$
$= a^i b^j$	

At each step there is only one rule that can be applied to successfully derive $a^i b^j$. Initially the S rule that produces an a must be employed, followed by exactly $i-1$ applications of the rule $A \rightarrow aA$. Selecting the other A rule prior to this point would produce a string with to few a's. Using more than $i-1$ applications would produce too many a's.

After the generation of a^i, the rule $A \rightarrow bB$ begins the generation of the b's. Any deviation from the pattern in the derivation above would produce the wrong number of b's. Consequently, there is only one leftmost derivation of $a^i b^j$ and the grammar is unambiguous.

39. a) The rules $S \rightarrow aABb$, $A \rightarrow aA$, and $B \rightarrow bB$ of G_1 are produced by the derivations $S \Rightarrow AABB \Rightarrow aABB \Rightarrow aABb$, $A \Rightarrow AA \Rightarrow aA$, and $B \Rightarrow BB \Rightarrow bB$ using the rules of G_2. Thus every rule of G_1 is either in G_2 or derivable in G_2.

    Now if w is derivable in G_1, it is also derivable in G_2 by replacing the application of a rule of G_1 by the sequence of rules of G_2 that produce the same transformation. If follows the $\mathrm{L}(\mathrm{G}_1) \subseteq \mathrm{L}(\mathrm{G}_2$.

    b) The language of G_1 is $\boldsymbol{aa^+b^+b}$. Since the rules

    $$A \rightarrow AA \mid a$$

    generate $\boldsymbol{a}^+$ and

    $$B \rightarrow BB \mid b$$

    generate $\boldsymbol{b}^+$, the language of G_2 is $\boldsymbol{a^+a^+b^+b^+} = \boldsymbol{aa^+b^+b}$.

40. A regular grammar is one with rules of the form $A \rightarrow aB$, $A \rightarrow a$, and $A \rightarrow \lambda$, where $A, B \in V$ and $a \in \Sigma$. The rules of a right-linear grammar have the form $A \rightarrow \lambda$, $A \rightarrow u$, and $A \rightarrow uB$, where $A, B \in V$ and $u \in \Sigma^+$. Since every regular grammar is also right-linear, all regular languages are generated by right-linear grammars.

    We now must show that every language generated by a right-linear grammar is regular. Let G be a right-linear grammar. A regular grammar G′ that generates L(G) is constructed from the rules of G. Rules of the form $A \rightarrow aB$, $A \rightarrow a$ and $A \rightarrow \lambda$ in G are also in G′. A right-linear rule $A \rightarrow a_1 \ldots a_n$ with $n > 1$ is transformed into a sequence of rules

    $$\begin{aligned} A &\rightarrow a_1T_1 \\ T_1 &\rightarrow a_2T_2 \\ &\vdots \\ T_{n-1} &\rightarrow a_n \end{aligned}$$

    where T_i are variables not occurring in G. The result of the application of the $A \rightarrow u$ of G is obtained by the derivation

    $$A \Rightarrow a_1T_1 \Rightarrow \cdots \Rightarrow a_1 \ldots a_{n-1}T_{n-1} \Rightarrow a_1 \ldots a_n$$

    in G′.

    Similarly, rules of the form $A \rightarrow a_1 \ldots a_nB$, with $n > 1$, can be transformed into a sequence of rules as above with the final rule having the form $T_{n-1} \rightarrow a_nB$. Clearly, the grammar G′ constructed in this manner generates L(G).

Chapter 4

Normal Forms for Context-Free Grammars

3. An equivalent essentially noncontracting grammar $\mathrm{G_L}$ with a nonrecursive start symbol is constructed following the steps given in the proof of Theorem 4.2.3. Because S is recursive, a new start symbol S' and the rule $S' \to S$ are added to $\mathrm{G_L}$.

   The set of nullable variables is $\{S', S, A, B\}$. All possible derivations of λ from the nullable variables are eliminated by the addition of five S rules, one A rule, and one B rule. The resulting grammar is

$$\begin{aligned} &S' \to S \mid \lambda \\ &S \to BSA \mid BS \mid SA \mid BA \mid B \mid S \mid A \\ &B \to Bba \mid ba \\ &A \to aA \mid a \end{aligned}$$

8. Following the technique of Example 4.3.1, an equivalent grammar $\mathrm{G_C}$ that does not contain chain rules is constructed. For each variable, Algorithm 4.3.1 is used to obtain the set of variables derivable using chain rules.

$$\begin{aligned} &\mathrm{CHAIN}(S) = \{S, A, B, C\} \\ &\mathrm{CHAIN}(A) = \{A, B, C\} \\ &\mathrm{CHAIN}(B) = \{B, C, A\} \\ &\mathrm{CHAIN}(C) = \{C, A, B\} \end{aligned}$$

   These sets are used to generate the rules of $\mathrm{G_C}$ according to the technique of Theorem 4.3.3, producing the grammar

$$\begin{aligned} &S \to aa \mid bb \mid cc \\ &A \to aa \mid bb \mid cc \\ &B \to aa \mid bb \mid cc \\ &C \to aa \mid bb \mid cc \end{aligned}$$

   In the grammar obtained by this transformation, it is clear that the A, B, and C rules do not contribute to derivations of terminal strings.

15. An equivalent grammar G_U without useless symbols is constructed in two steps. The first step involves the construction of a grammar G_T that contains only variables that derive terminal strings. Algorithm 4.4.2 is used to construct the set TERM of variables which derive terminal strings.

Iteration	TERM	PREV
0	$\{D, F, G\}$	-
1	$\{D, F, G, A\}$	$\{D, F, G\}$
2	$\{D, F, G, A, S\}$	$\{D, F, G, A\}$
3	$\{D, F, G, A, S\}$	$\{D, F, G, A, S\}$

Using this set, G_T is constructed.

$$\begin{aligned} V_T &= \{S, A, D, F, G\} \\ \Sigma_T &= \{a, b\} \\ P_T :\ & S \to aA \\ & A \to aA \mid aD \\ & D \to bD \mid b \\ & F \to aF \mid aG \mid a \\ & G \to a \mid b \end{aligned}$$

The second step in the construction of G_U involves the removal of all variables from G_T that are not reachable from S. Algorithm 4.4.4 is used to construct the set REACH of variables reachable from S.

Iteration	REACH	PREV	NEW
0	$\{S\}$	$\emptyset$	-
1	$\{S, A\}$	$\{S\}$	$\{S\}$
2	$\{S, A, D\}$	$\{S, A\}$	$\{A\}$
3	$\{S, A, D\}$	$\{S, A, D\}$	$\{D\}$

Removing all references to variables in the set $V_T -$ REACH produces the grammar

$$\begin{aligned} V_U &= \{S, A, D\} \\ \Sigma_U &= \{a, b\} \\ P_U :\ & S \to aA \\ & A \to aA \mid aD \\ & D \to bD \mid b \end{aligned}$$

19. To convert G to Chomsky normal form, we begin by transforming the rules of G to the form $S \to \lambda$, $A \to a$, or $A \to w$ where w is a string consisting solely of variables.

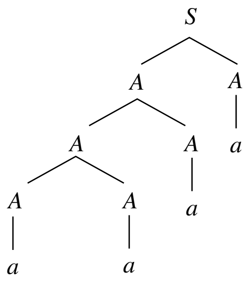
$$\begin{aligned} & S \to XAYB \mid ABC \mid a \\ & A \to XA \mid a \\ & B \to YBZC \mid b \\ & C \to XYZ \\ & X \to a \\ & Y \to b \\ & Z \to c \end{aligned}$$

The transformation is completed by breaking each rule whose right-hand side has length greater than 2 into a sequence of rules of the form $A \to BC$.

$$\begin{array}{l}
S \to XT_1 \mid AT_3 \mid a \\
T_1 \to AT_2 \\
T_2 \to YB \\
T_3 \to BC \\
A \to XA \mid a \\
B \to YT_4 \mid b \\
T_4 \to BT_5 \\
T_5 \to ZC \\
C \to XT_6 \\
T_6 \to YZ \\
X \to a \\
Y \to b \\
Z \to c
\end{array}$$

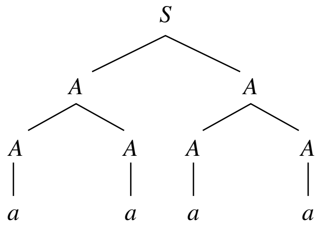
24. a) The derivation of a string of length 0, $S \Rightarrow \lambda$, requires one rule application.
A string of length $n > 0$ requires $2n - 1$; $n - 1$ rules of the form $A \to BC$ and n rules of the form $A \to a$.

b) The maximum depth derivation tree for a string of length 4 has the form



Generalizing this pattern we see that the maximum depth of the derivation tree of a string of length $n > 0$ is n. The depth of the derivation tree for the derivation of λ is 1.

c) The minimum depth derivation tree of a string of length 4 has the form



Intuitively, the minimum depth is obtained when the tree is the "bushiest" possible. The minimum depth of a derivation tree for a string of length $n > 0$ is $\lceil \lg(n) + 1 \rceil$.

28. To obtain an equivalent grammar with no direct left recursion, the rule modification that removes direct left recursion is applied to the A and B rules producing

$$\begin{array}{l}
S \to A \mid C \\
A \to BX \mid aX \mid B \mid a \\
X \to aBX \mid aCX \mid aB \mid aC
\end{array}$$

$$\begin{aligned}
&B \to CbY \mid Cb\\
&Y \to bY \mid b\\
&C \to cC \mid c
\end{aligned}$$

The application of an A rule generates B or a and the X rule produces elements of the form described by $\boldsymbol{(aB \cup aC)^+}$ using right recursion.

Similarly, the B and Y rules produce $\boldsymbol{Cbb^*}$ using the right recursive Y rule to generate $\boldsymbol{b^+}$.

32. The transformation of a grammar G from Chomsky normal form to Greibach normal form is accomplished in two phases. In the first phase, the variables of the grammar are numbered and an intermediate grammar is constructed in which the first symbol of the right-hand side of every rule is either a terminal or a variable with a higher number than the number of the variable on the left-hand side of the rule. This is done by removing direct left recursion and by applying the rule replacement schema of Lemma 4.1.3. The variables S, A, B, and C are numbered 1, 2, 3, and 4 respectively. The S rules are already in the proper form. Removing direct left recursion from the A rules produces the grammar

$$\begin{aligned}
&S \to AB \mid BC\\
&A \to aR_1 \mid a\\
&B \to AA \mid CB \mid b\\
&C \to a \mid b\\
&R_1 \to BR_1 \mid B
\end{aligned}$$

Applying the rule transformation schema of Lemma 4.1.3, the rule $B \to AA$ can be converted into the desired form by substituting for the first A, resulting in the grammar

$$\begin{aligned}
&S \to AB \mid BC\\
&A \to aR_1 \mid a\\
&B \to aR_1A \mid aA \mid CB \mid b\\
&C \to a \mid b\\
&R_1 \to BR_1 \mid B
\end{aligned}$$

The second phase involves transformation of the rules of the intermediate grammar to ensure that the first symbol of the right-hand side of each rule is a terminal symbol. Working backwards from the C rules and applying Lemma 4.1.3, we obtain

$$\begin{aligned}
&S \to aR_1B \mid aB \mid aR_1AC \mid aAC \mid aBC \mid bBC \mid bC\\
&A \to aR_1 \mid a\\
&B \to aR_1A \mid aA \mid aB \mid bB \mid b\\
&C \to a \mid b\\
&R_1 \to BR_1 \mid B
\end{aligned}$$

Finally, Lemma 4.1.3 is used to transform the R_1 rules created in the first phase into the proper form, producing the Greibach normal form grammar

$$\begin{aligned}
&S \to aR_1B \mid aB \mid aR_1AC \mid aAC \mid aBC \mid bBC \mid bC\\
&A \to aR_1 \mid a\\
&B \to aR_1A \mid aA \mid aB \mid bB \mid b\\
&C \to a \mid b\\
&R_1 \to aR_1AR_1 \mid aAR_1 \mid aBR_1 \mid bBR_1 \mid bR_1\\
&\quad\ \to aR_1A \mid aA \mid aB \mid bB \mid b
\end{aligned}$$

35. We begin by showing how to reduce the length of strings on the right-hand side of the rules of a grammar in Greibach normal form. Let G = (V, Σ, S, P) be a grammar in Greibach normal form in which the maximum length of the right-hand side of a rule is n, where n is any natural number greater than 3. We will show how to transform G to a Greibach normal form grammar G$'$ in which the right-hand side of each rule has length at most $n - 1$.

If a rule already has the desired form, no change is necessary. We will outline the approach to transform a rule of the form $A \to aB_2 \cdots B_{n-1}B_n$ into a set of rules of the desired form. The procedure can be performed on each rule with right-hand side of length n to produce a grammar in which the right-hand side of each rule has length at most $n - 1$.

A new variable $[B_{n-1}B_n]$ is added to grammar and the rule

$$A \to aB_2 \cdots B_{n-1}B_n$$

is replaced with the rule

$$A \to aB_2 \cdots B_{n-2}[B_{n-1}B_n]$$

whose right-hand side has length $n - 1$. It is now necessary to define the rules for the new variable $[B_{n-1}B_n]$. Let the B_{n-1} rules of G be

$$B_{n-1} \to \beta_1 u_1 \mid \beta_1 u_2 \mid \cdots \mid \beta_m u_m,$$

where $\beta_i \in \Sigma$, $u_i \in$ V*, and $length(\beta_i u_i) \leq n$. There are three cases to consider in creating the new rules.

Case 1: $length(\beta_i u_i) \leq n - 2$. Add the rule

$$[B_{n-1}B_n] \to \beta_i u_i B_n$$

to G$'$.

Case 2: $length(\beta_i u_i) = n-1$. In this case, the B_{n-1} rule can be written $B_{n-1} \to \beta_i C_2 \cdots C_{n-1}$. Create a new variable $[C_{n-1}B_n]$ and add the rule

$$B_{n-1} \to \beta_i C_2 \cdots C_{n-2}[C_{n-1}B_n]$$

to G$'$.

Case 3: $length(\beta_i u_i) = n$. In this case, the B_{n-1} rule can be written $B_{n-1} \to \beta_i C_2 \cdots C_{n-1}C_n$. Create two new variables $[C_{n-2}C_{n-1}]$ and $[C_nB_n]$ and add the rule

$$B_{n-1} \to \beta_i C_2 \cdots C_{n-3}[C_{n-2}C_{n-1}][C_nB_n]$$

to G$'$.

Each of the rules generated in cases 1, 2, and 3 have right-hand sides of length $n - 1$. The process must be repeated until rules have been created for each variable created by the rule transformations.

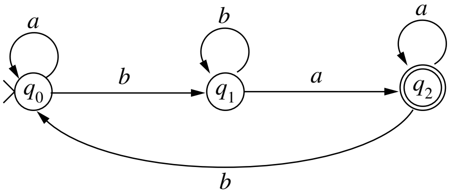
The construction in case 3 requires the original rule to have at least three variables. This length reduction process can be repeated until each rule has a right-hand side of length at most 3. A detailed proof that this construction produces an equivalent grammar can be found in Harrison[1].

[1] M. A. Harrison, *Introduction to Formal Languages*, Addison-Wesley, 1978

Chapter 5

Finite Automata

1. a) The state diagram of M is



b) i)

$$\begin{array}{ll} & [q_0, abaa] \\ \vdash & [q_0, baa] \\ \vdash & [q_1, aa] \\ \vdash & [q_2, a] \\ \vdash & [q_2, \lambda] \end{array}$$

ii)

$$\begin{array}{ll} & [q_0, bbbabb] \\ \vdash & [q_1, bbabb] \\ \vdash & [q_1, babb] \\ \vdash & [q_1, abb] \\ \vdash & [q_2, bb] \\ \vdash & [q_0, b] \\ \vdash & [q_1, \lambda] \end{array}$$

iii)

$$\begin{array}{ll} & [q_0, bababa] \\ \vdash & [q_1, ababa] \\ \vdash & [q_2, baba] \\ \vdash & [q_0, aba] \\ \vdash & [q_0, ba] \\ \vdash & [q_1, a] \\ \vdash & [q_2, \lambda] \end{array}$$

iv)

$$\begin{array}{ll} & [q_0, bbbaa] \\ \vdash & [q_1, bbaa] \\ \vdash & [q_1, baa] \\ \vdash & [q_1, aa] \\ \vdash & [q_2, a] \\ \vdash & [q_2, \lambda] \end{array}$$

c) The computations in (i), (iii), and (iv) terminate in the accepting state q_2. Therefore, the strings *abaa*, *bababa*, and *bbbaa* are in L(M).

d) Two regular expressions describing L(M) are $\boldsymbol{a^*b^+a^+(ba^*b^+a^+)^*}$ and $\boldsymbol{(a^*b^+a^+b)^*a^*b^+a^+}$.

4. The proof is by induction on length of the input string. The definitions of $\hat{\delta}$ and $\hat{\delta}'$ are the same for strings of length 0 and 1.

Assume that $\hat{\delta}(q_i, u) = \hat{\delta}'(q_i, u)$ for all strings u of length n or less. Let $w = bva$ be a string of length of $n + 1$, with $a, b \in \Sigma$ and $length(v) = n - 1$.

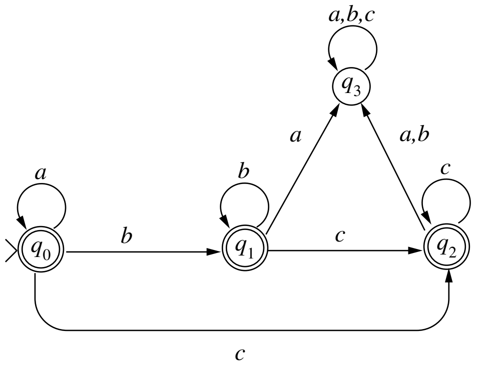
$$
\begin{aligned}
\hat{\delta}(q_i, w) &= \hat{\delta}(q_i, bva) & \\
&= \delta(\hat{\delta}(q_i, bv), a) & \text{(definition of } \hat{\delta}\text{)} \\
&= \delta(\hat{\delta}'(q_i, bv), a) & \text{(inductive hypothesis)} \\
&= \delta(\hat{\delta}'(\delta(q_i, b), v), a) & \text{(definition of } \hat{\delta}'\text{)}
\end{aligned}
$$

Similarly,

$$
\begin{aligned}
\hat{\delta}'(q_i, w) &= \hat{\delta}'(q_i, bva) & \\
&= \hat{\delta}'(\delta(q_i, b), va) & \text{(definition of } \hat{\delta}'\text{)} \\
&= \hat{\delta}(\delta(q_i, b), va) & \text{(inductive hypothesis)} \\
&= \delta(\hat{\delta}(\delta(q_i, b), v), a) & \text{(definition of } \hat{\delta}\text{)} \\
&= \delta(\hat{\delta}'(\delta(q_i, b), v), a) & \text{(inductive hypothesis)}
\end{aligned}
$$

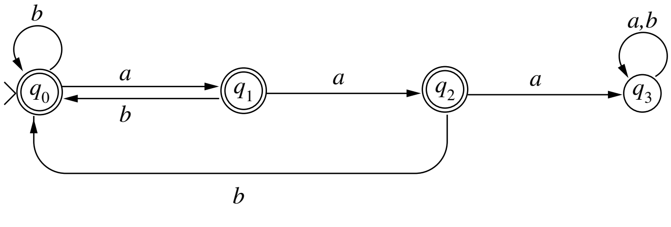
Thus $\hat{\delta}(q_i, w) = \hat{\delta}'(q_i, w)$ for all strings $w \in \Sigma^*$ as desired.

5. The DFA



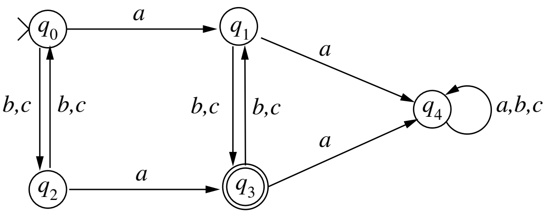
that accepts $\boldsymbol{a^*b^*c^*}$ uses the loops in states q_0, q_1, and q_2 to read a's, b's, and c's. Any deviation from the prescribed order causes the computation to enter q_3 and reject the string.

8. A DFA that accepts the strings over $\{a, b\}$ that do not contain the substring aaa is given by the state diagram



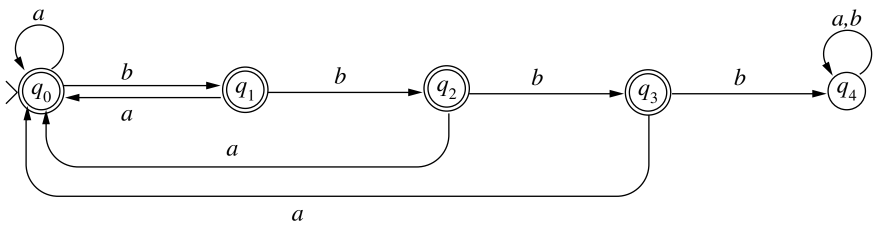
The states are used to count the number of consecutive a's that have been processed. When three consecutive a's are encountered, the DFA enters state q_3, processes the remainder of the input, and rejects the string.

15. The DFA



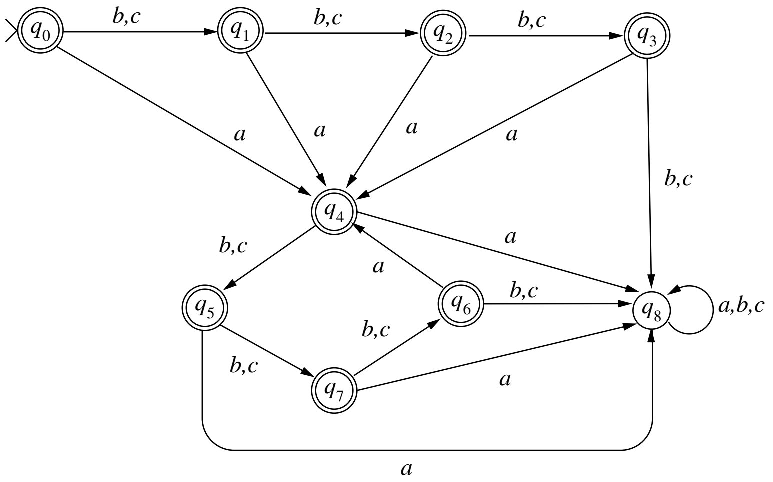
accepts strings of even length over $\{a, b, c\}$ that contain exactly one a. A string accepted by this machine must have exactly one a and the total number of b's and c's must be odd. States q_1 or q_3 are entered upon processing a single a. The state q_3 represents the combination of the two conditions required for acceptance. Upon reading a second a, the computation enters the nonaccepting state q_4 and rejects the string.

20. The states of the machine are used to count the number of symbols processed since an a has been read or since the beginning of the string, whichever is smaller.



A computation is in state q_i if the previous i elements are not a's. If an input string has a substring of length four without an a, the computation enters q_4 and rejects the string. All other strings accepted.

21. The DFA to accept this language differs from the machine in Exercise 20 by requiring every fourth symbol to be an a. The first a may occur at position 1, 2, 3, or 4. After reading the first a, the computation enters state q_4 and checks for a pattern consisting of a combination of three b's or c's followed by an a. If the string varies from this pattern, the computation enters state q_8 and rejects the input.



Strings of length 3 or less are accepted, since the condition is vacuously satisfied.

23. a) The transition table for the machine M is

δ	a	b
q_0	$\{q_0, q_1\}$	$\emptyset$
q_1	$\emptyset$	$\{q_1, q_2\}$
q_2	$\{q_0, q_1\}$	$\emptyset$

b)

$$\begin{array}{ll} & [q_0, aaabb] \\ \vdash & [q_1, aabb] \end{array}$$

$$\begin{array}{ll} & [q_0, aaabb] \\ \vdash & [q_0, aabb] \\ \vdash & [q_1, abb] \end{array}$$

$$\begin{array}{ll} & [q_0, aaabb] \\ \vdash & [q_0, aabb] \\ \vdash & [q_0, abb] \\ \vdash & [q_1, bb] \\ \vdash & [q_1, b] \\ \vdash & [q_1, \lambda] \end{array}$$

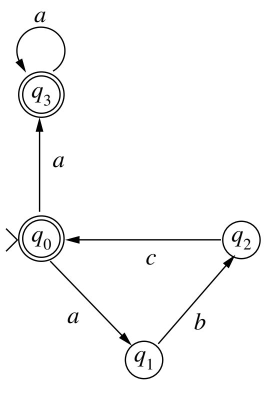
$$\begin{array}{ll} & [q_0, aaabb] \\ \vdash & [q_0, aabb] \\ \vdash & [q_0, abb] \\ \vdash & [q_0, bb] \end{array}$$

$$\begin{array}{ll} & [q_0, aaabb] \\ \vdash & [q_0, aabb] \\ \vdash & [q_0, abb] \\ \vdash & [q_1, bb] \\ \vdash & [q_1, b] \\ \vdash & [q_2, \lambda] \end{array}$$

c) Yes, the final computation above reads the entire string and halts in the accepting state q_2.

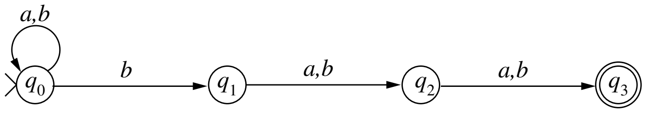
d) A regular expressions for L(M) is $\boldsymbol{a^+b^+(ab^+ \cup a^+ab^+)^*}$. Processing $\boldsymbol{a^+b^+}$ leaves the computation in state q_2. After arriving at q_2 there are two ways to leave and return, taking the cycle q_0, q_1, q_2 or the cycle q_1, q_2. The first path processes a string from $\boldsymbol{a^+ab^+}$ and the second $\boldsymbol{ab^+}$. This expression can be reduced to $\boldsymbol{(a^+b^+)^+}$.

25. c) The NFA



accepts the language $\boldsymbol{(abc)^*a^*}$. Strings of the form $\boldsymbol{(abc)^*}$ are accepted in q_0. State q_3 accepts $\boldsymbol{(abc)^*a^+}$.

29. The set of strings over $\{a, b\}$ whose third to the last symbol is b is accepted by the NFA



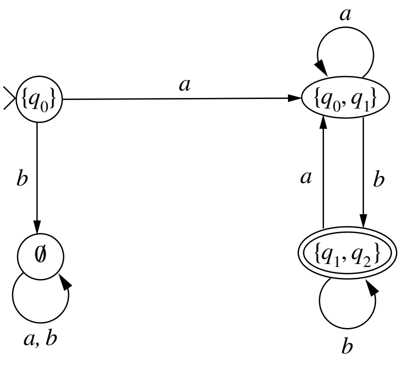
When a b is read in state q_0, nondeterminism is used to choose whether to enter state q_1 or remain in state q_0. If q_1 is entered upon processing the third to the last symbol, the computation accepts the input. A computation that enters q_1 in any other manner terminates unsuccessfully.

39. Algorithm 5.6.3 is used to construct a DFA that is equivalent to the NFA M in Exercise 17. Since M does not contain λ-transitions, the input transition function used by the algorithm is the transition function of M. The transition table of M is given in the solution to Exercise 17.

The nodes of the equivalent NFA are constructed in step 2 of Algorithm 5.6.3. The generation of Q′ is traced in the table below. The start state of the determininstic machine is $\{q_0\}$, which is the λ-closure of q_0. The algorithm proceeds by choosing a state X $\in$ Q′ and symbol $a \in \Sigma$ for which there is no arc leaving X labeled a. The set Y consists of the states that may be entered upon processing an a from a state in X.

X	input	Y	Q′
			$Q' := \{\{q_0\}\}$
$\{q_0\}$	a	$\{q_0, q_1\}$	$Q' := Q' \cup \{\{q_0, q_1\}\}$
$\{q_0\}$	b	$\emptyset$	$Q' := Q' \cup \{\emptyset\}$
$\{q_0, q_1\}$	a	$\{q_0, q_1\}$	
$\{q_0, q_1\}$	b	$\{q_1, q_2\}$	$Q' := Q' \cup \{\{q_1, q_2\}\}$
$\emptyset$	a	$\emptyset$	
$\emptyset$	b	$\emptyset$	
$\{q_1, q_2\}$	a	$\{q_0, q_1\}$	
$\{q_1, q_2\}$	b	$\{q_1, q_2\}$	

The accepting state is $\{q_1, q_2\}$. The state diagram of the deterministic machine is



43. To prove that q_m is equivalent to q_n we must show that $\hat{\delta}(q_m, w) \in$ F if, and only if, $\hat{\delta}(q_n, w) \in$ F for every string $w \in \Sigma^*$.

Let w be any string over Σ. Since q_i and q_j are equivalent, $\hat{\delta}(q_i, uw) \in$ F if, and only if, $\hat{\delta}(q_j, uw) \in$ F. The equivalence of q_m and q_n follows since $\hat{\delta}(q_i, uw) = \hat{\delta}(q_m, w)$ and $\hat{\delta}(q_n, uw) = \hat{\delta}(q_n, w)$.

44. Let $\delta'([q_i], a) = [\delta(q_i, a)]$ be the transition function defined on classes of equivalent states of a DFA and let q_i and q_j be two equivalent states. To show that δ' is well defined, we must show that $\delta'([q_i], a) = \delta'([q_j], a)$.

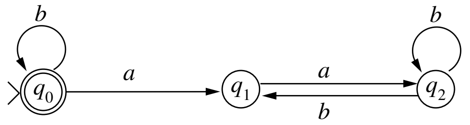
    If $\delta'([q_i], a)q_m$ and $\delta'([q_j], a) = q_n$, this reduces to showing that $[q_m] = [q_n]$. However, this equality follows from the result in Exercise 43.

Chapter 6

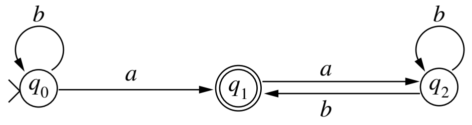
Properties of Regular Languages

2. a) This problem illustrates one technique for determining the language of a finite automaton with more than one accepting. The node deletion algorithm must be employed individually for each accepting state. Following this strategy, a regular expression is obtained by deleting nodes from the graphs

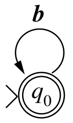
G$_1$



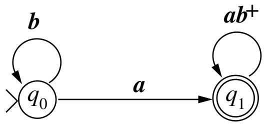
G$_2$



Deleting q_1 and q_2 from G$_1$ produces



with the corresponding regular expression $\boldsymbol{b^*}$. Removing q_2 from G$_2$ we obtain the expression graph



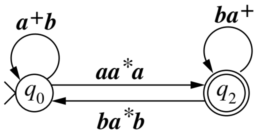
accepting $\boldsymbol{b^*a(ab^+)^*}$. Consequently, the original NFA accepts strings of the form $\boldsymbol{b^*} \cup \boldsymbol{b^*a(ab^+)^*}$.

5. a) Associating the variables S, A, and B with states q_0, q_1, and q_2 respectively, the regular grammar

$$\begin{array}{l} S \to aA \mid \lambda \\ A \to aA \mid aB \mid bS \\ B \to bA \mid \lambda \end{array}$$

is obtained from the arcs and the accepting states of M.

b) A regular expression for L(M) can be obtained using the node deletion process. Deleting the state q_1 produces the graph



The set of strings accepted by q_2 is described by the regular expression $\boldsymbol{(a^+b)^*aa^*a(ba^+ \cup ba^*b(a^+b)^*aa^*a)^*}$.

By deleting states q_2 then q_1, we see that q_0 accepts $\boldsymbol{(a(a \cup ab)^*b)^*}$. The language of M is the union of the strings accepted by q_0 and q_2.

6. Let G be a regular grammar and M the NFA that accepts L(G) obtained by the construction in Theorem 6.3.1. We will use induction on the length of the derivations of G to show that there is a computation $[q_0, w] \overset{*}{\vdash} [C, \lambda]$ in M whenever there is a derivation $S \overset{*}{\Rightarrow} wC$ in G.

The basis consists of derivations of the form $S \Rightarrow aC$ using a rule $S \rightarrow aC$. The rule produces the transition $\delta(q_0, a) = C$ in M and the corresponding derivation is $[q_0, a] \vdash [C, \lambda]$ as desired.

Now assume that $[q_0, w] \overset{*}{\vdash} [C, \lambda]$ whenever there is a derivation $S \overset{n}{\Rightarrow} wC$.

Let $S \overset{*}{\Rightarrow} wC$ be a computation of length $n + 1$. Such a derivation can be written

$$\begin{aligned} S &\overset{n}{\Rightarrow} uB \\ &uaC, \end{aligned}$$

where $B \rightarrow aC$ is a rule of G. By the inductive hypothesis, there is a computation $[q_0, u] \vdash$ ${}^* [B, \lambda]$. Combining the transition $\delta(a, B) = C$ obtained from the rule $B \rightarrow aC$ with the preceding computation produces

$$\begin{aligned} [q_0, w] &= [q_0, ua] \\ &\overset{*}{\vdash} [B, a] \\ &\vdash [C, \lambda] \end{aligned}$$

as desired.

7. a) Let L be any regular language over $\{a, b, c\}$ and let L$'$ be the language consisting of all strings over $\{a, b, c\}$ that end in aa. L$'$ is regular since it is defined by the regular expression $\boldsymbol{(a \cup b \cup c)^*aa}$. The set

$$\mathrm{L} \cap \mathrm{L}' = \{w \mid w \in \mathrm{L} \text{ and } w \text{ contains an } a\},$$

which consists of all strings in L that end with aa, is regular by Theorem 6.4.3.

11. a) Let G = (V, Σ, S, P) be a regular grammar that generates L. Without loss of generality, we may assume that G does not contain useless symbols. The algorithm to remove useless symbols, presented in Section 4.4, does not alter the form of rules of the grammar. Thus the equivalent grammar obtained by this transformation is also regular.

Derivations in G have the form $S \overset{*}{\Rightarrow} uA \overset{*}{\Rightarrow} uv$ where $u, v \in \Sigma^*$. A grammar G$'$ that generates $\mathrm{P} = \{u \mid uv \in \mathrm{L}\}$, the set of prefixes of L, can be constructed by augmenting the rules of G with rules $A \rightarrow \lambda$ for every variable $A \in$ V. The prefix u is produced by the derivation $S \overset{*}{\Rightarrow} uA \Rightarrow u$ in G$'$.

12. A DFA M$'$ that accepts the language L$'$ can be constructed from a DFA M $= (\mathrm{Q}, \Sigma, \delta, q_0, \mathrm{F})$ that accepts L. The states of M$'$ are ordered pairs of the form $[q_i, \mathrm{X}]$, where $q_i \in \mathrm{Q}$ and X is a subset of Q. Intuitively (but not precisely), the state q_i represents the number of transitions from the start state when processing a string and the set X consists of states that require the same number of transitions to reach an accepting state.

    The construction of the states Q$'$ and the transition function δ' of M$'$ is iterative. The start state of the machine is the ordered pair $[q_0, \mathrm{F}]$. For each state $[q_i, \mathrm{X}]$ and symbol a such that $\delta([q_i, \mathrm{X}], a)$ is undefined, the transition is created by the rule

    $$\delta([q_i, \mathrm{X}], a) = [q_i(a), \mathrm{Y}],$$

    where $\mathrm{Y} = \{q_j \mid \delta(q_j, a) \in \mathrm{X}\}$. If it is not already in Q$'$, the ordered pair $[q_i(a), \mathrm{Y}]$ is added to the states of M$'$. The process is repeated until a transition is defined for each state-symbol pair.

    A state $[q_i, \mathrm{X}]$ is accepting if $q_i \in \mathrm{X}$. Back to the intuition, this occurs when the number of transitions from the start state is the same as the number of transitions needed to reach an accepting state. That is, when half of a string in L has been processed.

13. b) To show that the set P of even length palindromes is not regular, it suffices to find sequences of strings u_i and v_i, $i = 0, 1, 2, \dots$, from $\{a, b\}^*$ that satisfy

    i) $u_i v_i \in \mathrm{P}$, for $i \geq 0$

    ii) $u_i v_j \notin \mathrm{P}$ whenever $i \neq j$.

    Let $u_i = a^i b$ and $v_i = ba^i$. Then $u_i v_i$ is the palindrome $a^i bba^i$. For $i \neq j$, the string $u_i v_j = a^i bba^j$ is not in P. We conclude, by Corollary 6.5.2, that P is not regular.

14. b) Assume that $\mathrm{L} = \{a^n b^m \mid n < m\}$ is regular. Let k be the number specified by the pumping lemma and z be the string $a^k b^{k+1}$. By the pumping lemma, z can be written uvw where

    i) $v \neq \lambda$

    ii) $length(uv) \leq k$

    iii) $uv^i w \in \mathrm{L}$ for all $i \geq 0$.

    By condition (ii), v consists solely of a's. Pumping v produces the string uv^2w that contains at least as many a's as b's. Thus $uv^2w \notin \mathrm{L}$ and L is not regular.

    e) The language L consists of strings of the form $\lambda, a, ab, aba, abaa, abaab$, and so forth. Assume that L is regular and let $z = abaab \cdots ba^{k-1}ba^k b$, where k is the number specified by the pumping lemma. We will show that there is no decomposition $z = uvw$ in which v satisfies the conditions of the pumping lemma.

    The argument utilizes the number of b's in the substring v. There are three cases to consider.

    Case 1: v has no b's. In this case, the string uv^0w has consecutive sequences of a's in which the second sequence has at most the same number of a's as its predecessor. Thus $uv^0w \notin \mathrm{L}$.

    Case 2: v has one b. In this case, v has the form $a^s ba^t$ and is obtained from a substring $ba^j ba^{j+1} b$ in z. The string uv^2w obtained by pumping v has a substring $ba^j ba^{s+t} ba^{j+1} b$, which does not follow pattern of increasing the number of a's by one in each subsequent substring of a's. Thus $uv^2w \notin \mathrm{L}$.

    Case 3: v has two or more b's. In this case, v contains a substring $ba^t b$. Pumping v produces two substrings of the form $ba^t b$ in uv^2w. Since no string in L can have two distinct substrings of this form, $uv^2w \notin \mathrm{L}$.

Thus there is no substring of z that satisfies the conditions of the pumping lemma and we conclude the L is not regular.

17. b) Let L_1 be a nonregular language and L_2 be a finite language. Since every finite language is regular, $L_1 \cap L_2$ is regular. Now assume that $L_1 - L_2$ is regular. By Theorem 6.4.1,

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

is regular. But this is a contradiction, and we conclude that $L_1 - L_2$ is not regular. This result shows that a language cannot be "a little nonreguar"; removing any finite number of elements from a nonregular set cannot make it regular.

18. a) Let $L_1 =\boldsymbol{a^*b^*}$ and $L_2 = \{a^ib^i \mid i \geq 0\}$. Then $L_1 \cup L_2 = L_1$, which is regular.

    b) In a similar manner, let $L_1 = \boldsymbol{\emptyset}$ and $L_2 = \{a^ib^i \mid i \geq 0\}$. Then $L_1 \cup L_2 = L_2$, which is nonregular.

19. a) The regular languages over a set Σ are constructed using the operations union, concatenation, and Kleene star. We begin by showing that these set operations are preserved by homomorphisms. That is, for sets X and Y and homomorphism h

    i) $h(XY) = h(X)h(Y)$
    ii) $h(X \cup Y) = h(X) \cup h(Y)$
    iii) $h(X^*) = h(X)^*$.

    The set equality $h(XY) = h(X)h(Y)$ can be obtained by establishing the inclusions $h(XY) \subseteq h(X)h(Y)$ and $h(X)h(Y) \subseteq h(XY)$.

    Let x be an element of $h(XY)$. Then $x = h(uv)$ for some $u \in X$ and $v \in Y$. Since h is a homomorphism, $x = h(u)h(v)$ and $x \in h(X)h(Y)$. Thus $h(XY) \subseteq h(X)h(Y)$. To establish the opposite inclusion, let x be an element in $h(X)h(Y)$. Then $x = h(u)h(v)$ for some $u \in X$ and $v \in Y$. As before, $x = h(uv)$ and $h(X)h(Y) \subseteq h(XY)$. The other two set equalities can be established by similar arguments.

    Now let Σ_1 and Σ_2 be alphabets, X a language over Σ_1, and h a homomorphism from Σ_1^* to Σ_2^*. We will use the recursive definition of regular sets to show that $h(X)$ is regular whenever X is. The proof is by induction on the number of applications of the recursive step in Definition 2.3.1 needed to generate X.

    Basis: The basis consists of regular sets $\emptyset$, $\{\lambda\}$, and $\{a\}$ for every $a \in \Sigma_1$. The homomorphic images of these sets are the sets $\emptyset$, $\{\lambda\}$, and $\{h(a)\}$, which are regular over Σ_2.

    Inductive hypothesis: Now assume that the homomorphic image of every regular set definable using n or fewer applications of the recursive step is regular.

    Inductive step: Let X be a regular set over Σ_1 definable by $n + 1$ applications of the recursive step in Definition 2.3.1. Then X can be written $Y \cup Z$, YZ, or Y^* where Y and Z are definable by n or fewer applications. By the inductive hypothesis, $h(Y)$ and $h(Z)$ are regular. It follows that $h(X)$, which is either $h(Y) \cup h(Z)$, $h(Y)h(Z)$, or $h(Y)^*$, is also regular.

21. For every regular grammar $G = (V, \Sigma, S, P)$ there is a corresponding a left-regular grammar $G' = (V, \Sigma, S, P')$, defined by

    i) $A \to Ba \in P'$ if, and only if, $A \to aB \in P$
    ii) $A \to a \in P'$ if, and only if, $A \to a \in P$
    iii) $A \to \lambda \in P'$ if, and only if, $A \to \lambda \in P$.

The following Lemma establishes a relationship between regular and left-regular grammars.

Lemma. Let G and G$'$ be corresponding regular and left-regular grammars. Then L(G$'$) = L(G)^R.

The lemma can be proven by showing that $S \overset{*}{\Rightarrow} u$ in G if, and only if, $S \overset{*}{\Rightarrow} u^R$ in G$'$. The proof is by induction on the length of the derivations.

a) Let L be a language generated by a left-regular grammar G$'$. By the preceding lemma, the corresponding regular grammar G generates L^R. Since regularity is preserved by the operation of reversal, (L^R)^R = L is regular. Thus every language generated by a left-regular grammar is regular.

b) Now we must show that every regular language is generated by a left-regular grammar. If L is a regular language, then so is L^R. This implies that there is a regular grammar G that generates L^R. The corresponding left-regular grammar G$'$ generates (L^R)^R = L.

29. The $\equiv_{\mathrm{M}}$ equivalence class of a state q_i consists of all strings for which the computation of M halts in q_i. The equivalence classes of the DFA in Example 5.3.1 are

state	equivalence class
q_0	$(\boldsymbol{a} \cup \boldsymbol{ba})^*$
q_1	$\boldsymbol{a}^*\boldsymbol{b}(\boldsymbol{a}^+\boldsymbol{b})^*$
q_2	$\boldsymbol{a}^*\boldsymbol{b}(\boldsymbol{a}^+\boldsymbol{b})^*\boldsymbol{b}(\boldsymbol{a} \cup \boldsymbol{b})^*$

31. Let M = (Q, Σ, δ, q_0, F) be the minimal state DFA that accepts a language L as constructed by Theorem 6.7.4 and let n be the number of states of M. We must show that any other DFA M$'$ with n states that accepts L is isomorphic to M.

By Theorem 6.7.5, the $\equiv_{\mathrm{M}}$ equivalence classes of the states M are identical to the $\equiv_{\mathrm{L}}$ equivalence classes of the language L.

By the argument in Theorem 6.7.5, each $\equiv_{\mathrm{M}'}$ equivalence class is a subset of a $\equiv_{\mathrm{L}}$ equivalence class. Since there are same number of $\equiv_{\mathrm{M}'}$ and $\equiv_{\mathrm{L}}$ equivalence classes, the equivalence classes defined by L and M$'$ must be identical.

The start state in each machine is the state associated with $[\lambda]_{\equiv_{\mathrm{L}}}$. Since the transition function can be obtained from the $\equiv_{\mathrm{L}}$ equivalence classes as outlined in Theorem 6.7.4, the transition functions of M and M$'$ are identical up to the names given to the states.

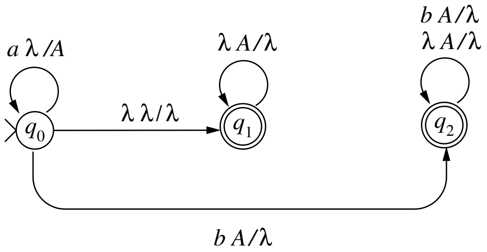
Chapter 7

Pushdown Automata and Context-Free Languages

1. a) The PDA M accepts the language $\{a^i b^j \mid 0 \le j \le i\}$. Processing an a pushes A onto the stack. Strings of the form a^i are accepted in state q_1. The transitions in q_1 empty the stack after the input has been read. A computation with input $a^i b^j$ enters state q_2 upon processing the first b. To read the entire input string, the stack must contain at least j A's. The transition $\delta(q_2, \lambda, A) = [q_2, \lambda]$ will pop any A's remaining on the stack.

   The computation for an input string that has fewer a's than b's or in which an a occurs after a b halts in state q_2 without processing the entire string. Thus strings with either of these forms are rejected.

   b) The state diagram of M is



   d) To show that the strings $aabb$ and $aaab$ are in L(M), we trace a computation of M that accepts these strings.

State	String	Stack
q_0	$aabb$	λ
q_0	abb	A
q_0	bb	AA
q_2	b	A
q_2	λ	λ

State	String	Stack
q_0	$aaab$	λ
q_0	aab	A
q_0	ab	AA
q_0	b	AAA
q_2	λ	AA

q_2	λ	A
q_2	λ	λ

Both of these computations terminate in the accepting state q_2 with an empty stack.

3. d) The pushdown automaton defined by the transitions

$$\begin{aligned}
\delta(q_0, \lambda, \lambda) &= \{[q_1, C]\}\\
\delta(q_1, a, A) &= \{[q_2, A]\}\\
\delta(q_1, a, C) &= \{[q_2, C]\}\\
\delta(q_1, b, B) &= \{[q_3, B]\}\\
\delta(q_1, b, C) &= \{[q_3, C]\}\\
\delta(q_1, a, B) &= \{[q_1, \lambda]\}\\
\delta(q_1, b, A) &= \{[q_1, \lambda]\}\\
\delta(q_1, \lambda, C) &= \{[q_5, \lambda]\}\\
\delta(q_2, \lambda, \lambda) &= \{[q_1, A]\}\\
\delta(q_3, \lambda, \lambda) &= \{[q_4, B]\}\\
\delta(q_4, \lambda, \lambda) &= \{[q_1, B]\}
\end{aligned}$$

accepts strings that have twice as many a's as b's. A computation begins by pushing a C onto the stack, which serves as a bottom-marker throughout the computation. The stack is used to record the relationship between the number of a's and b's scanned during the computation. The stacktop will be a C when the number of a's processed is exactly twice the number of b's processed. The stack will contain n A's if the automaton has read n more a's than b's. If n more b's than a's have been read, the stack will hold $2n$ B's.

When an a is read with an A or C on the top of the stack, an A is pushed onto the stack. This is accomplished by the transition to q_2. If a B is on the top of the stack, the stack is popped removing one b. If a b is read with a C or B on the stack, two B's are pushed onto the stack. Processing a b with an A on the stack pops the A.

The lone accepting state of the automaton is q_5. If the input string has the twice as many a's as b's, the transition to q_5 pops the C, terminates the computation, and accepts the string.

h) The language $\mathrm{L} = \{a^i b^j \mid 0 \leq i \leq j \leq 2 \cdot i\}$ is generated by the context-free grammar

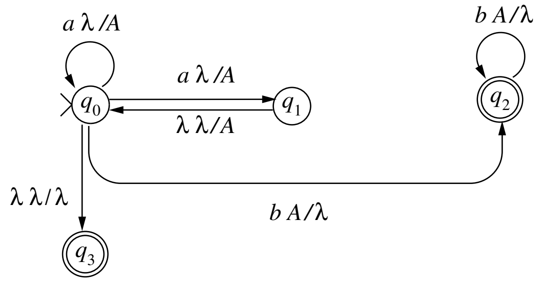
$$\begin{aligned}
S &\to aSB \mid \lambda\\
B &\to bb \mid b
\end{aligned}$$

The B rule generates one or two b's for each a. A pushdown automaton M that accepts L uses the a's to record an acceptable number of matching b's on the stack. Upon processing an a, the computation nondeterministically pushes one or two A's onto the stack. The transitions of M are

$$\begin{aligned}
\delta(q_0, a, \lambda) &= \{[q_1, A]\}\\
\delta(q_0, \lambda, \lambda) &= \{[q_3, \lambda]\}\\
\delta(q_0, a, \lambda) &= \{[q_0, A]\}\\
\delta(q_0, b, A) &= \{[q_2, \lambda]\}\\
\delta(q_1, \lambda, \lambda) &= \{[q_0, A]\}\\
\delta(q_2, b, A) &= \{[q_2, \lambda]\}.
\end{aligned}$$

The states q_2 and q_3 are the accepting states of M. The null string is accepted in q_3. For a nonnull string $a^i b^j \in \mathrm{L}$, one of the computations will push exactly j A's onto the stack. The stack is emptied by processing the b's in q_2.

The state diagram of the PDA is



8. Let M = (Q, Σ, Γ, δ, q_0, F) be a PDA that accepts L by final state and empty stack. We construct a PDA M′ = (Q ∪ $\{q_s, q_f\}$, Σ, Γ ∪ $\{C\}$, δ', q_s, $\{q_f\}$) from M by adding a new stack symbol C, new states q_s and q_f, and transitions for these states. The transition function δ' is constructed by augmenting δ with the transitions

$$\delta'(q_i,\ \lambda,\ C) = \{[q_f,\ \lambda]\} \quad \text{for all } q_i \in \mathrm{F}$$
$$\delta'(q_s,\ \lambda,\ \lambda) = \{[q_0,\ C]\}$$

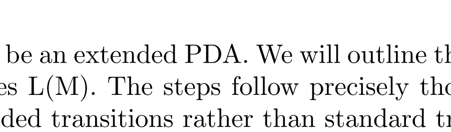
Let w be a string accepted by M in an accepting state q_i. A computation of M′ with input w begins by putting C on the stack and entering q_0. The stack symbol C acts as a marker designating the bottom of the stack. From q_0, M′ continues with a computation identical to that of M with input w. If the computation of M with w ends in an accepting state q_i with an empty stack, the computation of M′ is completed by the λ-transition from q_i to q_f that pops the C from the stack. Since the computation of M′ halts in q_f, $w \in$ L(M′).

We must also guarantee that the any string w accepted by M′ by final state is accepted by M by final state and empty stack. The lone final state of M′ is q_f, which can be entered only by a λ-transition from a final state q_i of M with C as the stacktop. Thus the computation of M′ must have the form

$$\begin{aligned} &[q_s, w, \lambda] \\ \vdash\ &[q_0, w, C] \\ \overset{*}{\vdash}\ &[q_i, \lambda, C] \\ \vdash\ &[q_f, \lambda, \lambda] \end{aligned}$$

Deleting the first and last transition provides a computation of M that accepts w by final state and empty stack.

12. The state diagram for the extended PDA obtained from the grammar is



16. Let M = (Q, Σ, Γ, δ, q_0, F) be an extended PDA. We will outline the construction of a context-free grammar that generates L(M). The steps follow precisely those given in Theorem 7.3.2, except we begin with extended transitions rather than standard transitions.

The first step is to construct an equivalent PDA M′ with transition function δ' by augmenting δ with the transitions:

$[q_j, B_1 \cdots B_n A] \in \delta'(q_i, u, B_1 \cdots B_n A)$ for every $A \in \Gamma$, whenever $[q_j, \lambda] \in \delta(q_i, u, B_1 \cdots B_n)$ is a transition of M. The string of variables $B_1 \cdots B_n$ in the preceding rules may be empty.

These rules transform a transition of M that does not remove an element from the stack into one that initially pops the stack and later replaces the same symbol on the top of the stack. Any string accepted by a computation that utilizes a new transition can also be obtained by applying the original transition; hence, L(M) = L(M′).

A grammar G = (V, Σ, P, S) is constructed from the transitions of M′. The alphabet of G is the input alphabet of M′. The variables of G consist of a start symbol S and objects of the form $\langle q_i, A, q_j\rangle$ where the q's are states of M′ and $A \in \Gamma \cup \{\lambda\}$. The variable $\langle q_i, A, q_j\rangle$ represents a computation that begins in state q_i, ends in q_j, and removes the symbol A from the stack. The rules of G are constructed as follows:

1. $S \to \langle q_0, \lambda, q_j\rangle$ for each $q_j \in$ F.
2. Each transition $[q_j, B_1 \cdots B_n] \in \delta'(q_i, x, A)$, where $A \in \Gamma \cup \{\lambda\}$, generates the set of rules

$$\{\langle q_i, A, q_k\rangle \to x\langle q_j, B_1, q_{m_1}\rangle\langle q_{m_1}, B_2, q_{m_2}\rangle \cdots \langle q_{m_{n-1}}, B_m, q_k\rangle \mid q_{m_i}, q_k \in \mathrm{Q}\}.$$

3. Each transition $[q_j, B_1 \cdots B_n A] \in \delta'(q_i, x, A)$, where $A \in \Gamma$, generates the set of rules

$$\{\langle q_i, A, q_k\rangle \to x\langle q_j, B_1, q_{m_1}\rangle\langle q_{m_1}, B_2, q_{m_2}\rangle \cdots \langle q_{m_{n-1}}, B_m, q_{m_n}\rangle\langle q_{m_n}, A, q_k\rangle \mid q_k, q_{m_i} \in \mathrm{Q}\}.$$

4. For each state $q_k \in$ Q,

$$\langle q_k, \lambda, q_k\rangle \to \lambda.$$

A derivation begins with a rule of type 1 whose right-hand side represents a computation that begins in state q_0, ends in a final state, and terminates with an empty stack, in other words, a successful computation in M′. Rules of types 2 and 3 trace the action of the machine. Rules of type 4 are used to terminate derivations. The rule $\langle q_k, \lambda, q_k\rangle \to \lambda$ represents a computation from a state q_k to itself that does not alter the stack, that is, the null computation.

The proof that the rules generate L(M) follows the same strategy as Theorem 7.3.2. Lemmas 7.3.3 and 7.3.4 relate the derivations from a variable $\langle q_i, A, q_k\rangle$ to computations of M.

17. a) Assume that language L consisting of strings over $\{a\}$ whose lengths are a perfect square is context-free. By the pumping lemma, there is a number k such that every string in L with length k or more can be written $uvwxy$ where

i) $length(vwx) \le k$
ii) v and x are not both null
iii) $uv^iwx^iy \in$ L, for $i \ge 0$.

The string a^{k^2} must have a decompositon $uvwxy$ that satisfies the preceding conditions. Consider the length of the string $z = uv^2wx^2y$ obtained by pumping $uvwxy$.

$$\begin{aligned} length(z) &= length(uv^2wx^2y) \\ &= length(uvwxy) + length(u) + length(v) \\ &= k^2 + length(u) + length(v) \\ &\le k^2 + k \\ &< (k+1)^2 \end{aligned}$$

Since the length of z is greater than k^2 but less than $(k+1)^2$, we conclude that $z \notin$ L and that L is not context-free.

f) Assume that the language L consisting of prefixes of the string

$$abaabaaabaaaab \dots ba^n ba^{n+1} b \dots$$

is context-free and let k be the number specified by the pumping lemma. Consider the string $z = abaab \cdots ba^k b$, which is in the language and has length greater than k. Thus z can be written $uvwxy$ where

i) $length(vwx) \leq k$

ii) v and x are not both null

iii) $uv^i wx^i y \in$ L, for $i \geq 0$.

To show that the assumption that L is context-free produces a contradiction, we examine all possible decompositions of z that satisfy the conditions of the pumping lemma. By (ii), one or both of v and x must be nonnull. In the following argument we assume that $v \neq \lambda$.

Case 1: v has no b's. In this case, v consists solely of a's and lies between two consecutive b's. That is, v occurs in z in a position of the form

$$\dots ba^n ba^i va^j ba^{n+2} b \dots$$

where $i + length(v) + j = n+1$. Pumping v produces an incorrect number of a's following $ba^n b$ and, consequently, the resulting string is not in the language.

Case 2: v has two or more b's. In this case, v contains a substring $ba^n b$. Pumping v produces a string with two substrings of the form $ba^n b$. No string with this property is in L.

Case 3: v has one b. Then v can be written $a^i ba^j$ and occurs in z as

$$\dots ba^{n-1} ba^{n-i} va^{n+1-j} b \dots .$$

Pumping v produces the substring

$$\dots ba^{n-1} ba^{n-i} a^i ba^j a^i ba^j a^{n+1-j} b \dots = \dots ba^{n-1} ba^n ba^{j+i} ba^{n+1} b \dots ,$$

which cannot occur in a string in L.

Regardless of its makeup, pumping any nonnull substring v of z produces a string that is not in the language L. A similar argument shows that pumping x produces a string not in L whenever x is nonnull. Since one of v or x is nonnull, there is no decomposition of z that satisfies the requirements of the pumping lemma and we conclude that the language is not context-free.

18. a) The language $\text{L}_1 = \{a^i b^{2i} c^j \mid i, j \geq 0\}$ is generated by the context-free grammar

$$\begin{aligned} &S \to AC \\ &A \to aAbb \mid \lambda \\ &C \to cC \mid \lambda \end{aligned}$$

b) Similarly, $\text{L}_2 = \{a^j b^i c^{2i} \mid i, j \geq 0\}$ is generated by

$$\begin{aligned} &S \to AB \\ &A \to aA \mid \lambda \\ &B \to bBcc \mid \lambda \end{aligned}$$

c) Assume $L_1 \cap L_2 = \{a^i b^{2i} c^{4i} \mid i \geq 0\}$ is context-free and let k be the number specified by the pumping lemma. The string $z = a^k b^{2 \cdot k} c^{4 \cdot k}$ must admit a decomposition $uvwxy$ that satisfies the conditions of the pumping lemma. Because of the restriction on its length, the substring vwx must have the form a^i, b^i, c^i, $a^i b^j$, or $b^i c^j$. Pumping z produces the string uv^2wx^2y. This operation increases the number of at least one, possibly two, but not all three types of terminals in z. Thus $uv^2wx^2y \notin$ L, contradicting the assumption that L is context-free.

20. Let z be any string of length 2 or more in L. Then z can be written in the form $z = pabq$ or $z = pbaq$, where p and q are strings from Σ^*. That is, z must contain the substring ab or the substring ba. A decomposition of z that satisfies the pumping lemma for $z = pabq$ is

$$u = p,\ v = ab,\ w = \lambda,\ x = \lambda,\ y = q$$

A similar decomposition can be produced for strings of the form $pbaq$.

22. Let L be a linear language. Then there is a linear grammar G = (V, Σ, S, P) that generates L and, by Theorem 4.3.3, we may assume that G has no chain rules. Let r by the number of variables of G and t be the maximum number of terminal symbols on the right-hand of any rule. Any derivation that produces a string of length $k = (r + 1)t$ or more must have at least $r + 1$ rule applications. Since there are only r variables, variable that is transformed on the $r + 1$st rule application must have occurred previously in the derivation.

Thus any derivation of a string $z \in$ L of length greater than k can be written

$$\begin{aligned} S &\overset{*}{\Rightarrow} uAy \\ &\overset{*}{\Rightarrow} uvAxy \\ &\overset{*}{\Rightarrow} uvwxy, \end{aligned}$$

where the second occurrence of A in the preceding derivation is the $r + 1$st rule application. Repeating the subderivation $A \overset{*}{\Rightarrow} vAx$ i times prior to completing the derivation with the subderivation $A \overset{*}{\Rightarrow} w$ produces a string uv^iwx^iy in L.

Since the sentential form $uvAxy$ is generated by at most r rule applications, the string $uvxy$ must have length less than k as desired.

26. Let G = (V, Σ, S, P) be a context-free grammar. A grammar G′ that generates L(G)^R is built from G. The variables, alphabet, and start symbol of G′ are the same as those of G. The rules of G′ are obtained by reversing the right-hand of the rules of G; $A \to w^R \in$ P′ if, and only if, $A \to w \in$ P.

Induction on the length of derivations is used to show that a string u is a sentential form of G if, and only if, u^R is a sentential form of G′. We consider sentential forms of G generated by leftmost derivations and of G′ by rightmost derivations.

Basis: The basis consists of sentential forms produced by derivations of length one. These are generated by the derivation $S \Rightarrow u$ in G and $S \Rightarrow u^R$ in G′ where $S \to u$ is a rule in P.

Inductive hypothesis: Assume that u is a sentential form of G derivable by n or fewer rule applications if, and only if, u^R is a sentential form of G′ derivable by n or fewer rule applications.

Inductive step: Let u be a sentential form of G derived by $n + 1$ rule applications. The derivation has the form

$$S \overset{n}{\Rightarrow} xAy \Rightarrow xwy = u$$

where A is the leftmost variable in xAy and $A \to w$ is a rule in P.

By the inductive hypothesis, $y^R A x^R$ is a sentential form of G′ derivable by n rule applications. Using the rule $A \to w^R$ in P′, u^R is derivable by the rightmost derivation

$$S \overset{n}{\Rightarrow} y^R A x^R \Rightarrow y^R w^R x^R = u^R.$$

In a like manner, we can prove that the reversal of every sentential form derivable in G′ is derivable in G. It follows that L(G′) = L(G)^R.

27. Let G = (V, Σ, S, P) be a context-free grammar that generates L and let G′ = (V ∪{X}, Σ − {a}, S, P′ ∪ {$X \to \lambda$}) be the grammar obtained from G by replacing all occurrences of the terminal a in the right-hand side of the rules of G with a new variable X. The set of rules of G′ is completed by adding the rule $X \to \lambda$. The grammar G′ generates er_a(L).

The proof follows from a straightforward examination of the derivations of G and G′. For simplicity, let $u : y/x$ represent a string $u \in \Sigma^*$ in which all occurrences of the symbol y in u are replaced by x.

Let u be a string in L. Then there is a derivation $S \overset{*}{\Rightarrow} u$ in G. Performing the same derivation with the transformed rules of G′ yields $S \overset{*}{\Rightarrow} u : a/X$. Completing the derivation with the rule $X \to \lambda$ produces $u : a/\lambda$. Thus, for every $u \in$ L, the erasure of u is in L(G′) and er_a(L) ⊆ L(G′).

Conversely, let u be any string in L(G′). Reordering the rules in the derivation of u so that applications of the rule $X \to \lambda$ come last, u is generated by a derivation of the form $S \overset{*}{\Rightarrow} v \overset{*}{\Rightarrow} u$. The subderivation $S \overset{*}{\Rightarrow} v$ uses only modified rules of G. Thus $v : X/a$ is in G and u is in er_a(L). It follows that L(G′) ⊆ er_a(L).

The two preceding inclusions show that L(G′) = er_a(L) and consequently that er_a(L) is context-free.

28. a) Let L be a context-free language over Σ, G = (V, Σ, S, P) a context-free grammar that generates L, and h a homomorphism from Σ^* to Σ^*. We will build a context-free grammar G′ that generates h(L) = {$h(w)$ | $w \in$ L}.

Let $\Sigma = \{a_1, \ldots, a_n\}$ and $h(a_i) = x_i$. The rules of G′ are obtained from those of G by substitution. If $A \to u$ is a rule of G, then

$$A \to u : a_i/u_i, \text{ for all } i$$

is a rule of G′. (See the solution to Exercise 27 for an explanation of the notation.)

To show that h(L) is context-free, it suffices to show that it is generated by the grammar G′. For any string $w \in$ L, mimicking its derivation in G using the corresponding rules of G′ produces $h(w)$. Moreover the derivations in G′ can only produce strings of this form and L(G′) = h(L) as desired.

b) First we note that a mapping $h : \Sigma \to \Sigma$ on the elements of the alphabet can be extended to a homomorphism on Σ^* as follows: a string $u = a_{i_1} a_{i_2} \cdots a_{i_k}$ maps to the string $h(a_{i_1}) h(a_{i_2}) \cdots h(a_{i_k})$.

Using the preceding observation, the mapping on Σ defined by $h(a) = \lambda$, $h(b) = b$, for all $b \neq a$, is a homomorphism from Σ^* to Σ^*. The image of this mapping, which is context-free by part a), is er_a(L).

c) The mapping from $\{a^i b^i c^i \mid i \geq 0\}$ that erases the c's maps a noncontext-free language to a context-free language.

30. b) Let $\Sigma = \{a, b, c\}$ and let $h : \Sigma^* \to \Sigma^*$ be the homomorphism obtained by using concatenation to extend the mapping $h(a) = a$, $h(b) = bb$, and $h(c) = ccc$ from elements of Σ to strings over Σ^*. The inverse image of $\{a^i b^{2i} c^{3i} \mid i \geq 0\}$ under the homomorphism h is $\{a^i b^i c^i \mid i \geq 0\}$. In Example 7.4.1, the pumping lemma was used to show that the latter language is not context-free. Since the class of context-free languages is closed under inverse homomorphic images (Exercise 29), it follows that $\{a^i b^{2i} c^{3i} \mid i \geq 0\}$ is not context-free.

Chapter 8

Turing Machines

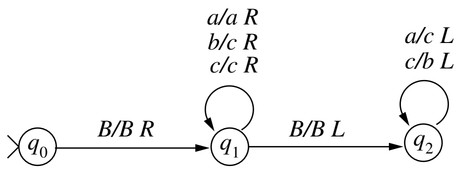
1. a)

$$\begin{aligned} & q_0BaabcaB \\ \vdash\ & Bq_1aabcaB \\ \vdash\ & Baq_1abcaB \\ \vdash\ & Baaq_1bcaB \\ \vdash\ & Baacq_1caB \\ \vdash\ & Baaccq_1aB \\ \vdash\ & Baaccaq_1B \\ \vdash\ & Baaccq_2aB \\ \vdash\ & Baacq_2ccB \\ \vdash\ & Baaq_2cbcB \\ \vdash\ & Baq_2abbcB \\ \vdash\ & Bq_2acbbcB \\ \vdash\ & q_2BccbbcB \end{aligned}$$

b)

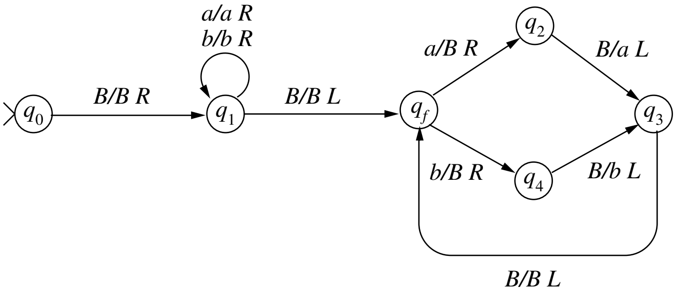
$$\begin{aligned} & q_0BbcbcB \\ \vdash\ & Bq_1bcbcB \\ \vdash\ & Bcq_1cbcB \\ \vdash\ & Bccq_1bcB \\ \vdash\ & Bcccq_1cB \\ \vdash\ & Bccccq_1B \\ \vdash\ & Bcccq_2cB \\ \vdash\ & Bccq_2cbB \\ \vdash\ & Bcq_2cbbB \\ \vdash\ & Bq_2cbbbB \\ \vdash\ & q_2BbbbbB \end{aligned}$$

c) The state diagram of M is



d) The result of a computation is to replace the *a*'s in the input string with *c*'s and the *c*'s with *b*'s.

3. a) Starting with the rightmost symbol in the input and working in a right-to-left manner, the machine diagrammed below moves each symbol one position to the right.



On reading the first blank following the input, the head moves to the left. If the input is the null string, the computation halts in state q_f with the tape head in position zero as desired. Otherwise, an a is moved to the right by transitions to states q_2 and q_3. Similarly, states q_4 and q_3 shift a b. This process is repeated until the entire string has been shifted.

c) A Turing machine to insert blanks between the symbols in the input string can iteratively use the machine from Exercise 3 a) to accomplish the task. The strategy employed by the machine is

i. Mark tape position 0 with a #.

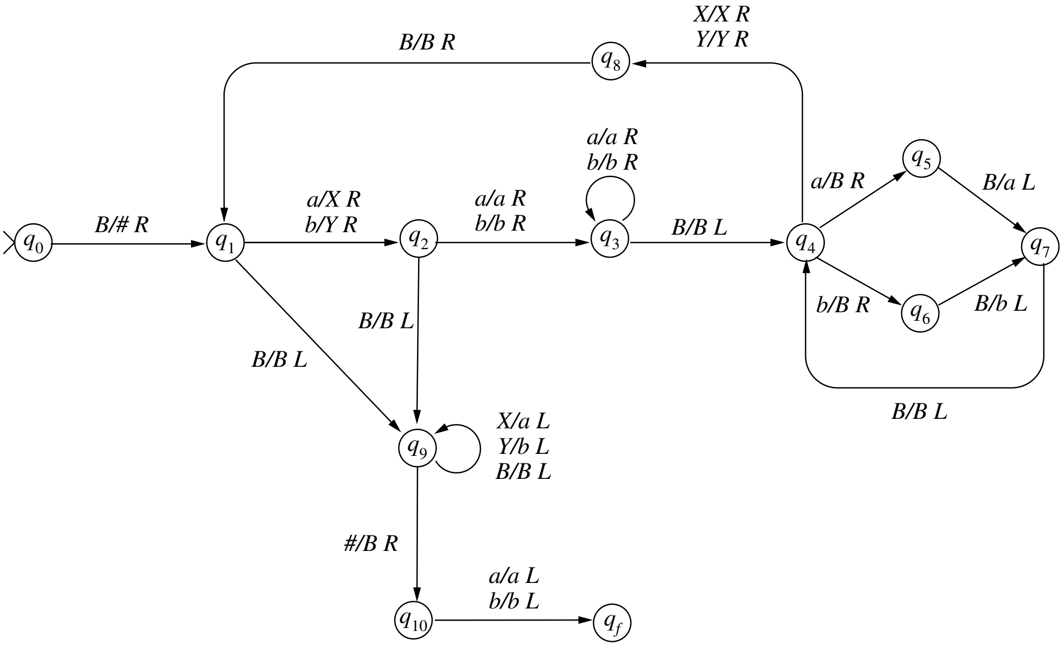
ii. If the input string is not empty, change the first symbol to X or Y to record if it was an a or b, respectively.

iii. Move to the end of the string and use strategy from the machine in part a) to move the unmarked input on square to the right.

iv. If there are no unmarked symbols on the tape, change all of the X's to a's and Y's to b's, and halt in position 0.

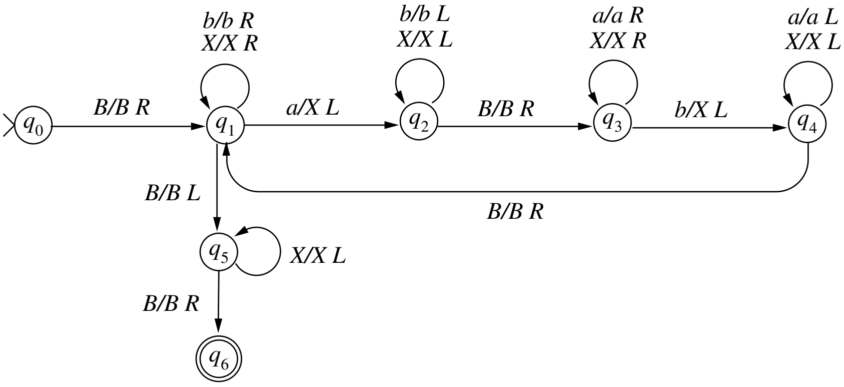
v. Otherwise, repeat the process on the unmarked string.

A Turing machine that accomplishes these tasks is



Processing the null string uses the arc from q_1 to q_f. Shifting a string one square to the right is accomplished by states q_3 to q_7. The arc from q_4 to q_8 when the shift has been completed. If another shift is needed, the entire process is repeated by entering q_1.

5. c) A computation of



begins by finding the first a on the tape and replacing it with an X (state q_1). The tape head is then returned to position zero and a search is initiated for a corresponding b. If a b is encountered in state q_3, an X is written and the tape head is repositioned to repeat the cycle q_1, q_2, q_3, q_4. If no matching b is found, the computation halts in state q_3 rejecting the input. After all the a's have been processed, the entire string is read in q_1 and q_5 is entered upon reading the trailing blank. The computation halts in the accepting state q_6 if no b's remain on the tape.

7. We refer to the method of acceptance defined in the problem as *acceptance by entering.* We show that the languages accepted by entering are precisely those accepted by final state, that is, the recursively enumerable languages.

Let M = (Q, Σ, Γ, δ, q_0, F) be a Turing machine that accepts by entering. The machine M′ = (Q, Σ, Γ, δ', q_0, F) with transition function δ' defined by

i) $\delta'(q_i, x) = \delta(q_j, x)$ for all $q_i \notin$ F

ii) $\delta'(q_i, x)$ is undefined for all $q_i \in$ F

accepts L(M) be final state. Computations of M′ are identical to those of M until M′ enters an accepting state. When this occurs, M′ halts accepting the input. A computation of M that enters an accepting state halts in that state in M′.

Now let M = (Q, Σ, Γ, δ, q_0, F) be a Turing machine that accepts by final state. A computation of M may enter and leave accepting states prior to termination; the intermediate states have no bearing on the acceptance of the input. We construct a machine M′ that accepts L(M) by entering. M′ is defined by the quintuple (Q ∪ $\{q_f\}$, Σ, Γ, δ', q_0, $\{q_f\}$). The transitions of M′ are defined by

i) $\delta'(q_i, x) = \delta(q_i, x)$ whenever $\delta(q_i, x)$ is defined

ii) $\delta'(q_i, x) = [q_f, x, R]$ if $q_i \in$ F and $\delta(q_i, x)$ is undefined.

A computation of M that accepts an input string halts in an accepting state q_i. The corresponding computation of M′ reaches q_i and then enters q_f, the sole accepting state of M′. Thus entering q_f in M′ is equivalent to halting in an accepting state of M and L(M) = L(M′).

8. Clearly, every recursively enumerable language is accepted by a Turing machine in which stationary transitions are permitted. A standard Turing machine can be considered to be such a machine whose transition function does not include any stationary transitions.

   Let M be a Turing machine (Q, Σ, Γ, δ, q_0, F) with stationary transitions. We construct a standard Turing machine M′ that accepts L(M). The transition function δ' of M′ is constructed from that of M. A transition of M that designates a movement of the tape head generates an identical transition of M′.

   i) $\delta'(q_i, x) = \delta(q_i, x)$ whenever $\delta(q_i, x) = [q_j, y, d]$ where $d \in \{L, R\}$

   A pair of standard transitions are required to perform the same action as a stationary transition $\delta(x, q_i) = [q_j, y, S]$. The first transition prints a y, moves right and enters a new state. Regardless of the symbol being scanned, the subsequent transition moves left to the original position and enters q_j.

   ii) $\delta'(q_i, x) = [q_t, y, R]$ for every transition $\delta(q_i, x) = [q_j, y, S]$, where a new state $q_t \notin$ Q is is added for every stationary transition

   iii) $\delta'(q_t, z) = [q_j, z, L]$ for every $z \in \Gamma$ and every state q_t added in (ii)

10. a) Let M = (Q, Σ, Γ, δ, q_0, F) be a standard Turing machine. We will build a context-sensitive Turing machine M′ that accepts L(M). The components of M′ are the same as those of M except for the transition function δ'. For every transition $\delta(q_i, x) = [q_j, y, d]$ of M, M′ has the transitions

$$\delta'(q_i, xz) = [q_j, yz, d] \qquad \text{for all } z \in \Gamma.$$

   Intuitively, a computation of M′ ignores the second input symbol in the transition.

   b) Assume that the context-sensitive transition function has transitions

$$\delta'(q_i, xz_i) = [q_{j_i}, y_i z_i, d]$$

   for symbols $z_1, z_2, \ldots, z_n$ from the tape alphabet. The computation of the standard machine should read the x in state q_i, determine the symbol on the right, and rewrite the original position accordingly. This can be accomplished using the transitions

$$\begin{aligned}
&\delta'(q_i, x) = [q_i', x', R]\\
&\delta'(q_i', z_1) = [q_{i_1}, z_1, L]\\
&\delta'(q_{i_1}, x) = [q_{j_1}, y_1, d]\\
&\qquad\quad \vdots\\
&\delta'(q_i', z_n) = [q_{i_n}, z_n, L]\\
&\delta'(q_{i_n}, x') = [q_{j_n}, y_n, d]\\
&\delta'(q_i', w) = [q_i, w, L] \qquad \text{for } w \neq z
\end{aligned}$$

   where x' is a symbol not in tape alphabet of the context-sensitive machine. If the context-sensitive transition is not applicable, the sequence of standard transitions will check the symbol to the right and halt in state q_i.

   c) By part a) we see that every recursively enumerable language is accepted by a context-sensitive Turing machine. Part b) provides the opposite inclusion; given a context-sensitive machine we can build a standard machine that accepts the same language. Thus the family of languages accepted by these types of machines are identical.

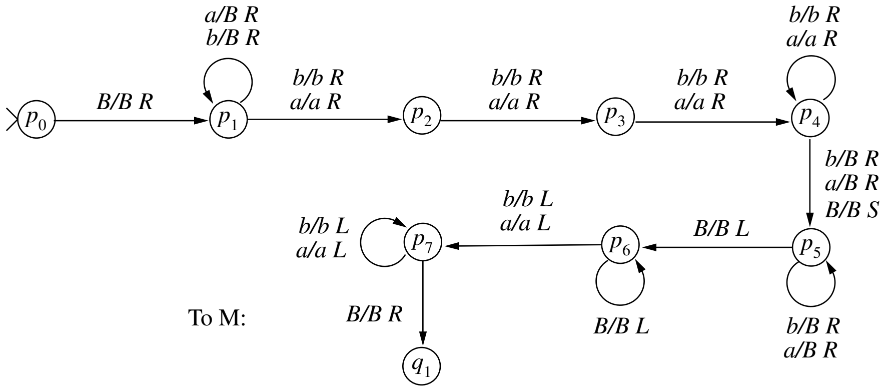
14. b) The two-tape Turing machine with tape alphabet $\{a, b, B\}$ and accepting state is q_4 defined by the transitions

$$\begin{array}{lll}
\delta(q_0, B, B) & = [q_1; B, R; B, R] & \\
\delta(q_1, x, B) & = [q_1; x, R; x, R] & \text{for every } x \in \{a, b\} \\
\delta(q_1, B, B) & = [q_2; B, L; B, S] & \\
\delta(q_2, x, B) & = [q_2; x, L; B, S] & \text{for every } x \in \{a, b\} \\
\delta(q_2, B, B) & = [q_3; B, R; B, L] & \\
\delta(q_3, x, x) & = [q_3; x, R; x, L] & \text{for every } x \in \{a, b\} \\
\delta(q_3, B, B) & = [q_4; B, S; B, S] &
\end{array}$$

accepts the palindromes over {a, b}. The input is copied onto tape 1 in state q_1. State q_2 returns the head reading tape 1 to the initial position. With tape head 1 moving left-to-right and head 2 moving right-to-left, the strings on the two tapes are compared. If both heads simultaneously read a blank, the computation terminates in q_4.

The maximal number of transitions of a computation with an input string of length n occurs when the string is accepted. Tape head 1 reads right-to-left, left-to-right, and then right-to-left through the input. Each pass requires $n + 1$ transitions. The computation of a string that is not accepted halts when the first mismatch of symbols on tape 1 and tape 2 is discovered. Thus, the maximal number of transitions for an input string of length n is $3(n + 1)$.

19. A deterministic machine M was constructed in Exercise 5 (c) that accepts strings over $\{a, b\}$ with the same number of a's and b's. Using M, a composite machine is constructed to determine whether an input string contains a substring of length three or more with the same number of a's and b's. Nondeterminism is used to 'choose' a substring to be examined. States p_0 to p_7 nondeterministically select a substring of length three or more from the input.

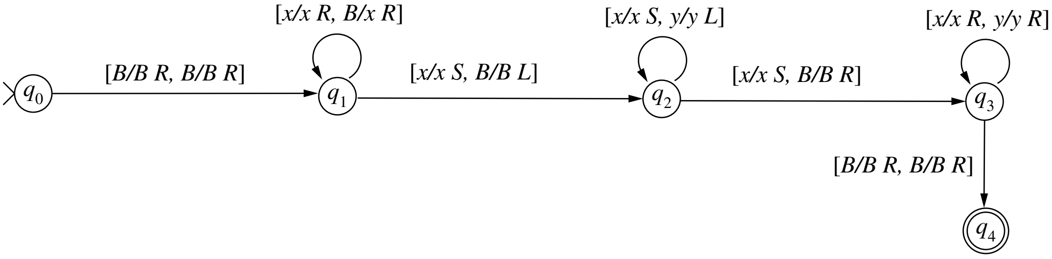


The transition to state q_1 of M begins the computation that checks whether the chosen substring has the same number of a's and b's. The accepting states of the composite machine are the accepting states of M.

This problem illustrates the two important features in the design of Turing machines. The first is the ability to use existing machines as components in more complex computations. The machine constructed in Exercise 5 (c) provides the evaluation of a single substring needed in this computation. The flexibility and computational power that can be achieved by combining submachines will be thoroughly examined in Chapter 9.

The second feature is the ability of a nondeterministic design to remove the need for the consideration of all possible substrings. The computation of a deterministic machine to solve this problem must select a substring of the input and decide if it satisfies the conditions. If not, another substring is is generated and examined. This process must be repeated until an acceptable substring is found or all substrings have been generated and evaluated.

21. The two-tape nondeterministic machine whose state diagram is given below accepts $\{uu \mid u \in \{a,b\}^*\}$. In the diagram, x and y represent arbitrary elements from $\{a,b\}$.



The computation begins by copying an initial segment of the input on tape 1 to tape 2. A nondeterministic transition to state q_2 represents a guess that the current position of the head on tape 1 is the beginning of the second half of the string. The head on tape 2 is repositioned at the leftmost square in state q_2. The remainder of the string on tape 1 is then compared with the initial segment that was copied onto tape 2. If a blank is read simultaneously on these tapes, the string is accepted. Finding any mismatch between the tapes in this comparison causes the computation to halt without accepting the input.

The maximal number of transitions occurs when all but the final symbol of the input string is copied onto tape 2. In this case, the head on tape 2 moves to tape position n, back to position 0, and begins the comparison. This can be accomplished in $2(length(w) + 1)$ transitions.

23. Let M be a nondeterministic Turing machine that halts for all inputs. The technique introduced in the construction of an equivalent deterministic machine M′ given in Section 9.7 demonstrates that L(M) is recursively enumerable. M′ systematically simulates all computations of M, beginning with computations of length 0, length 1, length 2, etc. The simulation of a computation of M is guided by a sequence of numbers that are generated on tape 3. When one of the simulated computations accepts the input string, M′ halts and accepts. If the input is not in L(M), M′ indefinitely continues the cycle of generating a sequence of numbers on tape 3 and simulating the computation of M associated with that sequence.

Unfortunately, the preceding approach cannot be used to show that L is recursive since the computation of M′ will never terminate for a string that is not in the language. To show that L(M) is recursive, we must construct a deterministic machine that halts for all inputs. This can be accomplished by adding a fourth tape to M′ that records the length of the longest simulated computation that has not halted prematurely. A computation terminates prematurely if the simulation terminates before executing all of the transitions indicated by the sequence on tape 3. Upon the termination of a simulated computation that did not halt prematurely, the length of the sequence on tape 3 is recorded on tape 4.

The machine M′ checks the length of non-prematurely terminating computations of M prior to simulating a subsequent computation of M. After a numeric sequence of length n is generated tape 3, tape 4 is checked to see if there has been a previous simulated computation using $n-1$ transitions. If not, M′ halts and rejects. Otherwise, M′ simulates the next computation.

If a string u is in L(M), the computation of M′ will simulate an accepting computation of M and accept u. If u in not in L(M), then all computations of M will terminate. Let k be the length of the longest computation of M with input u. When the simulations reach computations with $k+2$ transitions, M′ will halt and reject u.

26. Let M = (Q, Σ, Γ, δ, q_0, F) be a Turing machine that demonstrates that L is a recursive language. Since every computation of M halts, the machine (Q, Σ, Γ, δ, q_0, Q − F) accepts the complement of L. Thus $\overline{\mathrm{L}}$ is not only recursively enumerable, but also recursive.

    Conversely, we must show that L is recursive when both L and its complement are recursively enumerable. By Theorem 8.3.2, there are machines M_1 = (Q_1, Σ, Γ_1, δ_1, q_0) and M_2 = (Q_2, Σ, Γ_2, δ_2, p_0) that accept L and $\overline{\mathrm{L}}$ by halting. These machines provide sufficient information to decide whether a string u is in L. If $u \in$ L, then the computation of M_1 halts. On the other hand, M_2 halts when $u \notin$ L.

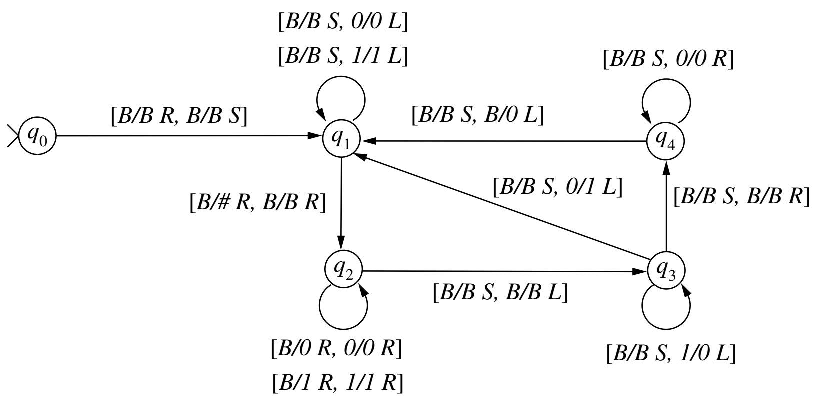
    We construct a two-tape machine M that simulates the computations of both M_1 and M_2. The input alphabet of M is Σ, the tape alphabet is $\Gamma_1 \cup \Gamma_2$ and the start state is r_0. A computation begins by copying the input onto tape 2 and repositioning the tape heads at the initial position. This is accomplished by the transitions defined in (i) through (v) below. M then 'runs' M_1 on tape 1 and M_2 on tape 2. The states of M that simulate the computation of the composite machines are elements of the Cartesian product $\mathrm{Q}_1 \times \mathrm{Q}_2 \times \{1, 2\}$. That is, a state is an ordered triple consisting of a state of M_1, a state of M_2 and an element of $\{1, 2\}$.

    i) $\delta(r_0, B, B) = [r_1; B, R; B, R]$
    ii) $\delta(r_1, x, B) = [r_1; x, R; x, R]$ for every $x \in \Sigma$
    iii) $\delta(r_1, B, B) = [r_2; B, L; B, L]$
    iv) $\delta(r_2, x, x) = [r_2; x, L; x, L]$ for every $x \in \Sigma$
    v) $\delta(r_2, B, B) = [[q_0, p_0, 1]; B, S; B, S]$
    vi) $\delta([q_i, p_j, 1], x, y) = [[q_n, p_j, 2]; s, d_1; y, S]$ whenever $\delta_1(q_i, x) = [q_n, s, d_1]$
    vii) $\delta([q_i, p_j, 2], x, y) = [[q_i, p_m, 1]; x, S; t, d_2]$ whenever $\delta_2(p_j, x) = [p_n, t, d_2]$

    When a 1 occurs in the state, M processes a transition of M_1 on tape 1. The status of the computation of M_2 is unaffected by such a transition. Similarly, a transition of the form $\delta([q_i, p_j, 2], x, y)$ indicates that a transition of M_2 is to be executed on tape 2.

    For every input string, either the computation of M_1 or M_2 terminates. A string is accepted if the computation halts due to the lack of a transition for a configuration on tape 1. Thus the set of final states of M consists of triples of the form $[q_i, p_j, 1]$.

32. The machine in Figure 8.1 generates all nonnull strings over an alphabet $\{\mathit{1}, 2, \ldots, n\}$. The strategy of this machine can be employed to build a two-tape machine that enumerates all strings over $\{\mathit{0}, 1\}$. Intuitively, the strings are generated on tape 2 and then output on tape 1.



The cycle of producing and writing a string on tape 1 begins in q_1. A # is written on tape 1 followed by the string in the input position on tape 2. For the remainder of the cycle, the tape head on tape 1 remains stationary while the next string is generated on tape 2.

33. We will outline the strategy employed by a Turing machine that enumerates all ordered pairs of natural numbers. The number n will be represented by the string 1^{n+1} and we will denote this representation by $\overline{n}$.

    The enumerating machine has three tapes: the enumeration tape, the diagonal tape, and the work tape. The computation consists of an initialization phase followed by a loop that enumerates the ordered pairs.

    Initialization:

    1. *1* is written on the diagonal tape.
    2. *#1B1#* is written on the enumerating tape.

    The initialization writes the ordered pair $[0, 0]$ on enumerating tape. Entering the loop that comprises the main portion of the computation, the diagonal tape holds the value of a diagonal element $[n, n]$ in the grid **N**×**N**. All the elements in the square bounded by $[0, 0]$ and $[n, n]$ have already been written on the enumerating tape.

    Loop:

    1. The number on the diagonal tape is incremented, producing $n + 1$.
    2. The diagonal pair $[n + 1, n + 1]$, $\overline{n+1}B\overline{n+1}\#$, is written on the enumerating tape.
    3. $\overline{n}$ is written on the work tape.
    4. While the work tape is not blank
       - the string on the work is copied tape to the enumeration tape,
       - B is written on the enumeration tape,
       - the string on the diagonal tape is copied to the enumeration tape,
       - # is written on the enumeration tape,
       - a *1* is erased from the work tape.
    5. $\overline{n}$ is written on the work tape.
    6. While the work tape is not blank
       - the string on the diagonal is copied tape to the enumeration tape,
       - B is written on the enumeration tape,
       - the string on the work tape is copied to the enumeration tape,
       - # is written on the enumeration tape,
       - a *1* is erased from the work tape.

    The diagonal element $[n + 1, n + 1]$ is written on the enumerating tape in step 2. The loop beginning at step 4 writes the pairs $[n, n + 1]$, $[n - 1, n + 1]$, $\ldots$, $[0, n + 1]$ on the enumerating tape. In a like manner, step 6 writes the pairs $[n + 1, n]$, $[n + 1, n - 1]$, $\ldots$, $[n + 1, 0]$. Each subsequent iteration through the loop increases the size of the square in the **N**×**N** that has been enumerated.

34. Definition 8.8.1, the definition of the enumeration of a language by a Turing machine, does not require the machine to halt when the entire language has been written on tape 1. After completely listing the strings in a finite language, an enumerating machine may continue indefinitely. For a string not in the language that is lexicographically greater than any string in the language, the halting condition given in Case 2 of Theorem 8.8.7 is never triggered.

Chapter 9

Turing Computable Functions

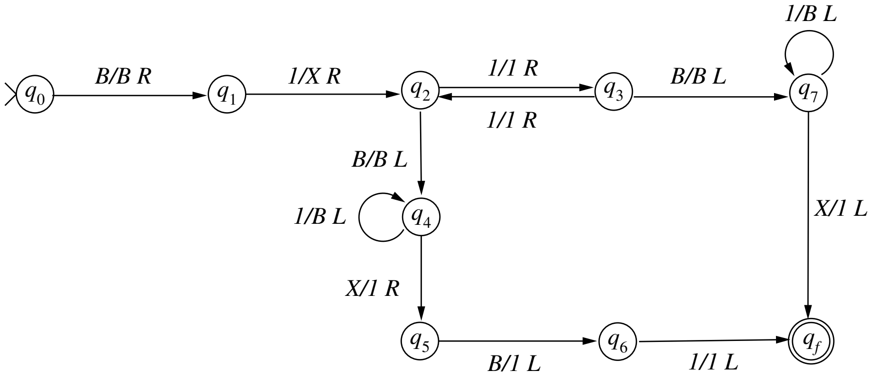
2. Let M a machine that computes the partial characteristic function of a language L. A machine M′ that accepts L can be built by extending M. When a computation of M halts, M′

    - checks to see if $\overline{1}$ is on the tape,
    - if so, M′ enters an accepting state and halts, otherwise
    - M′ enters a rejecting state and halts.

5. d) The Turing machine whose state diagram is given below computes the relation

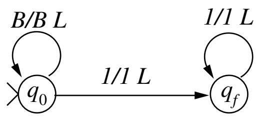
$$even(n) = \begin{cases} 1 & \text{if } n \text{ is even} \\ 0 & \text{otherwise.} \end{cases}$$

The input string represents an even number if a blank is read in q_2 and an odd number if a blank is read in q_3.



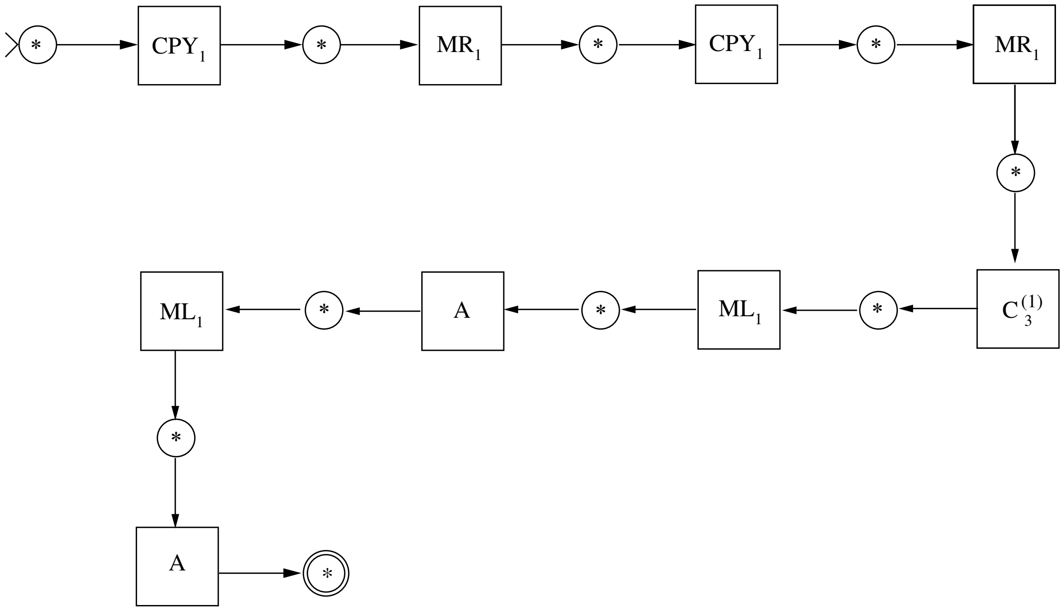
After determining the parity, the input is erased and the appropriate answer is written on the tape.

6. b) The find left macro FL



moves right-to-left reading blanks until it encounters a *1*. The computation then traverses the sequence of *1*'s and halts in state q_f.

7. a) A machine M that computes the function $f(n) = 2n + 3$ can be obtained by combining the copy, add, and constant macros.



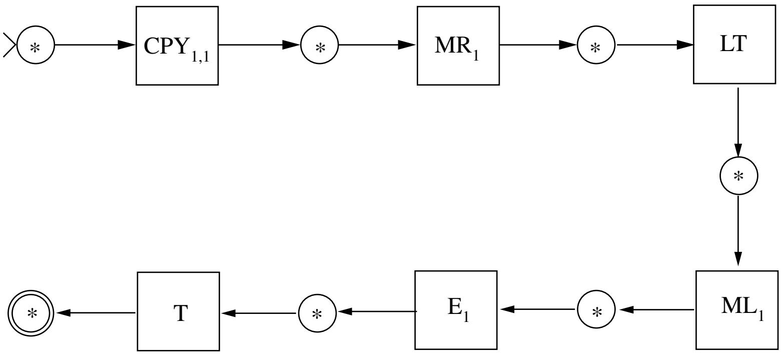
The machine $\mathrm{C}^{(1)}_3$ computes the function $c^{(1)}_3$, which was shown to be Turing computable in Example 9.4.2. Tracing the computation of M with input n illustrates the interaction among the machines that make up M.

Machine	Configuration
	$\underline{B}\overline{n}B$
CPY_1	$\underline{B}\overline{n}B\overline{n}B$
MR_1	$B\overline{n}\underline{B}\overline{n}B$
CPY_1	$B\overline{n}\underline{B}\overline{n}B\overline{n}B$
MR_1	$B\overline{n}B\overline{n}\underline{B}\overline{n}B$
$\mathrm{C}^{(1)}_3$	$B\overline{n}B\overline{n}\underline{B}\overline{3}B$
ML_1	$B\overline{n}\underline{B}\overline{n}B\overline{3}B$
A	$B\overline{n}\underline{B}\overline{n+3}B$
ML_1	$\underline{B}nB\overline{n+3}B$
A	$\underline{B}\overline{2n+3}B$

8. a) A machine can be constructed to compute the characteristic function of the greater than relation

$$gt(n,m) = \begin{cases} 1 & \text{if } n > m \\ 0 & \text{otherwise.} \end{cases}$$

The relations lt and gt satisfy the equality $gt(n,m) = lt(m,n)$. The machine to compute gt uses this relationship and the machine LT from Exercise 5 f):



The action of the composite machine GT is exhibited by tracing a computation. The underscore indicates the position of the tape head in the configuration.

Machine	Configuration
	$\underline{B}\overline{n}B\overline{m}B$
CPY$_{1,1}$	$\underline{B}\overline{n}B\overline{m}B\overline{n}B$
MR$_1$	$B\overline{n}\underline{B}\overline{m}B\overline{n}B$
LT	$B\overline{n}\underline{B}\overline{lt(m,n)}B$
ML$_1$	$\underline{B}\overline{n}B\overline{lt(m,n)}B$
E$_1$	$\underline{B}\dots B\overline{lt(m,n)}B$
T	$\underline{B}\overline{lt(m,n)}B$

10. a) The function

$$f = add \circ (mult \circ (id, id),\ add \circ (id, id))$$

is a one-variable function; the evaluation begins with the input provided to the occurrences of the identity function in f. Evaluating f with argument n yields

$$\begin{aligned} f(n) &= add(mult(id(n), id(n)), add(id(n), id(n))) \\ &= add(mult(n, n), add(n, n)) \\ &= add(n \cdot n, n + n) \\ &= n^2 + 2 \cdot n \end{aligned}$$

b) The function $p^{(2)}_1 \circ (s \circ p^{(2)}_1,\ e \circ p^{(2)}_2)$ is the two-variable empty function. Regardless of the value of the arguments, $e \circ p^{(2)}_2$ produces an undefined value in the second component of the ordered pair. This, in turn, causes the entire function to be undefined.

12. a) Let g be the function $g(n) = 2n$ and let h be

$$h(n) = \begin{cases} n/2 & \text{if } n \text{ is even} \\ \uparrow & \text{otherwise.} \end{cases}$$

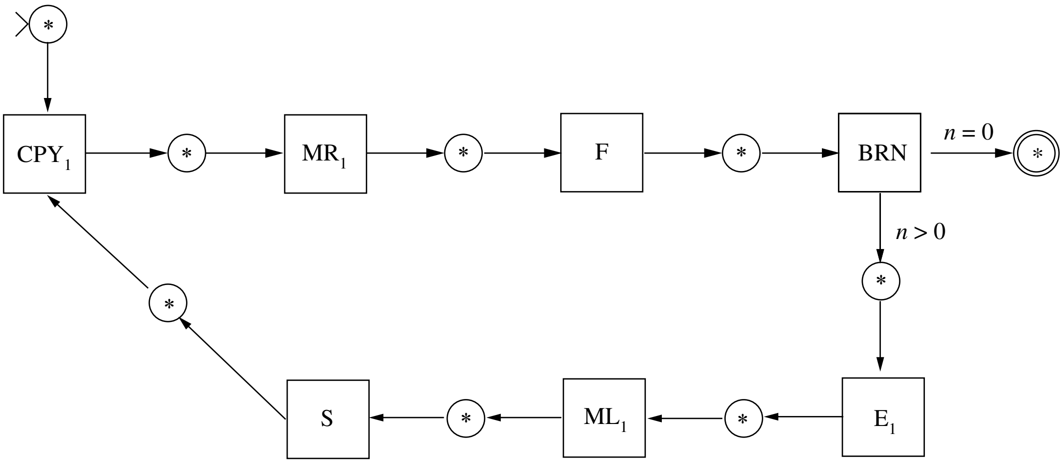
Not only is $h \circ g$ total, it is the identity function.

Modifying h by defining $h(n)$ to be 0 for all odd n produces a pair of functions g and h that provide a solution to Exercise 11 a).

b) Let g be $g(n) = 2n + 1$ and h be the function defined in part a). For any n, $g(n)$ is an odd number and, consequently, $h \circ g$ is undefined.

13. Let f be a Turing computable function and F a machine that computes f. We construct a machine M that returns the first natural number n for which $f(n) = 0$. A computation consists

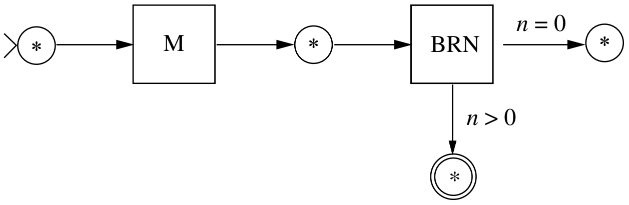
of a cycle that produces $B\overline{n}B\overline{f(n)}B$, for $n = 0, 1, \ldots$, until a value n for which $f(n) = 0$ is encountered.



The computation begins with the counter $\overline{0}$ on the tape. A working copy of $\overline{0}$ is made and F is run on the copy. The BRN macro is used to determine whether the computed value of f is 0. If not, the computed value is erased, the counter is incremented, and the subsequent f value is computed. The cycle of generating a natural number and computing the f value halts when a natural number n is found for which $f(n) = 0$.

If f never assumes the value 0, the computation continues indefinitely. If f is not total, the computation will return n where n is the first value for which $f(n) = 0$ only if f is defined for all $m < n$. Otherwise, M loops upon encountering the first value for which $f(m)\uparrow$.

15. Let R be a Turing computable unary relation and M a machine that computes the characteristic function of R. A machine that accepts R can be built using M and the branch macro.



The computation of M halts with $\overline{0}$ on the tape if, and only if, the input string is not an element of R. Consequently, the composite machine accepts R.

17. Let L $\subseteq \{1\}^+$ be a recursive language and let M = (Q, Σ, Γ, δ, q_0, F) be a machine that accepts L. The language L defines a unary relation R on the natural numbers by

$$R(n) = \begin{cases} 1, & 1^{n+1} \in \mathrm{L}; \\ 0, & \text{otherwise.} \end{cases}$$

To show that R is computable, we build a machine M′ that writes the correct answer on the tape at the end of a computation of M. The process is not quite as simple as it sounds, since M′ must erase the tape prior to writing the answer. That means, M′ must record the furthest right on the tape reached during a computation of M. To do this, we augment the tape alphabet to include symbols $[L, x]$ and $[R, x]$, for each $x \in \Gamma$, that act as left and right endmarkers of the portion of the tape read during a computation of M.

A computation of M$'$ with input u consists of the following steps:

(a) Initially M$'$ writes $[L, B]$ and $[R, B]$ in the squares on either of the input, producing $[L, B]u[R, B]$.

(b) The computation of M on the string BuB is simulated by M$'$ on $[L, B]u[R, B]$.

(c) If the simulation of M indicates that M would halt in an accepting state, M$'$ erases the tape and writes *11* in the input position.

(d) If the simulation of M indicates that M would halt in a nonaccepting state, M$'$ erases the tape and writes *1* in the input position.

The start state of M$'$ is $q_{0'}$ and the transition function is denoted δ'. The transitions to insert the endmarkers are

$$\begin{array}{ll}
\delta'(q_{0'}, B) = [q_{1'}, [L, B], R] & \\
\delta(q_{1'}, x) = [q_{1'}, x, R] & \text{for all } x \in \Sigma \\
\delta(q_{1'}, B) = [q_{2'}, [R, B], L] & \\
\delta(q_{2'}, x) = [q_{2'}, x, L] & \text{for all } x \in \Sigma \\
\delta(q_{2'}, x) = \delta(q_0, x) &
\end{array}$$

Once the endmarkers are in place, the computation of M is simulated by the transitions

$$\begin{array}{ll}
\delta'(q_i, x) = \delta(q_i, x) & \text{for every transition } \delta(q_i, x) \\
\delta'(q_i, [L, x]) = [q_j, [L, y], R] & \text{for every transition } \delta(q_i, x) = [q_j, y, R] \\
\delta'(q_i, [R, x]) = [q_j, [R, y], L] & \text{for every transition } \delta(q_i, x) = [q_j, y, L] \\
\delta'(q_i, [R, x]) = [q_{i,x,1}, y, R] & \text{for every transition } \delta(q_i, x) = [q_j, y, R] \\
\delta'(q_{i,x,1}, B) = [q_{i,x,2}, [R, B], R] & \\
\delta'(q_{i,x,2}, B) = [q_j, B, L] &
\end{array}$$

When a move to the right is indicated with the tape head reading the right endmarker, the appropriate symbol is written and the tape head moves right. At this point it writes the new endmarker $[R, B]$ on the tape and enters q_j to continue the simiulation of M.

When M halts in an accepting state, M$'$ finds the right endmarker, erases the tape, and writes the answer on the tape. For every $q_i \in \mathrm{F}$ with $\delta(q_i, x)$ undefined,

$$\begin{array}{ll}
\delta'(q_i, x) = [q_{3'}, x, R] & \\
\delta'(q_i, [R, x]) = [q_{4'}, B, L] & \text{for } x \in \Gamma \\
\delta'(q_i, [L, x]) = [q_{3'}, [L, x], R] & \text{for } x \in \Gamma \\
\delta'(q_{3'}, x) = [q_{3'}, x, R] & \text{for } x \in \Sigma \\
\delta'(q_{3'}, [R, x]) = [q_{4'}, B, L] & \text{for } x \in \Gamma \\
\delta'(q_{4'}, x) = [q_{4'}, B, L] & \text{for } x \in \Gamma \\
\delta'(q_{4'}, [L, x]) = [q_{5'}, B, R] & \text{for } x \in \Gamma \\
\delta'(q_{5'}, B) = [q_{6'}, \mathit{1}, R] & \\
\delta'(q_{6'}, B) = [q_{7'}, \mathit{1}, L] & \\
\delta'(q_{7'}, \mathit{1}) = [q_f, \mathit{1}, L] &
\end{array}$$

where q_f is the terminal state of M′.

A similar set of transitions write a negative answer when M halts in a nonaccepting state.

19. A comparison of cardinalities will be used to show that there are uncomputable functions that satisfy $f(i) = i$ for all even natural numbers. By Theorem 9.5.1 we know that the set of computable number-theoretic functions is countable.

    Diagonalization can be used to show that there are uncountably many functions that are the identity on the set of even natural numbers. Assume that this set of functions is countably infinite. Countability implies that there is an exhaustive listing $f_0, f_1, \ldots, f_i, \ldots$ of these functions. Consider the function

$$f(n) = \begin{cases} n & \text{if } n \text{ is even} \\ f_{(n-1)/2}(n) + 1 & \text{otherwise} \end{cases}$$

    Clearly f is the identity on all even numbers. However,

$$\begin{array}{l}
f(1) \neq f_0(1) \\
f(3) \neq f_1(3) \\
f(5) \neq f_2(5) \\
\quad\vdots \\
f(2n+1) \neq f_n(2n+1) \\
\quad\vdots
\end{array}$$

    Since $f \neq f_n$ for all n, f is not in the listing $f_0, f_1, \ldots, f_i, \ldots$. This contradicts the assumption that the set is countably infinite.

    Since the set Turing computable functions is countable and the set of number-theoretic functions that are the identity on the even natural numbers is uncountable, there functions of the latter form that are uncomputable.

Chapter 10

The Chomsky Hierarchy

1. a) The unrestricted grammar

$$\begin{aligned}
&S \to X \mid Y \mid aPAbQb\\
&X \to aaX \mid \lambda\\
&Y \to bbY \mid \lambda\\
&P \to aPA \mid \lambda\\
&Q \to bQb \mid E\\
&Ab \to bA\\
&AE \to a\\
&Aa \to aa
\end{aligned}$$

generates the language $\{a^ib^ja^ib^j \mid i, j \geq 0\}$. Rules $S \to X$ and $S \to Y$ initiate derivations that generate λ, a^{2i}, and b^{2i}. The derivation of a string $a^ib^ja^ib^j$, with $i, j > 0$, begins with an application of $S \to aPAbQb$. Repeated applications of the P and Q rules produce $a^iA^ib^jEb^j$. The rules $Ab \to bA$, $AE \to a$, and $Aa \to aa$ are then used to obtain the desired configuration of the terminals. The derivation of $aabaab$ demonstrates the repositioning of the variable A and its transformation to the terminal a.

$$\begin{aligned}
S &\Rightarrow aPAbQb\\
&\Rightarrow aaPAAbQb\\
&\Rightarrow aaAAbQb\\
&\Rightarrow aaAAbEb\\
&\Rightarrow aaAbAEb\\
&\Rightarrow aabAAEb\\
&\Rightarrow aabAab\\
&\Rightarrow aabaab
\end{aligned}$$

3. Let G = (V, Σ, P, S) be an unrestricted grammar. We construct a grammar G$'$ = (V$'$, Σ, P$'$, S) that generates L(G) in which each rule has the form $u \to v$ where $u \in (\mathrm{V}')^+$ and $v \in (\mathrm{V}' \cup \Sigma)^*$. The set of variables of V$'$ consists of the variables of V and a new variable T_a for every $a \in \Sigma$. For a string $u \in (\mathrm{V} \cup \Sigma)^*$, let $T(u)$ denote the string obtained by replacing each terminal a in u with the corresponding variable T_a. The rules of G$'$ are

i) $T(u) \to T(v)$ for every rule $u \to v \ \in P$

ii) $T_a \to a$ for every $a \in \Sigma$.

The grammar G contains a derivation $S \overset{*}{\Rightarrow} w$ if, and only if, $S \overset{*}{\Rightarrow} T(w)$ is a derivation of G′. Using the rules defined in (ii) to transform each variable T_a to the corresponding terminal, we see that L(G) = L(G′).

4. c) Let L_1 and L_2 be two recursively enumerable languages. We will construct an unrestricted grammar G that generates L_1L_2 from grammars $G_1 = (V_1, \Sigma_1, P_1, S_1)$ and $G_2 = (V_2, \Sigma_2, P_2, S_2)$ that generate L_1 and L_2 respectively.

   Let us first recall the proof of Theorem 7.5.1, which established that the context-free languages are closed under concatenation. The grammar

   $$G = (V_1 \cup V_2, \Sigma_2 \cup \Sigma_2, P_1 \cup P_2 \cup \{S \to S_1S_2\}, S)$$

   was constructed from G_1 and G_2 where $S \notin V_1 \cup V_2$ and V_1 and V_2 are assumed to be disjoint. Unfortunately, we must be careful in combining unrestricted grammars in this manner since the left-hand side of a rule may contain more than one symbol. For example, consider derivations $S_1 \overset{*}{\Rightarrow} x$ in G_1 and $S_2 \overset{*}{\Rightarrow} y$ in G_2. It follows that $S \overset{*}{\Rightarrow} xy$. Now either P_1 or P_2 may contain a rule $u \to v$ where $xy = x'uy'$. In this case, the transformation obtained by the application of the rule $u \to v$ may produce a string that is not in the concatenation of L_1 and L_2.

   The grammars G_1 and G_2 must be constructed to insure that no rule can overlap the strings derived from S_1 and S_2. Using Exercise 3, we can assume that the left-hand side of every rule in G_1 and G_2 consists solely of variables. As before, we also assume that V_1 and V_2 are disjoint. When G_1 and G_2 satisfy these conditions, the grammar G defined above generates L_1L_2 .

6. a)

$$\begin{aligned} S &\Rightarrow SBA \\ &\Rightarrow aBA \\ &\Rightarrow aAB \\ &\Rightarrow aaBB \\ &\Rightarrow aabB \\ &\Rightarrow aabb \end{aligned}$$

   b) The language of G is $\{a^{i+1}b^{2i} \mid i \geq 0\}$.

   c) L(G) is generated by the rules $S \to aSbb \mid a$.

8. a) The language L satisfies the pumping lemma. Let z be any string in L, then $z = a^ib^jc^k$ with $0 < i \leq j \leq k$. Selecting $u = a^ib^{j-1}$, $v = b$, $w = \lambda$, $x = c$, and $y = c^{k-1}$ is a decomposition in which v and x are pumpable. Even though the pumping lemma is satiafied, L is nevertheless not context-free.

   b) The context-sensitive grammar

$$\begin{aligned} &S \to aAbc \\ &A \to aAbC \mid B \\ &B \to bBC \mid BC \mid bC \mid C \\ &Cc \to cc \end{aligned}$$

   generates the language $\{a^ib^jc^k \mid 0 < i \leq j \leq k\}$. The S rule puts a c on the end of the string. The recursive A rules generate a matching number of a's, b's, and C's. The B rules produce a matching C for each b and, possibly, additional C's. The C's are moved to the end of the string by the $Cb \to bC$ rule and then turned to c's by $Cc \to cc$.

10. Let $u \to v$ be a monotonic rule with $length(v) > 2$. We define a set P of monotonic rules that produce the same transformation as $u \to v$. The rules of P are constructed so that a string

on the right-hand side of a rule has length less than $length(v)$. Without loss of generality we assume that $u, v \in \mathrm{V}^*$. If not, every occurrence of a terminal a in u and v is replaced by a new variable T_a and the rule $T_a \to a$ is added to P as in Exercise 3. The construction of P is divided into two cases.

i) If $length(u) = 1$, then $u \to v$ can be written $A \to BCDv'$ where $A, B, C, D \in \mathrm{V}$ and $v' \in \mathrm{V}^*$. Using two new variables X and Y, the monotonic rules

$$\begin{aligned} &A \to XY \\ &X \to BC \\ &Y \to Dv' \end{aligned}$$

define the same transformation as $u \to v$.

ii) If $length(u) > 1$, then $u \to v$ can be written $AEu' \to BCDv'$. The rules

$$\begin{aligned} &AE \to XY \\ &X \to B \\ &Yu' \to CDv' \end{aligned}$$

generate the transformation $u \to v$.

In either case, the right-hand side of each rule produced using these transformations has length at most $length(v) - 1$. This procedure can be repeated until the length of the right-hand of each rule is at most two.

11. The string transformation obtained by applying the rule $AB \to CD$ can be obtained by the sequential application of the three rules

$$\begin{aligned} &AB \to XB \\ &XB \to XD \\ &XD \to CD \end{aligned}$$

Each of these rules has the context-sensitive form $uAv \to uwv$ with $w \neq \lambda$.

13. The answer depends on the depth of the tree T. If the depth is 0, the language consists only of the null string, which is regular.

If the tree has depth 1, the language is $(\boldsymbol{LU} \cup \boldsymbol{RU})^*$. This follows since the most steps down the tree that can be taken from the root is one, either right or left, followed by a move up the tree. This process can be repeated any number of times to produce a string in L.

In general, a string is accepted if it satisfies the following two conditions:

a) the number of steps that move down tree equals the number of steps that move up, snd

b) the steps do not specify a move down from a leaf or up from the root.

For a tree of depth n, a DFA with $2n + 2$ states can be constructed to accept the language. State q_0 is the accepting state and represents a string with the same number of U's as L's and R's. States q_i, $i = 1, \ldots, n$, are entered when a string has i more U's than L's and R's. States q_i, $i = n + 1, \ldots, 2n$, are entered when a string has i more L's and R's than U's. If the string represents a path the would leave the tree, the DFA enters an error state q_e and rejects the string.

If the full binary tree has infinite depth, the strings λ, LU, $LLUU$, $\ldots$, L^iU^i, $\ldots$, are in the language. This sequence can be used with Corollary 6.5.2 to establish that the language is not regular.

To show that the language is context-free, a pushdown automaton can be constructed to accept it. Whenever an L or R is read, a A is pushed onto the stack. When a U is read, if there is an A on the stack it is popped. Otherwise, the computation rejects the string. A string is accepted if the computation processes the entire string and halts with an empty stack.

15. Let L be a recursively enumerable language accepted by the standard Turing machine M = (Q, Σ, Γ, δ, q_0, F) and let c be a terminal symbol not in Σ. The language

$$\text{L}' = \{\ wc^i \mid \text{M accepts } w \text{ by a computation in which the tape head does not read tape position } length(w) + i + 1\}$$

is accepted by a linear-bounded machine M′ constructed from M. A computation of M′ with initial tape configuration $\langle u \rangle$ begins by checking if the u has the form wc^{j+1} for some string $w \in \Sigma^*$ and $j \geq 0$. If so, then the final c is erased and the input shifted one position to the right producing $\langle Bwc^j \rangle$. The tape head is then moved to position 1 to read the blank preceding the input string. For the remainder of the computation, the transitions of M′ are identical to those of M except M′ treats the c as a blank. A transition of $\delta'[q_i, c]$ of M′ performs the same actions as $\delta[q_i, B]$ of M.

Let k be the furthest tape position scanned during the computation of M that accepts a string w. Choosing i so that $length(w) + i + 1 > k$, the computation of M′ will remain within the segment of tape enclosed within the endmarkers. Consequently M′ accepts wc^i.

Moreover, the only way that M′ accepts a string wc^i is to mimic a computation of M that accepts w that remains within the tape the first $length(w) + i$ tape positions.

Chapter 11

Decision Problems and the Church-Turing Thesis

4. A three-tape Turing machine can be designed to solve the "2^n" problem. In this solution we represent the natural number n by $\mathit{1}^n$. The numbers 2^0, 2^1, 2^2, . . . are sequentially produced on tape 2 and compared with the input string on tape 1. Tape 3 is used in the generation of the sequence on tape 2. The computation of the machine with input u consists of the following actions.

    1. A *1* is written on tapes 2 and 3.
    2. Reading left to right, the length of the strings on tapes 1 and 2 is compared.
    3. If blanks are simultaneously read on tapes 1 and 2, the computation halts accepting the input.
    4. If a blank is read on tape 1 and a *1* on tape 2, the computation halts rejecting the input.
    5. If a blank is read on tape 2 and a *1* on tape 1, then

        - the string on tape 3 is concatenated to the string on to tape 2,

        - tape 3 is erased and the string from tape 2 is copied to tape 3,

        - all tape heads are repositioned at the initial position, and

        - the computation continues with step 2.

    At the beginning of step 2, tapes 2 and 3 both contain the representation of 2^i. When the length of the input string u is compared with the string on tape 2, there are three possible outcomes: $length(u) = 2^i$ and the input is accepted in step 3, $length(u) < 2^i$ and the input is rejected in step 4, or $length(u) > 2^i$. In the latter case, the string 2^{i+1} is generated in step 5 and the comparison is repeated.

7. a) A regular grammar G = (V, Σ, P, S) has rules of the form $A \to aB$, $A \to a$, or $A \to \lambda$ where $A, B \in$ V and $a \in \Sigma$. To construct the encoding of the rules of the grammar over the set $\{\mathit{0}, \mathit{1}\}$, we assign a number to each of the variables and terminals. If $|V| = n$ then we number the variables from 1 to n with S assigned 1. The terminals are numbered from $n + 1$ to $n + m$ where $m = |\Sigma|$. The symbol assigned number n is encoded by the string $\mathit{1}^n$. The encoding of a variable A and a terminal a will be denoted $en(A)$ and $en(a)$. The encoding of a string w over $(\text{V} \cup \Sigma)^*$, denoted $en(w)$, consists of the encoding of each of the symbols in w separated by *0*'s.

Rules of the form $A \to \lambda$, $A \to a$, and $A \to aB$ are encoded by the strings $en(A)$, $en(A)\mathit{0}en(a)$, and $en(A)\mathit{0}en(a)\mathit{0}en(B)$, respectively. Rules are separated by *00*. The representation of a regular grammar G, denoted $R(\mathrm{G})$, is the concatenation of the encoding of the rules.

Consider the regular grammar G

$$\begin{aligned} S &\to aS \mid aA \\ A &\to bA \mid \lambda \end{aligned}$$

that generates the language $\boldsymbol{a}^+\boldsymbol{b}^*$. The grammar G is represented by the string

$$en(S)\mathit{0}en(a)\mathit{0}en(S)\mathit{00}en(S)\mathit{0}en(a)\mathit{0}en(A)\mathit{00}en(A)\mathit{0}en(b)\mathit{0}en(A)\mathit{00}en(A).$$

b) A nondeterministic two-tape Turing machine M is designed to solve the question of derivability in a regular grammar. A computation of M that decides whether a string w is in L(G) begins with input consisting of the encoding of the rules of the grammar followed by *000* and the encoding of w.

The actions of a machine M that solves the derivability problem are described below. M uses the representation of the rules of G to simulate a derivation of G on tape 2.

1. The computation of M begins by determining whether the input has the form $R(\mathrm{G})\mathit{000}en(w)$. If not, M halts rejecting the input.
2. The encoding of S is written in position one of tape 2.
3. Let uA be the string that is encoded on tape 2. A rule $A \to v$ is chosen nondeterministically from the encodings on tape 1. The encoding of the variable A is then replaced with the encoding of v on tape 2.
4. If v is a terminal symbol or the null string, the string on tape 2 is compared with the encoding of w on tape 1. If they are identical, M halts and accepts the input.
5. If $v = aB$ for some $a \in \Sigma$ and $B \in \mathrm{V}$, then the lengths of ua and w are compared. The input is rejected if $length(ua) > length(w)$. If not, steps 3–5 are repeated.

Step 5 insures that a computation of M halts after the simulation of at most $length(w) + 1$ rule applications. Consequently, all computations of M terminate and M solves the derivability problem.

7. c) A strategy for a machine to reduce membership in the language $\{x^iy^{i+1} \mid i \geq 0\}$ to $\{a^ib^i \mid i \geq 0\}$ is described below. The appropriate action is determined by the form of input string.

i) If there is an x that occurs after a y, the input is erased and a is written in the input position.

ii) If the input is λ, an a is written in the input position.

iii) If the input string has the form $\boldsymbol{x}^+$, the x's are changed to a's.

iv) If the string has the form $\boldsymbol{x}^*\boldsymbol{y}^+$, a y is erased from the end of the input and the remaining x's and y's are changed to a's and b's.

11. The universal machine U is modified to construct a machine U_{1} that accepts strings over $\{\mathit{0}, \mathit{1}\}$ of the form R(M)w where the computation of M with input w prints a *1*. U_{1} accepts by final state and has a single accepting state.

A computation of U_{1} initially determines whether the input has the form R(M)w. If so, the computation continues with the simulation of the transitions of M in the manner of the universal machine. Before the simulation of a transition of M by U_{1}, step 4 (b) of the description of the universal machine in Theorem 11.4.1, the symbol y to be printed is examined. If y is *1*, then the U_{1} enters the accepting state and halts. Otherwise, the simulation of the computation of M continues.

14. At first glance, this problem might seem to require the computation of the machine M to be tested on every possible input string—but this is not case. The first 10 transitions of a computation can read at most the first 10 symbols in the input string. Thus all that is necessary is to check the computation of M on all strings of length 10 or less. If a computation on this finite set of strings requires more than 10 transitions, the answer is yes. If no computation with any string from this set requires more than 10 transitions, the answer is no.

    A modification of the universal machine can be used to perform this check. Two tapes are added to the universal; one used in the generation of the strings from Σ^* of length 10 or less and the other to count the number of transitions in the simulation of a computation of M. Tape 4 will hold the input and tape 5 will serve as a counter. The input to this machine is the representation of M.

    The computation consists of the following cycle:

    1. The null string is written on tape 4.
    2. The string 1^n is written on tape 5.
    3. The string on tape 4 is copied to tape 3 and the encoding of state q_o is written on tape 2.
    4. The computation of M with the input string from tape 4 is simulated on tapes 2 and 3. With each simulated transition, a *1* is erased from tape 5.
    5. If the simulation of M specifies a transition and tape 5 is scanning a blank, the computation halts and accepts.
    6. If the simulation of M halts and a *1* is read on tape 5 then

        - tapes 2, 3 and 5 are erased,

        - the string 1^n is written on tape 5, and

        - the next string is generated on tape 4.
    7. If the string on tape 4 has length greater than 10, the computation halts and rejects the input. Otherwise, the computation continues with step 3.

Chapter 12

Undecidability

3. Let M be a Turing machine that accepts a nonrecursive language L. Without loss of generality, we may assume that the computation of M continues indefinitely whenever the input is not in L.

   Assume that there is a machine H that solves the Halting Problem for M. The halting machine H accepts a string w if M halts with input w. That is, if $w \in$ L. In addition, the computation of H halts in a nonaccepting state for all strings in $\overline{\text{L}}$. Consequently, L(H) = L. Since H halts for all input strings, it follows that L is recursive. This is a contradiction since our original assumption was that L is nonrecursive. Thus we conclude that there is no halting machine H.

6. We will show that the Halting Problem is reducible to the question of whether a Turing machine enters a designated state. The reduction has the form

Reduction	**Input**	**Condition**
Halting Problem	M, w	M halts with w
to	↓	↓
State Problem	M′, w, q_f	M′ enters state q_f when run with w

   The string w is the same for both problems. The machine M′ is constructed from M by adding an additional state q_f. The idea behind the reduction that M′ enters q_f whenever M halts.

   Let M = (Q, Σ, Γ, δ, q_0, F) be a Turing machine. The machine M′ = (Q ∪ $\{q_f\}$, Σ, Γ, δ', q_0, $\{q_f\}$) with transition function

   i) $\delta'(q_i, x) = \delta(q_i, x)$ whenever $\delta(q_i, x)$ is defined
   ii) $\delta'(q_i, x) = [q_f, x, R]$ whenever $\delta(q_i, x)$ is undefined

   enters q_f for precisely the strings for which M halts. The computations of M′ are identical to those of M until M halts. When this occurs, M′ enters q_f. Thus M halting is equivalent to M′ entering q_f.

   Assume that there is a machine S that solves the state problem. The input to S is a representation of a Turing machine M, input string w, and state q_i. S accepts the input if the computation of M with input w enters state q_i. Adding a preprocessor N to S, we construct a machine H that solves the halting problem.

   The input to N is $R(\text{M})w$, the representation of a Turing machine M followed by an input string. By adding the encoding of transitions described above to R(M), N constructs the representation of the machine M′. N then adds q_f to the end of the string producing $R(\text{M}')wq_f$.

The machine H combines N and S. A computation of H

1. runs N on input $R(\mathrm{M})w$ producing $R(\mathrm{M}')wq_f$
2. runs S on $R(\mathrm{M}')wq_f$.

H accepts input $R(\mathrm{M})w$ if, and only if, M′ enters state q_f when run with input w. That is, whenever M halts with input w. Step 1 is a reduction of the Halting Problem to the State Problem. We conclude that the machine S cannot exist and that the State Problem is undecidable.

12. a) The language L consisting solely of the string w is recursive, since every finite language is recursive. The complement of L is also recursive and clearly does not contain w. Consequently the property of containing the string w is a nontrivial property of recursively enumerable languages. Thus, by Rice's Theorem, the question of w being in L(M) is undecidable.

14. a) Rice's Theorem can be used to show that $\mathrm{L}_{\neq\emptyset}$ is not recursive. The languages $\{a\}^*$ and $\emptyset$ are both recursive, with the former language satisfying the property 'is not the empty set' the latter not satisfying the property. Thus the property 'is not the empty set' is nontrivial and membership in $\mathrm{L}_{\neq\emptyset}$ is undecidable.

b) To accept a string R(M) we need to determine if M will accept at least one input string. Intuitively, the strategy might appear to use a universal machine to simulate computations of M until one string in $\Sigma*$ is found that is accepted by M or all strings have been checked and found not to be accepted by M. Unfortunately, if the simulation encounters one string for which the computation of M does not halt, this strategy will not be able to check subsequent strings in Σ^*. This is the same difficulty encountered in showing that a recursively enumerable language can be enumerated (Theorem 8.8.6). The solution to our difficulty is the same as used in that proof.

A modification of the universal machine can be used to show that $\mathrm{L}_{\neq\emptyset}$ is recursively enumerable. We add three tapes and use two enumerating machines that have previously been described. Let u_0, u_1, u_2, ...be a lexicographic ordering of the strings in Σ^*. Ordered pairs $[i, j]$ are generated on tape 4 using the enumerating the machine introduced in Exercise 8.33. When a pair $[i, j]$ is written the tape, the string u_i is generated on tape 5 and i^j is written on tape 6.

At this point, the computation of M with input u_i is simulated on tapes 2 and 3 for j transitions. The string on tape 6 is used as a counter to halt the simulation when j transitions have been executed. If u_i is accepted by M in j transitions, L(M) is accepted. Otherwise, the work tapes are reinitialized, the next ordered pair is generated, and the simulation continues.

If M accepts some string u_n, it does so in some finite number m of transitions. When $[n, m]$ is generated, u_n will be checked and L(M) accepted. If M does not accept any string, the loop of generating and testing will continue indefinitely.

16. a) A solution to the Post correspondence system $\{[a, aa], [bb, b], [a, bb]\}$ is obtained by playing the dominoes

a	a	bb	bb
aa	bb	b	b

17. a) We will show that no sequence of dominoes can be constructed that solves this Post correspondence problem. A solution to the Post correspondence system

$$\{[ab, a], [ba, bab], [b, aa], [ba, ab]\}$$

must begin with $[ab, a]$ or $[ba, bab]$ since these are the only pairs that have the same leftmost symbol. A domino that extends the sequence beginning with $[ab, a]$ must have a b as the first symbol in the bottom string. The only such ordered pair is $[ba, bab]$. Playing this produces

ab	ba
a	bab

in which the top and bottom strings differ in the third symbol.

There are three ways to extend a sequence that begins with $[ba, bab]$.

i)

ba	ba
bab	bab

ii)

ba	b
bab	aa

iii)

ba	ab
bab	ba

Playing $[ba, bab]$ produces strings that disagree at the fourth position. We now turn our attention to trying to extend sequence (ii). The next domino must have an a in the first position in the top string. Playing the only domino of this form we obtain the sequence

ba	b	ab
bab	aa	a

in which the strings disagree in the fifth position.

The sole remaining candidate is sequence (iii). Note that the rightmost end of these strings has the same form as the sequence consisting solely of the domino $[ba, bab]$. That is, the length of the bottom string is one greater than that of the top string with a b being the rightmost symbol on the bottom. The preceding argument can be repeated to show that

ba	ba	ba
bab	ab	ab

is the sole extension of (iii) that satisfies the conditions of a solution to this Post correspondence system.

The only way in which this sequence can be extended is to continually play $[ba, ab]$. Utilizing this strategy, the length of the top string is always one less than that of the bottom. It follows that there is no sequence of dominoes that form a solution to the Post correspondence system.

18. We can determine whether a Post correspondence system for an alphabet $\Sigma = \{a\}$ has a solution by examining the relationship of number of symbols in the top of each domino with the number in the bottom. We will consider four cases.

    Case 1: the number of a's in the top is greater than the number of a's in the bottom of each domino. In this case, the Post correspondence system has no solution since every way to play the dominoes produces a longer top string than bottom string.

    Case 2: the number of a's in the top is less than the number of a's in the bottom of each domino. This case also is analogous to case 1 and there is no solution.

Case 3: there is a domino with the same number of a's in the top and bottom. This case has a solution consisting of that single domino.

Case 4: none of the previous cases hold. This means that the Post correspondence system contains two dominoes $[a^i, a^j]$ and $[a^m, a^n]$ such that $i > j$ and $m < n$. Playing $n - m$ dominoes of the form $[a^i, a^j]$ followed by $i - j$ dominoes of the form $[a^m, a^n]$produces a top string with

$$(n-m)i + (i-j)m = ni - mi + mi - jm = ni - mj$$

a's and a bottom string with

$$(n-m)j + (i-j)m = nj - mj + in - jn = in - mj$$

a's. Thus any Post correspondence system satisfying this condition has a solution.

Examining the preceding four cases, we can determine whether any Post correspondence system over a one symbol alphabet has a solution. Thus this special case of the Post Correspondence Problem is decidable.

21. Let $\mathrm{G_U}$ be a grammar whose rules have the form

$$S \to u_iSi \mid u_ii, \qquad i = 1, \dots, n$$

generated from a Post correspondence system. We will show that $\overline{\mathrm{L(G_U)}}$ is context-free.

The language $\mathrm{L(G_U)}$ consists of strings in which each u_i is matched to the integer i in reverse order. We can build a context-free grammar $\mathrm{G'}$ that generates $\overline{\mathrm{L(G_U)}}$. The derivations of $\mathrm{G'}$ produce all strings in which the matching does not occur. To describe the construction we will consider the substrings u_i and integers i to be units. There are two classes of strings that do not satisfy the matching criteria;

a) strings with an odd number of units, and
b) strings with m units in which there is some position k such that u_i is at position k and i is not at position $m - k$.

The variables O and E generate the strings in $\overline{\mathrm{L(G_U)}}$ with an odd number of units and units with a mismatch, respectively.

$$\begin{array}{ll}
S \to O \mid E & \\
O \to u_ijO \mid ju_iO \mid ijO \mid u_iu_jO \mid u_i \mid j & \text{where } i,j = 1,\dots,n \\
E \to u_iEi \mid A & \text{where } i = 1,\dots,n \\
A \to u_jBk \mid iBj \mid u_iBu_j \mid iBu_j & \text{where } i,j,k = 1,\dots,n \text{ and } j \neq k \\
B \to u_iB \mid iB \mid \lambda & \text{where } i = 1,\dots,n
\end{array}$$

The variable E produces strings with matching units. The application of an A rule inserts a mismatch into the string. This is followed by a derivation from B that generates any combination of units.

22. First we note that a Post correspondence problem C has infinitely many solutions whenever it is solvable. If w is the string spelled by a solution, then ww, www, ... are also spelled by solutions. These solutions are obtained by repeating the sequence of dominoes that spell w.

Let C be a Post correspondence system and let $\mathrm{G_U}$ and $\mathrm{G_V}$ be the upper and lower grammars defined by C. By Theorem 12.7.1, C has a solution if, and only if, $\mathrm{L(G_U)} \cap \mathrm{L(G_V)} \neq \emptyset$. Combining this with the previous observation, C having a solution is equivalent to $\mathrm{L(G_U)}$ and $\mathrm{L(G_V)}$ having infinitely many elements in common. It follows that there is no algorithm that can determine whether the intersection of the languages generated by two context-free grammars is an infinite set.

24. We will use a proof by contradiction to show that the question of determining whether the languages of two arbitrary context-free grammars G_1 and G_2 satisfy $\mathrm{L}(\mathrm{G}_1) \subseteq \mathrm{L}(\mathrm{G}_2)$. Assume such an algorithm exists and let G_2 be an arbitrary context-free grammar with alphabet Σ. We know that there is a context-free grammar G_1 that generates Σ^*. In this case, an algorithm that determines whether $\mathrm{L}(\mathrm{G}_1) \subseteq \mathrm{L}(\mathrm{G}_2)$ also determines whether $\mathrm{L}(\mathrm{G}_2) = \Sigma^*$.

    However, by Theorem 12.7.3, we know that there is no algorithm that can determine whether the language of an arbitrary context-free grammar G with input alphabet Σ is Σ^*. Thus our assumption must be false and there is no algorithm that determines whether $\mathrm{L}(\mathrm{G}_1) \subseteq \mathrm{L}(\mathrm{G}_2)$.

Chapter 13

μ-Recursive Functions

3. b) The proper subtraction function $sub(x, y)$ was defined in Table 13.2.1 by

$$\begin{aligned}&sub(x, 0) = x\\&sub(x, y + 1) = pred(sub(x, y))\end{aligned}$$

where $pred$ is the predecessor function. The functions $g = p_1^{(1)}$ and $h = pred \circ p_3^{(3)}$ formally define sub. Using g and h in the definition of primitive recursion produces

$$\begin{aligned}&sub(x, 0) = g(x) = x\\&sub(x, y + 1) = h(x, y, sub(x, y)) = pred(sub(x, y))\end{aligned}$$

as desired.

5. a)

$$\begin{aligned}f(3, 0) &= g(3)\\&= 3\end{aligned}$$

$$\begin{aligned}f(3, 1) &= h(3, 0, f(3, 0))\\&= h(3, 0, 3)\\&= 3 + 3\\&= 6\end{aligned}$$

$$\begin{aligned}f(3, 2) &= h(3, 1, f(3, 1))\\&= h(3, 0, 6)\\&= 9\end{aligned}$$

b) A closed form for $f(3, n)$ is $3(n + 1)$ and $f(m, n) = m(n + 1)$.

8. d) The values of the predicate

$$even(y) = \begin{cases} 1 & \text{if } y \text{ is even} \\ 0 & \text{otherwise} \end{cases}$$

alternate between 0 and 1. The value of $even(y+1)$ can be obtained directly from $even(y)$ using proper subtraction.

$$\begin{aligned} &even(0) = 1 \\ &even(y+1) = 1 \dot{-} even(y) \end{aligned}$$

The right-hand side of the definition by primitive recursion given above uses only constant functions, proper subtraction, and the previous value of *even*. Consequently, *even* is also primitive recursive.

e) To define $half(y)$ using primitive recursion, we must determine the relationship between $half(y)$ and $half(y+1)$.

$$half(y+1) = \begin{cases} half(y) & \text{if } y+1 \text{ is odd} \\ half(y)+1 & \text{if } y+1 \text{ is even} \end{cases}$$

Translating this into the language of primitive recursion, we obtain the primitive recursive definition

$$\begin{aligned} &half(0) = 0 \\ &half(y+1) = half(y) + even(y+1). \end{aligned}$$

f) If $y+1$ is not a perfect square, the $sqrt(y+1) = sqrt(y)$. When $y+1$ is a perfect square, $sqrt(y)$ is incremented.

$$\begin{aligned} sqrt(0) &= 0 \\ sqrt(y+1) &= sqrt(y) \\ &\qquad + eq(mult(sqrt(y+1), sqrt(y+1)), y+1). \end{aligned}$$

9. c) The function *btw*, between, can be obtained from the conjunction of two instances of *lt*:

$$btw(x,y,z) = lt(y,x) \cdot lt(x,z).$$

d) The integer square root function defined in Exercise 7 f) can be used to show that the relation *prsq* is primitive recursive. A natural number x is a perfect square if it is the product of the square root of x with itself. The equality predicate is used in the definition of *prsq* to check this condition,

$$prsq(x) = eq(x, sqrt(x) \cdot sqrt(x)).$$

11. a) The two-variable predicate

$$f(x,y) = \begin{cases} 1 & \text{if } g(i) < g(x) \text{ for all } i \le y \\ 0 & \text{otherwise} \end{cases}$$

is true only when the inequality $g(i) < g(x)$ is satisfied by every natural number i between 0 and y. The bounded product

$$f(x,y) = \prod_{i=0}^{y} lt(g(i), g(x))$$

computes f using the primitive recursive function g. Since the family of primitive recursive functions is closed under bounded products, f is primitive recursive.

e) The bounded sum can be used to count the number of times $g(i) = x$ in the range $0 \le i \le y$. The predicate $eq(g(i), x)$ is 1 only if $g(i) = x$. The summation

$$n(x,y) = \sum_{i=0}^{y} eq(g(i), x)$$

records the number times this occurs in the designated range.

f) The function $n(x, y)$, defined above, provides the counter needed for the construction of a function that finds the third occurrence of x in the sequence $g(0), g(1), \ldots, g(y)$. The first i that satisfies the predicate $eq(n(x, i), 3)$ is the third integer whose value $g(i)$ is x. Bounded minimalization is the primitive recursive tool for finding the first value that satisfies a predicate:

$$thrd(x, y) = ge(n(x, y), 3) \cdot \overset{y}{\mu} z\ [eq(n(x, z), 3)].$$

The factor $ge(n(x, y), 3)$ in the definition of $thrd$ provides the default value whenever there are fewer than three integers in the range 0 to y for which g assumes the value x.

12. a) First we note that, by definition, no number is a divisor of 0. Thus, the greatest common divisor of 0 and any number is 0. Finding the greatest common divisor of x and y can be thought of as a search procedure. The objective is to find the greatest number less than or equal to x that divides both x and y. The predicate $divides(x, x \dot{-} z) \cdot divides(y, x \dot{-} z)$ is 1 whenever $x \dot{-} z$ is a divisor of both x and y. The function

$$g(x, y) = \overset{x}{\mu} z\ [divides(x, x \dot{-} z) \cdot divides(y, x \dot{-} z)]$$

sequentially examines the predicates

$$\begin{array}{c} divides(x, x) \cdot divides(y, x) \\ divides(x, x \dot{-} 1) \cdot divides(y, x \dot{-} 1) \\ divides(x, x \dot{-} 2) \cdot divides(y, x \dot{-} 2) \\ \vdots \end{array}$$

The greatest common divisor can be obtained directly from g.

$$gcd(x, y) = sg(x) \cdot sg(y) \cdot ((x \dot{-} g(x, y))$$

The sg predicates handle the special case when one or both of the arguments are 0.

13. The function f is obtained using primitive recursion and the primitive recursive functions g and min as follows:

$$\begin{aligned} &f(0) = g(0) \\ &f(y + 1) = min(f(y), g(y + 1)) \end{aligned}$$

16. a) The prime decomposition of 18000 is $2^4 \cdot 3^2 \cdot 5^3$. This is a Gödel number since it consists of an initial sequence of the primes. The encoded sequence is 3, 1, 2.

17. a) To construct a primitive recursive one-variable predicate gdn whose value indicates whether the input is the encoding of a Gödel number, we begin by defining an auxiliary function

$$g(x) = \overset{x}{\mu} z\ [cosg(divides(x, pn(z)))]$$

that returns the number of the first prime that does not occur in the prime decomposition of x. If there are no primes greater than $pn(g(x))$ that divide x, then x is a Gödel number Deciding whether there is prime greater than $pn(g(x))$ that occurs in the prime decomposition of x seems like it may require an unbounded search. However, we can determine if there are primes greater than $pn(g(x))$ that divide x by constructing the product of the powers of the primes less than $pn(g(x))$.

$$gdn(x) = eq(x, \prod_{i=0}^{g(x)-1} pn(i)^{dec(i,x)+1})$$

If the product equals x, then all the primes that divide x occur before $pn(g(x))$. In this case x is the encoding of a Gödel number. The equality predicate returns zero when there are primes greater than $pn(g(x))$ that divide x.

18. If z is the encoding of an ordered pair $[x, y]$, the primitive recursive function $gn_1(dec(1, z), dec(0, z))$ produces the encoding of $[y, x]$.

20. Functions f_1 and f_2 defined by simultaneous recursion are intuitively computable, the definition provides a description for calculating the values $f_1(x, y)$ and $f_2(x, y)$. However, the definition does not follow the form of a definition by primitive recursion since the computation of $f_1(x, y+1)$ requires both $f_1(x, y)$ and $f_2(x, y)$.

    The encoding and decoding functions can be used to define a two-variable primitive recursive function f that encodes both f_1 and f_2. The initial value of f is obtained directly from those of f_1 and f_2:

$$f(x, 0) = gn_1(f_1(x, 0), f_2(x, 0)) = gn_1(g_1(x), g_2(x)).$$

    The functions h_1 and h_2 and $f(x, y)$ are used to compute the values of the ordered pair $[f_1(x, y+1), f_2(x, y+1)]$. The value of $f(x, y+1)$ is obtained by encoding this ordered pair.

$$\begin{aligned} f(x, y+1) = gn_1(&h_1(x, y, dec(0, f(x, y)), dec(1, f(x, y))), \\ &h_2(x, y, dec(0, f(x, y)), dec(1, f(x, y)))) \end{aligned}$$

    Since each of the functions used in the construction of f is primitive recursive, f is primitive recursive.

    The functions f_1 and f_2 are obtained directly from f by

$$f_1(x, y) = dec(0, f(x, y))$$

    and

$$f_2(x, y) = dec(1, f(x, y)).$$

23. b) The value of the function $root(c_2, c_1, c_0)$ is the first nonnegative integral root of the quadratic equation $c_2 \cdot x^2 + c_1 \cdot x + c_0$. If there is no such root, $root(c_2, c_1, c_0)$ is undefined. The function

$$\mu z[eq(c_2 \cdot z^2 + c_1 \cdot z + c_0, 0)]$$

    obtained using the unbounded μ-operator computes $root$.

24. Since the new μ-operator extends the original operator to partial predicates, every function generated using the standard operator is also definable using the extended μ-operator. Thus every μ-recursive and, consequently, Turing computable function can be obtained in this manner.

    Using an argument similar to that in Theorem 13.6.4, we can show that a function $f(x_1, \ldots, x_n)$ definable using the extended μ-operator is Turing computable. This reduces to showing that a function obtained from a Turing computable partial predicate $p(x_1, \ldots, x_n, z)$ using the extended μ-operator is Turing computable. The proof uses the strategy employed in Theorem 13.6.4 for the standard μ-operator. If the evaluation of $p(x_1, \ldots, x_n, j)$ is undefined in step 3, then the computation never halts indicating that $f(x_1, \ldots, x_n)$ is undefined. Thus the proper value of $\mu z[p(x_1, \ldots, x_n, z)]$ is produced by the Turing machine.

    The preceding arguments have shown that family of functions obtained utilizing both the μ-operator and the extended μ-operator are precisely the Turing computable functions.

28. a) The value of $tr_{\mathrm{M}}(x, y-1)$ is the configuration of the computation of M with input x prior to the yth transition. The value of the new symbol function $ns(tr_{\mathrm{M}}(x, y-1))$ is the encoding of the symbol to be written on the yth transition. The function prt is defined by

$$prt(x, y) = eq(ns(tr_{\mathrm{M}}(x, y-1)), 1),$$

which checks if the symbol to be printed is a *1*.

b) Unbounded minimalization $\mu z[prt(x, z)]$ returns the first value z for which $prt(x, z)$ is true. Composing this with the sign function produces the desired result when the computation of M prints a *1*. The function

$$fprt(x) = sg \circ \mu z[prt(x, z)]$$

is undefined if M never prints a *1*.

c) If the function $fprt$ were primitive recursive, then we would always obtain an answer indicating the first occurrence of the printing of a *1* or a default answer (probably 0) if *1* is never printed.

This total function could be used to produce an algorithm for determining whether the computation of an arbitrary Turing machine M will print a *1* when run with input x. The answer to this question could be obtained as follows:

1. Build tr_{M}.
2. Build prt.
3. Build $fprt$.
4. Obtain the value $fprt(x)$.

By Exercise 12.7 we know that the problem of determining whether the computation of an arbitrary Turing machine will print a *1* is undecidable. Consequently, there is no algorithm capable of making this determination and $fprt$ must not be primitive recursive.

Chapter 14

Time Complexity

1. c) The function $f(n) = n^2 + 5n + 2 + 10/n$ is obtained by multiplying the polynomials in the numerator and dividing each term by n^2. Intuitively, n^2 is the most significant contributor to the growth of f. Formally, the rate of growth of f can be obtained directly from Definition 14.2.1. Clearly, n^2 is $\mathrm{O}(f)$. Conversely,

$$\begin{array}{rcl} n^2 + 5n + 2 + 10/n & \leq & n^2 + 5n^2 + 2n^2 + 10n^2 \\ & = & 18n^2 \end{array}$$

for all $n > 0$. Thus $f = \mathrm{O}(n^2)$. Combining the two relationships we see that f and n^2 have the same rates of growth.

3. e) To show that the factorial function grows faster than an exponential function, we prove that $n! > 2^n$ for all $n \geq 4$.

$$\begin{array}{rcl} n! & = & \prod_{i=0}^{n} i \\ & = & 2 \cdot 3 \cdot 4 \cdot \prod_{i=5}^{n} i \\ & > & 2^4 \cdot \prod_{i=5}^{n} i \\ & > & 2^4 \cdot \prod_{i=5}^{n} 2 \\ & > & 2^n \end{array}$$

Thus, setting c to 1 and n_0 to 4 we see that $2^n \leq c \cdot n!$ for all $n \geq n_0$. Thus, $2^n = \mathrm{O}(n!)$.

f) The inequality $n! > 2^n$ for $n \geq 4$, established in Exercise 3 (e), is used to prove that $n! \neq \mathrm{O}(2^n)$. Assume that $n! = \mathrm{O}(2^n)$. Then there are constants c and n_0 such that

$$n! \leq c \cdot 2^n,$$

whenever $n \geq n_0$. Let m be a number greater than the maximum of n_0, $2 \cdot c$, and 4. Then

$$\begin{array}{rcl} m! & = & m \cdot (m-1)! \\ & > & 2 \cdot c \cdot (m-1)! \\ & > & 2 \cdot c \cdot 2^{m-1} \\ & = & c \cdot 2^m \end{array}$$

with $m > n_0$.

Consequently, there are no constants that satisfy the requirements of Definition 14.2.1 and $n! \neq \mathrm{O}(2^n)$.

4. It is not the case that $3^n = \mathrm{O}(2^n)$. If so, then there are constants c and n_0 such that $3^n \leq c \cdot 2^n$ for all $n \geq n_0$. It follows that $(3/2)^n \leq c$ for all n greater than n_0. Taking the logarithm base 3/2 of each side of this inequality, we see that the assumption that $3^n = \mathrm{O}(2^n)$ implies that $n \leq \log_{3/2} c$ for all $n > n_0$. This is a contradiction since $\log_{3/2} c$ is a constant and we can select a natural number n larger than this constant. Since the assumption that $3^n = \mathrm{O}(2^n)$ produced a contradiction, we conclude that $3^n \neq \mathrm{O}(2^n)$.

7. b) We will show, directly from the definition of big Θ, that $fg \in \Theta(n^{r+t})$.

Since $f \in \Theta(n^r)$, there are positive constants c_1, c_2, and n_0 such that

$$0 \leq c_1 n^r \leq f(n) \leq c_2 n^r$$

for all $n \geq n_0$. Similarly, $g \in \Theta(n^t)$ implies there are positive constants d_1, d_2, and m_0 such that

$$0 \leq d_1 n^t \leq g(n) \leq d_2 n^t$$

for all $n \geq m_0$.

First we show that $fg \in \mathrm{O}(n^{r+t})$. The preceding inequalities show that

$$\begin{aligned} f(n)g(n) &\leq (c_2 n^r) \cdot (d_2 n^t) \\ &= (c_2 d_2) n^{r+t} \end{aligned}$$

for all $n \geq \max\{n_0, m_0\}$ as desired.

The same approach establishes the lower bound:

$$\begin{aligned} f(n)g(n) &\geq (c_1 n^r) \cdot (d_1 n^t) \\ &= (c_1 d_1) n^{r+t} \end{aligned}$$

for all $n \geq \max\{n_0, m_0\}$.

8. c) The length of the input of a Turing machine that computes a function of more than one variable is considered to be the length of the arguments plus the blanks that separate them. The Turing machine in Example 9.1.2 computes the binary function of string concatenation. The input has the form $BuBvB$ with length $length(u) + length(v) + 1$.

A computation moves through u and then shifts each element of v one position to the left. For an input of length n, the worst case number of transitions occurs when $u = \lambda$ and $length(v) = n - 1$. The values of the trace function tr_{M} are given for the first four values of n.

n	$tr_{\mathrm{M}}(n)$
1	4
2	8
3	12
4	16

A computation with input $BBvB$ begins by reading the two blanks that precede v. Translating each symbol of v requires three transitions. The computation is completed by returning the tape head to the leftmost tape position. This requires $length(v) + 2$ transitions. Thus $tr_{\mathrm{M}}(n) = 2 + 3(n - 1) + n + 1 = 4n$.

11. The computation of a standard machine that accepts the language $\mathrm{L} = \{a^i b^i \mid 0 \leq i \leq 50\} \cup \{u \mid \mathit{length}(u) > 100\}$ begins using 100 states to check the length of the input. If the machine has not encountered a blank after 100 transitions, the computation halts and accepts the input string.

    If the machine reads a blank prior to reading 100 input symbols, the computation

    1. moves the tape head to position 0, and
    2. runs a machine from that accepts $\{a^i b^i \mid i \geq 0\}$.

    The latter step can be accomplished in $(n+2)(n+1)/2$ transitions for an input string of length n.

    The maximum number of transitions in the entire computation for an input of length n is

$$tc_{\mathrm{M}}(n) = \begin{cases} 2n+2+(n+2)(n+1)/2, & \text{if } n \leq 100 \\ 101, & \text{otherwise.} \end{cases}$$

    The asymptotic complexity of M is constant, $tc_{\mathrm{M}}(n) = \Theta(1)$, since the number of transitions is constant as n approaches infinity.

13. A one-tape deterministic Turing machine can be designed to accept the language $\{a^i b^i \mid i \geq 0\}$ in time $\mathrm{O}(n \lg(n))$ by marking half of the a's and half of the b's on each pass through the input.

    The computation begins reading the input, marking every other a with an X, and recording whether the number of a's was even or odd. Upon scanning a b, it is marked as well as every other subsequent b. As with the a's, the states are used to record parity of the number of b's read. If an a is encountered after the first b, the computation halts and rejects the input. Upon completing the scan of the input, the following halting conditions are checked:

    (a) If no a's and b's were found, the string is accepted.

    (b) If there were a's but no b's, the string is rejected.

    (c) If there were b's but no a's, the string is rejected.

    (d) If the parity of the number of a's and b's was different, the string is rejected.

    If none of the halting conditions were triggered, tape head returns to tape position 0 to make another pass of the input. On this pass, the X's are skipped while marking every other a and b. On the completion of the pass, the termination conditions are checked. The process is repeated until one of the terminating conditions is satisfied.

    The maximum number of transitions for a string of length n occurs with the input $a^{n/2}b^{n/2}$ if n is even and $a^{(n+1)/2}b^{(n-1)/2}$ if n is odd. Each iteration of the preceding cycle requires $2n+2$ transitions and the maximum number of iterations is $\lceil \lg(n) \rceil$. Thus the maximum number of transitions is $(2n+2)\lceil \lg(n) \rceil$.

    The number of transitions could be reduced by one-half if we marked symbols going both left-to-right and right-to-left. However, this would not alter the big oh evaluation of the time complexity.

Chapter 15

$\mathcal{P}, \mathcal{NP}$, and Cook's Theorem

1. a) The computations for the string aa are

$$\begin{array}{rrrr}
q_0BaaB & q_0BaaB & q_0BaaB & q_0BaaB \\
\vdash Bq_1aaB & \vdash Bq_1aaB & \vdash Bq_1aaB & \vdash Bq_1aaB \\
\vdash Baq_1aB & \vdash Baq_1aB & \vdash Baq_1aB & \vdash q_2BaaB \\
\vdash Baaq_1B & \vdash Bq_2aaB & \vdash Bq_2aaB & \\
 & \vdash q_2BaaB & \vdash Baq_3aB & \\
 & & \vdash Baaq_3B &
\end{array}$$

   b) The worst-case performance with input a^n occurs when the computation reads the first $n-1$ a's in q_1 and transitions to q_2, stays in q_2 and moves left until it reads the first a, then moves right in state q_3 until the computation halts.

   c) The time complexity is

$$tc_{\mathrm{M}}(n) = \begin{cases} 1, & \text{if } n = 0; \\ 3n-1, & \text{otherwise.} \end{cases}$$

5. a) We will use a proof by contradiction to show that the language

$$\mathrm{L} = \{\mathrm{R(M)}w \mid \mathrm{M} \text{ accepts } w \text{ using at most } 2^{length(w)} \text{ transitions}\}$$

is not accepted by any deterministic polynomial-time Turing machine. The contradiction is obtained using diagonalization and self-reference.

Assume that L is in $\mathcal{P}$. Then $\overline{\mathrm{L}}$ is also in $\mathcal{P}$. Since $\overline{\mathrm{L}}$ is in $\mathcal{P}$, there is some Turing machine M′ that accepts it in polynomial time. The self-reference occurs in asking if R(M′)R(M′) is in $\overline{\mathrm{L}}$; that is, does M′ accept R(M′)?

Consider the first possibility, that R(M′)R(M′) is in $\overline{\mathrm{L}}$. This means that M′ does not accept R(M′) in $2^{length(\mathrm{R(M')})}$ steps. Thus either the computation takes more than $2^{\mathrm{R(M')}}$ or M′ does not accept R(M′). The first case indicates that the computations of M′ are not polynomially bounded and the second case indicates that R(M′)R(M′) is not in $\overline{\mathrm{L}}$. Both of these possibilities yield a contradiction.

The second possibility is that R(M′)R(M′) is not in $\overline{\mathrm{L}}$. This means that M′ accepts R(M′) in at most $2^{length(\mathrm{R(M')})}$ transitions. Consequently R(M′)R(M′) is in $\overline{\mathrm{L}}$, a contradiction.

Both possibilities, that R(M′)R(M′) is in $\overline{\mathrm{L}}$ and R(M′)R(M′) is not in $\overline{\mathrm{L}}$, produce a contradiction. Thus our assumption could not be true and the language L is not in $\mathcal{P}$.

9. Intuitively, the polynomial time bound $p(n)$ can be used to terminate all computations of M that use more than $p(n)$ transitions. To accomplish this, a machine M$'$ is constructed that has one more tape than M, which we will call the counter tape. The computation of M$'$

   1. Writes $p(n)$ *1*'s on the counter tape and repositions the tape head at the beginning of the string of *1*'s.
   2. While the counter tape head is reading a *1*, M$'$
      - Moves the counter tape head one position to the right.
      - If a transition of M is defined, then M$'$ simulates the transition on original tapes. Otherwise M$'$ halts and returns the same result as M.
   3. M$'$ halts and does not accept when a B is read on the counter tape.

   For every string $u \in$ L(M), there is at least computation that requires $p(n)$ or fewer transitions and accepts u. The simulation of this computation in M$'$ will also accept u. Clearly, L(M$'$) is contained in L(M) since it only accepts strings that are accepted by M. Thus M$'$ accepts L(M) and halts for all input strings.

   The computation of M$'$ for an input string u of length n consists of the writing $p(n)$ *1*'s on the counter tape and simulating the computation of M. The number of transitions in the simulation is bounded by $p(n)$ and writing $p(n)$ *1*'s can be accomplished in time polynomial in n.

12. In the first printing of the book, the state diagram is incorrect. The arcs on the bottom should go from right-to-left and there should be a transition $B/B\ L$ on the arc from q_6 to q_f. The answers given are for the state diagram with these modifications.

   a)

$$\begin{aligned}
& q_0BabbaB \\
&\vdash \#q_1abbaB \\
&\vdash \#aq_2bbaB \\
&\vdash \#abq_3baB \\
&\vdash \#abbq_4aB \\
&\vdash \#abbBq_4B \\
&\vdash \#abbq_5BB \\
&\vdash \#abq_5bBB \\
&\vdash \#aq_5bdBB \\
&\vdash \#q_5addBB \\
&\vdash q_5\#cddBB \\
&\vdash Bq_6cddBB \\
&\vdash q_fBcddBB
\end{aligned}$$

   b) Any string will cause the maximum number of transitions. The computation reads the entire string from left-to-right, then again from right-to-left, and ends with two transitions to erase the endmarker.

   c) $tc_{\mathrm{M}}(n) = 2n + 4$

   d) The function computed by the machine is not a reduction of L to Q. The string ab is mapped to cd; the former is not in L while the latter is in Q.

15. Any combination of three of the clauses x, y, z, and $\neg x \vee \neg y \vee \neg z$ is satisfiable. The four clauses, however, are unsatisfiable. The satisfaction of the clause $\neg x \vee \neg y \vee \neg z$ requires the truth assignment for one of the Boolean variables to be 0. This precludes the clause that consists of that variable alone from being satisfied.

17. The assumption the $\mathcal{P} = \mathcal{NP}$ is critical in the arguments that follow. Of course, we do not know if this assumption is true.

a) Let L be a language in $\mathcal{NP}$ that is neither empty nor is its complement empty. That means that are strings u and v such that $u \in$ L and $v \in \overline{\text{L}}$.

To prove that L is NP-complete, we must show that every language Q in $\mathcal{NP}$ is reducible to L in polynomial time. Since $\mathcal{NP} = \mathcal{P}$, we know that there is a Turing machine M that accepts Q in polynomial time. A polynomial time reduction of Q to L can be obtained as follows: For a string w

1. Run M to determine if $w \in$ Q.
2. If $w \in$ Q, then erase the tape and write u in the input position. Else erase the tape and write v in the input position.

This is clearly a reduction of Q to L. Moreover, it can be accomplished in polynomial time since the computation of Q is polynomial.

b) In part (a) we have shown that every language in $\mathcal{NP}$ other than $\emptyset$ and its complement is NP-complete. These languages clearly are in $\mathcal{P} = \mathcal{NP}$, but are not NP-complete. To be able to reduce a nonempty language Q to the $\emptyset$, we must map strings in Q to a string in $\emptyset$ and strings not in Q to a string not in $\emptyset$. Unfortunately, we cannot do the former since there are no strings in $\emptyset$.

Chapter 16

NP-Complete Problems

1. A formula is in 2-conjunctive normal form if it is the conjunction of clauses each of which has two literals. The 2-Satisfiability Problem is to determine whether an arbitrary 2-conjunctive normal form is satisfiable.

   We will describe a polynomial-time algorithm that solves the 2-Satisfiability Problem. The algorithm constructs a truth assignment if the formula is satisfiable. If not, the attempt to construct the assignment fails.

   Before outlining the strategy, we first note that the assignment of a truth value to a variable may be used to reduce the length of a 2-conjunctive normal form formula that needs to be considered for determining satisfiability. Consider the formula

   $$(x_1 \vee x_2) \wedge (\neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3) \wedge (x_1 \vee \neg x_4).$$

   If $t(x_1) = 1$, then any clause that contains x_1 can be deleted since it is satisfied by t. This produces

   $$(\neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3).$$

   Moreover, the occurrence of the literal $\neg x_1$ can be deleted from clauses since its value will not satisfy the clause. Applying this reduction yields

   $$(\neg x_2 \vee x_3) \wedge (x_3).$$

   Since x_3 occurs as a singleton, it is necessary to assign $t(x_3) = 1$ to satisfy the formula. Doing the reduction produces

   $$(\neg x_2).$$

   The only truth assignment that satisfies this clause is $t(x_2) = 0$.

   In this example, we set $t(x_1) = 1$ and reduced based on this truth assignment. When all clauses are removed following the preceding rules, we have succeeded in constructing a satisfying truth assignment. In the preceding reduction, no value has been assigned to the variable x_4. This indicates that the value assigned to x_4 is irrelevant; either choice will yield an assignment that satisfies the formula.

   Before describing the algorithm in detail, let us consider two more examples; one which fails to find a satisfying truth assignment and one which requires several steps to complete. We attempt to find a satisfying truth assignment for

   $$(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_2 \vee x_3).$$

Setting $t(x_1) = 1$ yields

$$(\neg x_2) \wedge (\neg x_3) \wedge (x_2 \vee x_3).$$

Reducing by setting $t(x_2) = 0$ produces the contradictory clauses

$$(\neg x_3) \wedge (x_3).$$

The reduction fails because there is no assignment that can satisfy both of these clauses. At this point we conclude that there is no satisfying truth assignment that can be obtained by setting $t(x_1)$ to 1. The next step would be to begin the process by setting $t(x_1) = 0$ and trying to find a satisfying truth assignment with this assignment to x_1.

Our final example illustrates the need to select the truth value for more than one variable to produce a satisfying assignment. The result of assigning $t(x_1) = 1$ in

$$(x_1 \vee x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_3 \vee x_2) \wedge (x_4 \vee x_5)$$

produces

$$(x_3) \wedge (\neg x_3 \vee x_2) \wedge (x_4 \vee x_5),$$

requiring $t(x_3) = 1$ in order to satisfy the formula. The effect of this assignment yields

$$(x_2) \wedge (x_4 \vee x_5),$$

which requires $t(x_2) = 1$ in a satisfying truth assignment. The resulting formula

$$(x_4 \vee x_5),$$

matches neither of the termination conditions; the formula is not empty and it does not contain contradictory clauses. We continue the process by selecting a truth assignment for one of the variables in $(x_4 \vee x_5)$ and applying the reduction procedure.

Each reduction process is initiated with the selection of a truth assignment for a variable x_i. If the reduction does not produce contradictory clauses, this truth assignment, along with of those obtained during the reduction, are incorporated into the final truth assignment.

To describe the algorithm we let $u = w_1 \wedge \cdots \wedge w_m$ be a 2-conjunctive normal form formula and $\mathrm{V} = \{x_1, \ldots, x_n\}$ be the set of variables of u. The objective is to build a satisfying truth assignment t on V. For a formula v, we use the reduction procedure to build a truth assignment T on the set of variables in v, denoted $var(v)$. If the reduction does not fail, we then assign the truth values of T to t.

The algorithm proceeds as follows:

```
1. Let $v = u$.
2. Set $T(x) \uparrow$ for all $x \in var(v)$, and pick a variable $x_i$ in $var(v)$.
3. Set $T(x_i) = 1$ and reduce $v$.
4. If the reduction did not fail and produced the formula $v'$, then
     - Set $t(x) = T(x)$ for all $x \in var(v)$ for which $T(x)$ is defined.
     - If $v'$ is empty, continue with step 8,
     - otherwise set $v = v'$ and continue with step 2.
5. The reduction for $T(x_i) = 1$ has failed. Set $T(x_i) = 0$ and reduce $v$.
6. If the reduction did not fail and produced the formula $v'$, then
     - Set $t(x) = T(x)$ for all $x \in var(v)$ for which $T(x)$ is defined.
```

- If v' is empty, continue with step 8,
- otherwise set $v = v'$ and continue with step 2.

7. If both reductions failed, the formula is unsatisfiable.
8. For each x_j with $t(x_j)\uparrow$, select any truth value for $t(x_j)$. The truth assignment t satisfies u.

The truth assignment t clearly satisfies u, since the reduction removes a clause only when it has been satisfied. Moreover, the algorithm will produce a truth assignment whenever u is satisfiable. Assume that u is satisfied by some truth assignment t'. The selection of truth values $T(x_i) = t'(x_i)$ in steps 3, 5, and 7 will produce the truth assignment $t = t'$. Consequently, we are assured of the construction of some truth assignment whenever u is satisfiable.

Once a variable is assigned a truth value in step 4 or 6, that value does not change. At most $2n$ reductions are possible and each reduction can alter at most m clauses. Thus the algorithm completes in time polynomial to the number of variables and clauses in the formula.

6. A guess-and-check strategy that yields a nondeterministic polynomial time solution to the minimum set cover problem is straightforward. The guess nondeterministically selects a subcollection $\mathcal{C}'$ of $\mathcal{C}$ of size k. The check verifies that every element in S is in the union of the sets in $\mathcal{C}'$.

We show that the Minimum Set Cover Problem is NP-hard by reducing the 3-Satisfiability Problem to it. First we will outline the construction of a set S, a collection $\mathcal{C}$ of subsets of S, and a number k from a 3-conjunctive normal form formula u. We will then show that S is covered by k sets in $\mathcal{C}$ if, and only if, u is satisfiable.

To accomplish the reduction, we must reformulate truth assignments and clauses in terms of sets. Let

$$\begin{aligned} u &= w_1 \wedge \cdots \wedge w_m \\ &= (u_{1,1} \vee u_{1,2} \vee u_{1,3}) \wedge \cdots \wedge (u_{m,1} \vee u_{m,2} \vee u_{m,3}) \end{aligned}$$

be a 3-conjunctive normal form formula with m clauses. The variables of u are $\{x_1, \ldots, x_n\}$ and the symbol $u_{j,k}$ represents the kth literal in clause w_j.

We will construct the elements of S and the sets of $\mathcal{C}$ in three steps. The sets are built so that the smallest cover $\mathcal{C}'$ of S consists of $2mn$ sets from $\mathcal{C}$. We begin with the sets that represent truth assignments. For each variable x_i, construct the sets

$$\begin{gathered} \{\neg x_{i,1}, a_{i,1}, b_{i,1}\} \\ \{x_{i,1}, a_{i,2}, b_{i,1}\} \\ \{\neg x_{i,2}, a_{i,2}, b_{i,2}\} \\ \{x_{i,2}, a_{i,3}, b_{i,2}\} \\ \{\neg x_{i,3}, a_{i,3}, b_{i,3}\} \\ \{x_{i,3}, a_{i,4}, b_{i,3}\} \\ \vdots \\ \{\neg x_{i,m}, a_{i,m}, b_{i,m}\} \\ \{x_{i,m}, a_{i,1}, b_{i,m}\} \end{gathered}$$

There are $2mn$ sets of this form, $2m$ for each variable x_i. The elements $x_{i,j}, \neg x_{i,j}$, $i = 1, \ldots, n$ and $j = 1, \ldots, m$, and $a_{i,1}, \ldots, a_{i,m}$ and $b_{i,1}, \ldots, b_{i,m}$, $i = 1, \ldots, n$ are added to the set S.

The preceding sets are the only sets of $\mathcal{C}$ that contain the elements $a_{i,1}, \ldots, a_{i,m}$ and $b_{i,1}, \ldots, b_{i,m}$. For a fixed variable i, we can cover $a_{i,1}, \ldots, a_{i,m}$ and $b_{i,1}, \ldots, b_{i,m}$ with m of these sets. This is possible only if we choose all the sets that contain $\neg x_{i,k}$ or all of the sets that contain $x_{i,k}$.

This choice provides our intuition for a truth assignment. If the cover contains all the sets with $\neg x_{i,k}$, we associate this with a truth assignment that assigns 1 to $x_{i,k}$. In a like manner, if the cover contains all the sets with $x_{i,k}$, we associate this with a truth assignment that assigns 0 to $x_{i,k}$.

Now we add elements and sets used to represent the clauses of u. For each clause w_j,

- If the literal x_i occurs in w_j, then construct set $\{x_{i,j}, s_{1,j}, s_{2,j}\}$.
- If the literal $\neg x_i$ occurs in w_j, then construct set $\{\neg x_{i,j}, s_{1,j}, s_{2,j}\}$.

These are the only sets of $\mathcal{C}$ that contain $s_{1,j}$ and $s_{2,j}$. Thus at least one of the three sets generated from w_j is required for any cover of S.

To this point, the cover that we are building requires $mn + m$ sets: mn from the truth setting sets and m from the clause sets. Each of the $a_{i,j}$'s, $b_{i,j}$'s, $s_{1,j}$'s, and $s_{2,j}$ are covered by such a selection. If the sets from the truth setting component for a variable x_i all contain $\neg x_{i,j}$, then the $x_{i,j}$'s are not covered. Similarly, if the sets from the truth setting component all contain $x_{i,j}$, then the $\neg x_{i,j}$'s are not covered. We must add more sets so that these elements will be covered. We will refer to these sets as the completion sets, since they complete the coverage of the $x_{i,j}$'s or $\neg x_{i,j}$'s

For each variable x_i and clause w_j, we add the sets

$$\{\neg x_{i,j}, c_{1,k}, c_{2,k}\}$$
$$\{x_{i,j}, c_{1,k}, c_{2,k}\}$$

for $k = 1, \ldots, m(n-1)$. The elements $c_{1,k}$ and $c_{2,k}$ appear only in these sets. Thus $m(n-1)$ completion sets are necessary to cover the c's

This completes the construction of S and $\mathcal{C}$ from u. From the form of the sets, we see that a cover of S must contain at least $2mn$ sets from $\mathcal{C}$. We now must show that u is satisfiable if, and only if, there are $2mn$ sets in $\mathcal{C}$ that cover S.

If u is satisfied by a truth assignment t, we can build a cover of S consisting of $2mn$ sets as follows. For each variable x_i,

- If $t(x_i) = 1$, then add the mn sets of the form $\{\neg x_{i,1}, a_{i,k}, b_{i,k}\}$ to $\mathcal{C}'$.
- Otherwise add the sets $\{x_{i,1}, a_{i,k}, b_{i,k}\}$ to $\mathcal{C}'$.

Each clause w_j is satisfied by at least one literal. For one of the satisfying literals we will add a clause set to $\mathcal{C}'$. If $t(x_i) = 1$ and x_i is in w_j, then add the set $\{x_{i,j}, s_{1,j}, s_{2,j}\}$ to $\mathcal{C}'$. If $t(x_i) = 0$ and $\neg x_i$ is in w_j, then add $\{\neg x_{i,j}, s_{1,j}, s_{2,j}\}$ to $\mathcal{C}'$. In either case, this set covers $s_{1,j}$ and $s_{2,j}$ and adds one set to $\mathcal{C}'$.

For a clause w_j that was satisfied by a positive literal x_i in the preceding step, the elements $x_{t,j}$, where $t \neq i$, are not covered by our sets. These are covered by the addition of the $m-1$ completion sets

$$\{x_{t,j}, c_{1,k}, c_{2,k}\},$$

where $t \neq i$ and a different k is selected for each set. These $m-1$ sets simultaneously complete the coverage of the $x_{t,j}$'s and the $c_{1,k}$'s and $c_{2,k}$'s.

Similarly, for a clause w_j that was satisfied by a negative literal $\neg x_i$, the elements $\neg x_{t,j}$, where $t \neq i$, are not covered by the truth setting or clause sets. These are covered by the addition of the $m - 1$ sets of the form

$$\{\neg x_{t,j}, c_{1,k}, c_{2,k}\},$$

where $t \neq i$.

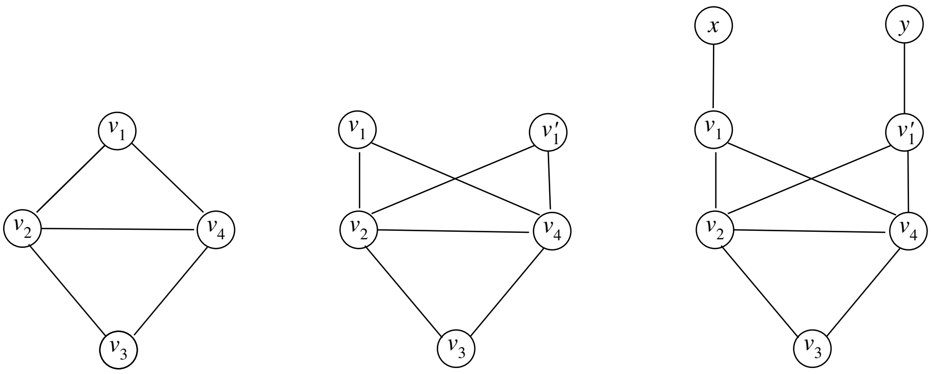
Thus we have a total of mn truth setting sets, m clause sets, and $m(n-1)$ completion sets for a cover of S by $2mn$ sets.

Conversely, if S is covered by $2mn$ sets from $\mathcal{C}'$. The cover must consist of mn truth setting sets, m linking sets, and $m(n-1)$ of the completion sets. We can obtain a truth assignment from the truth setting sets in $\mathcal{C}'$. If the sets $\{x_{i,1}, a_{i,k}, b_{i,k}\}$ are in the cover, then $t(x_i) = 0$; otherwise, $t(x_i) = 1$.

For each clause w_j, one set of the form $\{x_{i,j}, s_{1,j}, s_{2,j}\}$ or $\{\neg x_{i,j}, s_{1,j}, s_{2,j}\}$ is in the cover. In the former case, the x_i satisfies w_j. In the latter case, the $\neg x_i$ is satisfies w_j.

8. We will show that the Longest Path Problem is NP-hard by reducing the Hamiltonian Circuit Problem to it. The transformation converts a graph G = (N, A) into a graph G′ = (N′, A′) in two steps.

   We construct G′ from G by adding three nodes and the arcs incident to these nodes. Let N $= \{v_1, \ldots, v_n\}$ be the set of nodes of G. First we make a copy v_1' of v_1. For each arc $[v_i, v_1]$ in A, $v_i \neq v_1$, add an arc $[v_i, v_1']$ as illustrated in the diagram below.



   The construction of G′ is completed by adding nodes x and y and arcs $[x, v_1]$ and $[v_1', y]$. The reduction is complete by selecting the number $k = n + 2$ as the input in the Longest Path Problem.

   To show that the preceding construction is a reduction, we must show that G has a tour if, and only if, G′ has an acyclic path of length $n + 2$.

   Assume that G has a tour. The tour can be written as a path $v_1, v_{i_2}, \ldots, v_{i_n}, v_1$ of length n. Replacing the v_1 at the end of the path with v_1' and adding the arcs $[x, v_1]$ and $[v_1', y]$ creates an acyclic path of length $n + 2$ in G′.

   Now assume that G′ has an acyclic path of length $n + 2$. Since G′ has $n + 3$ nodes, this path must visit every node. To reach nodes x and y, the path must begin with the arc $[x, v_1]$ and end with $[v_1', y]$. Deleting these produces an acyclic path with endpoints v_1 and v_1'. Equating v_1 and v_1' produces a tour $v_1, v_{i_2}, \ldots, v_{i_n}, v_1$ in G.

9. a) The Multiprocessor Scheduling Problem can be converted into a decision problem by adding a deadline for the computations as follows:

Input:	set of tasks A, length of task function $l : \mathrm{A} \to \mathbf{N}$ number of processors k, and deadline m
Output:	yes, if there is a partition $\mathrm{A}_1, \dots, \mathrm{A}_k$ such that $l(A_i) \leq m$ for all i, no, otherwise.

With the deadline, the problem is to determine if there an assignment of tasks to processors that ensures that all tasks will be completed by time m.

b) Following the strategy employed in the proof of the NP-completeness of the Bin Packing Problem, we will reduce the Partition Problem to the Multiprocessor Scheduling Problem. The reduction transforms an instance of the Partition Problem into an instance of the Multiprocessor Scheduling Problem. The transformation is accomplished by the mapping

	Partition	Multiprocessor Scheduling
Set	A	A
Function	v	$l = v$
Number of sets		$k = 2$
Bound		$m = v(\mathrm{A})/2$

Now we must show that an instance (A, v) of the Partition Problem has a solution if, and only if, the corresponding instance (A, l, k, m) of the Multiprocessor Scheduling Problem has a solution.

Assume that there is a subset $\mathrm{A}' \subseteq \mathrm{A}$ such that $v(\mathrm{A}') = v(\mathrm{A})/2$. Then the partition $\mathrm{A}_1 = \mathrm{A}'$, $\mathrm{A}_2 = \mathrm{A} - \mathrm{A}'$ is a solution to the corresponding Multiprocessor Scheduling Problem.

Conversely, assume that the Multiprocessor Scheduling Problem has a solution A_1 and A_2. Since $\mathrm{A} = \mathrm{A}_1 \cup \mathrm{A}_2$, $l(\mathrm{A}) = l(\mathrm{A}_1) + l(\mathrm{A}_2)$. However, satisfaction of the deadline bound ensures us that $l(\mathrm{A}_1) \leq l(\mathrm{A})/2$ and $l(\mathrm{A}_2) \leq l(\mathrm{A})/2$. It follows that $l(\mathrm{A}_1) = l(\mathrm{A}_2) = l(\mathrm{A})/2$ and A_1 is a solution to the Partition Problem.

10. To show that the Integer Linear Programming Problem is NP-hard, we reduce the question of the satisfiability of a 3-conjunctive normal form formula to that of satisfying a system of integral inequalities. Let

$$u = w_1 \wedge w_2 \wedge \cdots \wedge w_m$$

be a 3-conjunctive normal form formula with clauses $w_i = u_{i,1} \vee u_{i,2} \vee u_{i,3}$ and let $\mathrm{V} = \{x_1, x_2, \dots, x_n\}$ be the set of Boolean variables occurring in w.

We will consider the literals as variables that may assume integer values. For each pair of literals x_i and $\neg x_i$ in V, the four equations

1. $x_i \geq 0$
2. $\neg x_i \geq 0$
3. $x_i + \neg x_i \geq 1$
4. $-x_i - \neg x_i \geq -1$

are satisfied only if one of x_i and $\neg x_i$ is one and the other is 0. A set of inequalities 1–4) is constructed for each variable x_i in V. Satisfying the $4n$ inequalities defines a truth assignment for V. For each clause w_i in u, we define the inequality

5. $u_{i,1} + u_{i,2} + u_{i,3} \geq 1$,

where $u_{i,j}$ is the literal in the jth position of clause i. This inequality is satisfied only if one of the literals in the clause is satisfied.

An integer linear programming system $A\mathbf{x} \geq \mathbf{b}$ can be constructed from the $4 \cdot n$ inequalities whose satisfaction defines a truth assignment for the variables of V and the m inequalities the represent the clauses. The matrix A has a column for each literal x_i and its negation $\neg x_i$. A row of A consists of the coefficients of the left-hand side of one of the $4 \cdot n + m$ inequalities. The column vector $\mathbf{b}$ is obtained from the right-hand side of the inequalities. The system $A\mathbf{x}$ $\geq \mathbf{b}$ is solvable if, and only if, the solution vector $\mathbf{x}$ defines a truth assignment that satisfies u. Clearly, a representation of A and $\mathbf{b}$ can be constructed in polynomial time from the clause u. Thus the Integer Linear Programming Problem is NP-hard.

13. Consider an instance of the Bin Packing Problem with items A $= \{a_1, \ldots, a_n\}$, size function s, and bin size m with the optimal number of bins denoted b^*. We will show that the first-fit algorithm generates a packing which requires at most $2b^*$ bins.

Let
$$T = \sum_{i=1}^{n} s(a_i),$$
be the total size of the items in the set A. T can be used to obtain lower bound on the number of bins in the optimal solution:
$$b^* \geq \lceil T/m \rceil.$$

First we note that at the completion of the first-fit algorithm, there is at most one bin that is less than half full. If a packing produces two bins numbered r and s that are both less than half full, then that packing could not have been produced by the first-fit algorithm. Assume that packing bin r occurred before packing bin s. When encountering the items in bin s, the first-fit algorithm would have placed these items into bin r. Thus whatever strategy was used to assign the items, it was not the first-fit algorithm.

Now assume that $\lceil 2T/m \rceil + 1$ or more bins are used in the first-fit packing. By the argument in the preceding paragraph, at least $\lceil 2T/m \rceil$ are more than half full. Thus the total size of the contents is greater than
$$m/2\lceil 2T/m \rceil \geq T,$$
which is a contradiction since T is the sum of the size of all the a_i's.

Thus we conclude that at most $\lceil 2T/m \rceil = 2\lceil T/m \rceil = 2b^*$ are required in the solution produced by the first-fit algorithm.

15. We will show that the modification to the greedy algorithm produces a 2-approximation algorithm for the Knapsack Problem. Let S $= \{a_1, \ldots, a_n\}$ be the set of objects, b the size bound, and $s : \mathrm{S} \to \mathbf{N}$ and $v : \mathrm{S} \to \mathbf{N}$ be the size and value functions, respectively. Also let c^* be the value of the optimal solution and c_g be the value produced by the greedy algorithm.

We assume that the items have been sorted by relative value,
$$v(a_1)/s(a_1) \geq v(a_2)/s(a_2) \geq \cdots \geq v(a_n)/s(a_n),$$
as required by the greedy algorithm and that every item in the list has size at most b. Any item with size greater than b can be removed from consideration since it cannot appear in a solution.

If all the items fit into the knapsack, then the optimal solution consists of all the items in S and this solution is discovered by the greedy algorithm.

Now we consider the case where

$$\sum_{i=1}^{n} s(a_i) > b.$$

Let a_k be the first item not able to be put into the knapsack by the greedy algorithm. Then

$$\sum_{i=1}^{k-1} v(a_i) \leq c_g \leq c^*.$$

The elements a_1, a_2, ..., a_k provide an optimal solution for a Knapsack Problem with the same set S but having size bound $b' = \sum_{i=1}^{k} s(a_i)$. The total value for this problem is $\sum_{i=1}^{k} v(a_i) = c'$. Now $c^* \leq c'$ since any set of items that can be packed into a knapsack with size bound b can also be packed into the larger knapsack with size bound b'.

Let a^* denote the item that has the largest value in S. Since a^* has the maximum value of any item in S, it follows that $v(a^*) \geq v(a_k)$. Combining the inequalities, $v(a^*) \geq v(a_k)$ and $\sum_{i=1}^{k-1} v(a_i) \leq c_g$ we get

$$\begin{aligned}
&\max\{v(a^*),\ c_g\} \\
&\geq \max\big\{v(a_k),\ \sum_{i=1}^{k-1} v(a_i)\big\} \\
&\geq \big(v(a_k) + \sum_{i=1}^{k-1} v(a_i)\big)/2 \\
&\geq \big(\sum_{i=1}^{k} v(a_i)\big)/2 \\
&\geq c'/2 \\
&\geq c^*/2
\end{aligned}$$

as desired. The third step follows from the inequality

$$\max\{x, y\} \geq (x + y)/2$$

that compares the maximum and the average of two numbers.

Chapter 17

Additional Complexity Classes

2. A three-tape Turing machine can be designed to accept the language $\{a^i b^i \mid i \geq 0\}$ in logarithmic space. The input is on tape 1 and tapes 2 and 3 serve as counters. Using the binary representation, the number of a's that have been read is recorded on tape 2 and the number of b's on tape three. A computation consists of two steps: accumulating the counts and comparing the results. The machine accepting $\{a^i b^i \mid i \geq 0\}$

   1. Reads the input on tape 1 and
      - If an a is read after a b, the computation halts and rejects the input.
      - On reading an a, the counter on tape 2 is incremented.
      - On reading a b, the counter on tape 3 is incremented.
   2. After processing the input, the strings on tapes 2 and 3 are compared.
      - If the strings are identical, the input is accepted.
      - Otherwise, it is rejected.

   The worst case performance for a string of length n occurs with the input string a^n. In this case, the counter on tape 2 uses $\lceil \lg(n) \rceil + 1$ tape squares.

3. The tape bound limits the number of configurations that can occur in an accepting computation. By Corollary 17.3.3, we know that the number of transitions is bounded by $c^{s(n)}$, where c is determined by the number of states and tape symbols of M.

   The strategy of employing a counter tape to terminate computations in which the number of transitions exceeds the bound can be used to construct a Turing machine M′ that accepts L and halts for all inputs. This strategy was described in the solution to Exercise 9 in Chapter 15. The machine M′ is no longer subject to the space bound, but it does demonstrate that L is recursive.

7. We sketch the design of a one-tape deterministic machine M for which every computation with input of length $n > 1$ reads exactly 2^n tape squares. The input alphabet for M is $\{a\}$.

   For an input string a^n, the computation begins with tape configuration Ba^nB. The computation begins by writing a # in tape position 0 and reading the input. If the input string has length 0, 1, or 2. The computation terminates after reading the entire input string. For input of length 2, the computation reads the initial blank, the input aa, and the following blank using 4 tape squares as required.

   For computations with input a^n, $n > 2$, we construct a loop that iteratively doubles the length of a string produced on previous iteration. The objective is to construct a string of length 2^n

on the tape beginning at position 0. Since this string must overwrite the input string, we will use the additional tape symbols to differentiate the input from the generated string.

The string beginning at position 0 that has length 4, 8, 16, and so forth will consist of #, b's, c's and X's and be referred to as the generated string. Each time we double the generated string, we will mark one input symbol as processed. The c's represent unprocessed input that are already in the generated string and the a's represent unprocessed whose tape positions have not already been incorporated into the string. After all the a's have been changed to b's or c's, the generated string is expanded by adding X's.

The computation will halt when all the input symbols have been processed. The meanings associated with the tape symbols are

#	left tape boundary marker
a, a'	unprocessed input symbol
c, c'	unprocessed input symbol in the generated string
b, b'	processed input symbol
X, X'	elements added to the generated string beyond the $n-1$st tape position
Y	elements used in the copying

The symbols with primes are used in the extension of the generated string.

As previously noted, the computation begins by writing a # in tape position 0 and reading the input. For input of length greater than 2, the final a is erased and the head returns to tape position 0. The first a in the input is then changed to b and the second a is changed to a c. If the next symbol is blank, an X is written. Otherwise, the next a is also changed to a c. This produces the tape configuration $\#bcX$ for an input string of length 3 and $\#bcca^{n-4}$ for an input of length 4 or more.

The # and b indicate that two symbols have been processed and the length of the generated string is 4.

The loop that generates the string proceeds as follows:

(a) The tape head moves to the right to find an a or c. If none is found, the computation halts.

(b) The tape head returns to position 0.

(c) One symbol at a time, each symbol b, c, X in the generated string is replaced by a b', c', X' and a symbol is added to the end of the generated string.

  - If an a is encountered, it is changed to an a'.
  - If a blank is encountered, it is changed you Y.

(d) After writing the final Y or a', the tape head returns to position 0. On this pass it changes Y and X' to X, a' and c' to c, and b' to b.

(e) The first occurrence of an a or c is changed to a b.

(f) The tape head returns to position 0 and the computation continues with step a).

For example, for input $aaaaa$, the tape configurations at the beginning of each loop are

$$\#bccaBBBBBBBBBBBB^{16}$$
$$\#bbccXXXBBBBBBBBB^{16}$$
$$\#bbbcXXXXXXXXXXXB^{16}$$
$$\#bbbbXXXXXXXXXXXX^{16}$$

12. Doubling the amount of available tape does not change the set of languages that can be accepted. We show that a language accepted by a Turing machine M whose computations require at most $2s(n)$ tape squares can also be accepted by Turing machine M′ with tape bound $s(n)$. By Theorem 17.2.2, we can restrict our attention to Turing machines with one work tape.

    Let M = (Q, Σ, Γ, δ, q_0, F) be a Turing machine whose computation have a $2s(n)$ tape bound. The input and tape alphabets Σ′ and Γ′ of M′ are

    Σ′ = Σ, and
    $\Gamma' = \{B\} \cup (\Gamma \times \Gamma \times \{1, 2\})$.

    Since M′ is designed to accept L(M), the input alphabets of M and M′ are identical. The tape alphabet of M′ has triples $[x, y, i]$ with $x, y \in \Gamma$ and $i \in \{1, 2\}$. The reduction in amount of tape required by the computations of M′ is achieved by the storing more information in a single tape symbol; a tape symbol of M′ contains two symbols from M. The integer value in the ordered triple indicates which of the two symbols from Γ would be scanned by M.

    When M′ reads a B on the work tape, M′ interprets the blank as the triple $[B, B, 1]$ and the transition converts it into a triple. The transition function δ' of M′ is obtained from the transition function δ of M as follows:

Transition of M	Transitions of M′
$\delta(q_i, a, x) = [q_j;\, b, d;\, z, R]$	$\delta'(q_i, a, [x, y, 1]) = [q_j;\, b, d;\, [z, y, 2], S]$ $\delta'(q_i, a, [y, x, 2]) = [q_j;\, b, d;\, [y, z, 2], R]$
$\delta(q_i, a, x) = [q_j;\, b, d;\, z, L]$	$\delta'(q_i, a, [x, y, 1]) = [q_j;\, a, d;\, [z, y, 1], L]$ $\delta'(q_i, a, [y, x, 2]) = [q_j;\, b, d;\, [y, z, 1], S]$
$\delta(q_i, a, B) = [q_j;\, b, d;\, z, R]$	$\delta'(q_i, a, B) = [q_j;\, b, d;\, [z, B, 2], S]$
$\delta(q_i, a, B) = [q_j;\, b, d;\, z, L]$	$\delta'(q_i, a, B) = [q_j;\, b, d;\, [z, B, 1], L]$

    where x, y, z are elements of Γ, $d \in \{L, R\}$, and S indicates that the tape head remains stationary in the transition.

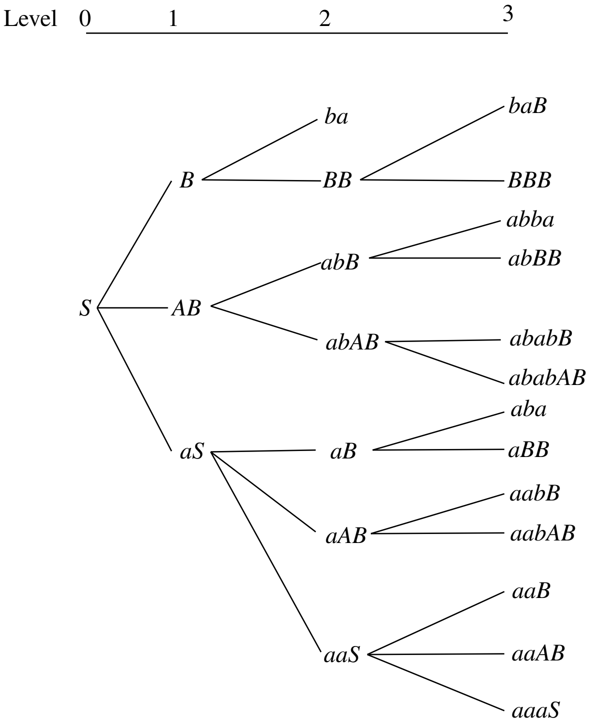
    The computation of M′ with an input string u simulates that of M with u but reads at most $s(n)$ tape squares on the work tape.

15. The NP-hardness of a $\mathcal{P}$-space complete language follows directly from the definition of $\mathcal{P}$-Space and the inclusions shown in Figure 17.4. Let Q be a $\mathcal{P}$-Space complete language. This means that Q is in $\mathcal{P}$-Space and that every language in $\mathcal{P}$-Space is reducible to Q in polynomial time. Since $\mathcal{NP}$ is a subset of $\mathcal{P}$-space, it follows that every language in $\mathcal{NP}$ is reducible to Q in polynomial time and, consequently, Q is NP-hard.

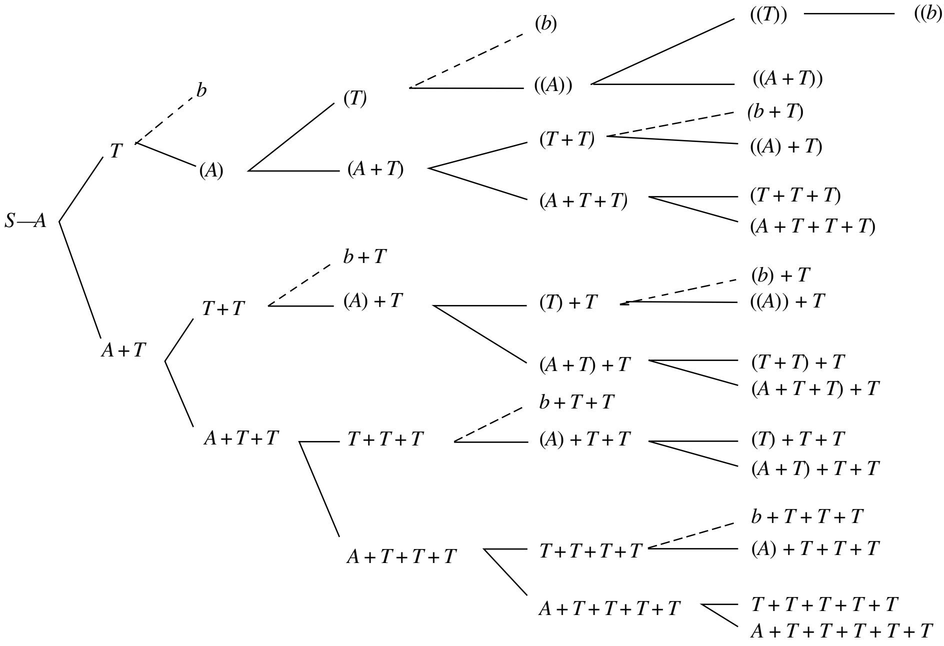
Chapter 18

Parsing: An Introduction

2. The subgraph of the graph of the grammar G consisting of all leftmost derivations of length three or less is



5. The search tree created by Algorithm 18.2.1 while parsing $((b))$ is given below. The nodes on each level are listed in the order of their generation.



The derivation of $((b))$ is obtained by following the arcs from the root S to the goal node $((b))$.

Derivation	Rule
$S \Rightarrow A$	$S \to A$
$\Rightarrow T$	$A \to T$
$\Rightarrow (A)$	$T \to (A)$
$\Rightarrow (T)$	$A \to T$
$\Rightarrow ((A))$	$T \to (A)$
$\Rightarrow ((T))$	$A \to T$
$\Rightarrow ((b))$	$T \to b$

11. Algorithm 18.4.1 will not terminate when run with input string aa and grammar

$$\begin{aligned} &S \to A \mid a \mid aa \\ &A \to S \end{aligned}$$

The rule $S \to a$ establishes Sa as the result of the first reduction for the string aa. The parse proceeds by alternating reductions with the rule $A \to S$ and the rule $S \to A$.

Chapter 19

LL(k) Grammars

2. a) The lookahead sets are given for the rules of the grammar. The lookahead set for a variable A is the union of the lookahead sets of the A rules.

Rule	Lookahead set
$S \to ABab$	$\{abab, acab, aab, cbab, ccab, cab\}$
$S \to bAcc$	$\{bacc, bccc\}$
$A \to a$	$\{abab, acab, aab, acc\}$
$A \to c$	$\{cbab, ccab, cab, ccc\}$
$B \to b$	$\{bab\}$
$B \to c$	$\{cab\}$
$B \to \lambda$	$\{ab\}$

3. b) The FIRST_1 and FOLLOW_1 sets for the variables of the grammar are

Variable	FIRST_1	FOLLOW_1
S	$a, \#$	λ
A	a	$a, \#$
B	a	$\#$

To determine whether the grammar is LL(1), the techniques outlined in Theorem 19.2.5 are used to construct the lookahead sets from the FIRST_1 and FOLLOW_1 sets.

Rule	Lookahead set
$S \to AB\#$	$\{a, \#\}$
$A \to aAb$	$\{a\}$
$A \to B$	$\{a, \#\}$
$B \to aBc$	$\{a\}$
$B \to \lambda$	$\{\#\}$

Since the lookahead sets for the alternative A rules both contain the symbol a, the grammar is not strong LL(1).

6. a) Algorithm 19.4.1 is used to construct the FIRST_2 sets for each of the variables of the grammar

$$\begin{aligned} S &\to ABC\#\#\# \\ A &\to aA \mid a \\ B &\to bB \mid \lambda \\ C &\to cC \mid a \mid b \mid c. \end{aligned}$$

The rules of the grammar produce the following assignment statements for step 3.2 of the algorithm.

$$\begin{aligned}
F(S) &:= F(S) \cup trunc_2(F'(A)F'(B)F'(C)\{cc\}) \\
F(A) &:= F(A) \cup trunc_2(\{a\}F'(A)) \cup \{a\}) \\
F(B) &:= F(B) \cup trunc_2(\{b\}F'(B)) \\
F(C) &:= F(C) \cup trunc_2(\{c\}F'(C)) \cup \{a,b,c\}
\end{aligned}$$

The algorithm is traced by exhibiting the sets produced after each iteration.

Step	$F(S)$	$F(A)$	$F(B)$	$F(C)$
0	$\emptyset$	$\emptyset$	$\{\lambda\}$	$\emptyset$
1	$\emptyset$	$\{a\}$	$\{\lambda, b\}$	$\{a, b, c\}$
2	$\{aa, ab, ac\}$	$\{a, aa\}$	$\{\lambda, b, bb\}$	$\{a, b, c, ca, cb, cc\}$
3	$\{aa, ab, ac\}$	$\{a, aa\}$	$\{\lambda, b, bb\}$	$\{a, b, c, ca, cb, cc\}$

The rules that contain a variable on the right-hand side generate the assignment statements used in steps 3.2.2 and 3.2.3 of Algorithm 18.5.1. The rules $S \to ABC\#\#$, $B \to bB$, and $C \to cC$ produce

$$\begin{aligned}
FL(C) &:= FL(C) \cup trunc_2(\{\#\#\}FL'(S)) \\
&= FL(C) \cup \{\#\#\}
\end{aligned}$$

$$\begin{aligned}
FL(B) &:= FL(B) \cup trunc_2(FIRST_2(C)\{\#\#\}FL'(S)) \\
&= FL(B) \cup \{a\#, b\#, c\#, ca, cb, cc\}
\end{aligned}$$

$$\begin{aligned}
FL(A) &:= FL(A) \cup trunc_2(\mathrm{FIRST}_2(B)\mathrm{FIRST}_2(C)\{\#\#\}FL'(S)) \\
&= FL(A) \cup \{a\#, b\#, c\#, ba, bb, bc\}
\end{aligned}$$

$$FL(B) := FL(B) \cup FL'(B)$$

$$FL(C) := FL(C) \cup FL'(C)$$

The last two assignment statements may be omitted from consideration since they do not contribute to the generation of $FL(B)$ and $FL(C)$.

Step	$FL(S)$	$FL(A)$	$FL(B)$	$FL(C)$
0	$\{\lambda\}$	$\emptyset$	$\emptyset$	$\emptyset$
1	$\{a\#, b\#, c\#, ba, bb, bc\}$	$\{a\#, b\#, c\#, ca, cb, cc\}$	$\{\#\#\}$	$\emptyset$
2	$\{a\#, b\#, c\#, ba, bb, bc\}$	$\{a\#, b\#, c\#, ca, cb, cc\}$	$\{\#\#\}$	$\emptyset$

The length two lookahead sets are used to construct the FIRST_2 and FOLLOW_2 sets.

Rule	Lookahead set
$S \to ABC\#\#$	$\{aa, ab, ac\}$
$A \to aA$	$\{aa\}$
$A \to a$	$\{aa, ab, ac\}$
$B \to bB$	$\{ba, bb, bc\}$
$B \to \lambda$	$\{a\#, b\#, c\#, ca, cb, cc\}$
$C \to cC$	$\{ca, cb, cc\}$
$C \to a$	$\{a\#\}$
$C \to b$	$\{b\#\}$
$C \to c$	$\{c\#\}$

The lookahead sets of the A rules show that the grammar is not strong LL(2).

7. To establish part (3) of Lemma 19.2.2, we must show that the identity

$$\text{FIRST}_k(au) = \{av \mid v \in \text{FIRST}_{k-1}(u)\}$$

holds for all $k \geq 1$. For convenience we let $\text{FIRST}_0(u) = \{\lambda\}$ for all strings u. Let $v \in FIRST_{k-1}(u)$ for some $k \geq 1$. This implies that there is a derivation $u \overset{*}{\Rightarrow} vx$ where $vx \in \Sigma^*$, $length(v) = k-1$ or $length(v) < k-1$ and $x = \lambda$. Consequently, $au \overset{*}{\Rightarrow} avx$ and av is in $\text{FIRST}_k(au)$.

The preceding argument shows that $\text{FIRST}_k(au) \subseteq \{a\}\text{FIRST}_{k-1}(u)$. We must now establish the opposite inclusion. Let $av \in \text{FIRST}_{k-1}(au)$. Then $au \overset{*}{\Rightarrow} avx$ where $length(v) = k-1$ or $length(v) < k-1$ and $x\lambda$. The derivation $u \overset{*}{\Rightarrow} vx$, obtained by deleting the leading a from the preceding derivation, shows that $v \in \text{FIRST}_k(u)$.

8. We will use a proof by contradiction to show every LL(k) grammar is unambiguous. Let G be an LL(k) grammar and assume that G is ambiguous. Then there is some string z that has two distinct leftmost derivations. Let $S \overset{*}{\Rightarrow} uAv$, $u \in \Sigma^*$, be the initial subderivation that is identical in both derivations of z. The two derivations can be written

$$1)\, S \overset{*}{\Rightarrow} uAv \overset{*}{\Rightarrow} ux_1v \overset{*}{\Rightarrow} uw = z$$

$$2)\, S \overset{*}{\Rightarrow} uAv \overset{*}{\Rightarrow} ux_2v \overset{*}{\Rightarrow} uw = z.$$

The derivations continue by applying rules $A \to x_1$ and $A \to x_2$ to the variable A.

These derivations show that the string $trunc_k(w)$ is in both $\text{LA}_k(uAv, A \to x_1)$ and $\text{LA}_k(uAv, A \to x_2)$. This is a contradiction; the sets $\text{LA}_k(uAv, A \to x_1)$ and $\text{LA}_k(uAv, A \to x_2)$ are disjoint since G is an LL(k) grammar. Thus our assumption, that G is ambiguous, must be false.

The exercise asked us to show that every strong LL(k) grammar is unambiguous. This follows immediately since the strong LL(k) grammars are a subfamily of the LL(k) grammars.

9. a) The lookahead sets for the rules of G_1 are

Rule	Lookahead set
$S \to aSb$	$\{a^j a^i c^i b^j \mid j > 0, i \geq 0\}$
$S \to A$	$\{a^i c^i \mid i \geq 0\}$
$A \to aAc$	$\{a^i c^i \mid i > 0\}$
$A \to \lambda$	$\{\lambda\}$

For any $k > 0$, the string a^k is a lookahead string for both of the S rules. Thus G_1 is not strong LL(k).

A pushdown automaton that accepts $\text{L}(\text{G}_1)$ is defined by the following transitions.

$$\begin{aligned}
\delta(q_0, a, \lambda) &= [q_1, A] \\
\delta(q_1, a, \lambda) &= [q_1, A] \\
\delta(q_1, b, A) &= [q_2, \lambda] \\
\delta(q_1, c, A) &= [q_3, \lambda] \\
\delta(q_2, b, A) &= [q_2, \lambda] \\
\delta(q_3, b, A) &= [q_2, \lambda] \\
\delta(q_3, c, A) &= [q_3, \lambda]
\end{aligned}$$

States q_0 and q_1 read an a and push A onto the stack. State q_3 reads the c's and q_2 the b's with each transition popping the stack. The accepting states are q_0 and q_2.

11. a) The process of transforming the grammar into an equivalent strong LL(1) grammar begins by removing the directly left recursive A rules, producing

$$\begin{aligned} &S \to A\# \\ &A \to aB \mid ZaB \\ &B \to bBc \mid \lambda \\ &Z \to bZ \mid cZ \mid b \mid c \end{aligned}$$

The resulting grammar is not strong LL(1) since there are two Z rules that begin with b and two that begin with c. Left factoring these rules we obtain

$$\begin{aligned} &S \to A\# \\ &A \to aB \mid ZaB \\ &B \to bBc \mid \lambda \\ &Z \to bX \mid cY \\ &X \to Z \mid \lambda \\ &Y \to Z \mid \lambda \end{aligned}$$

Examining the length one lookahead sets, we see that this grammar is strong LL(1).

Rule	Lookahead set
$S \to A\#$	$\{a, b, c\}$
$A \to ab$	$\{a\}$
$A \to Zab$	$\{b, c\}$
$B \to bBc$	$\{b\}$
$B \to \lambda$	$\{\#\}$
$Z \to bX$	$\{b\}$
$Z \to cY$	$\{c\}$
$X \to Z$	$\{b, c\}$
$X \to \lambda$	$\{a\}$
$Y \to Z$	$\{b, c\}$
$Y \to \lambda$	$\{a\}$

13. a) The lookahead sets for the grammar G

$$\begin{aligned} &S \to aAcaa \mid bAbcc \\ &A \to a \mid ab \mid \lambda \end{aligned}$$

are

Rule	Lookahead set
$S \to aAcaa$	$\{aacaa, aabcaa, acaa\}$
$S \to bAbcc$	$\{babcc, babbcc, bbcc\}$
$A \to a$	$\{acaa, abcc\}$
$A \to ab$	$\{abcaa, abbcc\}$
$A \to \lambda$	$\{caa, bcc\}$

One symbol lookahead is sufficient to discriminate between the S rules. Four symbols are required to choose the appropriate A rule. Thus G is strong LL(4).

To show that G is LL(3), the lookahead sets are constructed for the sentential forms of the grammar.

Rule	Sentential form	Lookahead set
$S \to aAcaa$	S	$\{aaccaa, aabcaa, acaa\}$

$S \to bAbcc$	S	$\{babcc, babbcc, bbcc\}$
$A \to a$	$aAcaa$	$\{acaa\}$
$A \to ab$	$aAcaa$	$\{abcaa\}$
$A \to \lambda$	$aAcaa$	$\{caa\}$
$A \to a$	$bAbcc$	$\{abcc\}$
$A \to ab$	$bAbcc$	$\{abbcc\}$
$A \to \lambda$	$bAbcc$	$\{bcc\}$

14. For every $i > 0$, the string a^i is a prefix of both a^i and a^ib^i.

    A nondeterministic pushdown automaton that accepts $\{a^i \mid i \geq 0\} \cup \{a^ib^i \mid i \geq 0\}$ was given in Example 7.1.2. A deterministic approach to recognizing strings of this form uses an endmarker # on the strings. The computation reads leading a's and pushes an A on the stack in state q_0 until either the endmarker or a b is read. If the endmarker is read, the stack is popped and the string is accepted. If a b is read, a loop is entered that compares the number of b's with the number of A's stored on the stack. The pushdown automaton defined by the transitions

    $$\begin{aligned}\delta(q_0, a, \lambda) &= \{[q_0, A]\}\\ \delta(q_0, \#, \lambda) &= \{[q_1, \lambda]\}\\ \delta(q_0, b, A) &= \{[q_2, \lambda]\}\\ \delta(q_1, \lambda, A) &= \{[q_1, \lambda]\}\\ \delta(q_2, b, A) &= \{[q_2, \lambda\}\\ \delta(q_2, \#, \lambda) &= \{[q_2, \lambda\}\end{aligned}$$

    with accepting states q_1 and q_2 accepts the language.

15. It follows immediately from the definitions of LL(1) and strong LL(1) grammars that a strong LL(1) grammar G is also LL(1). To conclude that these families of grammars are identical, we must show that every LL(1) grammar is also strong LL(1).

    Let G = (V, Σ, S, P) be an LL(1) grammar and assume that it is not strong LL(1). Since G is not strong LL(1), there are derivations

    $$S \overset{*}{\Rightarrow} u_1Az_1 \Rightarrow u_1xz_1 \overset{*}{\Rightarrow} u_1v_1z_1 \overset{*}{\Rightarrow} u_1v_1w_1$$
    $$S \overset{*}{\Rightarrow} u_2Az_2 \Rightarrow u_2yz_2 \overset{*}{\Rightarrow} u_2v_2z_2 \overset{*}{\Rightarrow} u_2v_2w_2$$

    where

    a) u_i, v_i, and w_i are in Σ^*, $z_i \in (\Sigma \cup \mathrm{V})^*$,

    b) $u_1v_1w_1 \neq u_2v_2w_2$,

    c) $A \to x$ and $A \to y$ are distinct rules of G, and

    d) $\mathrm{FIRST}_1(v_1w_1) = \mathrm{FIRST}_1(v_2w_2)$.

    That is, the FIRST sets do not distinguish between the rule $A \to x$ and $A \to y$.

    We will use the preceding derivations to show that the original grammar G is not LL(1). But this is a contradiction, so our assumption that G is not strong LL(1) must be false. There are two cases to consider: when $v_1 = v_2 = \lambda$ and when at least one of v_1 or v_2 is not λ.

Case 1: $v_1 = v_2 = \lambda$. In this case, both of the rules $A \to x$ and $A \to y$ initiate derivations of the form $A \overset{*}{\Rightarrow} \lambda$. Using these subderivations, we have two distinct derivations

$$S \overset{*}{\Rightarrow} u_1Az_1 \Rightarrow u_1xz_1 \overset{*}{\Rightarrow} u_1z_1 \overset{*}{\Rightarrow} u_1w_1$$
$$S \overset{*}{\Rightarrow} u_1Az_1 \Rightarrow u_1yz_1 \overset{*}{\Rightarrow} u_1z_1 \overset{*}{\Rightarrow} u_1w_1$$

of the string u_1w_1. Since the lookahead sets $\mathrm{LA}_1(u_1Az_1, A \to x)$ and $\mathrm{LA}_1(u_1Az_1, A \to y)$ are not disjoint, G is not LL(1).

Case 2: At least one of v_1 or v_2 is not λ. Without loss of generality, assume that $v_1 \neq \lambda$. From d) above we note that

$$\mathrm{FIRST}_1(v_1w_1) = \mathrm{FIRST}_1(v_1) = \mathrm{FIRST}_1(v_2w_2).$$

We can build derivations

$$S \overset{*}{\Rightarrow} u_2Az_2 \Rightarrow u_2xz_2 \overset{*}{\Rightarrow} u_2v_1z_2 \overset{*}{\Rightarrow} u_2v_1w_2$$
$$S \overset{*}{\Rightarrow} u_2Az_2 \Rightarrow u_2yz_2 \overset{*}{\Rightarrow} u_2v_2z_2 \overset{*}{\Rightarrow} u_2v_2w_2,$$

which demonstrate that $\mathrm{LA}_1(u_2Az_2, A \to x)$ and $\mathrm{LA}_1(u_2Az_2, A \to y)$ have the same FIRST_1 set and consequently G is not LL(1).

In both cases, we have shown that a grammar that is strong LL(1) is also LL(1). Combining this with inclusion of the familty of LL(1) grammars in the family of strong LL(1) grammars yields the equality of these families of grammars.

Chapter 20

LR(k) Grammars

1. a) The LR(0) contexts for the rules of G

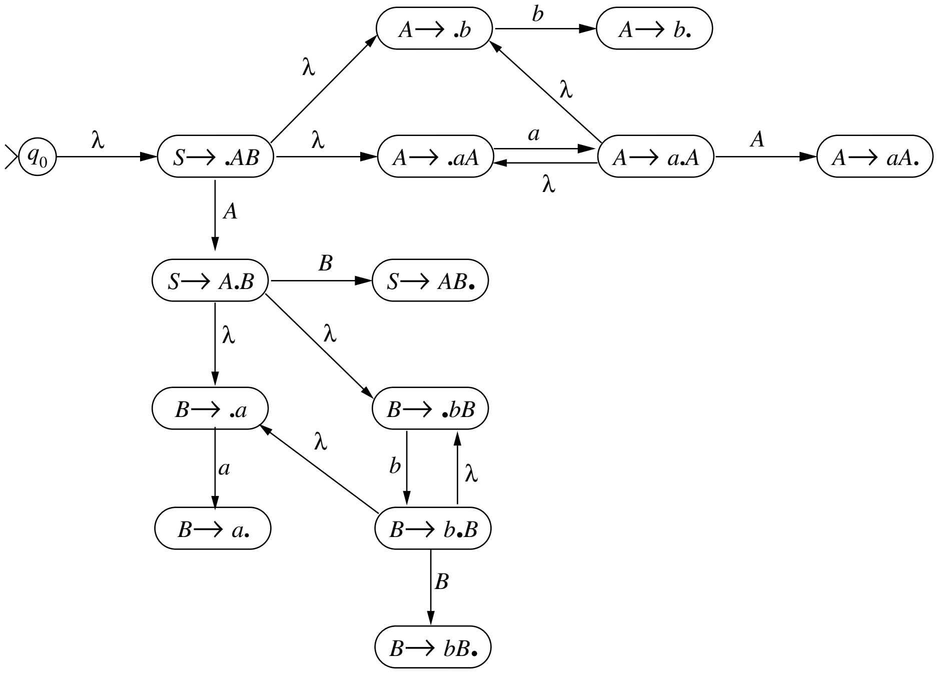
$$\begin{aligned} &S \to AB \\ &A \to aA \mid b \\ &B \to bB \mid a \end{aligned}$$

are obtained from the rightmost derivations

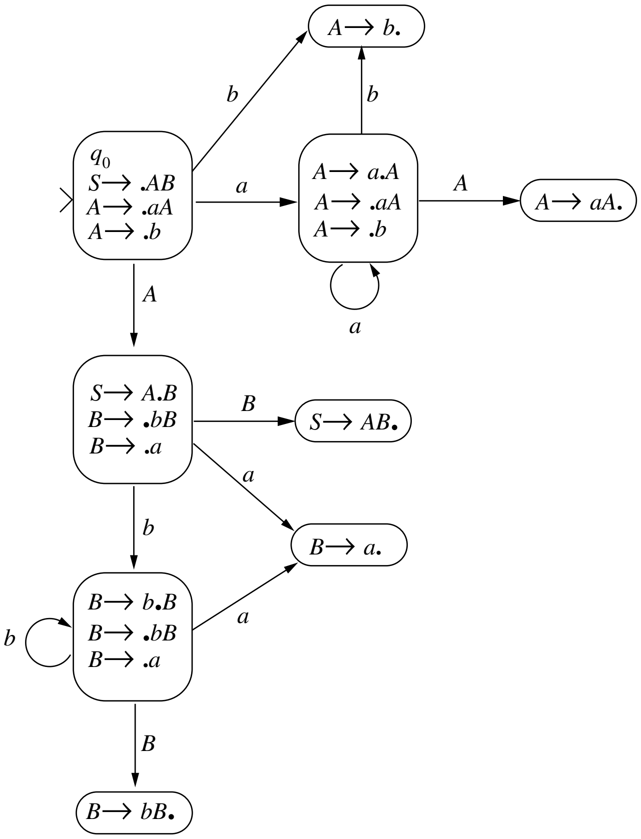
$$S \Rightarrow AB \overset{i}{\Rightarrow} Ab^iB \Rightarrow Ab^ia \overset{j}{\Rightarrow} a^jAb^iB \Rightarrow a^jbb^ia$$

Rule	LR(0) contexts
$S \to AB$	$\{AB\}$
$A \to aA$	$\{a^iA \mid i > 0\}$
$A \to b$	$\{a^ib \mid i \geq 0\}$
$B \to bB$	$\{Ab^iB \mid i > 0\}$
$B \to a$	$\{Ab^ia \mid i \geq 0\}$

The nondeterministic LR(0) machine of G is constructed directly from the specifications of Definition 20.3.2.



The deterministic LR(0) machine of G is



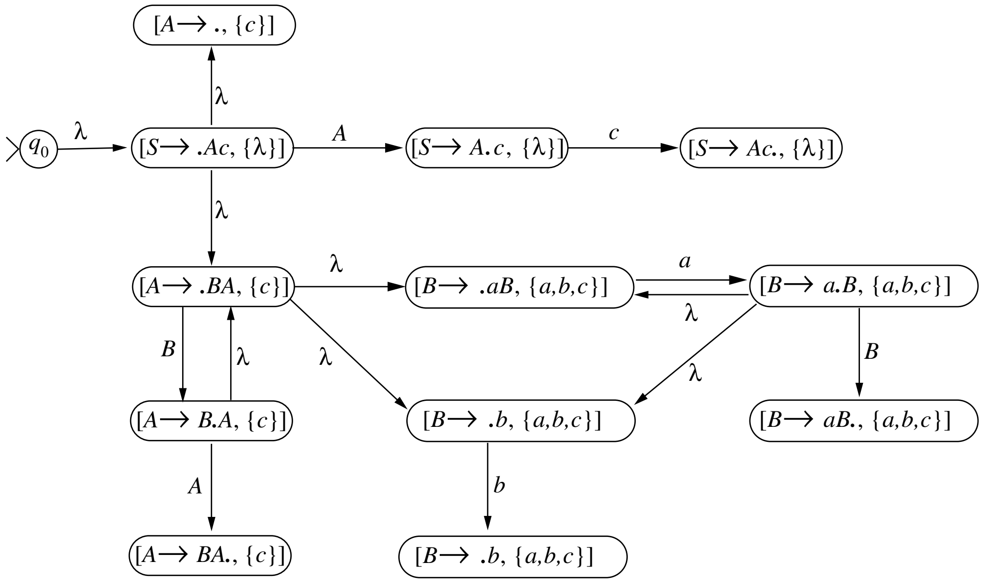
Since every state with a complete item contains only that item, the grammar G is LR(0).

3. The grammar AE is not LR(0) since A is an LR(0) context of the rule $S \to A$ and a viable prefix of the rule $A \to A + T$.

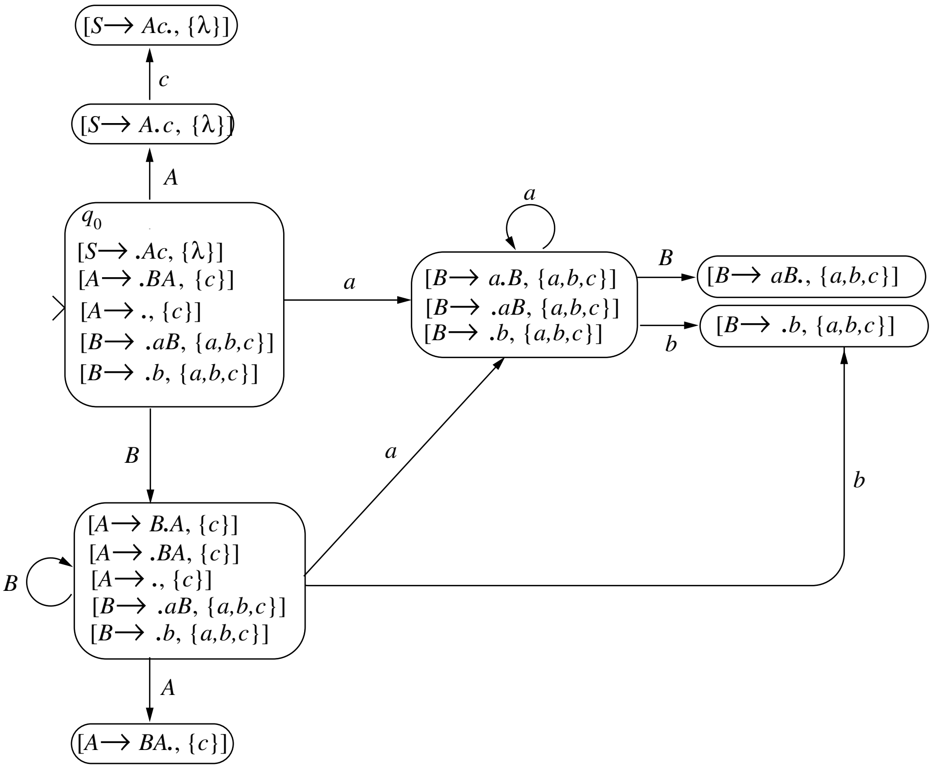
7. a) The nondeterministic LR(1) machine is constructed for the grammar

$$\begin{aligned} S &\to Ac \\ A &\to BA \mid \lambda \\ B &\to aB \mid b \end{aligned}$$

The λ-transitions from the item $[S \to .Ac, \{\lambda\}]$ generate LR(1) items with lookahead set $\{c\}$ since the symbol c must follow all reductions that produce A. Similarly the lookahead set $\{a, b, c\}$ is generated by λ-transitions from $[A \to .BA, \{c\}]$ since the string that follows B, A in this case, generates a, b and λ.



The corresponding deterministic LR(1) is



Since the deterministic machine satisfies the conditions of Definition 20.5.3, the grammar is LR(1).