

$$g) n - m = \begin{cases} n - m & \text{if } n \geq m \\ 0 & \text{otherwise} \end{cases}$$

6. Construct Turing machines that perform the actions specified by the following macros. The computation should not leave the segment of the tape specified in the input configuration.
- ZR; input $\underline{B}BB$, output $\underline{B}\bar{0}B$
 - FL; input $\underline{B}\bar{n}B^i\underline{B}$, output $\underline{B}\bar{n}B^iB$
 - E_2 ; input $\underline{B}\bar{n}B\bar{m}B$, output $\underline{B}B^{n+m+3}B$
 - T; input $\underline{B}B^i\bar{n}B$, output $\underline{B}\bar{n}B^iB$
 - BRN; input $\underline{B}\bar{n}B$, output $\underline{B}\bar{n}B$
 - INT; input $\underline{B}\bar{n}B\bar{m}B$, output $\underline{B}\bar{m}B\bar{n}B$
7. Use the macros and machines constructed in Sections 9.2 through 9.4 to design machines that compute the following functions:
- $f(n) = 2n + 3$
 - $f(n) = n^2 + 2n + 2$
 - $f(n_1, n_2, n_3) = n_1 + n_2 + n_3$
 - $f(n, m) = m^3$
 - $f(n_1, n_2, n_3) = n_2 + 2n_3$
8. Design machines that compute the following relations. You may use the macros and machines constructed in Sections 9.2 through 9.4 and the machines constructed in Exercise 5.
- $gt(n, m) = \begin{cases} 1 & \text{if } n > m \\ 0 & \text{otherwise} \end{cases}$
 - $persq(n) = \begin{cases} 1 & \text{if } n \text{ is a perfect square} \\ 0 & \text{otherwise} \end{cases}$
 - $divides(n, m) = \begin{cases} 1 & \text{if } n > 0, m > 0, \text{ and } m \text{ divides } n \\ 0 & \text{otherwise} \end{cases}$
9. Trace the actions of the machine MULT for computations with input
- $n = 0, m = 4$
 - $n = 1, m = 0$
 - $n = 2, m = 2$.
10. Describe the mapping defined by each of the following composite functions:
- $add \circ (mult \circ (id, id), add \circ (id, id))$
 - $p_1^{(2)} \circ (s \circ p_1^{(2)}, e \circ p_2^{(2)})$
 - $mult \circ (c_2^{(3)}, add \circ (p_1^{(3)}, s \circ p_2^{(3)}))$
 - $mult \circ (mult \circ (p_1^{(1)}, p_1^{(1)}), p_1^{(1)})$.

11. Give examples of total unary number-theoretic functions that satisfy the following conditions:

 - g is not id and h is not id but $g \circ h = id$.
 - g is not a constant function and h is not a constant function but $g \circ h$ is a constant function.

12. Give examples of unary number-theoretic functions that satisfy the following conditions:

 - g is not one-to-one, h is not total, $h \circ g$ is total.
 - $g \neq e$, $h \neq e$, $h \circ g = e$, where e is the empty function.
 - $g \neq id$, $h \neq id$, $h \circ g = id$, where id is the identity function.
 - g is total, h is not one-to-one, $h \circ g = id$.

* 13. Let F be a Turing machine that computes a total unary number-theoretic function f . Design a machine that returns the first natural number n such that $f(n) = 0$. A computation should continue indefinitely if no such n exists. What will happen if the function computed by F is not total?

14. Let F be a Turing machine that computes a total unary number-theoretic function f . Design a machine that computes the function

$$g(n) = \sum_{i=0}^n f(i).$$

15. Let F and G be Turing machines that compute total unary number-theoretic functions f and g , respectively. Design a Turing machine that computes the function

$$h(n) = \sum_{i=0}^n eq(f(i), g(i)).$$

That is, $h(n)$ is the number of values in the range 0 to n for which the functions f and g assume the same value.

16. A unary relation R over \mathbb{N} is Turing computable if its characteristic function is computable. Prove that every computable unary relation over \mathbb{N} defines a recursive language. *Hint:* Construct a machine that accepts R from the machine that computes its characteristic function.
 - * 17. Let $R \subseteq \{I\}^+$ be a recursive language. Prove that R defines a computable unary relation over \mathbb{N} .
 18. Prove that there are unary relations over \mathbb{N} that are not Turing computable.
 19. Let F be the set consisting of all total unary number-theoretic functions that satisfy $f(i) = i$ for every even natural number i . Prove that there are functions in F that are not Turing computable.

Bibliographic

The Turing machine family of abstract how, 1973]. Rand and a finite number of instructions of a range arithmetic operations computer architecture to Turing machines

- following
- constant
- g condi-
- function
 $i) = 0$. A
pen if the
- unction f .
- functions
- ions f and
- on is com-
ursive lan-
computes its
- ary relation
- that satisfy
n F that are
- have (
20. Let v_1, v_2, v_3, v_4 be a listing of the variables used in a TM program and assume register 1 contains a value. Trace the action of the instruction STOR $v_2, 1$. To trace the actions, use the technique in Example 9.3.2.
21. Give a TM program that computes the function $f(v_1, v_2) = v_1 \div v_2$.

Bibliographic Notes

The Turing machine assembly language provides an architecture that resembles another family of abstract computing devices known as random access machines [Cook and Reckhow, 1973]. Random access machines consist of an infinite number of memory locations and a finite number of registers, each of which is capable of storing a single integer. The instructions of a random access machine manipulate the registers and memory and perform arithmetic operations. These machines provide an abstraction of the standard von Neumann computer architecture. An introduction to random access machines and their equivalence to Turing machines can be found in Aho, Hopcroft, and Ullman [1974].

CHAPTER 10

The Chomsky Hierarchy

In Chapter 3, regular and context-free grammars were introduced as rule-based systems for generating the strings of language. A rule defines a string transformation, and a sentence of the language is obtained by a sequence of permissible transformations. The regular and context-free grammars are subsets of the more general class of phrase-structure grammars. Phrase-structure grammars were proposed as syntactic models of natural language by Noam Chomsky. In this chapter we will consider two additional families of phrase-structure grammars, unrestricted grammars and context-sensitive grammars. The four families of grammars, regular, context-free, context-sensitive, and unrestricted, make up the Chomsky hierarchy of phrase-structure grammars, with each successive family in the hierarchy permitting additional flexibility in the definition of a rule.

Automata were designed to mechanically recognize regular and context-free languages; deterministic finite automata accept the languages generated by regular grammars and push-down automata accept the languages generated by context-free grammars. The relationship between grammatical generation and mechanical acceptance is extended to the new families of grammars. Turing machines are shown to accept the languages generated by unrestricted grammars. A class of machines obtained by limiting the memory available to a Turing machine accepts the languages generated by context-sensitive grammars.

10.1 Unrestricted Grammars

Phrase-structure grammars were designed to provide formal models of the syntax of natural language. The name, phrase-structure, is based on the proposition that the sentences of

language may have several different syntactic patterns. The sentences themselves are made up of phrases: noun phrases, verb phrases, and the like, that are arranged as specified by one of the sentence patterns. The rules of the grammar define the structure of both the sentences and the phrases.

The components of a phrase-structure grammar are the same as those of the regular and context-free grammars studied in Chapter 3. A phrase-structure grammar consists of a finite set V of variables, an alphabet Σ , a start variable, and a set of rules. A rule has the form $u \rightarrow v$, where u and v can be any combination of variables and terminals, and defines a permissible string transformation. The application of a rule to a string z is a two-step process that consists of

- i) matching the left-hand side of the rule to a substring of z , and
- ii) replacing the left-hand side with the right-hand side.

The application of the rule $u \rightarrow v$ to the string xuy , written $xuy \Rightarrow xv y$, produces the string $xv y$. A string q is derivable from p , $p \xrightarrow{*} q$, if there is a sequence of rule applications that transforms p to q . The language of G , denoted $L(G)$, is the set of terminal strings derivable from the start symbol S . Symbolically, $L(G) = \{w \in \Sigma^* \mid S \xrightarrow{*} w\}$.

A family of grammars is defined by the restrictions placed on the form of the rules. A context-free grammar is a phrase-structure grammar in which the left-hand side of every rule is a single variable. The right-hand side can be any combination of variables and terminals. Each rule of a regular grammar is required to have one of the following forms:

- i) $A \rightarrow aB$,
- ii) $A \rightarrow a$, or
- iii) $A \rightarrow \lambda$,

where $A, B \in V$, and $a \in \Sigma$.

The unrestricted grammars are the largest class of phrase-structure grammars. There are no constraints on a rule other than requiring that the left-hand side must not be null.

Definition 10.1.1

An **unrestricted grammar** is a quadruple (V, Σ, P, S) , where V is a finite set of variables; Σ (the alphabet) is a finite set of terminal symbols; P is a set of rules; and S is a distinguished element of V . A production of an unrestricted grammar has the form $u \rightarrow v$, where $u \in (V \cup \Sigma)^+$ and $v \in (V \cup \Sigma)^*$. The sets V and Σ are assumed to be disjoint.

Two examples are given that illustrate the generative power of unrestricted grammars. Example 10.1.1 shows that the language $\{a^i b^i c^i \mid i \geq 0\}$, which we know is not derivable by any context-free grammar, can be generated by an unrestricted grammar with six rules. The second example shows how unrestricted rules can be used to generate copies of a string.

Example 10.1.1

The unrestricted

with start symbol generated by a d

using the rule A final C to pass the reaching the leftmost occurrence of the

Example 10.1.2

The unrestricted

generates the lan

The addition of the variable A to a terminal, the deletion enclosed in the brackets terminal is added

e made
by one
ntences

regular
sts of a
has the
defines
wo-step

he string
ions that
lerivable

rules. A
very rule
erminals.

urs. There
e null.

variables;
s a distin-
v, where

grammars.
erivable by
rules. The
a string.

Example 10.1.1

The unrestricted grammar

$$\begin{aligned} V &= \{S, A, C\} & S \rightarrow aAbc \mid \lambda \\ \Sigma &= \{a, b, c\} & A \rightarrow aAbC \mid \lambda \\ & & Cb \rightarrow bC \\ & & Cc \rightarrow cc \end{aligned}$$

with start symbol S generates the language $\{a^i b^i c^i \mid i \geq 0\}$. The string $a^i b^i c^i$, $i > 0$, is generated by a derivation that begins

$$\begin{aligned} S &\Longrightarrow aAbc \\ &\xrightarrow{i-1} a^i A(bC)^{i-1} bc \\ &\Longrightarrow a^i (bC)^{i-1} bc, \end{aligned}$$

using the rule $A \rightarrow aABC$ to generate the i leading a 's. The rule $Cb \rightarrow bC$ allows the final C to pass through the b 's that separate it from the c 's at the end of the string. Upon reaching the leftmost c , the variable C is replaced with c . This process is repeated until each occurrence of the variable C is moved to the right of all the b 's and transformed into a c . \square

Example 10.1.2

The unrestricted grammar with terminal alphabet $\{a, b, [,]\}$ defined by the productions

$$\begin{aligned} S &\rightarrow aT[a] \mid bT[b] \mid [] \\ T[&\rightarrow aT[A \mid bT[B \mid [\\ Aa &\rightarrow aA \\ Ab &\rightarrow bA \\ Ba &\rightarrow aB \\ Bb &\rightarrow bB \\ A] &\rightarrow a] \\ B] &\rightarrow b] \end{aligned}$$

generates the language $\{u[u] \mid u \in \{a, b\}^*\}$.

The addition of an a or b to the left of the variable T is accompanied by the generation of the variable A or B after $T[$. Using the rules that interchange the position of a variable and a terminal, the derivation progresses by passing the variable through the copy of the string enclosed in the brackets. When the variable is adjacent to the symbol $]$, the appropriate terminal is added to the second string. The entire process may be repeated to generate

additional terminal symbols or be terminated by the application of the rule $T[\rightarrow]$. The derivation

$S \Rightarrow aT[a]$
 $\Rightarrow aaT[Aa]$
 $\Rightarrow aaT[aA]$
 $\Rightarrow aaT[aa]$
 $\Rightarrow aabT[Baa]$
 $\Rightarrow aabT[aBa]$
 $\Rightarrow aabT[aaB]$
 $\Rightarrow aabT[aab]$
 $\Rightarrow aab[aab]$

exhibits the roles of the variables in a derivation

In the grammars in the two preceding examples, the left-hand side of each rule contained a variable. This is not required by the definition of unrestricted grammar. However, the imposition of the restriction that the left-hand side of a rule contain a variable does not reduce the set of languages that can be generated (Exercise 3).

Throughout our study of formal languages, we have demonstrated a correspondence between the generation of a language by a grammar and its acceptance by a finite-state machine. Regular languages are accepted by finite automata and context-free languages by pushdown automata. Unrestricted grammars provide the most flexible type of string transformation; there are no conditions on the matching substring, nor on the replacement. It would seem reasonable that generation by an unrestricted grammar corresponds to acceptance by the most powerful type of abstract machine. This is indeed the case. The next two theorems show that a language is generated by an unrestricted grammar if, and only if, it is accepted by a Turing machine.

Theorem 10.1.2

Let $G = (V, \Sigma, P, S)$ be an unrestricted grammar. Then $L(G)$ is a recursively enumerable language.

Proof. We will sketch the design of a three-tape nondeterministic Turing machine M that accepts $L(G)$. We will design M so that its computations simulate derivations of the grammar G . Tape 1 holds an input string p from Σ^* . A representation of the rules of G is written on tape 2. A rule $u \rightarrow v$ is represented by the string $u\#v$, where $\#$ is a tape symbol reserved for this purpose. Rules are separated by two consecutive $\#$'s. The derivations of G are simulated on tape 3.

A computation of the machine M that accepts $L(G)$ consists of the following actions:

1. S is written on position one of tape 3.
 2. The rules of G are written on tape 2.

3. A rule $u \# v$ is chosen.
4. An instance of the problem halts in a rejection state.
5. The string u is removed.
6. If the strings on the stack are empty, the computation halts in a rejection state.
7. The computation continues.

Since the length of u require shifting the p

For any string p , derivation will be exact and M will accept p , derivable from S in G . $L(M) = L(G)$.

Example 10.1.3

The language $L = \{a\}$

Computations of the
of the grammar are re-

P. S-

The rule $S \rightarrow \lambda$ is redundant, as it is derived from the rule $S \rightarrow S_1 S_2$. The case, is followed by the rule $S \rightarrow S_1 S_2$.

Theorem 10.1.3

Let L be a recursively

Proof. Since L is recursive, $M = (Q, \Sigma, \Gamma, \delta, q_0)$ derivations simulate the configuration as a string on the configuration tape $uyq;vB$.

$[\rightarrow [$. The

3. A rule $u\#v$ is chosen from tape 2.
4. An instance of the string u is chosen on tape 3, if one exists. Otherwise, the computation halts in a rejecting state.
5. The string u is replaced by v on tape 3.
6. If the strings on tape 3 and tape 1 match, the computation halts in an accepting state.
7. The computation continues with step 3 to simulate another rule application.

Since the length of u and v may differ, the simulation of a rule application $xuy \Rightarrow xvy$ may require shifting the position of the string y .

For any string $p \in L(G)$, there is a sequence of rule applications that derives p . This derivation will be examined by one of the nondeterministic computations of the machine M , and M will accept p . Conversely, the actions of M on tape 3 generate precisely the strings derivable from S in G . The only strings accepted by M are terminal strings in $L(G)$. Thus, $L(M) = L(G)$. ■

□

le contained
owever, the
le does not

espondence
finite-state
e languages
pe of string
lacement. It
o acceptance
he next two
and only if,

enumerable

chine M that
the grammar
is written on
l reserved for
are simulated

ving actions:

Example 10.1.3

The language $L = \{a^i b^i c^i \mid i \geq 0\}$ is generated by the rules

$$\begin{aligned} S &\rightarrow aAbc \mid \lambda \\ A &\rightarrow aAbC \mid \lambda \\ Cb &\rightarrow bc \\ Cc &\rightarrow cc. \end{aligned}$$

Computations of the machine that accepts L simulate derivations of the grammar. The rules of the grammar are represented on tape 2 by

$$BS\#aAbc\#\#S\#\#A\#aAbC\#\#A\#\#Cb\#bC\#\#Cc\#ccB.$$

The rule $S \rightarrow \lambda$ is represented by the string $S\#\#$. The first # separates the left-hand side of the rule from the right-hand side. The right-hand side of the rule, the null string in this case, is followed by the string $\#\#$. □

Theorem 10.1.3

Let L be a recursively enumerable language. Then there is an unrestricted grammar G with $L(G) = L$.

Proof. Since L is recursively enumerable, it is accepted by a deterministic Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$. An unrestricted grammar $G = (V, \Sigma, P, S)$ is designed whose derivations simulate the computations of M . Using the representation of a Turing machine configuration as a string, the effect of a Turing machine transition $\delta(q_i, x) = [q_j, y, R]$ on the configuration $uq_i x v B$ can be represented by the string transformation $uq_i x v B \Rightarrow uyq_j v B$.

The derivation of a terminal string in G consists of three distinct subderivations:

- the generation of a string $u[q_0Bu]$ where $u \in \Sigma^*$,
- the simulation of a computation of M on the string $[q_0Bu]$, and
- if M accepts u , the removal of the simulation substring.

The grammar G contains a variable A_i for each terminal symbol $a_i \in \Sigma$. These variables, along with S , T , $[$, and $]$, are used in the generation of the string $u[q_0Bu]$. The simulation of a computation uses variables corresponding to the states of M. The variables E_R and E_L are used in the third phase of a derivation. The terminal symbols of the grammar are the elements of the input alphabet of M. Thus the alphabets of G are

$$\Sigma = \{a_1, a_2, \dots, a_n\}$$

$$V = \{S, T, E_R, E_L, [,], A_1, A_2, \dots, A_n\} \cup Q.$$

The rules for each of the three parts of a derivation are given separately. A derivation begins by generating $u[q_0Bu]$, where u is an arbitrary string in Σ^* . The strategy used for generating strings of this form was presented in Example 10.1.2.

- $S \rightarrow a_i T[a_i] | [q_0B]$ for $1 \leq i \leq n$
- $T \rightarrow a_i T[A_i] | [q_0B]$ for $i \leq i \leq n$
- $A_i a_j \rightarrow a_j A_i$ for $1 \leq i, j \leq n$
- $A_i] \rightarrow a_i]$ for $1 \leq i \leq n$

The computation of the Turing machine with input u is simulated on the string $[q_0Bu]$. The rules are obtained by rewriting the transitions of M as string transformations.

- $q_i xy \rightarrow z q_j y$ whenever $\delta(q_i, x) = [q_j, z, R]$ and $y \in \Gamma$
- $q_i x] \rightarrow z q_j B]$ whenever $\delta(q_i, x) = [q_j, z, R]$
- $y q_i x \rightarrow q_j y z$ whenever $\delta(q_i, x) = [q_j, z, L]$ and $y \in \Gamma$

Example 10.1.4

The construction of is demonstrated usin

that accepts $a^*b(a \cup b)^*$. The variables an

The rules are given in
Input-generating rules:

Simulation rules:

If the computation of M halts in an accepting state, the derivation erases the string within the brackets. The variable E_R erases the string to the right of the halting position of the tape head. Upon reaching the endmarker $]$, the variable E_L (erase left) is generated.

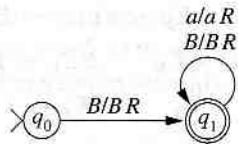
- $q_i x \rightarrow E_R$ whenever $\delta(q_i, x)$ is undefined and $q_i \in F$
- $E_R x \rightarrow E_R$ for $x \in \Gamma$
- $E_R] \rightarrow E_L$
- $x E_L \rightarrow E_L$ for $x \in \Gamma$
- $[E_L \rightarrow \lambda$

The derivation that begins by generating $u[q_0Bu]$ terminates with u whenever $u \in L(M)$. If $u \notin L(M)$, the brackets enclosing the simulation of the computation are never erased and the derivation does not produce a terminal string. ■

ons:

Example 10.1.4

The construction of a grammar that generates the language accepted by a Turing machine is demonstrated using the machine M



that accepts $a^*b(a \cup b)^*$. When the first b is encountered, M halts and accepts in state q_1 .

The variables and terminals of G are

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

$$V = \{S, T, E_R, E_L, [], A, X\} \cup \{q_0, q_1\}.$$

The rules are given in three sets.

Input-generating rules:

$$\begin{aligned}
 S &\rightarrow aT[a] \mid bT[b] \mid [q_0B] \\
 T &\rightarrow aT[A \mid bT[X \mid [q_0B \\
 Aa &\rightarrow aA \\
 Ab &\rightarrow bA \\
 A] &\rightarrow a] \\
 Xa &\rightarrow aX \\
 Xb &\rightarrow bX \\
 X] &\rightarrow b]
 \end{aligned}$$

Simulation rules:

Transition	Rules
$\delta(q_0, B) = [q_1, B, R]$	$q_0Ba \rightarrow Bq_1a$
	$q_0Bb \rightarrow Bq_1b$
	$q_0BB \rightarrow Bq_1B$
	$q_0B] \rightarrow Bq_1B]$
$\delta(q_1, a) = [q_1, a, R]$	$q_1aa \rightarrow aq_1a$
	$q_1ab \rightarrow aq_1b$
	$q_1aB \rightarrow aq_1B$
	$q_1a] \rightarrow aq_1B]$
$\delta(q_1, B) = [q_1, B, R]$	$q_1Ba \rightarrow Bq_1a$
	$q_1Bb \rightarrow Bq_1b$
	$q_1BB \rightarrow Bq_1B$
	$q_1B] \rightarrow Bq_1B]$

These variables are used in the grammar. The variables Bu and Bv are used in the grammar. The variables Bu and Bv are used in the grammar.

derivation string used for

ing $[q_0Bu]$.
ns.

s the string
position of
generated.

$u \in L(M)$. If
erased and

Erasure rules:

$$\begin{array}{ll}
 q_1 b \rightarrow E_R & \\
 E_R a \rightarrow E_R & a E_L \rightarrow E_L \\
 E_R b \rightarrow E_R & b E_L \rightarrow E_L \\
 E_R B \rightarrow E_R & B E_L \rightarrow E_L \\
 E_R] \rightarrow E_L & [E_L \rightarrow \lambda
 \end{array}$$

The computation that accepts the string ab in M and the corresponding derivation in the grammar G that accepts ab are

$$\begin{array}{ll}
 q_0 BabB & S \Rightarrow aT[a] \\
 \vdash Bq_1 abB & \Rightarrow abT[Xa] \\
 \vdash Baq_1 bB & \Rightarrow ab[q_0 BXa] \\
 & \Rightarrow ab[q_0 BaX] \\
 & \Rightarrow ab[q_0 Bab] \\
 & \Rightarrow ab[Bq_1 ab] \\
 & \Rightarrow ab[Baq_1 b] \\
 & \Rightarrow ab[BaE_R] \\
 & \Rightarrow ab[BaE_L] \\
 & \Rightarrow ab[BE_L] \\
 & \Rightarrow ab[E_L] \\
 & \Rightarrow ab. \quad \square
 \end{array}$$

Properties of unrestricted grammars can be used to establish closure results for recursively enumerable languages. The proofs, similar to those presented in Theorem 7.5.1 for context-free languages, are left as exercises.

Theorem 10.1.4

The set of recursively enumerable languages is closed under union, concatenation, and Kleene star.

10.2 Context-Sensitive Grammars

The context-sensitive grammars represent an intermediate step between the context-free and the unrestricted grammars. No restrictions are placed on the left-hand side of a production, but the length of the right-hand side is required to be at least that of the left.

Definition 10.2.1

A phrase-structure grammar is said to be context-sensitive if the form $u \rightarrow v$

A rule that increases the application of a nonterminal increases. The λ rule is called a context-sensitive rule.

Context-sensitive grammars, which each rule of the form $(V \cup \Sigma)^*$. The nonterminal appears in the context in this manner and a context-sensitive rule can be generated.

The monotonicity of context-sensitivity guarantees that we obtain the unique

that generates the same language as the original context-sensitive grammar.

produces an equivalent language.

A nondeterministic finite state machine designed to accept a language permits the length of the string in the derivation. When it reaches a final state, it halts and rejects the string.

Theorem 10.2.2

Every context-sensitive grammar is a context-sensitive grammar.

Proof. Following the proof of Theorem 10.1.4, we show that every context-sensitive grammar is a context-sensitive grammar.

Definition 10.2.1

A phrase-structure grammar $G = (V, \Sigma, P, S)$ is called **context-sensitive** if each rule has the form $u \rightarrow v$, where $u \in (V \cup \Sigma)^+$, $v \in (V \cup \Sigma)^+$, and $\text{length}(u) \leq \text{length}(v)$.

A rule that satisfies the conditions of Definition 10.2.1 is called *monotonic*. With each application of a monotonic rule, the length of the derived string either remains the same or increases. The language generated by a context-sensitive grammar is called, not surprisingly, a context-sensitive language.

Context-sensitive grammars were originally defined as phrase-structure grammars in which each rule has the form $uAv \rightarrow uwv$, where $A \in V$, $w \in (V \cup \Sigma)^+$, and $u, v \in (V \cup \Sigma)^*$. The preceding rule indicates that the variable A can be replaced by w only when it appears in the context of being preceded by u and followed by v . Clearly, every rule defined in this manner is monotonic. On the other hand, a transformation defined by a monotonic rule can be generated by a set of rules of the form $uAv \rightarrow uwv$ (Exercises 10 and 11).

The monotonic property of the rules guarantees that the null string is not an element of a context-sensitive language. Removing the rule $S \rightarrow \lambda$ from the grammar in Example 10.1.1, we obtain the unrestricted grammar

$$\begin{aligned} S &\rightarrow aAbc \\ A &\rightarrow aAbC \mid \lambda \\ Cb &\rightarrow bC \\ Cc &\rightarrow cc \end{aligned}$$

that generates the language $\{a^i b^i c^i \mid i > 0\}$. The λ -rule violates the monotonicity property of context-sensitive rules. Replacing the S and A rules with

$$\begin{aligned} S &\rightarrow aAbc \mid abc \\ A &\rightarrow aAbC \mid abC \end{aligned}$$

produces an equivalent context-sensitive grammar.

A nondeterministic Turing machine, similar to the machine in Theorem 10.1.2, is designed to accept a context-sensitive language. The noncontracting nature of the rules permits the length of the input string to be used to terminate the simulation of an unsuccessful derivation. When the length of the derived string surpasses that of the input, the computation halts and rejects the string.

Theorem 10.2.2

Every context-sensitive language is recursive.

Proof. Following the approach developed in Theorem 10.1.2, derivations of the context-sensitive grammar are simulated on a three-tape nondeterministic Turing machine M . The entire derivation, rather than just the result, is recorded on tape 3. When a rule $u \rightarrow v$ is

applied to the string xuy on tape 3, the string xvy is written on the tape following $xuy\#$. The symbol $\#$ is used to separate the derived strings.

A computation of M with input string p performs the following sequence of actions:

1. $S\#$ is written beginning at position one of tape 3.
2. The rules of G are written on tape 2.
3. A rule $u\#v$ is chosen from tape 2.
4. Let $q\#$ be the most recent string written on tape 3:
 - a) An instance of the string u in q is chosen, if one exists. In this case, q can be written xuy .
 - b) Otherwise, the computation halts in a nonaccepting state.
5. $xvy\#$ is written on tape 3 immediately following $q\#$.
6. a) If $xvy = p$, the computation halts in an accepting state.
b) If xvy occurs at another position on tape 3, the computation halts in a nonaccepting state.
c) If $\text{length}(xvy) > \text{length}(p)$, the computation halts in a nonaccepting state.
7. The computation continues with step 3 to simulate another rule application.

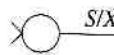
There are only a finite number of strings in $(V \cup \Sigma)^*$ with length less than or equal to $\text{length}(p)$. This implies that every derivation eventually halts, enters a cycle, or derives a string of length greater than $\text{length}(p)$. A computation halts at step 4 when the rule that has been selected cannot be applied to the current string. Cyclic derivations, $S \xrightarrow{*} w \xrightarrow{*} w$, are terminated in step 6(b). The length bound is used in step 6(c) to terminate all other unsuccessful derivations.

Every string in $L(G)$ is generated by a noncyclic derivation. The simulation of such a derivation causes M to accept the string. Since every computation of M halts, $L(G)$ is recursive (Exercise 8.23). ■

The initial positions. The e input. A comput may be read by a move to the r accepted by an I

We will sh automaton. An grammar. The T grammar begins the amount of tap of the LBA are u

The diagram rule $Sa \rightarrow aAS$ $uaASv$. The first at the position of traversed to dete to the computati shifted one posit



10.3 Linear-Bounded Automata

We have examined several modifications of the standard Turing machine that do not alter the set of languages accepted by the machines. Restricting the amount of the tape decreases the capabilities of a Turing machine computation. A linear-bounded automaton is a Turing machine in which the amount of available tape is determined by the length of the input string. The input alphabet contains two symbols, \langle and \rangle , that designate the left and right boundaries of the tape.

Definition 10.3.1

A **linear-bounded automaton** (LBA) is a structure $M = (Q, \Sigma, \Gamma, \delta, q_0, \langle, \rangle, F)$, where $Q, \Sigma, \Gamma, \delta, q_0$, and F are the same as for a nondeterministic Turing machine. The symbols \langle and \rangle are distinguished elements of Σ .

ing $xuy\#$.

f actions:

en A

reaching

accepting

be written

as rules

the amount

available

to an LBA

is limited

by the tape

length.

ate.

1.

an or equal

; or derives

the rule that

$\Rightarrow w \stackrel{+}{\Rightarrow} w$,

ate all other

tion of such

alts, $L(G)$ is

■

do not alter

pe decreases

n is a Turing

of the input

left and right

accepting

state.

do not alter

pe decreases

n is a Turing

of the input

left and right

\rangle , F), where
The symbols

The initial configuration of a computation is $q_0\langle w \rangle$, requiring $\text{length}(w) + 2$ tape positions. The endmarkers \langle and \rangle are written on the tape but not considered part of the input. A computation remains within the boundaries specified by \langle and \rangle . The endmarkers may be read by the machine but cannot be erased. Transitions scanning \langle must designate a move to the right and those reading \rangle a move to the left. A string $w \in (\Sigma - \{\langle, \rangle\})^*$ is accepted by an LBA if a computation with input $\langle w \rangle$ halts in an accepting state.

We will show that every context-sensitive language is accepted by a linear-bounded automaton. An LBA is constructed to simulate the derivations of the context-sensitive grammar. The Turing machine constructed to simulate the derivations of an unrestricted grammar begins by writing the rules of the grammar on one of the tapes. The restriction on the amount of tape available to an LBA prohibits this approach. Instead, states and transitions of the LBA are used to encode the rules.

The diagram in Figure 10.1 shows how transitions can simulate the application of the rule $Sa \rightarrow aAs$. The application of the rule generates a string transformation $uSav \Rightarrow uaAsv$. The first two transitions in the diagram verify that the string on the tape beginning at the position of the tape head matches Sa . Before Sa is replaced with aAs , the string v is traversed to determine whether the derived string fits on the segment of the tape available to the computation. If the \rangle is read, the computation terminates. Otherwise, the string v is shifted one position to the right and Sa is replaced by aAs .

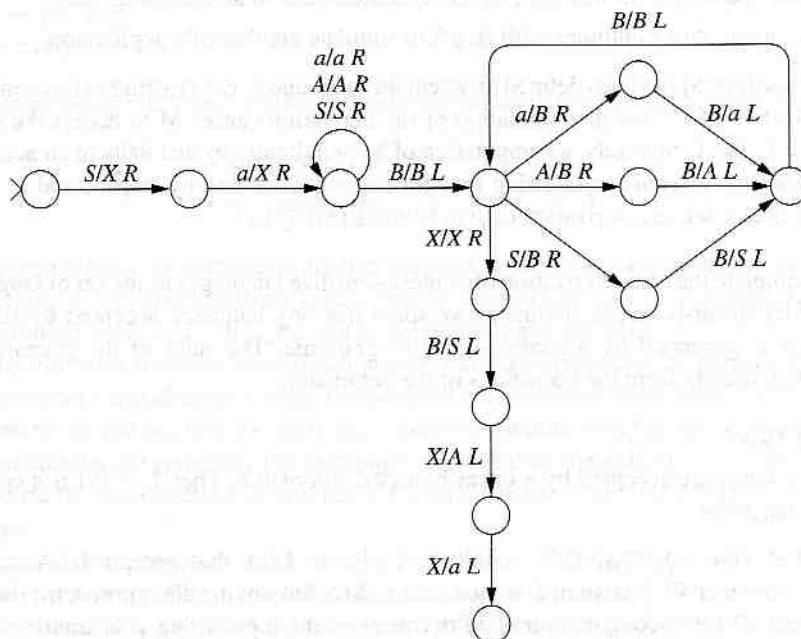


FIGURE 10.1 LBA simulation of application of $Sa \rightarrow aAs$.

Theorem 10.3.2

Let L be a context-sensitive language. Then there is a linear-bounded automaton M with $L(M) = L$.

Proof. Since L is a context-sensitive language, $L = L(G)$ for some context-sensitive grammar $G = (V, \Sigma, P, S)$. An LBA M with a two-track tape is constructed to simulate the derivations of G . The first track contains the input, including the endmarkers. The second track holds the string generated by the simulated derivation.

Each rule of G is encoded in a submachine of M . A computation of M with input $\langle p \rangle$ consists of the following sequence of actions:

1. S is written on track 2 in position one.
2. The tape head is moved into a position in which it scans a symbol of the string on track 2.
3. A rule $u \rightarrow v$ is nondeterministically selected, and the computation attempts to apply the rule.
 4. a) If a substring on track 2 beginning at the position of the tape head does not match u , the computation halts in a nonaccepting state.
 - b) If the tape head is scanning u but the string obtained by replacing u by v is greater than $length(p)$, then the computation halts in a nonaccepting state.
 - c) Otherwise, u is replaced by v on track 2.
5. If track 2 contains the string p , the computation halts in an accepting state.
6. The computation continues with step 2 to simulate another rule application.

The machine M has been defined to accept the language L . Every string in L is generated by a derivation of G , and the simulation of the derivation causes M to accept the string. Thus, $L \subseteq L(M)$. Conversely, a computation of M with input $\langle p \rangle$ that halts in an accepting state consists of a sequence of string transformations generated by steps 2 and 3. These transformations define a derivation of p in G and $L(M) \subseteq L$. ■

To complete the characterization of context-sensitive languages as the set of languages accepted by linear-bounded automata, we show that any language accepted by such an automaton is generated by a context-sensitive grammar. The rules of the grammar are constructed directly from the transitions of the automaton.

Theorem 10.3.3

Let L be a language accepted by a linear-bounded automaton. Then $L - \{\lambda\}$ is a context-sensitive language.

Proof. Let $M = (Q, \Sigma_M, \Gamma, \delta, q_0, \langle \cdot, \cdot \rangle, F)$ be an LBA that accepts L . A context-sensitive grammar G is designed to generate $L(M)$. Employing the approach presented in Theorem 10.1.3, a computation of M that accepts the input string p is simulated by a derivation of p in G . The techniques used to construct an unrestricted grammar that simulates

a Turing machine do not satisfy the to erase symbols a derived string objects as variables

The terminal endmarkers. Orde a terminal symbol possibly a state an

$$\Sigma_G = \Sigma_M$$

$$V = \{S, A\}$$

$$[a_i]$$

where $a_i \in \Sigma_G$, $x \in V$

The S and A and the initial con-

1. $S \rightarrow [a_i, q_0 \langle a_i \rangle] \rightarrow [a_i, q_0 \langle a_i \rangle] \dots \rightarrow [a_i, q_0 \langle a_i \rangle] \dots$ for every $a_i \in \Sigma_M$
2. $A \rightarrow [a_i, a_i] \rightarrow [a_i, a_i] \dots \rightarrow [a_i, a_i] \dots$ for every $a_i \in \Sigma_M$

Derivations using

The string obtain pairs, $a_{i_1}a_{i_2} \dots a_{i_n}$ nents produce $q_0 \langle a_{i_1}a_{i_2} \dots a_{i_n} \rangle$

The rules that transformations that components do not not simulated by the to produce the rule exercise.

Upon the com original input string accepting state in the symbol contained i

a Turing machine computation cannot be employed since the rules that erase the simulation do not satisfy the monotonicity restrictions of a context-sensitive grammar. The inability to erase symbols in the derivation of a context-sensitive grammar restricts the length of a derived string to that of the input. The simulation is accomplished by using composite objects as variables.

The terminal alphabet of G is obtained from the input alphabet of M by deleting the endmarkers. Ordered pairs are used as variables. The first component of an ordered pair is a terminal symbol. The second is a string consisting of a combination of a tape symbol and possibly a state and endmarker(s).

$$\Sigma_G = \Sigma_M - \{(\), \}) = \{a_1, a_2, \dots, a_n\}$$

$$V = \{S, A, [a_i, x], [a_i, \langle x \rangle], [a_i, x)], [a_i, \langle x \rangle], [a_i, q_k x], [a_i, q_k \langle x \rangle], [a_i, \langle q_k x \rangle], [a_i, q_k x)], [a_i, q_k \langle x \rangle], [a_i, \langle q_k x \rangle], [a_i, \langle x q_k \rangle]\},$$

where $a_i \in \Sigma_G$, $x \in \Gamma$, and $q_k \in Q$.

The S and A rules generate ordered pairs whose components represent the input string and the initial configuration of a computation of M .

1. $S \rightarrow [a_i, q_0 \langle a_i \rangle]A$
 $\quad \rightarrow [a_i, q_0 \langle a_i \rangle]$
for every $a_i \in \Sigma_G$
2. $A \rightarrow [a_i, a_i]A$
 $\quad \rightarrow [a_i, a_i])$
for every $a_i \in \Sigma_G$

Derivations using the S and A rules generate sequences of ordered pairs of the form

$[a_i, q_0 \langle a_i \rangle]$, or

$[a_{i_1}, q_0 \langle a_{i_1} \rangle] [a_{i_2}, a_{i_2}] \dots [a_{i_n}, a_{i_n}]$.

The string obtained by concatenating the elements in the first components of the ordered pairs, $a_{i_1}a_{i_2}\dots a_{i_n}$, represents the input string to a computation of M . The second components produce $q_0 \langle a_{i_1}a_{i_2}\dots a_{i_n} \rangle$, the initial configuration of the LBA.

The rules that simulate a computation are obtained by rewriting the transitions of M as transformations that alter the second components of the ordered pairs. Note that the second components do not produce the string $q_0 \langle \rangle$; the computation with the null string as input is not simulated by the grammar. The techniques presented in Theorem 10.1.3 can be modified to produce the rules needed to simulate the computations of M . The details are left as an exercise.

Upon the completion of a successful computation, the derivation must generate the original input string. When an accepting configuration is generated, the variable with the accepting state in the second component of the ordered pair is transformed into the terminal symbol contained in the first component.

3. $[a_i, q_k \langle x \rangle] \rightarrow a_i$
 $[a_i, q_k \langle x \rangle] \rightarrow a_i$
whenever $\delta(q_k, \langle \rangle) = \emptyset$ and $q_k \in F$
- $[a_i, xq_k] \rightarrow a_i$
 $[a_i, \langle xq_k \rangle] \rightarrow a_i$
whenever $\delta(q_k, \rangle) = \emptyset$ and $q_k \in F$
- $[a_i, q_k x] \rightarrow a_i$
 $[a_i, q_k x] \rightarrow a_i$
 $[a_i, \langle q_k x \rangle] \rightarrow a_i$
 $[a_i, \langle q_k x \rangle] \rightarrow a_i$
whenever $\delta(q_k, x) = \emptyset$ and $q_k \in F$

The derivation is completed by transforming the remaining variables to the terminal contained in the first component.

4. $[a_i, u]a_j \rightarrow a_i a_j$
 $a_j [a_i, u] \rightarrow a_j a_i$
for every $a_j \in \Sigma_G$ and $[a_i, u] \in V$

10.4 The Chomsky Hierarchy

Chomsky numbered the four families of grammars (and languages) that make up the hierarchy. Unrestricted, context-sensitive, context-free, and regular grammars are referred to as type 0, type 1, type 2, and type 3 grammars, respectively. The restrictions placed on the rules increase with the number of the grammar. The nesting of the families of grammars of the Chomsky hierarchy induces a nesting of the corresponding languages. Every context-free language containing the null string is generated by a context-free grammar in which $S \rightarrow \lambda$ is the only λ -rule (Theorem 4.2.3). Removing this single λ -rule produces a context-sensitive grammar that generates $L - \{\lambda\}$. Thus, the language $L - \{\lambda\}$ is context-sensitive whenever L is context-free. Ignoring the complications presented by the null string in context-sensitive languages, every type i language is also type $(i - 1)$.

The preceding inclusions are proper. The set $\{a^i b^i \mid i \geq 0\}$ is context-free but not regular (Theorem 6.5.1). Similarly, $\{a^i b^i c^i \mid i > 0\}$ is context-sensitive but not context-free (Example 7.4.1). In Chapter 11, the language of the Halting Problem is shown to be recursively enumerable but not recursive. Combining this result with Theorem 10.2.2 establishes the proper inclusion of context-sensitive languages in the set of recursively enumerable languages.

Each class of languages in the Chomsky hierarchy has been characterized as the languages generated by a family of grammars and accepted by a type of machine. The relationships developed between generation and recognition are summarized in the following table.

Grammars
Type 0 grammars
phrase-structure grammars
unrestricted grammars
Type 1 grammars
context-sensitive grammars
Type 2 grammars
context-free grammars
Type 3 grammars
regular grammars
left-linear grammars
right-linear grammars

Exercises

- Design unrestricted grammars for the following languages:
 - $\{a^i b^j a^i b^j \mid i, j \geq 0\}$
 - $\{a^i b^i c^i d^i \mid i > 0\}$
 - $\{www \mid w \in \{a, b, c\}^*\}$
- Prove that each of the following languages is context-free:
 - $\{a^i b^j \mid i \geq 0, j \geq 0, i \neq j\}$
 - $\{a^i b^i c^i \mid i > 0\}$
 - $\{a^i b^i c^i d^i \mid i > 0\}$
 - $\{a^i b^i c^i d^i e^i \mid i > 0\}$
- Prove that each rule has the form $A \rightarrow a$, where A is a non-terminal symbol and a is a terminal symbol.
- Prove that the following languages are context-sensitive:
 - union of two context-free languages
 - intersection of two context-free languages
 - concatenation of two context-free languages
 - Kleene star of a context-free language
 - homomorphism of a context-free language

Grammars	Languages	Accepting Machines
Type 0 grammars, phrase-structure grammars, unrestricted grammars	Recursively enumerable	Turing machine, nondeterministic Turing machine
Type 1 grammars, context-sensitive grammars,	Context-sensitive	Linear-bounded automata
Type 2 grammars, context-free grammars	Context-free	Pushdown automata
Type 3 grammars, regular grammars, left-linear grammars, right-linear grammars	Regular	Deterministic finite automata, nondeterministic finite automata

terminal

Exercises

1. Design unrestricted grammars to generate the following languages:

- a) $\{a^i b^j a^i b^j \mid i, j \geq 0\}$
- b) $\{a^i b^i c^i d^i \mid i \geq 0\}$
- c) $\{www \mid w \in \{a, b\}^*\}$

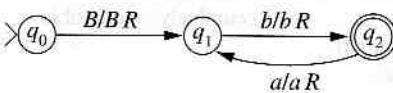
2. Prove that every terminal string generated by the grammar

$$\begin{aligned} S &\rightarrow aAbc \mid \lambda \\ A &\rightarrow aAbC \mid \lambda \\ Cb &\rightarrow bC \\ Cc &\rightarrow cc \end{aligned}$$

has the form $a^i b^j c^j$ for some $i \geq 0$.

- * 3. Prove that every recursively enumerable language is generated by a grammar in which each rule has the form $u \rightarrow v$ where $u \in V^+$ and $v \in (V \cup \Sigma)^*$.
- 4. Prove that the recursively enumerable languages are closed under the following operations:
 - a) union
 - b) intersection
 - c) concatenation
 - d) Kleene star
 - e) homomorphic images

5. Let M be the Turing machine



- a) Give a regular expression for $L(M)$.
- b) Using the techniques from Theorem 10.1.3, give the rules of an unrestricted grammar G that accepts $L(M)$.
- c) Trace the computation of M when run with input bab and give the corresponding derivation in G .

6. Let G be the context-sensitive grammar

$$\begin{aligned} G: \quad S &\rightarrow SBA \mid a \\ BA &\rightarrow AB \\ aA &\rightarrow aaB \\ B &\rightarrow b. \end{aligned}$$

- a) Give a derivation of $aabb$.
 - b) What is $L(G)$?
 - c) Construct a context-free grammar that generates $L(G)$.
7. Let L be the language $\{a^i b^{2i} a^i \mid i > 0\}$.
- a) Use the pumping lemma for context-free languages to show that L is not context-free.
 - b) Construct a context-sensitive grammar G that generates L .
 - c) Give the derivation of $aabbbbaa$ in G .
 - d) Construct an LBA M that accepts L .
 - e) Trace the computation of M with input $aabbbbaa$.

* 8. Let $L = \{a^i b^j c^k \mid 0 < i \leq j \leq k\}$.

- a) L is not context-free. Can this be proved using the pumping lemma for context-free languages? If so, do so. If not, show that the pumping lemma is incapable of establishing that L is not context-free.
- b) Give a context-sensitive grammar that generates L .

9. Let M be an LBA with alphabet Σ . Outline a general approach to construct monotonic rules that simulate the computation of M . The rules of the grammar should consist of variables in the set

$$\{[a_i, x], [a_i, \langle x \rangle], [a_i, x], [a_i, \langle x \rangle], [a_i, \langle x \rangle], [a_i, q_k x], [a_i, q_k \langle x \rangle], [a_i, \langle q_k x \rangle], [a_i, q_k x], [a_i, x q_k], [a_i, q_k \langle x \rangle], [a_i, \langle q_k x \rangle], [a_i, \langle x q_k \rangle]\},$$

where $a_i \in \Sigma$, $x \in \Gamma$, and $q_i \in Q$. This completes the construction of the grammar in Theorem 10.3.3.

- * 10. Let $u \rightarrow v$ be a right-hand side.
- 11. Construct a sequence of transformations each of whose steps is generated by $w \in (V \cup \Sigma)^*$.
- 12. Use the results of Exercise 11 to prove that $w \in (V \cup \Sigma)^*$ is generated by a grammar.
- * 13. Let T be a full binary tree. Consider the language $L(T)$ consisting of all strings $(R), \text{ or up } (U)$ that are rooted at the root of T . For example, if T is the Chomsky hierarchy tree, then $L(T)$ consists of all strings of the form $R_1 R_2 \dots R_n$ where R_i is a rule of the Chomsky hierarchy tree.
- 14. Prove that the class of context-sensitive grammars is closed under homomorphisms. A homomorphism h is a function from a grammar G to another grammar G' such that $h(S) \in V^*$ and $h(A) \in V^*$ for all $S \in V$ and $A \in T$.
- * 15. Let L be a recursive language and Σ a finite alphabet. Show that there exists a context-free grammar G such that $L = L(G)$ and $w \in \Sigma^*, w \in L$.
- 16. Prove that every context-free language is context-sensitive.
- 17. A grammar is said to be linear if it satisfies the condition $uAv \rightarrow uvw$, where $u, v, w \in \Sigma^*$ and $uAv \in L$. Prove that every context-free language is linear.
- 18. A linear-bound automaton (LBA) is a nondeterministic finite automaton with a linear bounded tape. Define a linear-bound automaton and prove that it accepts exactly the context-free languages. Hint: consider the number of configurations of each state and tape cell.
- 19. Let L be a context-free language. Let \bar{L} be the complement of L . Prove that \bar{L} is context-free if and only if L is context-free. Hint: use the fact that LBA need not halt.

Bibliographic Notes

The Chomsky hierarchy was first proposed by Chomsky [1956]. The proof that the unrestricted grammar generates the same language as the Turing machine was given by Post [1947]. The proof that the unrestricted grammar generates the same language as the linear-bounded automaton was given by Gold [1967]. The proof that the linear-bounded automaton accepts exactly the context-free languages was given by Ogden and Kuroda [1964]. The proof that the context-free grammar generates the same language as the linear-bounded automaton was given by Arbib [1968]. The proof that the context-free grammar generates the same language as the Turing machine was given by Cook [1971].

- * 10. Let $u \rightarrow v$ be a monotonic rule. Construct a sequence of monotonic rules, each of whose right-hand side has length two or less, that defines the same transformation as $u \rightarrow v$.
- 11. Construct a sequence of context-sensitive rules $uAv \rightarrow uwv$ that define the same transformation as the monotonic rule $AB \rightarrow CD$. Hint: A sequence of three rules, each of whose left-hand side and right-hand side is of length two, suffices.
- 12. Use the results from Exercises 10 and 11 to prove that every context-sensitive language is generated by a grammar in which each rule has the form $uAv \rightarrow uwv$, where $w \in (V \cup \Sigma)^+$ and $u, v \in (V \cup \Sigma)^*$.
- * 13. Let T be a full binary tree. A path through T is a sequence of left-down (L), right-down (R), or up (U) moves. Thus paths may be identified with strings over $\Sigma = \{L, R, U\}$. Consider the language $L = \{w \in \Sigma^* \mid w \text{ describes a path from the root back to the root}\}$. For example, $\lambda, LU, LRUULU \in L$, and $U, LRU \notin L$. Establish L 's place in the Chomsky hierarchy.
- 14. Prove that the context-sensitive languages are not closed under arbitrary homomorphisms. A homomorphism is λ -free if $h(u) = \lambda$ implies $u = \lambda$. Prove that the context-sensitive grammars are closed under λ -free homomorphisms.
- * 15. Let L be a recursively enumerable language over Σ and c a terminal symbol not in Σ . Show that there is a context-sensitive language L' over $\Sigma \cup \{c\}$ such that for every $w \in \Sigma^*$, $w \in L$ if, and only if, $wc^i \in L'$ for some $i \geq 0$.
- 16. Prove that every recursively enumerable language is the homomorphic image of a context-sensitive language. Hint: Use Exercise 15.
- 17. A grammar is said to be context-sensitive with erasing if every rule has the form $uAv \rightarrow uvw$, where $A \in V$ and $u, v, w \in (V \cup \Sigma)^*$. Prove that this family of grammars generates the recursively enumerable languages.
- 18. A linear-bounded automaton is deterministic if at most one transition is specified for each state and tape symbol. Prove that every context-free language is accepted by a deterministic LBA.
- 19. Let L be a context-sensitive language that is accepted by a deterministic LBA. Prove that \bar{L} is context-sensitive. Recall that a computation in an arbitrary deterministic LBA need not halt.

Bibliographic Notes

The Chomsky hierarchy was introduced by Chomsky [1956, 1959]. This paper includes the proof that the unrestricted grammars generate precisely recursively enumerable languages. Linear-bounded automata were presented in Myhill [1960]. The relationship between linear-bounded automata and context-sensitive languages was developed by Landweber [1963] and Kuroda [1964]. Solutions to Exercises 10, 11, and 12, which exhibit the relationship between monotonic and context-sensitive grammars, can be found in Kuroda [1964].

CHAPTER 11

Decision Problems and the Church-Turing Thesis

In the preceding chapters Turing machines were used to detect patterns in strings, to recognize languages, and to compute functions. Many interesting problems, however, are posed at a higher level than string recognition or manipulation. For example, we may be interested in determining answers to questions of the form: “Is a natural number a perfect square?” Or “Does a graph have a cycle?” Or “Does the computation of a Turing machine halt before the 20th transition?” Each of these general questions describes a decision problem.

Formally, a **decision problem** P is a set of related questions each of which has a yes or no answer. The decision problem of determining if a natural number is a perfect square consists of the following questions:

p_0 : Is 0 a perfect square?

p_1 : Is 1 a perfect square?

p_2 : Is 2 a perfect square?

⋮ ⋮

Each individual question is referred to as an instance of the problem. A solution to a decision problem P is an algorithm that determines the appropriate answer to every question $p \in P$. A decision problem is said to be **decidable** if it has a solution.

Since the solution to a decision problem is an algorithm, a review of our intuitive notion of algorithmic computation may be beneficial. We have not defined, and probably cannot precisely define, the term *algorithm*. This notion falls into the category of “I can’t describe it but I know one when I see one.” We can, however, list several properties that

seem fundamental to the concept of algorithm. An algorithm that solves a decision problem should be

- Complete: It produces the correct answer for each problem instance.
- Mechanistic: It consists of a finite sequence of instructions, each of which can be carried out without requiring insight, ingenuity, or guesswork.
- Deterministic: When presented with identical input, it always performs the same computation.

A procedure that satisfies the preceding properties is often called *effective*.

The computations of a standard Turing machine are clearly mechanistic and deterministic. A Turing machine solution that halts for every input string is also complete. Because of the intuitive effectiveness of their computations, we will use Turing machines as the framework for solving decision problems. The transformation of problem instances into input strings for a Turing machine constitutes the representation of the decision problem. A problem instance is answered affirmatively if the corresponding input string is accepted by the Turing machine and negatively if it is rejected.

The Church-Turing Thesis for decision problems asserts that a Turing machine can be designed to solve any decision problem that is solvable by any effective procedure. A more general interpretation of the Church-Turing Thesis is that any procedure or process that can be algorithmically computed can be realized by a suitably designed Turing machine. This chapter begins by establishing the relationship between decision problems, Turing machines, and recursive languages. The remainder of the chapter presents the Church-Turing Thesis and discusses the importance and implications of the assertion.

11.1 Representation of Decision Problems

The first step in a Turing machine solution of a decision problem is to express the problem in terms of the acceptance of strings. This requires constructing a representation of the problem. Recall the newspaper vending machine described at the beginning of Chapter 5. Thirty cents in nickels, dimes, and quarters is required to open the latch. If more than 30 cents is inserted, the machine keeps the entire amount. Now consider the problem of a miser who wants to buy a newspaper but refuses to pay more than the minimum. A solution to this problem is an effective procedure that determines whether a set of coins contains a combination that totals exactly 30 cents.

A Turing machine representation of the miser's problem transforms an instance of the problem from its natural domain of coins into an equivalent problem of accepting a string. This can be accomplished by representing a set of coins as an element of $\{n, d, q\}^*$ where n , d , and q designate a nickel, a dime, and a quarter, respectively. Using this representation, a Turing machine that solves the miser's problem accepts strings $qnnn$, $nddnd$ and rejects $nnnd$ and $qdqdqqq$. In Exercise 1 you are asked to build a Turing machine that solves this problem.

Problem
Instances

p_1 —

p_2 —

p_3 —

⋮

p_i —

⋮

⋮

Constructing
process outlined in
representation of the
in the design of the
representation by c
Two common repre
The alphabet of the
 I^{n+1} . The alphabet
The Turing ma

solves the even nu
whether an even or
string of odd length

The binary repr
machine

$\times q_0$

accepts precisely the
dence of the Turing

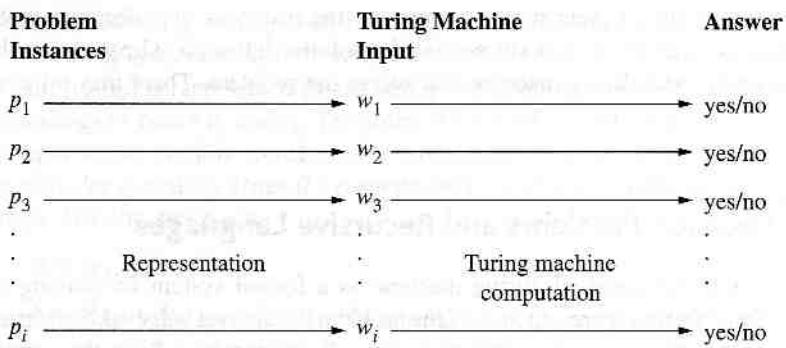
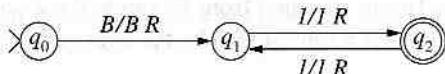


FIGURE 11.1 Solution to decision problem.

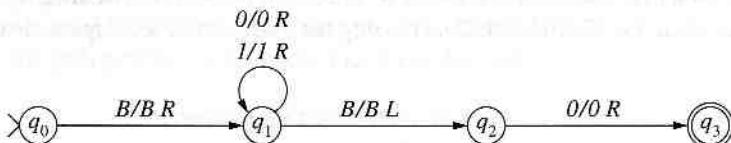
Constructing a Turing machine solution to a decision problem follows the two-step process outlined in Figure 11.1. The first step is the selection of an alphabet and a string representation of the problem instances. The properties of the representation are then utilized in the design of the Turing machine that solves the problem. We illustrate the impact of the representation by considering the problem of determining whether a natural number is even. Two common representations of natural numbers are the unary and binary representations. The alphabet of the unary representation is $\{I\}$ and the number n is represented by the string I^{n+1} . The alphabet $\{0, I\}$ is used by the standard binary representation of natural numbers.

The Turing machine



solves the even number problem for the unary representation. The states q_1 and q_2 record whether an even or odd number of I 's have been processed. In the unary representation, a string of odd length represents an even number. Thus the language of M_1 is $\{I^i \mid i \text{ is odd}\}$.

The binary representation of an even number has 0 in the rightmost position. The Turing machine



accepts precisely these strings. The strategies employed by M_1 and M_2 illustrate the dependence of the Turing machine on the choice of the representation.

There are many different ways to represent the instances of a decision problem as strings. A decision problem has a Turing machine solution if there is at least one combination of representation and Turing machine that solves the problem. There may, of course, be many.

11.2 Decision Problems and Recursive Languages

We have chosen the standard Turing machine as a formal system for solving decision problems. Once a string representation of the problem instances is selected, the remainder of the solution consists of the analysis of the input by a Turing machine. Since the completeness property requires the computation of the Turing machine to terminate for every input string, the language accepted by the machine is recursive. Thus every Turing machine solution of a decision problem defines a recursive language. Conversely, every recursive language L can be considered to be the solution of a decision problem. The decision problem, called the membership problem for L , consists of the questions “Is the string w in L ?” for every string w over the alphabet of L .

The duality between solvable decision problems and recursive languages can be exploited to broaden the techniques available for establishing the decidability of a decision problem. Since computations of deterministic multitrack and multitape machines can be simulated by a standard Turing machine, solutions using these machines also establish the decidability of a problem.

Example 11.2.1

The decision problem of determining whether a natural number is a perfect square is decidable. The three-tape Turing machine from Example 8.6.2 solves the perfect square problem with the natural number n represented by the string a^n . \square

Determinism is one of the fundamental properties of algorithms. However, it is often easier to design a nondeterministic Turing machine than a deterministic one to accept a language. In Section 8.7 it was shown that every language accepted by a nondeterministic Turing machine is also accepted by a deterministic one. A solution to a decision problem requires more than a machine that accepts the appropriate strings; it also demands that all computations terminate. A nondeterministic machine in which every computation terminates can be used to establish the existence of a decision procedure. The languages of such machines are recursive (Exercise 8.23), ensuring the existence of a complete deterministic solution.

Example 11.2.2

We will use nondeterminism to show that the problem of determining whether there is a path from a node v_i to a node v_j in a directed graph is decidable. A directed graph consists

of a set of nodes N : over $\{0, 1\}$, node v_k is a node. An arc $[v_s, v_t]$ are the encodings of

The input to the encoding of nodes v_i of the graph. The dir

$$N = \{v_1, v_2, \dots\}$$

$$A = \{[v_i, v_j] \mid v_i, v_j \in N\}$$

is represented by the determine whether the string
 11011100110110011

A nondeterministic The actions of M are

1. The input is checked against the graph followed by
2. The input is now in the representation of
3. The encoding of
4. Let v_s be the rightmost noncyclically chosen final state on tape 2, M halts
5. If $v_t = v_j$, then M accepts the string on tape 1

Steps 4 and 5 generate only noncyclic paths a that $L(M)$ is recursive

A decision problem condition that must be met by definition, the path pr

P
In

on problem as
ne combination
, of course, be

solving decision
he remainder of
ie completeness
ery input string,
hine solution of
sive language L
problem, called
in L?" for every

ages can be ex-
tly of a decision
machines can be
Iso establish the

perfect square is
e perfect square

□

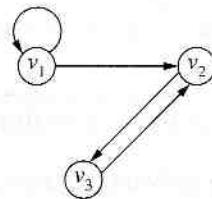
ever, it is often
one to accept a
nondeterministic
lecision problem
demands that all
mputation termi-
anguages of such
lete deterministic

whether there is a
ed graph consists

of a set of nodes $N = \{v_1, \dots, v_n\}$ and arcs $A \subseteq N \times N$. To represent a graph as a string over $\{0, 1\}$, node v_k is encoded as 1^{k+1} using the unary representation of the subscript of the node. An arc $[v_s, v_t]$ is represented by the string $en(v_s)0en(v_t)$, where $en(v_s)$ and $en(v_t)$ are the encodings of nodes v_s and v_t . The string 00 is used to separate arcs.

The input to the machine consists of a representation of the graph followed by the encoding of nodes v_i and v_j . Three 0 's separate $en(v_i)$ and $en(v_j)$ from the representation of the graph. The directed graph

$$\begin{aligned} N &= \{v_1, v_2, v_3\} \\ A &= \{[v_1, v_2], [v_1, v_1], [v_2, v_3], [v_3, v_2]\} \end{aligned}$$



is represented by the string $110111001101100111011110011110111$. A computation to determine whether there is a path from v_3 to v_1 in this graph begins with the input $1101110011011001110111100111101110001111011$.

A nondeterministic two-tape Turing machine M is designed to solve the path problem. The actions of M are summarized as follows:

1. The input is checked to determine if its format is that of a representation of a directed graph followed by the encoding of two nodes. If not, M halts and rejects the string.
2. The input is now assumed to have the form $R(G)000en(v_i)0en(v_j)$, where $R(G)$ is the representation of a directed graph G . If $v_i = v_j$, M halts in an accepting state.
3. The encoding of node v_i followed by 0 is written on tape 2.
4. Let v_s be the rightmost node encoded on tape 2. An arc from v_s to v_t is nondeterministically chosen from $R(G)$. If no such arc exists or v_t is already on the path encoded on tape 2, M halts in a rejecting state.
5. If $v_t = v_j$, then M halts in an accepting state. Otherwise, $en(v_t)0$ is written at the end of the string on tape 2 and the computation continues with step 4.

Steps 4 and 5 generate paths beginning with node v_i on tape 2. Since step 4 guarantees that only noncyclic paths are written on tape 2, every computation of M terminates. It follows that $L(M)$ is recursive and the problem is decidable. □

A decision problem will frequently be defined by describing its instances and the condition that must be satisfied to obtain a positive answer. Using this method of problem definition, the path problem of Example 11.2.2 can be written

Path Problem for Directed Graphs

Input: Directed graph $G = (N, A)$, nodes $v_i, v_j \in N$

Output: yes; if there is a path from v_i to v_j in G
no; otherwise.

With the correspondence between solvable decision problems and recursive languages, should we speak of problems or languages? We will use the terminology of decision problems when the problem statement is given using high-level concepts and a representation is required to transform the problem instances into strings. When a problem is specified in terms of the acceptance of strings, we will use the terminology of recursive languages. In either case, a decision problem or a language is decidable if there is an algorithm that produces the correct answer for each problem instance or the correct membership value for each string, respectively.

11.3 Problem Reduction

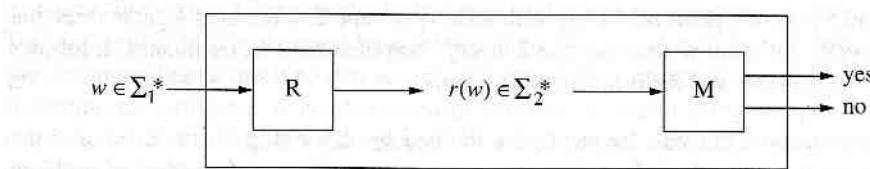
Reduction is a problem-solving technique commonly employed to avoid “reinventing the wheel” when encountering a new problem. The objective of a reduction is to transform the instances of the new problem into those of a problem that we already know how to solve. Reduction is an important tool for establishing the decidability of problems and, as we will see in Chapter 12, also for showing that certain problems do not have algorithmic solutions.

We will examine the mappings and requirements needed for problem reduction both on the level of languages and on the level of decision problems. We begin with the definition of reduction for membership in languages.

Definition 11.3.1

Let L be a language over Σ_1 and Q a language over Σ_2 . L is **many-to-one reducible** to Q if there is a Turing computable function $r : \Sigma_1^* \rightarrow \Sigma_2^*$ such that $w \in L$ if, and only if, $r(w) \in Q$.

If a language L is reducible to a decidable language Q by a function r , then L is also decidable. Let R be the Turing machine that computes the reduction and M the machine that accepts Q . The sequential execution of R and M on strings from Σ_1^* constitutes a solution to the membership problem for L .



Note that the reduction machine R does not determine membership in either L or Q ; it simply transforms strings from Σ_1^* to Σ_2^* . Membership in Q is determined by M and membership in L by the combination of R and M .

To illustrate the reduction of one language to another, we will show that $L = \{x^i y^i z^k \mid i \geq 0, k \geq 0\}$ is reducible to $Q = \{a^i b^i \mid i \geq 0\}$. A reduction of L to Q may be described in the tabular form

A string $w \in \{x, y, z\}^*$

- i) If w has no x :
and erase the z
- ii) If w has an x at
input position.

The following table

The examples show
strings in Σ_1^* can ma

The Turing mac

Reduction	Input	Condition
$L = \{x^i y^i z^k \mid i \geq 0, k \geq 0\}$ to $Q = \{a^i b^i \mid i \geq 0\}$	$w \in \{x, y, z\}^*$ $\downarrow r$ $v \in \{a, b\}^*$	$w \in L$ if, and only if, $r(w) \in Q$

A string $w \in \{x, y, z\}^*$ is transformed to the string $r(w) \in \{a, b\}^*$ as follows:

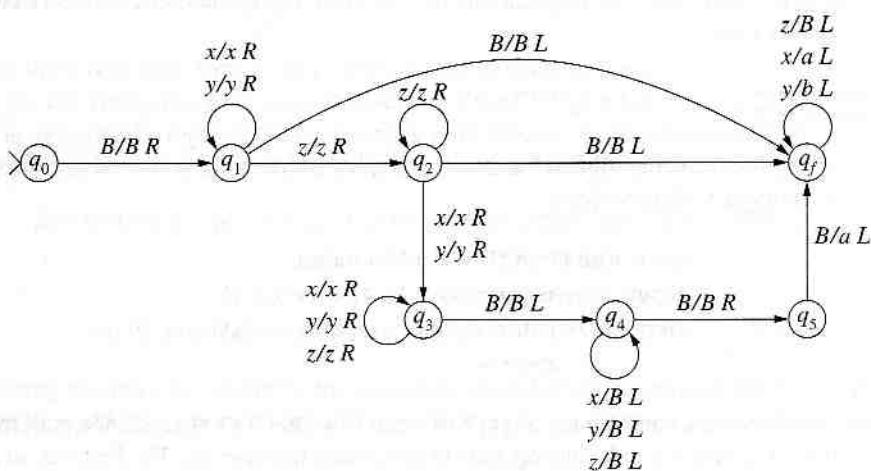
- i) If w has no x 's or y 's occurring after a z , replace each x with an a , each y with a b , and erase the z 's.
- ii) If w has an x or y occurring after a z , erase the entire string and write a single a in the input position.

The following table gives the result of the transformation of several strings in Σ_1^* .

$w \in \Sigma_1^*$	In L ?	$r(w) \in \Sigma_2^*$	In Q ?
$xxyy$	yes	$aabb$	yes
$xxyyzzz$	yes	$aabb$	yes
$yxxyz$	no	$baab$	no
$xxzyy$	no	a	no
$zyzx$	no	a	no
λ	yes	λ	yes
zzz	yes	λ	yes

The examples show why the transformation is called a many-to-one reduction; multiple strings in Σ_1^* can map to the same string in Σ_2^* .

The Turing machine



performs the reduction of L to Q. Strings that have the form $(x \cup y)^*z^*$ are identified in states q_1 and q_2 and transformed in state q_f . Strings in which a z precedes an x or y are erased in state q_4 and an a is written on the tape in the transition to q_f .

Example 11.3.1

Consider the problem of accepting strings in the language $L = \{uu \mid u = a^i b^i c^i \text{ for some } i \geq 0\}$. The machine M in Example 8.2.2 accepts the language $\{a^i b^i c^i \mid i \geq 0\}$. We will sketch a reduction of the membership problem of L to that of recognizing a single instance of $a^i b^i c^i$. The original problem can then be solved using the reduction and the machine M. The reduction is obtained as follows:

1. The input string w is copied. The copy of w is used to determine whether $w = uu$ for some string $u \in \{a, b, c\}^*$.
2. If $w \neq uu$, then the tape is erased and a single a is written in the input position.
3. If $w = uu$, then the copy and the second u in the input string are erased leaving u in the input position.

If the input string w has the form uu , then $w \in L$ if, and only if, $u = a^i b^i c^i$ for some i . The reduction does not check the number or the order of the a 's, b 's, and c 's; the machine M has been designed to perform that task.

If a string w does not have the form uu , the reduction produces the string a . This string is subsequently rejected by M, indicating that the input w is not in L. \square

A decision problem P is many-to-one reducible to a problem Q if there is a transformation of problem instances of P into instances of the Q that preserves the affirmative and negative answers. Formally, a reduction transforms the string representations of the problem instances. Frequently, we will define a reduction directly on the problem instances, with the assumption that the modifications could be performed at the string level if we so desire. This technique, along with the implications for the string representations, is illustrated in the following example.

Example 11.3.2

We will show that the path problem for directed graphs, which was introduced in Example 11.2.2, is reducible to the problem:

Cycle with Fixed Node (CFN) Problem

Input: Directed graph $G = (N, A)$, node $v_k \in N$

Output: yes; if there is a cycle containing v_k in G
no; otherwise.

The reduction requires constructing a graph G' from G so that the existence of a path from v_i to v_j in G is equivalent to G' having a cycle containing the node v_k . The first step in the

reduction is to identify problem. With the sele

Reduction

Path Problem
to
CFN Problem

The graph G' is obtaine

- i) Deleting all arcs [$v_i, v_j]$
- ii) Adding an arc $[v_j, v_i]$

If there is a path from a node; cycles in the path or terminal node. Conse or absence of a path fr contain v_i since there a

The addition of the is a path from v_i to v_j in the path problem to the

The reduction of in Example 11.2.2, produc



Since there is no path fr

On the Turing mach and node v_i may be repr G, v_3, v_1 of the path pro

$R(G)000en(v_3)e$

to

$R(G')000en$

A Turing machine that p entering v_i , add the repr input.

ified in
or y are

for some
We will
instance
chine M.

$= uu$ for

on.
ving u in

ne i . The
achine M

his string

□

transformative and
e problem
, with the
so desire.
strated in

Example

path from
step in the

reduction is to identify the node v_k in the CFN problem to the initial node v_i of the path problem. With the selection of v_i as the node in the CFN problem, the reduction becomes

Reduction	Instances	Condition
Path Problem to CFN Problem	Graph G, nodes v_i, v_j $\downarrow r$ Graph G' , node v_i	G has a path from v_i to v_j if, and only if, G' has a cycle containing v_i

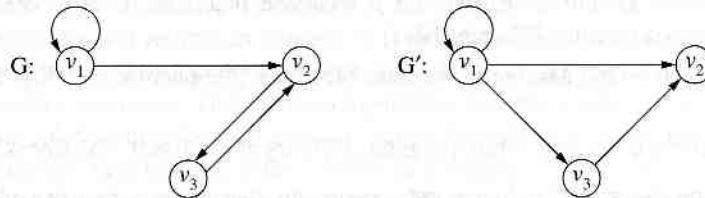
The graph G' is obtained by modifying G as follows:

- i) Deleting all arcs $[v_t, v_i]$ that enter v_i .
- ii) Adding an arc $[v_j, v_i]$.

If there is a path from v_i to v_j in G , then there is a path in which v_i occurs only as the first node; cycles in the path that reenter v_i may be removed without changing either the initial or terminal node. Consequently, the deletion of the arcs $[v_t, v_i]$ does not affect the presence or absence of a path from v_i to v_j . After the arc deletion in step (i), there are no cycles that contain v_i since there are no arcs that enter v_i .

The addition of the arc $[v_j, v_i]$ in step (ii) will produce a cycle in G' if, and only if, there is a path from v_i to v_j in the original graph G . Thus the modification of G is a reduction of the path problem to the CFN problem.

The reduction of instance G, v_3, v_1 of the path problem, where G is the graph from Example 11.2.2, produces



Since there is no path from v_3 to v_1 in G , G' has no cycle containing v_3 .

On the Turing machine level, an instance of the CFN problem consisting of a graph G' and node v_i may be represented by the string $R(G')000en(v_i)$. The reduction of the instance G, v_3, v_1 of the path problem to the instance G', v_3 of the CFN problem changes

$$R(G)000en(v_3)en(v_1) = 1101111001101100111100111101110001111011$$

to

$$R(G')000en(v_3) = 1101111001101100111100111101110001111.$$

A Turing machine that performs the reduction must delete the representations of the arcs entering v_i , add the representation of the arc from v_j to v_i , and erase v_j from the end of the input. □

As with languages, reducing a decision problem P to a decidable problem Q shows that P is also decidable. A solution to P can be obtained by sequentially combining the reduction with the algorithm that solves Q .

11.4 The Church-Turing Thesis

The notion of algorithmic computation is not new. In fact, the word *algorithm* comes from the name of the 9th-century Arabian mathematician Abu Ja'far Muhammad ibn Musa al-Khwarizmi. In what is generally considered the first book on algebra, Al-Khwarizmi presented a set of rules for solving linear and quadratic equations. Step-by-step mechanistic procedures have been employed for centuries to describe calculations, processes, and mathematical derivations. This informal usage matured in the early 20th century when mathematicians sought to precisely determine the meaning, capabilities, and limitations of algorithmic computation.

The investigation into the properties of computability led to a number of approaches and formalisms for performing algorithmic computation. Effective procedures have been defined by rules that transform strings, by the evaluation of functions, by the computations of abstract machines, and more recently, by programs in high-level programming languages. Examples of each of these types of systems include

- String Transformations: Post systems [Post, 1936], Markov systems [Markov, 1961], unrestricted grammars
 - Evaluation of Functions: partial and μ -recursive functions [Gödel, 1931; Kleene, 1936], lambda calculus [Church, 1941]
 - Abstract Computing Machines: Register Machines [Shepherdson, 1963], Turing machines
 - Programming languages: while-programs [Kfoury et al., 1982], TM from Chapter 9

While-programs, listed in a final category, are programs that can be written in a minimal programming language that consists of assignment, conditional, for, and while statements. Having a small number of statements facilitates the analysis of programs, but while-programs have the same computational ability as programs in standard programming languages such as C, C++, Java, and so on.

We have used Turing machines as the computational framework for solving decision problems. However, any of the other algorithmic systems could just as well have been selected. Would this in any way have changed our ability to solve problems? Ideally the answer should be no—the existence of a solution to a problem should be an inherent feature of the problem itself and not an artifact of our choice of an algorithmic system. The Church-Turing Thesis validates this intuition.

What do all of the previously mentioned algorithmic systems have in common? It has been shown that they are all capable of performing precisely the same computations. This claim may seem remarkable, since these systems were designed to perform different types of operations on different types of data. However, you have already seen one example of

the equivalence and computation of a Turing machine. Conversely, any language introduced by Gödel's algorithmic approach can be simulated by a Turing machine.

The realization that have the same constraints define the bounds of computation and no single system can define bound on what it can do. Thesis formalizes the notion of algorithmic computation of the types of computation Church-Turing Thesis.

The Church-Turing Thesis

A solution to a
instance of the problem.
A partial solution to
effective procedure
whose answer is yes
or fail to produce an

Just as a solution
ship in a recursive la
question of members
encompasses algorith

The Church-Turing Thesis
solvable if, and only if,
whose answer is yes.

Turing machines
halts to define the res-
lems uses the computa-
The method of specifi-
chine solutions (Exer-
of computable function

The Church-Turing Thesis
putable if, and only if,

the equivalence and will see another in Chapter 13. In Section 10.1 we proved that the computation of a Turing machine can be simulated by the rules of an unrestricted grammar. Conversely, any language generated by an unrestricted grammar is accepted by a Turing machine. Consequently, the power of Turing machines for recognizing languages is identical to that of unrestricted grammars for generating languages. In Chapter 13 we will show that the algorithmic approach to the definition and evaluation of number-theoretic functions introduced by Gödel and Kleene produces exactly the functions that can be computed by Turing machines.

The realization that the various approaches to effective computation produced systems that have the same computational power led to the belief that the capabilities of these systems define the bounds of algorithmic computation. There is no single definition of *algorithm* and no single system for performing effective computation. However, there is a well-defined bound on what can be accomplished in any of these systems. The Church-Turing Thesis formalizes this belief in a general statement about the capabilities and limitations of algorithmic computation. We will present three variations, one corresponding to each of the types of computations that we have studied. We begin with the interpretation of the Church-Turing Thesis for decision problems.

The Church-Turing Thesis for Decision Problems There is an effective procedure to solve a decision problem if, and only if, there is a Turing machine that halts for all input strings and solves the problem.

A solution to a decision problem requires the computation to return an answer for every instance of the problem. Relaxing this restriction, we obtain the notion of a partial solution. A partial solution to a decision problem P is a not necessarily complete but otherwise effective procedure that returns an affirmative response for every problem instance $p \in P$ whose answer is yes. If the answer to p is negative, however, the procedure may return no or fail to produce an answer. That is, the computation recognizes affirmative instances.

Just as a solution to a decision problem can be formulated as a question of membership in a recursive language, a partial solution to a decision problem is equivalent to the question of membership in a recursively enumerable language. The Church-Turing Thesis encompasses algorithms that recognize languages as well as those that decide languages.

The Church-Turing Thesis for Recognition Problems A decision problem P is partially solvable if, and only if, there is a Turing machine that accepts precisely the instances of P whose answer is yes.

Turing machines compute functions using the symbols on the tape when the machine halts to define the result of a computation. A functional approach to solving decision problems uses the computed values one and zero to designate affirmative and negative responses. The method of specifying the answer does not affect the set problems that have Turing machine solutions (Exercise 9.4). Thus the formulation of the Church-Turing Thesis in terms of computable functions subsumes and extends the two previous versions of the thesis.

The Church-Turing Thesis for Computable Functions A function f is effectively computable if, and only if, there is a Turing machine that computes f .

After establishing the equivalence of Turing computable functions and μ -recursive functions in Chapter 13, we will give a more concise version of the Church-Turing Thesis and present a natural generalization from computable number-theoretic functions to computable functions on arbitrary sets.

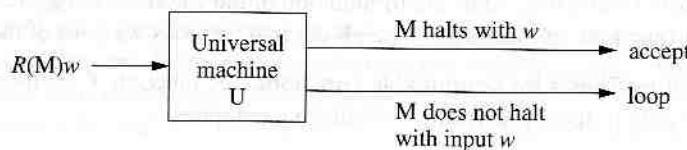
To appreciate the content of the Church-Turing Thesis, it is necessary to understand the nature of the assertion. The Church-Turing Thesis is not a mathematical theorem; it cannot be proved. This would require a formal definition of the intuitive notion of an effective procedure. The claim could, however, be disproved. This could be accomplished by discovering an effective procedure that cannot be computed by a Turing machine. The equivalence of Turing machines to other algorithmic systems, the robustness of the Turing machine architecture, and the lack of a counterexample highlight an impressive pool of evidence that suggests that such a procedure will not be found.

A proof by the Church-Turing Thesis is a shortcut often taken in establishing the existence of a decision algorithm. Rather than constructing a Turing machine solution to a decision problem, we describe an intuitively effective procedure that solves the problem. The Church-Turing Thesis guarantees that a Turing machine can be designed to solve the problem. We have tacitly been using the Church-Turing Thesis in this manner throughout the presentation of Turing computability. For complicated machines, we simply gave a description of the actions of a computation of the machine. We assumed that the complete machine could then be explicitly constructed, if desired.

11.5 A Universal Machine

One of the most significant advances in computer design occurred in the mid-1940s with the development of the stored program model of computation. Early computers were designed to perform a single task; the input could vary, but the same program would be executed for each input. Making a change to the instructions would frequently require reconfiguration of the hardware. In the stored program model, the instructions are electronically loaded into memory along with the data. A computation in a stored program computer is a cycle consisting of the retrieval of an instruction from memory followed by its execution.

The Turing machines in the preceding chapters, like the early computers, were designed to execute a single set of instructions. The Turing machine architecture has its own version of the stored program concept, which preceded the first stored program computer by a decade. A **universal Turing machine** is designed to simulate the computations of an arbitrary Turing machine M . To do so, the input to the universal machine must contain a representation of the machine M and the string w to be processed by M . For simplicity, we will assume that M is a standard Turing machine that accepts by halting. The action of a universal machine U is depicted by



where $R(M)$ is the representation of the computation of M . If M does not halt, the computation of any

The first step in the computation of a Turing machine M on input $w \in \{0, 1\}$, we consider the initial state. The states of a Turing machine are

A Turing machine has a tape with states $\{L, R\}$. We encode the

Let $en(z)$ denote the encoding of z by the string

The 0 's separate the encoded parts of w . U is constructed from the encoding of M . The beginning and end of the tape are indicated by

Example 11.5.1

The computation of the

and μ -recursive functions to com-

y to understand
ical theorem; it
ve notion of an
e accomplished
g machine. The
ss of the Turing
pressive pool of

establishing the
ine solution to a
es the problem.
ned to solve the
nner throughout
; simply gave a
at the complete

i-1940s with the
s were designed
be executed for
reconfiguration
ronically loaded
puter is a cycle
xecution.
s, were designed
s own version of
ter by a decade.
arbitrary Turing
epresentation of
will assume that
iversal machine

where $R(M)$ is the representation of the machine M . The output labeled loop indicates that the computation of U does not terminate. If M halts and accepts input w , U does the same. If M does not halt with w , neither does U . The machine U is called universal since the computation of any Turing machine M can be simulated by U .

The first step in the construction of a universal machine is to design the string representation of a Turing machine. Because of the ability to encode arbitrary symbols as strings over $\{0, 1\}$, we consider Turing machines with input alphabet $\{0, 1\}$ and tape alphabet $\{0, 1, B\}$. The states of a Turing machine are assumed to be named $\{q_0, q_1, \dots, q_n\}$, with q_0 the start state.

A Turing machine M is defined by its transition function. A transition of a standard Turing machine has the form $\delta(q_i, x) = [q_j, y, d]$, where $q_i, q_j \in Q$; $x, y \in \Gamma$; and $d \in \{L, R\}$. We encode the elements of M using strings of I 's:

Symbol	Encoding
0	I
1	II
B	III
q_0	I
q_1	II
\vdots	\vdots
q_n	I^{n+1}
L	I
R	II

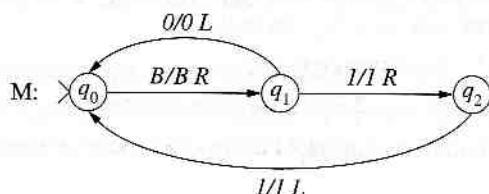
Let $en(z)$ denote the encoding of a symbol z . A transition $\delta(q_i, x) = [q_j, y, d]$ is encoded by the string

$\text{en}(q_i)O\text{en}(x)O\text{en}(q_j)O\text{en}(y)O\text{en}(d)$.

The O 's separate the components of the transition. A representation of the machine is constructed from the encoded transitions. Two consecutive O 's are used to separate transitions. The beginning and end of the representation are designated by three O 's.

Example 11.5.1

The computation of the Turing machine



halts for the null string and strings that begin with l , and does not terminate for strings beginning with 0 . The encoded transitions of M are given in the following table.

Transition	Encoding
$\delta(q_0, B) = [q_1, B, R]$	101110110111011
$\delta(q_1, 0) = [q_0, 0, L]$	1101010101
$\delta(q_1, l) = [q_2, l, R]$	110110111011011
$\delta(q_2, l) = [q_0, l, L]$	1110110101101

The machine M is represented by the string

$$00010111011011101100110101010011011011011001110110101101000. \quad \square$$

A Turing machine can be constructed to determine whether an arbitrary string $u \in \{0, 1\}^*$ is the encoding of a deterministic Turing machine. The computation examines u to see if it consists of a prefix 000 followed by a finite sequence of encoded transitions separated by 00 's followed by 000 . A string that satisfies these conditions is the representation of some Turing machine M . The machine M is deterministic if the combination of the state and input symbol in every encoded transition is distinct.

We will now outline the design of a three-tape, deterministic universal machine U . A computation of U begins with the input on tape 1. If the input string has the form $R(M)w$, the computation of M with input w is simulated on tape 3. A computation of U consists of the following actions:

1. If the input string does not have the form $R(M)w$ for a deterministic Turing machine M and string w , U moves to the right forever.
2. The string w is written on tape 3 beginning at position one. The tape head is then repositioned at the leftmost square of the tape. The configuration of tape 3 is the initial configuration of a computation of M with input w .
3. A single l , the encoding of state q_0 , is written on tape 2.
4. A transition of M is simulated on tape 3. The transition of M is determined by the symbol scanned on tape 3 and the state encoded on tape 2. Let x be the symbol from tape 3 and q_i the state encoded on tape 2.
 - a) Tape 1 is scanned for a transition whose first two components match $en(q_i)$ and $en(x)$. If there is no such transition, U halts accepting the input.
 - b) If tape 1 contains an encoded transition $en(q_i)0en(x)0en(q_j)0en(y)0en(d)$, then
 - i) $en(q_i)$ is replaced by $en(q_j)$ on tape 2.
 - ii) The symbol y is written on tape 3.
 - iii) The tape head of tape 3 is moved in the direction specified by d .
5. The computation continues with step 4 to simulate the next transition of M .

Theorem 11.5.1

The language L is

Proof. The universal representation of M is

The language L is the combination of

The computation of M with input w and simulates

Example 11.5.2

A solution to the

can be obtained by

input w " and cou

A machine U to the universal

problem instance

representation of

input string u con

1. If the input string does not have the form $R(M)w$ for a deterministic Turing machine M and string w , U moves to the right forever.
2. The string I^n is written on tape 3 beginning at position one. The tape head is then repositioned at the leftmost square of the tape. The configuration of tape 3 is the initial configuration of a computation of M with input w .
3. If the string w does not match the input string u , U halts accepting the input.
4. The string w is written on tape 3 beginning at position one. The tape head is then repositioned at the leftmost square of the tape. The configuration of tape 3 is the initial configuration of a computation of M with input w .
5. Following the computation of M with input w , the string w matches the string I^n on tape 3. The computation continues with step 4 to simulate the next transition of M .

for strings
le.

representations
of strings
and machines
and the
universal machine
accepts strings
of the form
 $R(M)w$ where
 $R(M)$ is the
representation of a
Turing machine
and M halts when
run with input w .
For all other strings,
the computation of U does not terminate. Thus the language of U is L_H . ■

ary string $u \in$
examines u to
ions separated
tation of some
state and input

machine U . A
form $R(M)w$,
f U consists of

Turing machine
e head is then
e 3 is the initial

etermined by the
e symbol from

atch $en(q_i)$ and
y) $0en(d)$, then

d.
of M .

Theorem 11.5.1

The language $L_H = \{R(M)w \mid M \text{ halts with input } w\}$ is recursively enumerable.

Proof. The universal machine accepts strings of the form $R(M)w$ where $R(M)$ is the representation of a Turing machine and M halts when run with input w . For all other strings, the computation of U does not terminate. Thus the language of U is L_H . ■

The language L_H is known as the language of the Halting Problem. A string is in L_H if it is the combination of the representation of a Turing M and a string w such that M halts when run with w .

The computation of the universal machine U with input $R(M)w$ simulates the computation M with input w . The ability to obtain the results of one machine via the computations of another facilitates the design of complicated Turing machines. When we say that a Turing machine M' “runs machine M with input w ,” we mean that M' is supplied with $R(M)$ and w and simulates the computation of M in the manner of the universal machine.

Example 11.5.2

A solution to the decision problem

Halts on n 'th Transition Problem

Input: Turing machine M , string w , integer n

Output: yes; if the computation of M with input w performs

exactly n transitions before halting

no; otherwise.

can be obtained by simulating the computations of M . Intuitively, a solution “runs M with input w ” and counts the transitions of M .

A machine U' that solves this problem can be constructed by adding a fourth tape to the universal machine to record the number of transitions in a computation of M . A problem instance will be represented by a string of the form $R(M)w000I^{n+1}$ with the unary representation of n separated from $R(M)w$ by three zeroes. The computation of U' with input string u consists of the following actions:

1. If the input string u does not end with $000I^{n+1}$, U' halts rejecting the input.
2. The string I^n is written on tape 4 beginning in position one; $000I^{n+1}$ is erased from the end of the string on tape 1; and the tape head on tape 4 moves to position one.
3. If the string remaining on tape 1 does not have the form $R(M)w$, U' halts rejecting the input.
4. The string w is copied to tape 3 and the encoding of state q_0 is written on tape 2.
5. Following the strategy of the universal machine, tape 1 is searched for a transition that matches the symbol x scanned on tape 3 and the state q_i encoded on tape 2.

- a) If there is no transition for q_i, x and a 1 is read on tape 4, then U' halts rejecting the input.
 - b) If there is no transition for q_i, x and a blank is read on tape 4, then U' halts accepting the input.
 - c) If there is a transition $\delta(q_i, x)$ encoded on tape 1 and a blank is read on tape 4, then U' halts rejecting the input.
 - d) If there is a transition $\delta(q_i, x)$ encoded on tape 1 and a 1 is read on tape 4, then the transition is simulated on tapes 2 and 3 and the tape head on tape 4 is moved one square to the right.
6. The computation continues with step 5 to examine the next transition of M .

If M halts prior to the n th transition, $R(M)w0001^{n+1}$ is rejected in step 5 (a). After the simulation of n transitions of M , the counter on tape 4 reads a blank. If M has no applicable transition at this point, U' accepts. Otherwise, the input is rejected in step 5 (c). \square

Exercises

1. Give a state diagram of a Turing machine M that solves the miser problem from Section 11.1. A set of coins is represented as an element of $\{n, d, q\}^*$ where n, d , and q designate a nickel, a dime, and a quarter, respectively.

In Exercises 2 through 7, describe a Turing machine that solves the specified decision problem. Use Example 11.2.2 as a model for defining the actions of a computation of the machine. You need not explicitly construct the transition function nor the state diagram of your solution. You may use multitape Turing machines and nondeterminism in your solutions.

2. Design a two-tape Turing machine that determines whether two strings u and v over $\{0, 1\}$ are identical. The computation begins with $BuBvB$ on the tape and should require no more than $3(\text{length}(u) + 1)$ transitions.
3. Using the unary representation of the natural numbers, design a Turing machine whose computations decide whether a natural number is prime.
4. Using the unary representation of the natural numbers, design a Turing machine that solves the “ 2^n ” problem. Hint: The input is the representation of a natural number i and the output is yes if $i = 2^n$ for some n , no otherwise.
5. A directed graph is said to be *cyclic* if it contains at least one cycle. Using the representation of a directed graph from Section 11.2, design a Turing machine whose computations decide whether a directed graph is cyclic.

6. A tour in a directed graph.
 - i) $p_0 = p_n$
 - ii) For $0 < i < n$, $p_i \neq p_{i+1}$
 - iii) Every node is reached from every other node.

That is, a tour in a directed graph is a path that starts and ends at the same node and passes through every node exactly once. Construct a Turing machine that solves the tour problem.
- *7. Let $G = (V, E)$.
 - a) Construct a Turing machine that decides whether G is connected.
 - b) Design a Turing machine that decides whether G is a complete graph.
8. Construct a Turing machine that decides whether a language L is $\{x, y\}^*$.
 - a) $L = (xy)^*$
 - b) $L = x^+y^*$
 - c) $L = \{x^i y^i\}$
 - d) $L = \{x^i y^j z^k\}$
 - e) $L = \{x^i(yz)^j\}$
 - f) $L = \{x^i y^j x^k\}$
9. Let M be the Turing machine defined by the state diagram in Figure 11.12.
 - a) What is $L(M)$?
 - b) Give the regular expression for $L(M)$.
10. Construct a Turing machine that decides whether a language L is a complete language of a nondeterministic Turing machine that computes a language M .
 - i) $u = R(M)$
 - ii) when M is a complete language

Your machine should accept all inputs u such that $u \in L$ and $R(u) = M$.
11. Design a Turing machine that decides whether a language L is a complete language of a nondeterministic Turing machine that computes a language M .
 - i) $u = R(M)$
 - ii) when M is a complete language

Your machine should accept all inputs u such that $u \in L$ and $R(u) = M$.

ejecting the
is accepting
tape 4, then
24, then the
moved one

1.
a). After the
o applicable
□

from Section
1q designate

ied decision
tation of the
tate diagram
ism in your

u and v over
should require

achine whose

machine that
ral number i

sing the rep-
chine whose

6. A tour in a directed graph is a path p_0, p_1, \dots, p_n in which

- i) $p_0 = p_n$.
- ii) For $0 < i, j \leq n, i \neq j$ implies $p_i \neq p_j$.
- iii) Every node in the graph occurs in the path.

That is, a tour visits every node exactly once and ends where it begins. Design a Turing machine that decides whether a directed graph contains a tour. Use the representation of a directed graph given in Section 11.2.

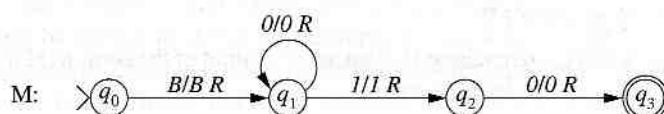
- *7. Let $G = (V, \Sigma, P, S)$ be a regular grammar.

- a) Construct a representation for the grammar G over $\{0, 1\}$.
- b) Design a Turing machine that decides whether a string $w \in \Sigma^*$ is in $L(G)$. The use of nondeterminism facilitates the construction of the desired machine.

8. Construct a Turing machine that reduces the language L to Q . In each case the alphabet of L is $\{x, y\}$ and the alphabet of Q is $\{a, b\}$.

- | | |
|--|---------------------------------|
| a) $L = (xy)^*$ | $Q = (aa)^*$ |
| b) $L = x^+y^*$ | $Q = a^+b$ |
| c) $L = \{x^i y^{i+1} \mid i \geq 0\}$ | $Q = \{a^i b^i \mid i \geq 0\}$ |
| d) $L = \{x^i y^j z^l \mid i \geq 0, j \geq 0, l \geq 0\}$ | $Q = \{a^i b^i \mid i \geq 0\}$ |
| e) $L = \{x^i (yy)^i \mid i \geq 0\}$ | $Q = \{a^i b^i \mid i \geq 0\}$ |
| f) $L = \{x^i y^i x^i \mid i \geq 0\}$ | $Q = \{a^i b^i \mid i \geq 0\}$ |

9. Let M be the Turing machine



- a) What is $L(M)$?
 - b) Give the representation of M using the encoding from Section 11.5.
10. Construct a Turing machine that decides whether a string over $\{0, 1\}^*$ is the encoding of a nondeterministic Turing machine. What would be required to change this to a machine that decides whether the input is the representation of a deterministic Turing machine?
11. Design a Turing machine with input alphabet $\{0, 1\}$ that accepts an input string u if
- i) $u = R(M)w$ for some Turing machine M and input string w , and
 - ii) when M is run with input w , there is a transition in the computation that prints a 1.
- Your machine need not halt for all inputs.

12. Given an arbitrary Turing machine M and input string w , will the computation of M with input w halt in fewer than 100 transitions? Describe a Turing machine that solves this decision problem.
13. Show that the decision problem

Input: Turing machine M

Output: yes; if the third transition of M prints a blank when run
with a blank tape
no; otherwise.

is decidable. The answer for a Turing machine M is no if M halts prior to its third transition.

- * 14. Show that the decision problem

Input: Turing machine M

Output: yes; if there is some string $w \in \Sigma^*$ for which the computation
of M takes more than 10 transitions
no; otherwise.

is decidable.

15. The universal machine introduced in Section 11.5 was designed to simulate the actions of Turing machines that accept by halting. Consequently, the representation scheme $R(M)$ did not encode accepting states.
- Extend the representation $R(M)$ of a Turing machine M to explicitly encode the accepting states of M .
 - Design a universal machine U_f that accepts input of the form $R(M)w$ if the machine M accepts input w by final state.

Bibliographic Notes

Turing [1936] envisioned the theoretical computing machine he designed to be capable of performing all effective computations. This viewpoint, now known as the Church-Turing Thesis, was formalized by Church [1936]. Turing's 1936 paper also included the design of a universal machine. The original plans for the development of a stored program computer were reported by von Neumann [von Neumann, 1945], and the first working models appeared in 1949.

In our construction of the universal machine, we limited the input and tape alphabets of the Turing machines to $\{0, 1\}$ and $\{0, 1, B\}$, respectively. A proof that an arbitrary Turing machine can be simulated by a machine with these alphabets can be found in Hopcroft and Ullman [1979].

The Church-decision problem is not encodable in a single device. Thus, it must be solved entirely on the processor time required to solve a problem can be bounded by a procedure. A

In Section 11.5, we saw that the number of states in a Turing machine is finite, but that there are many more strings in Σ^* than there are states. In fact, the cardinalities of Σ^* and $\{0, 1\}^*$ are equal. We have no idea of what particular decidable problems can be solved by a Turing machine.

The first thing you should appreciate the Church-Turing Thesis is that it is rather than Tu

utation of M
ne that solves

or to its third

on

ite the actions
ation scheme
ly encode the
f the machine

to be capable
the Church-
included the
ored program
first working

ape alphabets
bitrary Turing
Hopcroft and

CHAPTER 12

Undecidability

The Church-Turing Thesis asserts that a Turing machine can be designed to solve any decision problem that is solvable by any effective procedure. A Turing machine computation is not encumbered by the physical restrictions that are inherent in any “real” computing device. Thus the existence of a Turing machine solution to a decision problem depends entirely on the nature of the problem itself and not on the availability of memory or central processor time. The Church-Turing Thesis also has consequences for undecidability. If a problem cannot be solved by a Turing machine, it cannot be solved by any effective procedure. A decision problem that has no algorithmic solution is said to be **undecidable**.

In Section 9.5 it was shown that there are only countably many Turing machines. The number of languages over a nonempty alphabet, however, is uncountable. It follows that there are languages whose membership problem is undecidable. The comparison of cardinalities ensures us of the existence of undecidable decision problems but gives us no idea of what such a problem might look like. In this chapter we show that some particular decision problems concerning the computational capabilities of Turing machines, derivations in grammars, and even playing a game with dominoes are undecidable.

The first problem that we consider is the Halting Problem for Turing Machines. To appreciate the significance of the Halting Problem, we will describe it in terms of C programs rather than Turing machines. The Halting Problem for C Programs can be stated as

Halting Problem for C Programs

Input: C program *Prog*,

input file *inpt* for *Prog*

Output: yes; if *Prog* halts when run with input *inpt*
no; otherwise.

If the Halting Problem for C Programs were decidable, a bane of all programmers—the infinite loop—would be a thing of the past. The execution of a program would become a two-step process:

1. running the algorithm that solves the Halting Problem on *Prog* and *inpt*;
2. if the algorithm indicates *Prog* will halt, then running *Prog* with *inpt*.

A solution to the Halting Problem does not tell us the result of the computation, only that a result will be produced. After receiving an affirmative response from the halting algorithm, the result could be obtained by running *Prog* with the input file *inpt*. Unfortunately, the Halting Problem for C Programs, like its counterpart for Turing machines, is undecidable.

Throughout the first four sections of this chapter, we will consider Turing machines with input alphabet $\{0, 1\}$ and tape alphabet $\{0, 1, B\}$. The restriction on the alphabets imposes no limitation on the computational capabilities of Turing machines since the computation of an arbitrary Turing machine *M* can be simulated by a machine with these restricted alphabets. The simulation requires encoding the symbols of *M* as strings over $\{0, 1\}$. This is precisely the approach employed by digital computers, which use the ASCII (American Standard Code for Information Interchange), EBCDIC (Extended Binary Coded Decimal Interchange Code), or Unicode encodings to represent characters as binary strings.

12.1 The Halting Problem for Turing Machines

The most famous of the undecidable problems is concerned with the properties of Turing machines themselves. The Halting Problem may be formulated as follows: Given an arbitrary Turing machine *M* with input alphabet Σ and a string $w \in \Sigma^*$, will the computation of *M* with input w halt? We will show that there is no algorithm that solves the Halting Problem. The undecidability of the Halting Problem is one of the fundamental results in the theory of computer science.

It is important to understand the statement of the problem. We may be able to determine that a particular Turing machine will halt for a given string. In fact, the exact set of strings for which a Turing machine halts may be known. For example, the machine in Example 8.3.1 halts for all and only the strings that contain *aa* as a substring. A solution to the Halting Problem, however, requires a general algorithm that answers the halting question for every possible combination of Turing machine and input string.

Since the Halting Problem asks a question about a Turing machine, the input must contain a Turing machine, or more precisely the representation of a Turing machine. We will use the Turing machine representation developed in Section 11.5, which encodes a Turing machine with input alphabet $\{0, 1\}$ as a string over $\{0, 1\}$. The proof of the undecidability of the Halting Problem does not depend upon the features of this particular encoding. The argument is valid for any representation that encodes a Turing machine as a string over its input alphabet. As before, the representation of a machine *M* is denoted $R(M)$.

The proof of the undecidability of the Halting Problem for Turing machines shows that there is a Turing machine *H* that, given a representation of another Turing machine *M* and an input string *w*, can determine whether *M* halts on *w*. By making simple modifications to *H*, it is possible to construct a machine that, given a representation of a Turing machine *M*, can determine whether *M* halts on itself. This leads to a contradiction; an impossible situation is reached because a machine cannot accept its own representation as input. Therefore, the Halting Problem for Turing machines is undecidable.

Theorem 12.1.1

The Halting Problem for Turing machines is undecidable.

Proof. Assume that the Halting Problem for Turing machines is decidable. Then there is a machine *H* that accepts the language $L = \{(R(M), w) \mid M \text{ halts on } w\}$.

- i) z consists of the representation of a machine *M* followed by a blank tape symbol *B*.
- ii) the computation of *M* on *z* halts.

If either of these conditions holds, then *H* accepts $(R(M), z)$. The machine *H* is depicted by the diagram below.

$R(M)w$

The machine *H* is shown with a start state s_0 . It has three states: s_0 , s_1 , and s_2 . Transitions are labeled with pairs of symbols. The transitions are: $s_0 \xrightarrow{0, 0} s_1$, $s_0 \xrightarrow{1, 1} s_1$, $s_1 \xrightarrow{0, 0} s_2$, $s_1 \xrightarrow{1, 1} s_2$, and $s_2 \xrightarrow{B, B} s_0$. The final state is s_0 .

$R(M)w$

From this point on in the computation, the machine *H* continues to move left, reading the string $R(M)w$ from right to left. The machine *H* continues indefinitely.

The machine *H'* is constructed from *H* as follows. The input to *H'* is the string $R(M)R(M)w$. The machine *D* is a Turing machine that takes the string $R(M)R(M)w$ and creates the string $R(M)R(M)R(M)w$. The machine *H'* on input $R(M)R(M)w$ runs *H* on $R(M)w$. The machine *H'* moves to the right, reading the string $R(M)R(M)w$ from left to right. The machine *H'* continues indefinitely.

mers—the
I become a

The proof of the undecidability of the Halting Problem is by contradiction. We assume that there is a Turing machine H that solves the Halting Problem. We then make several simple modifications to H to obtain a new machine D that produces a self-referential contradiction; an impossible situation occurs when the machine D is run with its own representation as input. Since the assumption of the existence of a machine H that solves the Halting Problem produces a contradiction, the Halting Problem is not solvable.

only that a
algorithm,
ately, the
idecidable.
chines with
ts imposes
omputation
e restricted
 $\{0, 1\}$. This
(American
ed Decimal
ngs.

s of Turing
en an arbi-
omputation
the Halting
results in the

o determine
f strings for
ample 8.3.1
the Halting
on for every

input must
ine. We will
les a Turing
decidability
coding. The
ring over its

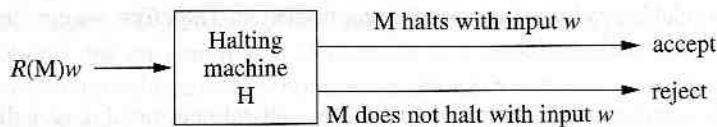
Theorem 12.1.1

The Halting Problem for Turing Machines is undecidable.

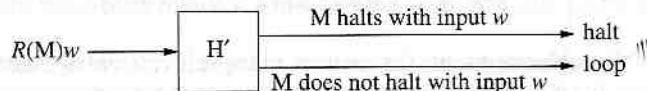
Proof. Assume that the Turing machine H solves the Halting Problem. A string $z \in \{0, 1\}^*$ is accepted by H if

- i) z consists of the representation of a Turing machine M followed by a string w and
- ii) the computation of M with input w halts.

If either of these conditions is not satisfied, H rejects the input. The operation of the machine H is depicted by the diagram

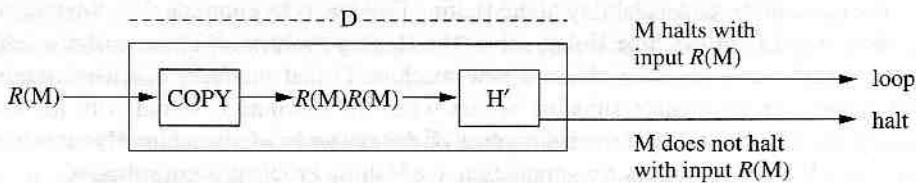


The machine H is modified to construct a new Turing machine H' . The computations of H' are the same as H except H' continues when H halts in an accepting state. At that point, H' moves to the right forever. The transition function of H' is obtained from that of H by adding transitions that cause H' to move indefinitely to the right upon entering an accepting configuration of H . The action of H' may be depicted by

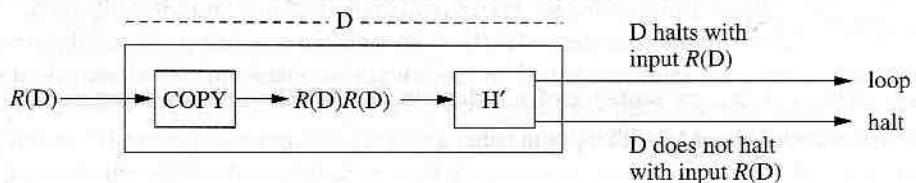


From this point on in the proof, we are concerned only with whether a computation halts or continues indefinitely. The latter case is denoted by the word *loop* in the diagrams.

The machine H' is combined with a copy machine to construct another Turing machine D . The input to D is a Turing machine representation $R(M)$. A computation of D begins by creating the string $R(M)R(M)$ from the input $R(M)$. The computation continues by running H' on $R(M)R(M)$.



The input to the machine D may be the representation of any Turing machine with alphabet $\{0, 1, B\}$. In particular, D itself is such a machine. Consider a computation of D with input $R(D)$. Rewriting the previous diagram with M replaced by D and $R(M)$ by $R(D)$, we get



Examining the diagram, we see that D halts with input $R(D)$ if, and only if, D does not halt with input $R(D)$. This is obviously impossible. However, the machine D can be constructed directly from a machine H that solves the Halting Problem. The assumption that the Halting Problem is decidable produces the preceding contradiction. Therefore, we conclude that the Halting Problem is undecidable. ■

The contradiction in the preceding proof uses self-reference and diagonalization. To obtain the standard relational table for a diagonalization argument, we consider every string $v \in \{0, 1\}^*$ to represent a Turing machine; if v does not have the form $R(M)$, the one-state Turing machine with no transitions is assigned to v . Thus the Turing machines can be listed $M_0, M_1, M_2, M_3, M_4, \dots$ corresponding to strings $\lambda, 0, 1, 00, 01, \dots$. Now consider a table that lists the Turing machines along the horizontal and vertical axes. The i, j th entry of the table is

$$\begin{cases} 1 & \text{if } M_i \text{ halts when run with } R(M_j) \\ 0 & \text{if } M_i \text{ does not halt when run with } R(M_j). \end{cases}$$

The diagonal of the table represents the answers to the self-referential question, "Does M_i halt when run on itself?" The machine D was constructed to produce a contradiction in response to that question.

A similar argument can be used to establish the undecidability of the Halting Problem for Turing Machines with arbitrary alphabets. The essential feature of this approach is the ability to encode the transitions of a Turing machine as a string over its own input alphabet. Two symbols are sufficient to construct such an encoding.

The undecidability of the Halting Problem and the ability of the universal machine to simulate computations of Turing machines combine to show that the recursive languages are

a proper subset of the
of the undecidability

Corollary 12.1.2

The language $L_H = \{$
halts with input $w\}$ is

Corollary 12.1.3

The recursive language

Proof. The universal form $R(M)w$ and M machine demonstrates that L_H is not recursive.

In Exercise 8.26

Corollary 12.1.4

The language $\overline{L_H}$ is no

Corollary 12.1.4 tells us that strings of the language can be used to detect patterns that are not recursively enumerable. This is too complex to be detected.

12.2 Problem

Reduction was introduced to solve problems. A decision rule r that transforms instances into answers is called a reduction from problem P to problem Q .

Reduction

to describe the compo-

Reduction has improved.

a proper subset of the recursively enumerable languages. Corollary 12.1.2 is the restatement of the undecidability of the Halting Problem in the terminology of recursive languages.

- loop
- halt

nine with
tion of D
by $R(D)$,

- loop
- halt

es not halt
onstructed
he Halting
ide that the
■

ization. To
every string
e one-state
an be listed
ider a table
entry of the
■

i, “Does M_i
radiction in

ing Problem
roach is the
ut alphabet.

l machine to
nguages are

Corollary 12.1.2

The language $L_H = \{R(M)w \mid R(M) \text{ is the representation of a Turing machine } M \text{ and } M \text{ halts with input } w\}$ over $\{0, 1\}^*$ is not recursive.

Corollary 12.1.3

The recursive languages are a proper subset of the recursively enumerable languages.

Proof. The universal machine U accepts L_H ; a string is accepted by U only if it is of the form $R(M)w$ and M halts when run with input w . The acceptance of L_H by the universal machine demonstrates that L_H is recursively enumerable, while Corollary 12.1.2 established that L_H is not recursive. ■

In Exercise 8.26 it was shown that a language L is recursive if both L and \bar{L} are recursively enumerable. Combining this with Corollary 12.1.2 yields

Corollary 12.1.4

The language \bar{L}_H is not recursively enumerable.

Corollary 12.1.4 tells us that there is no algorithm that can either accept or recognize the strings of the language \bar{L}_H . From a pattern recognition perspective, machines are designed to detect patterns that are common to all elements in a set of strings. When a language is not recursively enumerable, any common pattern among the elements of the language is too complex to be detected algorithmically.

12.2 Problem Reduction and Undecidability

Reduction was introduced in Chapter 11 as a tool for constructing solutions to decision problems. A decision problem P is reducible to Q if there is a Turing computable function r that transforms instances of P into instances of Q , and the transformation preserves the answer to the problem instance of P . As in Chapter 11, we will use a table of the form

Reduction	Input	Condition
P	instances p_0, p_1, \dots	the answer to p_i is yes
to	$\downarrow r$	if, and only if,
Q	instances q_0, q_1, \dots	the answer to $r(p_i)$ is yes

to describe the components and conditions of a reduction of P to Q .

Reduction has important implications for undecidability as well for decidability. If P is undecidable and reducible to a problem Q , then Q must also be undecidable. If Q were

decidable, combining the reduction of P to Q with the algorithm that solves Q produces a decision procedure for P as follows: For an input p_i to P

- i) Use the reduction to transform p_i to $r(p_i)$.
- ii) Use the algorithm for Q to determine the answer for $r(p_i)$.

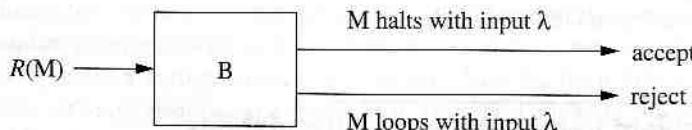
Since r is a reduction, the answer to the decision problem P for input p_i is the same as the answer to $r(p_i)$ for problem Q . The sequential execution of the reduction and the algorithm that solves Q produces a solution to P . This is a contradiction since P was known to be undecidable. Consequently, our assumption that Q is decidable must be false.

The *Blank Tape Problem* is the problem of deciding whether a Turing machine halts when a computation is initiated with a blank tape. The Blank Tape Problem is a special case of the Halting Problem since it is concerned only with the question of halting when the input is the null string. We will show that the Halting Problem is reducible to the Blank Tape Problem and, consequently, that the Blank Tape Problem is undecidable.

Theorem 12.2.1

There is no algorithm that determines whether an arbitrary Turing machine halts when a computation is initiated with a blank tape.

Proof. Assume that there is a machine B that solves the Blank Tape Problem. Such a machine can be represented

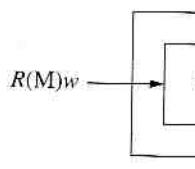


The reduction of the Halting Problem to the Blank Tape Problem is accomplished by a machine R . The input to R is the representation of a Turing machine M followed by an input string w . The result of a computation of R is the representation of a machine M' that

1. writes w on a blank tape,
2. returns the tape head to the initial position with the machine in the start state of M , and
3. runs M .

$R(M')$ is obtained by adding encoded transitions to $R(M)$ and suitably renaming the start state of M . The machine M' has been constructed so that it halts when run with a blank tape if, and only if, M halts with input w .

A new machine is constructed by adding R as a preprocessor to B . Sequentially running the machines R and B produces the composite machine



Tracing a computation, the preprocessor R reduces the Halting Problem to the Blank Tape Problem is undecidable.

The preprocessor R reduces the Halting Problem to the Blank Tape Problem, which is a representation of a Turing machine M that performs by the preprocessor R .

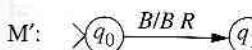
Example 12.2.1

Let M be the Turing machine

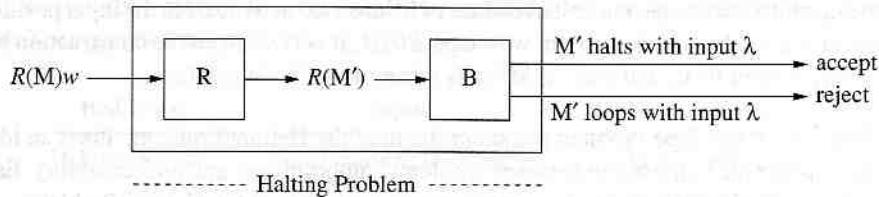
that halts whenever the input string

0001011101

With input $R(M)0001011101$, the machine M' halts.



uces a



as the
orithm
1 to be
dered.

e halts
special
; when
Blank

Tracing a computation, we see that the composite machine solves the Halting Problem. Since the preprocessor R reduces the Halting Problem to the Blank Tape Problem, the Blank Tape Problem is undecidable. ■

The preprocessor R, which performs the reduction of the Halting Problem to the Blank Tape Problem, modifies the representation of a Turing machine M to construct the representation of a Turing machine M'. Example 12.2.1 shows the result of a transformation performed by the preprocessor R.

Example 12.2.1

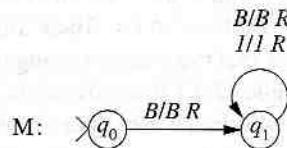
Let M be the Turing machine

Such a

ed by a
d by an
M' that

M, and

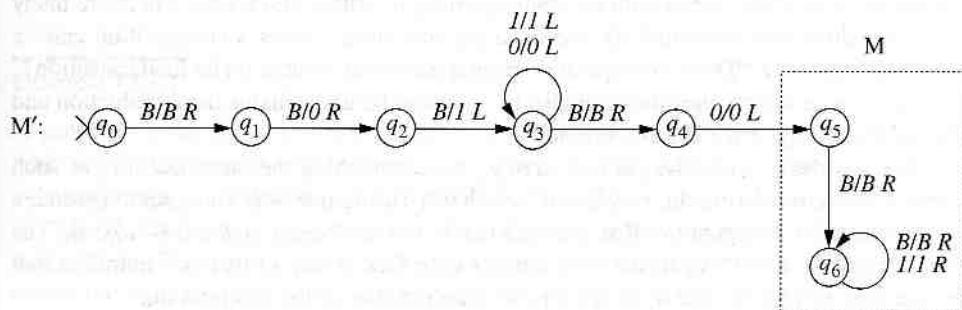
the start
ank tape
running



that halts whenever the input string contains 0. The encoding $R(M)$ of M is

$00010111011011101100110111011011101100110110110110110000$.

With input $R(M)01$, the preprocessor R constructs the encoding of the Turing machine M'.



When run with a blank tape, the first five states of M' are used to write 01 in the input position. A copy of the machine M is then run with tape $B01B$. It is clear from the construction that M halts with input 01 if, and only if, M' halts when run with a blank tape. \square

Since the Blank Tape Problem is a subproblem of the Halting Problem, this is an ideal time to consider the relationship between problems, subproblems, and undecidability. Each of following problems is obtained by fixing one of the inputs of the Halting Problem:

Subproblem	Input	Decidable?
Blank Tape Problem	$R(M)$, (input string fixed)	Undecidable
Halting of the universal machine U	$(\text{machine fixed}), R(M)w$	Undecidable
Halting of M from Example 8.3.1	$(\text{machine fixed}), w$	Decidable

The Halting Problem for the universal machine asks if U will halt with input $R(M)w$. A solution to this problem would determine if an arbitrary Turing machine M halts with input w and thus provide a solution to the Halting Problem. The preceding table shows that subproblems of an undecidable problem may or may not be undecidable depending upon which features of the problem are retained. On the other hand, if Q is a subproblem of a decision problem P and Q is undecidable, then P is necessarily undecidable; any algorithm that solves P is automatically a solution to all of its subproblems.

The reduction of the Halting Problem to the Blank Tape Problem was accomplished by a Turing computable function r that transformed strings of the form $R(M)w$ to a string $R(M')$. Theorem 12.2.1 and Example 12.2.1 showed how the Turing machine representation $R(M)$ is modified to produce $R(M')$. In the remainder of examples, we will give a high-level explanation of the reduction and omit the details of the manipulation of the string representations.

12.3 Additional Halting Problem Reductions

We have shown that there is no algorithm that determines whether a Turing machine computation will halt, either with an arbitrary string or with a blank tape. There are many other questions that we could ask about Turing machines: “Does a computation enter a particular state?” Or “Does a computation print a particular symbol on its final transition?” And so on. Many such questions can also be shown to be undecidable using reduction and the undecidability of the Halting Problem.

We will demonstrate the general strategy for establishing the undecidability of such questions by considering the problem of whether a Turing machine computation reenters its start state. A computation that reenters the start state begins $q_0BwB \not\models uq_0vB$. The computation need not halt in the start state or even halt at all; all that is required is that the machine returns to state q_0 at some point after the start of the computation.

We will show that to it. The reduction ha

Reduction

Halting Problem

to

Reenter Problem

As indicated, we will Problem and the mach

Let $M = (Q, \Sigma, \delta, q_0, F)$, construct a machine M' that halts when run with w . The computation is in no way related to the original machine M except for the fact that M' is designed to reenter its start state.

The idea behind the reduction is to use a state q'_0 that has the same transitions as q_0 but configuration of M . For example, if q_0 has a transition to q_1 on input a , then q'_0 also has a transition to q'_1 on input a .

with q'_0 the start state of M' . The computation of M' takes the same number of steps as the computation of M . If the transition to q'_0 is never taken, then the computation of M' does not reenter its start state and is therefore undecidable.

Example 12.3.1

A proof by contradiction. Suppose that there is an algorithm that decides whether an arbitrary Turing machine A enters its start state. Let M be a Turing machine A with start state q_0 and final state q_f . Let $v \in \{0, 1\}^*$. The input is a string v consisting of n input strings. The input is v and the representation of v is $R(v)$.

osition.
ion that

□

an ideal
ty. Each
em:

le?

able
able
le

$R(M)w$.
alts with
nows that
ing upon
lem of a
lgorithm

mplished
o a string
resentation
e a high-
the string

: machine
are many
on enter a
ansition?"
uction and

ity of such
on reenters
 q_0vB . The
ired is that

We will show that the Reenter Problem is undecidable by reducing the Halting Problem to it. The reduction has the form

Reduction	Input	Condition
Halting Problem to Reenter Problem	Turing machine M, string w ↓ Turing machine M' , string w	M halts with input w if, and only if, M' reenters its start state when run with w

As indicated, we will use the same string w as the input for the machine M in the Halting Problem and the machine M' in the Reenter Problem.

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ and w be an instance of the Halting Problem. We must construct a machine M' that reenters its start state when run with w if, and only if, M halts when run with w . First we note that, in an arbitrary Turing machine, the halting of a computation is in no way connected to whether the computation reenters the start state. In designing the reduction, it is our task to connect them.

The idea behind the construction of the machine M' is to start with M , add a new start state q'_0 that has the same transitions as q_0 , and add a transition to q'_0 for every halting configuration of M . Formally, M' is defined from the components of M :

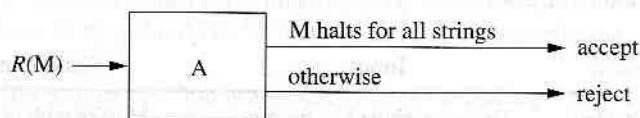
$$\begin{aligned} Q' &= (Q \cup \{q'_0\}), \Sigma' = \Sigma, \Gamma' = \Gamma, F' = F \\ \delta'(q_i, x) &= \delta(q_i, x) \text{ if } \delta(q_i, x) \text{ is defined} \\ \delta'(q'_0, x) &= \delta(q_0, x) \text{ for all } x \in \Gamma \\ \delta'(q_i, x) &= [q'_0, x, R] \text{ if } \delta(q_i, x) \text{ is undefined} \end{aligned}$$

with q'_0 the start state of M' . If the computation of M halts with input w , the corresponding computation of M' takes one additional transition and reenters q'_0 . If M does not halt, a transition to q'_0 is never taken and M' does not reenter its start state. The construction transforms the question of whether M halts with input w to the question of whether M' reenters its start state when run with w . It follows that the Reenter Problem is also undecidable.

Example 12.3.1

A proof by contradiction is used to show that the problem of determining whether an arbitrary Turing machine halts for all input strings is undecidable. Assume that there is a Turing machine A that solves this problem. The input to such a machine is a string $v \in \{0, 1\}^*$. The input is accepted if $v = R(M)$ for some Turing machine M that halts for all input strings. The input is rejected if either v is not the representation of a Turing machine or it is the representation of a machine that does not halt for some input string.

The computation of machine A can be depicted by



Problem reduction is used to create a solution to the Halting Problem from the machine A. It follows that the 'halts for all strings' problem is undecidable.

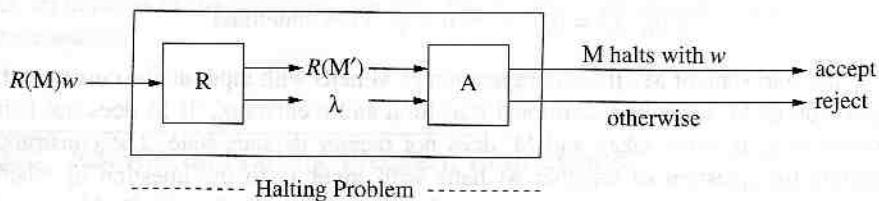
The language of the Halting Problem consists of strings of the form $R(M)w$, where the machine M halts when run with input w . The reduction is accomplished by a machine R. The first action of R is to determine whether the input string has the expected format of the representation of some Turing machine M followed by a string w . If the input does not have this form, R erases the input, leaving the tape blank.

When the input has the form $R(M)w$, the computation of R constructs the encoding of a machine M' that, when run with any input string y ,

1. erases y from the tape,
2. writes w on the tape, and
3. runs M on w .

$R(M')$ is obtained from $R(M)$ by adding the encoding of two sets of transitions: one set that erases the input that is initially on the tape and another set that then writes the w in the input position. The machine M' has been constructed to completely ignore its input. Every computation of M' halts if, and only if, the computation of M with input w halts.

The machine consisting of the combination of R and A



provides a solution to the Halting Problem. If the input does not have the form $R(M)w$, the null string is produced by R and subsequently rejected by A. Otherwise R generates $R(M')$. Tracing the sequential operation of the machines, the input is accepted if, and only if, it is the representation of a Turing machine M that halts when run with w .

Since the Halting Problem is undecidable and the reduction machine R is constructible, we conclude that there is no machine A that solves the 'halts for all strings' problem. \square

The relationship between Turing machines and unrestricted grammars developed in Section 10.1 can be used to convert undecidability results from the domain of machines to the domain of grammars. Consider the problem of deciding whether a string w is generated by

an unrestricted grammar. This problem has the form

Reduction

Halting Prob

to

Derivability Prob

Let M be a Turing machine. The problem is to modify M to obtain a machine that accepts the same language. This is accomplished by modifying M so that every computation that accepts are synonymous.

Using Theorem 12.1, we can construct an algorithm that decides whether M halts when run with any input string w . Thus no machine M exists that decides the Halting Problem.

12.4 Rice's Theorem

In the preceding section, we saw how to reduce the Halting Problem to answer certain questions about languages. A classic example of this was the proof that the Halting Problem is undecidable. We asked, "Given a machine M, does M halt when run with any input string?" In each of these reductions, the problem being reduced to was concerned with the Halting Problem.

Rather than asking whether a machine M halts when run with any input string, we will now focus on more specific properties of machines. We say that a machine satisfies a property if there is an algorithm that, given a machine M, answers the question "Does M satisfy the property?" In each of these reductions, the problem being reduced to was concerned with a specific property of machines.

- i) Is λ in $L(M)$?
- ii) Is $L(M) = \emptyset$?
- iii) Is $L(M)$ a regular language?
- iv) Is $L(M) = \Sigma^*$?

The ability to encode these properties into machines allows us to reduce the preceding questions to the Halting Problem. Given a machine M and an encoding, a set of Turing machines $\{M_i\}$ is said to be decidable if there is an algorithm that, given a string w , answers the question "Does M_i accept w ?"

an unrestricted grammar G . A reduction that establishes the undecidability of the derivability problem has the form

Reduction	Input	Condition
Halting Problem to Derivability Problem	Turing machine M , string w ↓ unrestricted grammar G , string w	M halts with input w if, and only if, there is a derivation $S \xrightarrow{*} w$ in G

Let M be a Turing machine and w an input string for M . The first step in the reduction is to modify M to obtain a machine M' that accepts every string for which M halts. This is accomplished by making every state of M an accepting state in M' . In M' , halting and accepting are synonymous.

Using Theorem 10.1.3, we can construct a grammar $G_{M'}$ with $L(G_{M'}) = L(M')$. An algorithm that decides whether $w \in L(G_{M'})$ also determines whether the computation of M' (and M) halts. Thus no such algorithm is possible.

12.4 Rice's Theorem

In the preceding sections we have shown that it is impossible to construct an algorithm to answer certain questions about a computation of an arbitrary Turing machine. The first example of this was the Halting Problem, which posed the question, “Will a Turing machine M halt when run with input w ?”. Problem reduction allowed us to establish that there is no algorithm that answers the question, “Will a Turing machine M halt when run with a blank tape?”. In each of these problems, the input contained a Turing machine and the decision problem was concerned with determining the result of the computation of the machine.

Rather than asking about the computation of a Turing machine with a particular input string, we will now focus on determining whether the language accepted by a Turing machine satisfies a prescribed property. For example, we might be interested in the existence of an algorithm that, when given a Turing machine M as input, produces an answer to questions of the form

- i) Is λ in $L(M)$?
- ii) Is $L(M) = \emptyset$?
- iii) Is $L(M)$ a regular language?
- iv) Is $L(M) = \Sigma^*$?

The ability to encode Turing machines as strings over $\{0, 1\}$ permits us to transform the preceding questions into questions about membership in a language. Employing the encoding, a set of Turing machines defines a language over $\{0, 1\}$ and the question of whether the set of strings accepted by a Turing machine M satisfies a property can be posed

as a question of membership $R(M)$ in the appropriate language. For example, the question, “Is $L(M) = \emptyset$?” can be rephrased in terms of membership as, “Is $R(M) \in L_\emptyset$?” Using this approach, the languages associated with the previous questions are

- i) $L_\lambda = \{R(M) \mid \lambda \in L(M)\}$
- ii) $L_\emptyset = \{R(M) \mid L(M) = \emptyset\}$
- iii) $L_{\text{reg}} = \{R(M) \mid L(M) \text{ is regular}\}$
- iv) $L_{\Sigma^*} = \{R(M) \mid L(M) = \Sigma^*\}$.

Example 12.3.1 showed that the question of membership in L_{Σ^*} is undecidable. That is, there is no algorithm that decides whether a Turing machine halts for all (and accepts) input strings.

The reduction strategy employed in Example 12.3.1 can be generalized to show that many languages consisting of representations of Turing machines are not recursive. A property \mathbb{P} of recursively enumerable languages describes a condition that a recursively enumerable language may satisfy. For example, \mathbb{P} may be “The language contains the null string”; “The language is the empty set”; “The language is regular”; or “The language contains all strings.” The language of a property \mathbb{P} is defined by $L_{\mathbb{P}} = \{R(M) \mid L(M) \text{ satisfies } \mathbb{P}\}$. Thus L_\emptyset , the language associated with the property “The language is the empty set” consists of the representations of all Turing machines that do not accept any strings.

A property \mathbb{P} of recursively enumerable languages is called *trivial* if there are no recursively enumerable languages that satisfy \mathbb{P} or if every recursively enumerable language satisfies \mathbb{P} . For a trivial property, $L_{\mathbb{P}}$ is either the empty set or consists of all representations of Turing machines. Membership in both of these languages is decidable. Rice’s Theorem shows that any property that is satisfied by some, but not all, recursively enumerable languages is undecidable.

Theorem 12.4.1 (Rice’s Theorem)

If \mathbb{P} is a nontrivial property of recursively enumerable languages, then $L_{\mathbb{P}}$ is not recursive.

Proof. Let \mathbb{P} be a nontrivial property that is not satisfied by the empty language. We will show that $L_{\mathbb{P}} = \{R(M) \mid L(M) \text{ satisfies } \mathbb{P}\}$ is not recursive.

Since $L_{\mathbb{P}}$ is nontrivial, there is at least one language $L \in L_{\mathbb{P}}$. Moreover, L is not \emptyset by the assumption that the empty language does not satisfy \mathbb{P} . Let M_L be a Turing machine that accepts L .

The reducibility of the Halting Problem to $L_{\mathbb{P}}$ will be used to show that $L_{\mathbb{P}}$ is not recursive. As in Example 12.3.1, a preprocessor R will be designed to transform input $R(M)w$ into the encoding of a machine M' . The action of M' when run with input y is to

1. write w to the right of y , producing $ByBwB$;
2. run the transitions of M on w ; and
3. if M halts when run with w , then run M_L with input y .

The role of the machine M and the string w is that of a gatekeeper. The processing of the input string y by M_L is allowed only if M halts with input w .

If the computation of $R(M)w$ halts with y . In this case the machine M' will accept y . The computation of $R(M)w$ halts with y . Thus M' accepts y .

The machine M' accepts y if and only if M halts with w . Since M halts with w if and only if $R(M)w$ halts with y . Thus M' accepts y if and only if $R(M)w$ halts with y .

Now assume that $R(M)w$ halts with y . Then M' accepts y .

$R(M)w \rightarrow y$

Consequently, the problem $R(M)w$ is undecidable.

Originally, we assumed that $R(M)w$ halts with y . By the preceding argument can conclude that $L_{\mathbb{P}}$ must also be undecidable.

Rice’s Theorem shows that any nontrivial property of recursively enumerable languages is undecidable. We will illustrate the proof of Rice’s Theorem with an example.

Example 12.4.1

The problem of determining whether a context-free grammar is context-free is undecidable. Recall that a grammar is context-free if and only if “is context-free” is a nontrivial property of context-free grammars. This result was accomplished by finding a many-one reduction from the Halting Problem to the problem of determining whether a grammar is context-free.

12.5 An Unsolved Problem

Semi-Thue Systems, named after the Norwegian mathematician Thoralf Skolem, are a special type of grammar. A semi-Thue system is a triple (A, Σ, R) , where A is a set of nonterminals, Σ is a set of terminals, and R is a set of rules of the form $u \rightarrow v$, where u and v are strings over Σ .

question,
sing this

. That is,
pts) input

show that
ursive. A
recursively
is the null
language
 \emptyset satisfies
empty set”
ngs.
ere are no
e language
esentations
s Theorem
numerable

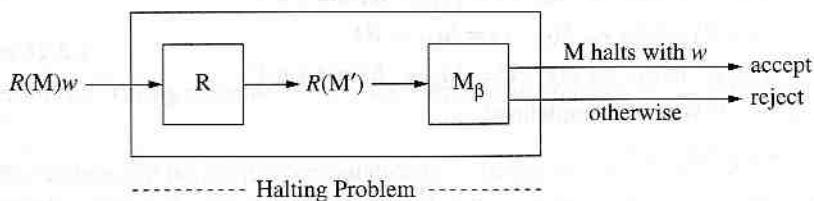
t recursive.
ge. We will
is not \emptyset by
ng machine
is not recur-
put $R(M)w$
to

essing of the

If the computation of M halts when run with w , then M_L is allowed to process input y . In this case the result of a computation of M' with an input string y is exactly that of the computation of M_L with y . Consequently, $L(M') = L(M_L) = L$ and $L(M')$ satisfies \mathbb{P} . If the computation of M does not halt when run with w , then M' never halts regardless of the input string y . Thus no string is accepted by M' and $L(M') = \emptyset$, which does not satisfy \mathbb{P} .

The machine M' accepts \emptyset when M does not halt with input w , and M' accepts L when M halts with w . Since L satisfies \mathbb{P} and \emptyset does not, $L(M')$ satisfies \mathbb{P} if, and only if, M halts when run with input w .

Now assume that $L_{\mathbb{P}}$ is recursive. Then there is a machine $M_{\mathbb{P}}$ that decides membership in $L_{\mathbb{P}}$. The machines R and $M_{\mathbb{P}}$ combine to produce a solution to the Halting Problem.



Consequently, the property \mathbb{P} is not decidable.

Originally, we assumed that \mathbb{P} was not satisfied by the empty set. If $\emptyset \in L_{\mathbb{P}}$, the preceding argument can be used to show that $L_{\mathbb{P}}$ is not recursive. It follows from Exercise 8.26 that $L_{\mathbb{P}}$ must also be nonrecursive. ■

Rice's Theorem makes it easy to demonstrate the undecidability of many questions about properties of languages accepted by Turing machines, as is seen in the following example.

Example 12.4.1

The problem of determining whether the language accepted by a Turing machine is context-free is undecidable. By Rice's Theorem, all that is necessary is to show that the property “is context-free” is a nontrivial property of recursively enumerable languages. This is accomplished by finding one recursively enumerable language that is context-free and another that is not. The languages \emptyset and $\{a^i b^i c^i \mid i \geq 0\}$ are both recursively enumerable; the former is context-free, and the latter is not (Example 7.4.1). □

12.5 An Unsolvable Word Problem

Semi-Thue Systems, named after their originator Norwegian mathematician Axel Thue, are a special type of grammar consisting of a single alphabet Σ and a set P of rules. A rule has the form $u \rightarrow v$, where $u \in \Sigma^+$ and $v \in \Sigma^*$. There is no division of the symbols into variables and terminals, nor is there a designated start symbol. The Word Problem for Semi-Thue

Systems is the problem of determining, for an arbitrary Semi-Thue System $S = (\Sigma, P)$ and strings $u, v \in \Sigma^*$, whether v is derivable from u in S . We will show that the Halting Problem is reducible to the Word Problem. The reduction is obtained by establishing a relationship between Turing machine computations and derivations in appropriately designed Semi-Thue Systems.

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a deterministic Turing machine. Using a modification of the construction presented in Theorem 10.1.3, we can construct a Semi-Thue System $S_M = (\Sigma_M, P_M)$ whose derivations simulate the computations of M . The alphabet of S_M is the set $Q \cup \Gamma \cup \{[,], q_f, q_R, q_L\}$. The set P_M of rules of S_M is defined by

1. $q_i xy \rightarrow zq_j y$ whenever $\delta(q_i, x) = [q_j, z, R]$ and $y \in \Gamma$
2. $q_i x] \rightarrow zq_j B]$ whenever $\delta(q_i, x) = [q_j, z, R]$
3. $yq_i x \rightarrow q_j yz$ whenever $\delta(q_i, x) = [q_j, z, L]$ and $y \in \Gamma$
4. $q_i x \rightarrow q_R$ if $\delta(q_i, x)$ is undefined
5. $q_R x \rightarrow q_R$ for $x \in \Gamma$
6. $q_R] \rightarrow q_L]$
7. $xq_L \rightarrow q_L$ for $x \in \Gamma$
8. $[q_L \rightarrow [q_f]$.

The rules that generate the string $[q_0 B w]$ in Theorem 10.1.3 are omitted since the Word Problem for a Semi-Thue System is concerned with derivability of a string v from another string u , not from a distinguished starting configuration. The erasing rules (5 through 8) have been modified to generate the string $[q_f]$ whenever the computation of M with input w halts.

The simulation of a computation of M in S_M manipulates strings of the form $[uqv]$ with $u, v \in \Gamma^*$, and $q \in Q \cup \{q_f, q_R, q_L\}$. Lemma 12.5.1 lists several important properties of derivations of S_M that simulate a computation of M .

Lemma 12.5.1

Let M be a deterministic Turing machine, S_M be the Semi-Thue System constructed from M , and $w = [uqv]$ be a string with $u, v \in \Gamma^*$, and $q \in Q \cup \{q_f, q_R, q_L\}$.

- i) There is at most one string z such that $\underset{S_M}{w} \Rightarrow z$.
- ii) If there is such a z , then z also has the form $[u' q' v']$ with $u', v' \in \Gamma^*$, and $q' \in Q \cup \{q_f, q_R, q_L\}$.

Proof. The application of a rule replaces one instance of an element of $Q \cup \{q_f, q_R, q_L\}$ with another. The determinism of M guarantees that there is at most one rule in P_M that can be applied to $[uqv]$ whenever $q \in Q$. If $q = q_R$ there is a unique rule that can be applied to $[uq_Rv]$. This rule is determined by the first symbol in the string $v]$. Similarly, there is only

one rule that can be applied to a string containing q_L .

Condition (ii) follows.

A computation of M with input w halts if and only if $[q_f]$ is derived from $[uqv]$.

The erasure rules trap the computation of M in a loop. See Lemma 12.5.2.

Lemma 12.5.2

A deterministic Turing machine M with input w halts if and only if $[q_f]$ is derived from $[uqv]$.

The relationship between the computation of M and the corresponding Semi-Thue system S_M is given by the following lemma.

Example 12.5.1

The language of the Turing machine M is $0^* I(0 \cup I)^*$. The rule

is $0^* I(0 \cup I)^*$. The rule

$q_0 \rightarrow q_0$
 $q_1 \rightarrow q_1$
 $q_1 \rightarrow q_1$
 $q_1 \rightarrow q_1$
 $q_1 \rightarrow q_1$
 $q_2 \rightarrow q_2$
 $q_2 \rightarrow q_2$
 $q_2 \rightarrow q_2$
 $q_2 \rightarrow q_2$
 $q_3 \rightarrow q_3$
 $q_3 \rightarrow q_3$
 $q_3 \rightarrow q_3$
 $q_3 \rightarrow q_3$

P) and
problem
ionship
1 Semi-

fication
System
of S_M is

one rule that can be applied to $[uq_L]$. Finally, there are no rules in P_M that can be applied to a string containing q_f .

Condition (ii) follows immediately from the form of the rules of P_M . ■

A computation of M that halts with input w produces a derivation

$$[q_0 B w B] \xrightarrow{S_M}^* [uq_R v].$$

The erasure rules transform this string to $[q_f]$. These properties are combined to yield Lemma 12.5.2.

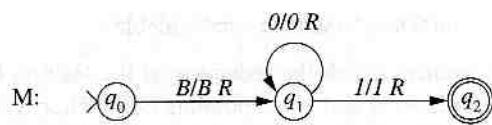
Lemma 12.5.2

A deterministic Turing machine M halts with input w if, and only if, $[q_0 B w B] \xrightarrow{S_M}^* [q_f]$.

The relationship between a computation of a Turing machine and a derivation in the corresponding Semi-Thue System is illustrated in the following example.

Example 12.5.1

The language of the Turing machine



is $0^* 1(0 \cup 1)^*$. The rules of the corresponding Semi-Thue System S_M are

$$\begin{array}{lll} q_0 B B \rightarrow B q_1 B & q_1 0 B \rightarrow 0 q_1 B & q_1 1 B \rightarrow 1 q_2 B \\ q_0 B 0 \rightarrow B q_1 0 & q_1 0 0 \rightarrow 0 q_1 0 & q_1 1 0 \rightarrow 1 q_2 0 \\ q_0 B 1 \rightarrow B q_1 1 & q_1 0 1 \rightarrow 0 q_1 1 & q_1 1 1 \rightarrow 1 q_2 1 \\ q_0 B] \rightarrow B q_1 B] & q_1 0] \rightarrow 0 q_1 B] & q_1 1] \rightarrow 1 q_2 B] \end{array}$$

the Word
1 another
rough 8)
ith input

rm $[uqv]$
roperties

cted from

and $q' \in$

$f, q_R, q_L\}$
 M that can
applied to
ere is only

$$\begin{array}{lll} q_0 0 \rightarrow q_R & q_R B \rightarrow q_R & B q_L \rightarrow q_L \\ q_0 1 \rightarrow q_R & q_R 0 \rightarrow q_R & 0 q_L \rightarrow q_L \\ q_1 B \rightarrow q_R & q_R 1 \rightarrow q_R & 1 q_L \rightarrow q_L \\ q_2 B \rightarrow q_R & q_R] \rightarrow q_L] & [q_L \rightarrow [q_f \\ q_2 0 \rightarrow q_R & & \\ q_2 1 \rightarrow q_R & & \end{array}$$

The computation of M that accepts 011 is given with the associated derivation of $[q_f]$ from $[q_0B011B]$ in the Semi-Thue System S_M .

$$\begin{array}{ll}
 q_0B011B & [q_0B011B] \\
 \vdash Bq_1011B & \Rightarrow [Bq_1011B] \\
 \vdash B0q_111B & \Rightarrow [B0q_111B] \\
 \vdash B01q_21B & \Rightarrow [B01q_21B] \\
 & \Rightarrow [B01q_R B] \\
 & \Rightarrow [B01q_R] \\
 & \Rightarrow [B01q_L] \\
 & \Rightarrow [B0q_L] \\
 & \Rightarrow [Bq_L] \\
 & \Rightarrow [q_L] \\
 & \Rightarrow [q_f]
 \end{array} \quad \square$$

The ability to simulate the computations of a Turing machine with derivations of a Semi-Thue System provides the basis for establishing the undecidability of the Word Problem for Semi-Thue Systems.

Theorem 12.5.3

The Word Problem for Semi-Thue Systems is undecidable.

Proof. The preceding lemmas sketch the reduction of the Halting Problem to the Word Problem. For a Turing machine M and corresponding Semi-Thue System S_M , the computation of M with input w halting is equivalent to the derivability of $[q_f]$ from $[q_0BwB]$ in S_M . An algorithm that solves the Word Problem could also be used to solve the Halting Problem. ■

By Theorem 12.5.3, there is no algorithm that solves the Word Problem for an arbitrary Semi-Thue System $S = (\Sigma, P)$ and pair of strings in Σ^* . The relationship between the computations of a Turing machine M and derivations of S_M developed in Lemma 12.5.2 can be used to prove that there are particular Semi-Thue Systems whose word problems are undecidable.

Theorem 12.5.4

Let M be a deterministic Turing machine that accepts a nonrecursive language. The Word Problem for the Semi-Thue System S_M is undecidable.

Proof. Since M recognizes a nonrecursive language, the Halting Problem for M is undecidable (Exercise 3). The correspondence between computations of M and derivations of S_M yields the undecidability of the Word Problem for this system. ■

12.6 The Post Correspondence Problem

The undecidable problem is the properties of Turing machines. The Post Correspondence Problem is a simple game of manipulating strings over a fixed alphabet, one on the left and one on the right.

A Post correspondence system is defined as follows:

The game begins with two rows of dominoes. One row is then placed to the immediate left of the other, producing a sequence of dominoes of each type.

A string is obtained by reading off the bottom dominoes. We refer to this string as the bottom string. The objective is to find a sequence of identical top and bottom dominoes.

The sequence

spells $acbbaacb$ in both directions.

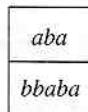
Formally, a Post correspondence system is an ordered pairs (u_i, v_i) , $i = 1, 2, \dots, n$. The sequence of strings u_1, u_2, \dots, u_n is called the top string and the sequence of strings v_1, v_2, \dots, v_n is called the bottom string. The Post correspondence problem is to determine if there is a sequence of indices i_1, i_2, \dots, i_k such that $u_{i_1}u_{i_2}\dots u_{i_k} = v_{i_1}v_{i_2}\dots v_{i_k}$.

The problem of determining whether a given Post Correspondence Problem has a solution is undecidable.

n of $[q_f]$

12.6 The Post Correspondence Problem

The undecidable problems examined in the preceding sections have been concerned with the properties of Turing machines or mathematical systems that simulate Turing machines. The Post Correspondence Problem is a combinatorial question that can be described as a simple game of manipulating dominoes. A domino consists of two nonnull strings from a fixed alphabet, one on the top half of the domino and the other on the bottom.



A Post correspondence system can be thought of as defining a finite set of domino types.

The game begins with one of the dominoes being placed on a table. Another domino is then placed to the immediate right of the domino on the table. This process is repeated, producing a sequence of adjacent dominoes. We assume that there is an unlimited number of dominoes of each type; playing a domino does not limit the number of future moves.

A string is obtained by concatenating the strings in the top halves of a sequence of dominoes. We refer to this as the top string. Similarly, a sequence of dominoes defines a bottom string. The object of the game is to find a finite sequence of plays that produces identical top and bottom strings. Consider the Post correspondence system defined by dominoes

a	c	ba	acb
ac	ba	a	b

The sequence

a	c	ba	a	acb
ac	ba	a	ac	b

spells *acbbaacb* in both the top and bottom strings.

Formally, a **Post correspondence system** consists of an alphabet Σ and a finite set of ordered pairs $[u_i, v_i]$, $i = 1, 2, \dots, n$, where $u_i, v_i \in \Sigma^+$. A solution to a Post correspondence system is a sequence i_1, i_2, \dots, i_k such that

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

The problem of determining whether a Post correspondence system has a solution is the Post Correspondence Problem.

Example 12.6.1

The Post correspondence system with alphabet $\{a, b\}$ and ordered pairs $[aaa, aa]$, $[baa, abaaa]$ has a solution

aaa	baa	aaa
aa	abaaa	aa

□

Example 12.6.2

Consider the Post correspondence system with alphabet $\{a, b\}$ and ordered pairs $[ab, aba]$, $[bba, aa]$, $[aba, bab]$. A solution must begin with the domino

ab
aba

since this is the only domino in which prefixes on the top and bottom agree. The string in the top half of the next domino must begin with a . There are two possibilities:

ab	ab
aba	aba

(a)

ab	aba
aba	bab

(b)

The fourth elements of the strings in (a) do not match. The only possible way of constructing a solution is to extend (b). Employing the same reasoning as before, we see that the first element in the top of the next domino must be b . This lone possibility produces

ab	aba	bba
aba	bab	aa

which cannot be the initial subsequence of a solution since the seventh elements in the top and bottom differ. We have shown that there is no way of “playing the dominoes” in which the top and bottom strings are identical. Hence, this Post correspondence system has no solution. □

We will show that the Post Correspondence Problem is undecidable by associating derivations in a Semi-Thue System with sequences of dominoes. By Theorem 12.5.4 we know that there is a Semi-Thue System $S = (\Sigma, P)$ whose word problem is undecidable;

that is, there is no algorithm using the rules in P .

Reduction

Derivability in
to
Post Correspondence
Problem

The reduction consists in a manner that playing

Theorem 12.6.1

There is no algorithm for a solution.

Proof. Let $S = (\Sigma, P)$ be a semi-thue system that is unsolvable. For each string $w \in \Sigma^*$ let $C_{u,v}$ be the semi-thue system S with u and v as new start symbols. Then there can be no general algorithm to decide whether $C_{u,v}$ is solvable or not.

We begin by analyzing the derivability of strings in $C_{u,v}$. Derivations in the resulting systems are sequences of rule applications that guarantee that whenever a string w is derived from u it is also derived from v . By abuse of notation, we will write $u \rightarrow w$ if w is derived from u .

Now let u and v be two strings that are not derived from each other. Let w be a string constructed from u , v , and some additional symbols. We want to show that there is no algorithm to decide whether w is derived from u or v .

Each production $u \rightarrow v$ consists of two dominoes

The system is complete.

that is, there is no algorithm that determines whether a string v is derivable from a string u using the rules in P . The components of the reduction are

$[aaa, aa]$,

Reduction	Input	Condition
Derivability in $S = (\Sigma, P)$ to Post Correspondence Problem	strings u, v \downarrow set of dominoes $C_{u,v}$	v is derivable from u if, and only if, the Post correspondence system $C_{u,v}$ has a solution

The reduction consists of producing dominoes from the rules of P and the strings u and v in a manner that playing the dominoes corresponds to derivations in the Semi-Thue System.

Theorem 12.6.1

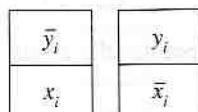
There is no algorithm that determines whether an arbitrary Post correspondence system has a solution.

Proof. Let $S = (\Sigma, P)$ be a Semi-Thue System with alphabet $\{0, 1\}$ whose word problem is unsolvable. For each pair of strings $u, v \in \Sigma^*$, we will construct a Post correspondence system $C_{u,v}$ that has a solution if, and only if, $u \xrightarrow[S]{} v$. Since the latter problem is undecidable, there can be no general algorithm that solves the Post Correspondence Problem.

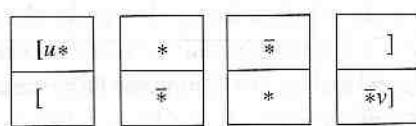
We begin by augmenting the set of productions of S with the rules $0 \rightarrow 0$ and $1 \rightarrow 1$. Derivations in the resulting system are identical to those in S except for the possible addition of rule applications that do not transform the string. The application of such a rule, however, guarantees that whenever $u \xrightarrow[S]{} v$, v may be obtained from u by a derivation of even length. By abuse of notation, the augmented system is also denoted S .

Now let u and v be strings over $\{0, 1\}^*$. A Post correspondence system $C_{u,v}$ is constructed from u , v , and S . The alphabet of $C_{u,v}$ consists of 0 , $\bar{0}$, 1 , $\bar{1}$, $[$, $]$, $*$, and $\bar{*}$. A string w consisting entirely of “barred” symbols is denoted \bar{w} .

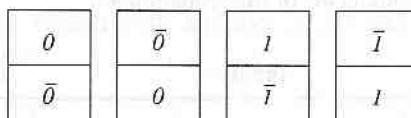
Each production $x_i \rightarrow y_i$, $i = 1, 2, \dots, n$, of S (including $0 \rightarrow 0$ and $1 \rightarrow 1$) defines two dominoes



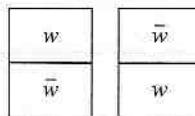
The system is completed by the dominoes



The dominoes



can be combined to form sequences of dominoes that spell



for any string $w \in \{0, 1\}^*$. We will feel free to use these composite dominoes when constructing a solution to a Post correspondence system $C_{u,v}$.

First we show that $C_{u,v}$ has a solution whenever $u \xrightarrow[s]{*} v$. Let

$$u = u_0 \Rightarrow u_1 \Rightarrow \dots \Rightarrow u_k \Rightarrow v$$

be a derivation of even length. The rules $0 \rightarrow 0$ and $1 \rightarrow 1$ ensure that there is derivation of even length whenever v is derivable from u . The i th step of the derivation can be written

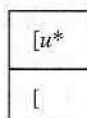
$$u_{i-1} = p_{i-1}x_{j_{i-1}}q_{i-1} \Rightarrow p_{i-1}y_{j_{i-1}}q_{i-1} = u_i,$$

where u_i is obtained from u_{i-1} by an application of the rule $x_{j_{i-1}} \rightarrow y_{j_{i-1}}$. The string

$$[u_0 * \bar{u}_1 * u_2 * \bar{u}_3 * \dots * \bar{u}_{k-1} * u_k]$$

is a solution to $C_{u,v}$. This solution can be constructed as follows:

- Initially play

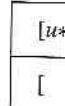


- To obtain a match, dominoes spelling the string $u = u_0$ on the bottom are played, producing

u^*	\bar{p}_0	\bar{y}_{j_0}	\bar{q}_0	*
[p_0	x_{j_0}	q_0	*

The dominoes spelling p_0 and q_0 are composite dominoes. The middle domino is generated by the rule $x_{j_0} \rightarrow y_{j_0}$.

- Since p_0, y_{j_0}, q_0 are all in u^* , the previous



producing $[u^*]$

- This process

u^*	\bar{p}_0
[p_0

- Completing t

produces the s
the correspon

We will now show
correspondence sy

since this is the o
argument, a soluti

Thus the string spe
can be written $[u^*]$
on both the top and

In light of the
the Post correspon

3. Since $p_0y_{j_0}q_0 = u_1$, the top string can be written $[u_0 * \bar{u}_1]$ and the bottom $[u_0]$. Repeating the previous strategy, dominoes must be played to spell \bar{u}_1 on the bottom

$[u^*$	\bar{p}_0	\bar{y}_{j_0}	\bar{q}_0	$*$	p_1	y_{j_1}	q_1	$*$
[p_0	x_{j_0}	q_0	*	\bar{p}_1	\bar{x}_{j_1}	\bar{q}_1	$*$

producing $[u_0 * \bar{u}_1 * u_2 * \dots]$ on the top.

4. This process is continued for steps 2, 3, ..., $k - 1$ of the derivation, producing

$[u^*$	\bar{p}_0	\bar{y}_{j_0}	\bar{q}_0	$*$	p_1	y_{j_1}	q_1	$*$...	p_{k-1}	$y_{j_{k-1}}$	q_{k-1}
[p_0	x_{j_0}	q_0	*	\bar{p}_1	\bar{x}_{j_1}	\bar{q}_1	$*$...	\bar{p}_{k-1}	$\bar{x}_{j_{k-1}}$	\bar{q}_{k-1}

5. Completing the sequence with the domino

]
*v]

produces the string $[u_0 * \bar{u}_1 * u_2 * \dots * \bar{u}_{k-1} * u_k]$ in both the top and the bottom, solving the correspondence system.

We will now show that a derivation $u \xrightarrow{*} w$ can be constructed from a solution to the Post correspondence system $C_{u,v}$. A solution to $C_{u,v}$ must begin with

$[u^*$
[

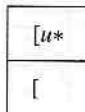
since this is the only domino whose strings begin with the same symbol. By the same argument, a solution must end with

]
*v]

Thus the string spelled by a solution has the form $[u * w\bar{v}]$. If w contains $]$, then the solution can be written $[u * x\bar{v}]y\bar{v}$. Since $]$ occurs in only one domino and is the rightmost symbol on both the top and the bottom of that domino, the string $[u * x\bar{v}]$ is also a solution of $C_{u,v}$.

In light of the previous observation, let $[u * \dots * v]$ be a string that is a solution of the Post correspondence system $C_{u,v}$ in which $]$ occurs only as the rightmost symbol. The

information provided by the dominoes at the ends of a solution determines the structure of the entire solution. The solution begins with



A sequence of dominoes that spell u on the bottom must be played in order to match the string already generated on the top. Let $u = x_{i_1}x_{i_2}\dots x_{i_k}$ be bottom strings in the dominoes that spell u in the solution. Then the solution has the form

[$u*$]	\bar{y}_{i_1}	\bar{y}_{i_2}	\bar{y}_{i_3}	...	\bar{y}_{i_k}	$\bar{*}$
[x_{i_1}	x_{i_2}	x_{i_3}	...	x_{i_k}	*

Since each domino represents a derivation $x_{i_j} \Rightarrow y_{i_j}$, we combine these to obtain the derivation $u \xrightarrow{*} u_1$, where $u_1 = y_{i_1}y_{i_2}\dots y_{i_k}$. The prefix of the top string of the dominoes that make up the solution has the form $[u * \bar{u}_1 \bar{*}]$, and the prefix of the bottom string is $[u*]$. Repeating this process, we see that a solution defines a sequence of strings

$$\begin{aligned} & [u * \bar{u}_1 \bar{*} u_2 * \dots \bar{*} v] \\ & [u * \bar{u}_1 \bar{*} u_2 * \bar{u}_3 \bar{*} \dots \bar{*} v] \\ & [u * \bar{u}_1 \bar{*} u_2 * \bar{u}_3 \bar{*} u_4 * \dots \bar{*} v] \\ & \vdots \\ & [u * \bar{u}_1 \bar{*} u_2 * \bar{u}_3 \bar{*} u_4 * \dots \bar{u}_{k-1} \bar{*} v], \end{aligned}$$

where $u_i \xrightarrow{*} u_{i+1}$ with $u_0 = u$ and $u_k = v$. Combining these produces a derivation $u \xrightarrow{*} v$.

The preceding two arguments constitute a reduction of the Word Problem for the Semi-Thue System S to the Post Correspondence Problem. It follows that the Post Correspondence Problem is undecidable. ■

Let $C = (\Sigma_C, \{$
Two context-free gr
follows:

$G_U:$

$G_L:$

Determining whether
the answers to certain
 G_L . The grammar G_U
of dominoes. The dig
(in reverse order). Si
half of a sequence of

The Post correspo
such that

In this case, G_U and G_L

where $u_{i_1}u_{i_2}\dots u_{i_{k-1}}$
section of $L(G_U)$ and

Conversely, assu
followed by a sequenc
is a solution to C .

Example 12.7.1

The grammars G_U and
 $[baa, abaaaa]$ from

$$\begin{aligned} G_U: S_U &\rightarrow a \\ &\rightarrow ba \\ &\rightarrow baa \\ &\rightarrow abaaa \\ &\rightarrow abaaaa \end{aligned}$$

12.7 Undecidable Problems in Context-Free Grammars

Context-free grammars provide an important tool for defining the syntax of programming languages. The undecidability of the Post Correspondence Problem can be used to establish the undecidability of several important questions concerning the languages generated by context-free grammars. To establish a link between Post correspondence systems and context-free grammars, the dominoes of a Post correspondence system are used to define the rules of two context-free grammars.

ecture of

Let $C = (\Sigma_C, \{[u_1, v_1], [u_2, v_2], \dots, [u_n, v_n]\})$ be a Post correspondence system. Two context-free grammars G_U and G_L are constructed from the ordered pairs of C as follows:

$$\begin{aligned} G_U: V_U &= \{S_U\} \\ \Sigma_U &= \Sigma_C \cup \{1, 2, \dots, n\} \\ P_U &= \{S_U \rightarrow u_i S_U i, S_U \rightarrow u_i i \mid i = 1, 2, \dots, n\} \\ G_L: V_L &= \{S_L\} \\ \Sigma_L &= \Sigma_C \cup \{1, 2, \dots, n\} \\ P_L &= \{S_L \rightarrow v_i S_L i, S_L \rightarrow v_i i \mid i = 1, 2, \dots, n\}. \end{aligned}$$

natch the
dominoes

obtain the
dominoes
ng is $[u^*$.

Determining whether a Post correspondence system C has a solution reduces to deciding the answers to certain questions concerning derivability in corresponding grammars G_U and G_L . The grammar G_U generates the strings that can appear in the upper half of a sequence of dominoes. The digits in the rule record the sequence of dominoes that generate the string (in reverse order). Similarly, G_L generates the strings that can be obtained from the lower half of a sequence of dominoes.

The Post correspondence system C has a solution if there is a sequence $i_1 i_2 \dots i_{k-1} i_k$ such that

$$u_{i_1} u_{i_2} \dots u_{i_{k-1}} u_{i_k} = v_{i_1} v_{i_2} \dots v_{i_{k-1}} v_{i_k}.$$

In this case, G_U and G_L contain derivations

$$\begin{aligned} S_U &\xrightarrow[G_U]{*} u_{i_1} u_{i_2} \dots u_{i_{k-1}} u_{i_k} i_k i_{k-1} \dots i_2 i_1 \\ S_L &\xrightarrow[G_L]{*} v_{i_1} v_{i_2} \dots v_{i_{k-1}} v_{i_k} i_k i_{k-1} \dots i_2 i_1, \end{aligned}$$

where $u_{i_1} u_{i_2} \dots u_{i_{k-1}} u_{i_k} i_k i_{k-1} \dots i_2 i_1 = v_{i_1} v_{i_2} \dots v_{i_{k-1}} v_{i_k} i_k i_{k-1} \dots i_2 i_1$. Hence, the intersection of $L(G_U)$ and $L(G_L)$ is not empty.

Conversely, assume that $w \in L(G_U) \cap L(G_L)$. Then w consists of a string $w' \in \Sigma_C^+$ followed by a sequence $i_k i_{k-1} \dots i_2 i_1$. The string $w' = u_{i_1} u_{i_2} \dots u_{i_{k-1}} u_{i_k} = v_{i_1} v_{i_2} \dots v_{i_{k-1}} v_{i_k}$ is a solution to C .

Example 12.7.1

The grammars G_U and G_L are constructed from the Post correspondence system $[aaa, aa]$, $[baa, abaaaa]$ from Example 12.6.1.

$$\begin{aligned} G_U: S_U &\rightarrow aaa S_U 1 \mid aaa 1 \\ &\rightarrow baa S_U 2 \mid baa 2 \end{aligned} \quad \begin{aligned} G_L: S_L &\rightarrow aa S_L 1 \mid aa 1 \\ &\rightarrow abaaa S_L 2 \mid abaaa 2 \end{aligned}$$

rogramming
to establish
generated by
ystems and
d to define

Derivations that exhibit the solution to the correspondence problem are

$$\begin{array}{ll} S_U \Rightarrow aaaS_U1 & S_L \Rightarrow aaS_L1 \\ \Rightarrow aaabaaS_U21 & \Rightarrow aaabaaaS_L21 \\ \Rightarrow aaabaaaaa121 & \Rightarrow aaabaaaaa121. \end{array}$$

□

The relationship between solutions to a Post correspondence system and derivations in the associated grammars G_U and G_L is used to demonstrate the undecidability of several questions about the languages generated by context-free grammars.

Theorem 12.7.1

There is no algorithm that determines whether the languages of two context-free grammars are disjoint.

Proof. Assume there is such an algorithm. Then the Post Correspondence Problem could be solved as follows:

1. For an arbitrary Post correspondence system C , construct the grammars G_U and G_L from the ordered pairs of C .
2. Use the algorithm to determine if $L(G_U)$ and $L(G_L)$ are disjoint.
3. C has a solution if, and only if, $L(G_U) \cap L(G_L)$ is nonempty.

Step 1 reduces the Post Correspondence Problem to the problem of determining whether two context-free languages are disjoint. Since the Post Correspondence Problem has already been shown to be undecidable, we conclude that the question of the intersection of context-free languages is also undecidable. ■

Theorem 12.7.2

There is no algorithm that determines whether an arbitrary context-free grammar is ambiguous.

Proof. A context-free grammar is ambiguous if it contains a string that can be generated by two distinct leftmost derivations. As before, we begin with an arbitrary Post correspondence system C and construct G_U and G_L . These grammars are combined to obtain the grammar

$$\begin{aligned} G: L = \{S, S_U, S_L\} \\ \Sigma = \Sigma_U \\ P = P_U \cup P_L \cup \{S \rightarrow S_U, S \rightarrow S_L\} \end{aligned}$$

with start symbol S that generates $L(G_U) \cup L(G_L)$.

Clearly, all derivations of G are leftmost; every sentential form contains at most one variable. A derivation of G consists of the application of an S rule followed by a derivation of G_U or G_L . The grammars G_U and G_L are unambiguous; distinct derivations generate distinct suffixes of integers. This implies that G is ambiguous if, and only if, $L(G_U) \cap L(G_L) \neq \emptyset$. But this condition is equivalent to the existence of a solution to the

original Post correspondence problem. Thus there is no algorithm that determines whether the languages of two context-free grammars are disjoint.

In Section 12.6, we showed that the Post Correspondence Problem is undecidable. The complementation of context-free languages is also undecidable. Let C be a Post correspondence system. Construct the grammars G_U and G_L from the ordered pairs of C . Then $L(G_U) \cap L(G_L) \neq \emptyset$ if and only if C has a solution. Thus there is no algorithm that determines whether the languages of two context-free grammars are disjoint.

Theorem 12.7.3

There is no algorithm that determines whether two context-free languages are disjoint. Let $G = (L, \Sigma, P, S)$ be a context-free grammar.

Proof. First, we show that there is no algorithm that determines whether $L(G) = \emptyset$.

Let C be a Post correspondence system. Construct the grammars G_U and G_L from the ordered pairs of C . Then $L(G_U) \cap L(G_L) \neq \emptyset$ if and only if C has a solution. Thus there is no algorithm that determines whether $L(G_U) \cap L(G_L) \neq \emptyset$.

1. For a Post correspondence system C , construct the grammars G_U and G_L from the ordered pairs of C .
2. Construct the grammar G from the ordered pairs of C .
3. Construct G' from the ordered pairs of C .
4. Use the decision algorithm for context-free grammars to determine if $L(G') = \emptyset$.

Thus there can be no algorithm that determines whether $L(G) = \emptyset$.

Theorem 12.7.4

There is no algorithm that determines whether two context-free grammars are identical.

Proof. Let C be a Post correspondence system. In the proof of Theorem 12.7.3, we showed that there is no algorithm that determines whether $L(G_U) \cap L(G_L) \neq \emptyset$. Let G_U and G_L be the grammars constructed from the ordered pairs of C . Then $L(G_U) \cap L(G_L) \neq \emptyset$ if and only if C has a solution. Thus there is no algorithm that determines whether $L(G_U) \cap L(G_L) \neq \emptyset$.

The language $L(G_U) \cup L(G_L)$ is the union of the languages generated by G_U and G_L . The language $L(G)$ is the union of the languages generated by G_U and G_L . Thus there is no algorithm that determines whether $L(G) \neq L(G_U) \cup L(G_L)$.

original Post correspondence system C . Since the Post Correspondence Problem is reducible to the problem of determining whether a context-free grammar is ambiguous, the latter problem is also undecidable. ■

In Section 7.5 we saw that the family of context-free languages is not closed under complementation. However, for an arbitrary Post correspondence system C , the languages $\overline{L(G_U)}$ and $\overline{L(G_L)}$ are context-free (Exercise 20). We will use this property to establish the undecidability of the problem of determining whether an arbitrary context-free grammar generates all strings over its alphabet and whether two context-free grammars generate the same language.

Theorem 12.7.3

There is no algorithm that determines whether the language of a context-free grammar $G = (L, \Sigma, P, S)$ is Σ^* .

Proof. First, note that $L = \Sigma^*$ is equivalent to $\overline{L} = \emptyset$. We will show that there is no algorithm that determines whether $\overline{L(G)}$ is empty for an arbitrary context-free grammar G .

Let C be a Post correspondence system with associated grammars G_U and G_L . A context-free grammar G' that generates $\overline{L(G_U)} \cup \overline{L(G_L)}$ can be obtained directly from the context-free grammars that generate $\overline{L(G_U)}$ and $\overline{L(G_L)}$. By DeMorgan's Law, $\overline{L(G')} = L(G_U) \cap L(G_L)$.

An algorithm that determines whether $\overline{L(G)} = \emptyset$ for an arbitrary context-free grammar G can be used to solve the Post Correspondence Problem as follows:

1. For a Post correspondence system C , construct the grammars G_U and G_L .
2. Construct the grammars that generate $\overline{L(G_U)}$ and $\overline{L(G_L)}$.
3. Construct G' from the grammars that generate $\overline{L(G_U)}$ and $\overline{L(G_L)}$.
4. Use the decision algorithm to determine whether $\overline{L(G')} = \emptyset$.
5. $\overline{L(G')} = \emptyset$ if, and only if, $L(G_U)$ and $L(G_L)$ are disjoint, if and only if, C has a solution.

Thus there can be no algorithm that decides whether $\overline{L(G)} = \emptyset$ or, equivalently, whether $L(G) = \Sigma^*$. ■

Theorem 12.7.4

There is no algorithm that determines whether the languages of two context-free grammars are identical.

Proof. Let C be a Post correspondence system with associated grammars G_U and G_L . As in the proof of Theorem 12.7.3, a context-free grammar G_1 can be constructed that generates $\overline{L(G_U)} \cup \overline{L(G_L)} = \overline{L(G_U) \cap L(G_L)}$. The second context-free grammar G_2 generates all strings over Σ_U .

The language $L(G_1)$ contains all strings in Σ_U^* that are not solutions of the Post correspondence system C . Thus $L(G_1) = L(G_2)$ if, and only if, C does not have a solution.

Consequently, an algorithm that determines whether two grammars generate the same language can be used to determine whether a Post correspondence system has a solution. ■

Exercises

1. Prove that the Halting Problem for the universal machine is undecidable. That is, there is no Turing machine that can determine whether the computation of U with an arbitrary input string will halt.
 2. Explain the fundamental difference between the Halts on n 'th Transition Problem from Example 11.5.2 and the Halting Problem that makes the former decidable and the latter undecidable.
 3. Let M be any deterministic Turing machine that accepts a nonrecursive language. Prove that the Halting Problem for M is undecidable. That is, there is no Turing machine that takes input w and determines whether the computation of M halts with input w .
- For Exercises 4 through 8, use reduction to establish the undecidability of each of the decision problems.
4. Prove that there is no algorithm that determines whether an arbitrary Turing machine halts when run with the input string 101 .
 5. Prove that there is no algorithm that determines whether an arbitrary Turing machine halts for at least one input string.
 6. Prove that there is no algorithm with input consisting of a Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, a state $q_i \in Q$, and a string $w \in \Sigma^*$ that determines whether the computation of M with input w enters state q_i .
 7. Prove that there is no algorithm that determines whether an arbitrary Turing machine prints a 1 on its final transition.
 8. Prove that there is no algorithm that determines whether an arbitrary Turing machine prints the symbol 1 on three consecutive transitions when run with a blank tape.
 9. Why can't we successfully argue that the Blank Tape Problem is undecidable as follows: The Blank Tape Problem is a subproblem of the Halting Problem, which is undecidable and therefore must be undecidable itself?
 10. Show that the problem of deciding whether a string over $\Sigma = \{1\}$ has even length is reducible to the Blank Tape Problem. Why is it incorrect to conclude from this that the problem of determining whether a string has even length is undecidable?
 11. Give an example of a property of languages that is not satisfied by any recursively enumerable language.
12. Use Rice's Theorem to show that the following properties of languages are undecidable. The property is undecidable if and only if the property is not recursive.
 - L contains a finite language.
 - L is finite.
 - L is regular.
 - L is $\{0, 1\}^*$. 13. Let $L = \{R(M) \mid M \text{ is a TM}\}$.
 - Show that L is undecidable.
 - Show that L is recursive.
 - * 14. Let $L_{\neq\emptyset} = \{R(M) \mid M \text{ is a TM and } L(M) \neq \emptyset\}$.
 - Show that $L_{\neq\emptyset}$ is undecidable.
 - Show that $L_{\neq\emptyset}$ is recursive.
 15. Let M be the Turing machine defined in Exercise 11.5.2. Let $S_M = \{q_0, q_f, \Sigma, \Gamma, \delta\}$.
 - Give the rule δ .
 - Trace the computation S_M on input 101 .
 16. Find a solution for the following subproblems of the Post Correspondence Problem.
 - $[a, aa], [bb, b]$
 - $[a, aaa], [aa, ab]$
 - $[aa, aab], [b, bb]$
 - $[a, ab], [ba, bb]$
 17. Show that the following subproblems of the Post Correspondence Problem are undecidable.
 - $[b, ba], [aa, ab]$
 - $[ab, a], [ba, bb]$
 - $[ab, aba], [b, bb]$
 - $[ab, bb], [aa, ab]$
 - $[abb, ab], [a, aab]$
 - * 18. Prove that the Post Correspondence Problem is decidable.

- the same solution. ■
- is, there arbitrary
- em from the latter
- ge. Prove fine that w .
- ch of the machine
- machine
- ine $M =$, whether machine type.
- s follows: decidable length is is that the recursively
12. Use Rice's Theorem to show that the following properties of recursively enumerable languages are undecidable. To establish the undecidability, all you need do is show that the property is nontrivial.

- L contains a particular string w .
- L is finite.
- L is regular.
- L is $\{0, 1\}^*$.

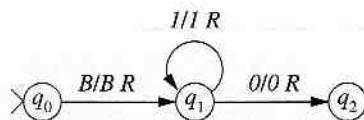
13. Let $L = \{R(M) \mid M \text{ halts when run with } R(M)\}$.

- Show that L is not recursive.
- Show that L is recursively enumerable.

- * 14. Let $L_{\neq\emptyset} = \{R(M) \mid L(M) \text{ is nonempty}\}$.

- Show that $L_{\neq\emptyset}$ is not recursive.
- Show that $L_{\neq\emptyset}$ is recursively enumerable.

15. Let M be the Turing machine



- Give the rules of the Semi-Thue System S_M that simulate the computations of M .
 - Trace the computation of M with input 01 and give the corresponding derivation in S_M .
16. Find a solution for each of the following Post correspondence systems.
- $[a, aa], [bb, b], [a, bb]$
 - $[a, aaa], [aab, b], [abaa, ab]$
 - $[aa, aab], [bb, ba], [abb, b]$
 - $[a, ab], [ba, aba], [b, aba], [bba, b]$
17. Show that the following Post correspondence systems have no solutions.
- $[b, ba], [aa, b], [bab, aa], [ab, ba]$
 - $[ab, a], [ba, bab], [b, aa], [ba, ab]$
 - $[ab, aba], [baa, aa], [aba, baa]$
 - $[ab, bb], [aa, ba], [ab, abb], [bb, bab]$
 - $[abb, ab], [aba, ba], [aab, abab]$
- * 18. Prove that the Post Correspondence Problem for systems with a one-symbol alphabet is decidable.

19. Let P be the Post correspondence system defined by $[b, bbb]$, $[babbb, ba]$, $[bab, aab]$, $[ba, a]$.
- Give a solution to P .
 - Construct the grammars G_U and G_L from P .
 - Give the derivations in G_U and G_L corresponding to the solution in (a).
20. Build the context-free grammars G_U and G_L that are constructed from the Post correspondence system $[b, bb]$, $[aa, baa]$, $[ab, a]$. Is $L(G_U) \cap L(G_L) = \emptyset$?
- *21. Let C be a Post correspondence system. Construct a context-free grammar that generates $\overline{L(G_U)}$.
- *22. Prove that there is no algorithm that determines whether the intersection of the languages of two context-free grammars contains infinitely many elements.
23. Prove that there is no algorithm that determines whether the complement of the language of a context-free grammar contains infinitely many elements.
- *24. Prove that there is no algorithm that determines whether the languages of two arbitrary context-free grammars G_1 and G_2 satisfy $L(G_1) \subseteq L(G_2)$.

Bibliographic Notes

The undecidability of the Halting Problem was established by Turing [1936]. The proof given in Section 12.1 is from Minsky [1967]. Techniques for establishing undecidability using properties of languages were presented in Rice [1953] and [1956]. The string transformation systems of Thue were introduced in Thue [1914] and the undecidability of the Word Problem for Semi-Thue Systems was established by Post [1947].

The undecidability of the Post Correspondence Problem was presented in Post [1946]. The proof of Theorem 12.6.1, based on the technique of Floyd [1964], is from Davis and Weyuker [1983]. Undecidability results for context-free languages, including Theorem 12.7.1, can be found in Bar-Hillel, Perles, and Shamir [1961]. The undecidability of ambiguity of context-free languages was established by Cantor [1962], Floyd [1962], and Chomsky and Schutzenberger [1963]. The question of inherent ambiguity was shown to be unsolvable by Ginsburg and Ullian [1966a].

CHAPTER

Mu-

In Chapter 9 we saw that the transitions of a Turing machine do not assert that every transition leads exactly what function does. This is related to this question.

We now consider the question of what objects of study are computable. We focus on elements of computation, and the μ -recursive functions. These are the most generally computable functions.

The computability of a function means that there exists an effective procedure for computing it. The question of what functions are computable is called the question of the computability of a function.

13.1 Primitive Recursive Functions

A family of functions that are computable by recursive functions.

CHAPTER 13

Mu-Recursive Functions

In Chapter 9 we introduced computable functions from a mechanical perspective; the transitions of a Turing machine produced the values of a function. The Church-Turing Thesis asserts that every algorithmically computable function can be realized in this manner, but exactly what functions are Turing computable? In this chapter we will provide an answer to this question and, in doing so, obtain further support for the Church-Turing Thesis.

We now consider computable functions from a macroscopic viewpoint. Rather than focusing on elementary Turing machine operations, functions themselves are the fundamental objects of study. We introduce two families of functions, the primitive recursive functions and the μ -recursive functions. The primitive recursive functions are built from a set of intuitively computable functions using the operations of composition and primitive recursion. The μ -recursive functions are obtained by adding unbounded minimalization, a functional representation of sequential search, to the function building operations.

The computability of the primitive and μ -recursive functions is demonstrated by outlining an effective method for producing the values of the functions. The analysis of effective computation is completed by showing the equivalence of the notions of Turing computability and μ -recursivity. This answers the question posed in the opening paragraph—the functions computable by a Turing machine are exactly the μ -recursive functions.

13.1 Primitive Recursive Functions

A family of intuitively computable number-theoretic functions, known as the primitive recursive functions, is obtained from the basic functions

- i) the successor function $s: s(x) = x + 1$
- ii) the zero function $z: z(x) = 0$
- iii) the projection functions $p_i^{(n)}: p_i^{(n)}(x_1, \dots, x_n) = x_i, 1 \leq i \leq n$

using operations that construct new functions from functions already in the family. The simplicity of the basic functions supports their intuitive computability. The successor function requires only the ability to add one to a natural number. Computing the zero function is even less complex; the value of the function is zero for every argument. The value of the projection function $p_i^{(n)}$ is simply its i th argument.

The primitive recursive functions are constructed from the basic functions by applications of two operations that preserve computability. The first operation is functional composition (Definition 9.4.2). Let f be defined by the composition of the n -variable function h with the k -variable functions g_1, g_2, \dots, g_n . If each of the components of the composition is computable, then the value of $f(x_1, \dots, x_k)$ can be obtained from h and $g_1(x_1, \dots, x_k), g_2(x_1, \dots, x_k), \dots, g_n(x_1, \dots, x_k)$. The computability of f follows from the computability of its constituent functions. The second operation for producing new functions is primitive recursion.

Definition 13.1.1

Let g and h be total number-theoretic functions with n and $n + 2$ variables, respectively. The $n + 1$ -variable function f defined by

- i) $f(x_1, \dots, x_n, 0) = g(x_1, \dots, x_n)$
- ii) $f(x_1, \dots, x_n, y + 1) = h(x_1, \dots, x_n, y, f(x_1, \dots, x_n, y))$

is said to be obtained from g and h by **primitive recursion**.

The x_i 's are called the *parameters* of a definition by primitive recursion. The variable y is the *recursive variable*.

The operation of primitive recursion provides its own algorithm for computing the value of $f(x_1, \dots, x_n, y)$ whenever g and h are computable. For a fixed set of parameters x_1, \dots, x_n , $f(x_1, \dots, x_n, 0)$ is obtained directly from the function g :

$$f(x_1, \dots, x_n, 0) = g(x_1, \dots, x_n).$$

The value $f(x_1, \dots, x_n, y + 1)$ is obtained from the computable function h using

- i) the parameters x_1, \dots, x_n ,
- ii) y , the previous value of the recursive variable, and
- iii) $f(x_1, \dots, x_n, y)$, the previous value of the function.

For example, $f(x_1, \dots,$

f
 f
 f

$f(x_1,$

Since h is computable
for any value of the i

Definition 13.1.2

A function is **primitive recursive** if it is obtained from the basic functions by a finite sequence of applications of the operations of composition and primitive recursion.

A function defined by primitive recursion is total. This is an immediate consequence of the fact that the computability of the basic functions implies the totality of the primitive recursive functions. Since the basic functions are total, it follows that any primitive recursive function is total.

Taken together, the operations of composition and primitive recursion provide a means for constructing all total computable functions. The construction of functions by composition and primitive recursion is called the **recursion theorem**.

Example 13.1.1

The constant function c is primitive recursive because it is obtained from the constant function z by composition.

Example 13.1.2

Let add be the function $add(x, y, z) = z + 1$. Then

add
 add

The function add is primitive recursive. It indicates that the sum of x and y is z . The function add defines the sum of x and y . The value of the recursive function $add(x, y, z)$ is $z + 1$.

For example, $f(x_1, \dots, x_n, y + 1)$ is obtained by the sequence of computations

$$\begin{aligned} f(x_1, \dots, x_n, 0) &= g(x_1, \dots, x_n) \\ f(x_1, \dots, x_n, 1) &= h(x_1, \dots, x_n, 0, f(x_1, \dots, x_n, 0)) \\ f(x_1, \dots, x_n, 2) &= h(x_1, \dots, x_n, 1, f(x_1, \dots, x_n, 1)) \\ &\vdots \\ f(x_1, \dots, x_n, y + 1) &= h(x_1, \dots, x_n, y, f(x_1, \dots, x_n, y)). \end{aligned}$$

Since h is computable, this iterative process can be used to determine $f(x_1, \dots, x_n, y + 1)$ for any value of the recursive variable y .

Definition 13.1.2

A function is **primitive recursive** if it can be obtained from the successor, zero, and projection functions by a finite number of applications of primitive recursion.

A function defined by composition or primitive recursion from total functions is itself total. This is an immediate consequence of the definitions of the operations and is left as an exercise. Since the basic primitive recursive functions are total and the operations preserve totality, it follows that all primitive recursive functions are total.

Taken together, composition and primitive recursion provide powerful tools for the construction of functions. The following examples show that arbitrary constant functions, addition, multiplication, and factorial are primitive recursive functions.

Example 13.1.1

The constant functions $c_i^{(n)}(x_1, \dots, x_n) = i$ are primitive recursive. Example 9.4.2 defines the constant functions as the composition of the successor, zero, and projection functions. \square

Example 13.1.2

Let add be the function defined by primitive recursion from the functions $g(x) = x$ and $h(x, y, z) = z + 1$. Then

$$\begin{aligned} \text{add}(x, 0) &= g(x) = x \\ \text{add}(x, y + 1) &= h(x, y, \text{add}(x, y)) = \text{add}(x, y) + 1. \end{aligned}$$

The function add computes the sum of two natural numbers. The definition of $\text{add}(x, 0)$ indicates that the sum of any number with zero is the number itself. The latter condition defines the sum of x and $y + 1$ as the sum of x and y (the result of add for the previous value of the recursive variable) incremented by one.

The preceding definition establishes that addition is primitive recursive. Both g and h , the components of the definition by primitive recursion, are primitive recursive since $g = p_1^{(1)}$ and $h = s \circ p_3^{(3)}$.

The result of the addition of two natural numbers can be obtained from the primitive recursive definition of add by repeatedly applying the condition $\text{add}(x, y + 1) = \text{add}(x, y) + 1$ to reduce the value of the recursive variable. For example,

$$\begin{aligned}\text{add}(2, 4) &= \text{add}(2, 3) + 1 \\ &= (\text{add}(2, 2) + 1) + 1 \\ &= ((\text{add}(2, 1) + 1) + 1) + 1 \\ &= (((\text{add}(2, 0) + 1) + 1) + 1) + 1 \\ &= (((2 + 1) + 1) + 1) + 1 \\ &= 6.\end{aligned}$$

When the recursive variable is zero, the function g is used to initiate the evaluation of the expression. \square

Example 13.1.3

Let g and h be the primitive functions $g = z$ and $h = \text{add} \circ (p_3^{(3)}, p_1^{(3)})$. Multiplication can be defined by primitive recursion from g and h as follows:

$$\begin{aligned}\text{mult}(x, 0) &= g(x) = 0 \\ \text{mult}(x, y + 1) &= h(x, y, \text{mult}(x, y)) = \text{mult}(x, y) + x.\end{aligned}$$

The infix expression corresponding to the primitive recursive definition is the identity $x \cdot (y + 1) = x \cdot y + x$, which follows from the distributive property of addition and multiplication. \square

Adopting the convention that a zero-variable function is a constant, we can use Definition 13.1.1 to define one-variable functions using primitive recursion and a two-variable function h . The definition of such a function f has the form

- i) $f(0) = n_0$, where $n_0 \in \mathbb{N}$
- ii) $f(y + 1) = h(y, f(y))$.

Example 13.1.4

The one-variable factorial function defined by

$$\text{fact}(y) = \begin{cases} 1 & \text{if } y = 0 \\ \prod_{i=1}^y i & \text{otherwise} \end{cases}$$

is primitive recursive.
function is defined using

f
 f

Note that the definition
applying the successor

The evaluation of
primitive recursive defi

The factorial function is

The primitive recursive
functions. The Church-Turing
thesis states that any function
that this is indeed the case.

Theorem 13.1.3

Every primitive recursive function is

Proof. Turing machines can compute every primitive recursive function. To complete the proof, it remains to show that any primitive recursive function that remains is to show that it is Turing computable. That is, if f is primitive recursive and g and h , then f is Turing computable.

Let g and h be primitive recursive functions.

$$\begin{aligned}f(x_1, \dots, x_n) &= \\ f(x_1, \dots, x_n) &= \dots\end{aligned}$$

defined from g and h by standard Turing machine configurations. A configuration is constructed to compute f from x_1, \dots, x_n . The initial configuration is $B\bar{x}_1B\bar{x}_2B\dots$

1. A counter, initially set to 0, is used to record the number of multiplications performed.

both g and h are primitive recursive. Let $h(x, y) = \text{mult} \circ (p_2^{(2)}, s \circ p_1^{(2)}) = y \cdot (x + 1)$. The factorial function is defined using primitive recursion from h by

$$\begin{aligned} \text{fact}(0) &= 1 \\ \text{fact}(y + 1) &= h(y, \text{fact}(y)) = \text{fact}(y) \cdot (y + 1). \end{aligned}$$

Note that the definition uses $y + 1$, the value of the recursive variable. This is obtained by applying the successor function to y , the value provided to the function h .

The evaluation of the function fact for the first five input values illustrates how the primitive recursive definition generates the factorial function.

$$\begin{aligned} \text{fact}(0) &= 1 \\ \text{fact}(1) &= \text{fact}(0) \cdot (0 + 1) = 1 \\ \text{fact}(2) &= \text{fact}(1) \cdot (1 + 1) = 2 \\ \text{fact}(3) &= \text{fact}(2) \cdot (2 + 1) = 6 \\ \text{fact}(4) &= \text{fact}(3) \cdot (3 + 1) = 24 \end{aligned}$$

The factorial function is usually denoted $\text{fact}(x) = x!$. □

The primitive recursive functions were defined as a family of intuitively computable functions. The Church-Turing Thesis asserts that these functions must also be computable using our Turing machine approach to functional computation. The Theorem 13.1.3 shows that this is indeed the case.

Theorem 13.1.3

Every primitive recursive function is Turing computable.

Proof. Turing machines that compute the basic functions were constructed in Section 9.2. To complete the proof, it suffices to prove that the Turing computable functions are closed under composition and primitive recursion. The former was established in Section 9.4. All that remains is to show that the Turing computable functions are closed under primitive recursion; that is, if f is defined by primitive recursion from Turing computable functions g and h , then f is Turing computable.

Let g and h be Turing computable functions and let f be the function

$$\begin{aligned} f(x_1, \dots, x_n, 0) &= g(x_1, \dots, x_n) \\ f(x_1, \dots, x_n, y + 1) &= h(x_1, \dots, x_n, y, f(x_1, \dots, x_n, y)) \end{aligned}$$

defined from g and h by primitive recursion. Since g and h are Turing computable, there are standard Turing machines G and H that compute them. A composite machine F is constructed to compute f . The computation of $f(x_1, x_2, \dots, x_n, y)$ begins with tape configuration $B\bar{x}_1B\bar{x}_2B\dots B\bar{x}_nB\bar{y}B$.

1. A counter, initially set to 0, is written to the immediate right of the input. The counter is used to record the value of the recursive variable for the current computation.

The parameters are then written to the right of the counter, producing the tape configuration

$$B\bar{x}_1B\bar{x}_2B \dots B\bar{x}_nB\bar{y}B\bar{0}B\bar{x}_1B\bar{x}_2B \dots B\bar{x}_nB.$$

2. The machine G is run on the final n values of the tape, producing

$$B\bar{x}_1B\bar{x}_2B \dots B\bar{x}_nB\bar{y}B\bar{0}B\bar{g}(x_1, x_2, \dots, x_n)B.$$

The computation of G generates $\bar{g}(x_1, x_2, \dots, x_n) = f(x_1, x_2, \dots, x_n, 0)$.

3. The tape now has the form

$$B\bar{x}_1B\bar{x}_2B \dots B\bar{x}_nB\bar{y}B\bar{i}B\bar{f}(x_1, x_2, \dots, x_n, i)B.$$

If the counter i is equal to y , the computation of $f(x_1, x_2, \dots, x_n, y)$ is completed by erasing the initial $n + 2$ numbers on the tape and translating the result to tape position one.

4. If $i < y$, the tape is configured to compute the next value of f .

$$B\bar{x}_1B\bar{x}_2B \dots B\bar{x}_nB\bar{y}B\bar{i} + 1B\bar{x}_1B\bar{x}_2B \dots B\bar{x}_nB\bar{i}B\bar{f}(x_1, x_2, \dots, x_n, i)B$$

The machine H is run on the final $n + 2$ values on the tape, producing

$$B\bar{x}_1B\bar{x}_2B \dots B\bar{x}_nB\bar{y}B\bar{i} + 1B\bar{h}(x_1, x_2, \dots, x_n, i, f(x_1, x_2, \dots, x_n, i))B,$$

where the rightmost value on the tape is $f(x_1, x_2, \dots, x_n, i + 1)$. The computation continues with the comparison in step 3. ■

13.2 Some Primitive Recursive Functions

A function is primitive recursive if it can be constructed from the zero, successor, and projection functions by a finite number of applications of composition and primitive recursion. Composition permits g and h , the functions used in a primitive recursive definition, to utilize any function that has previously been shown to be primitive recursive.

Primitive recursive definitions are constructed for several common arithmetic functions. Rather than explicitly detailing the functions g and h , a definition by primitive recursion is given in terms of the parameters, the recursive variable, the previous value of the function, and other primitive recursive functions. Note that the definitions of addition and multiplication are identical to the formal definitions given in Examples 13.1.2 and 13.1.3, with the intermediate step omitted.

Because of the compatibility with the operations of composition and primitive recursion, the definitions in Tables 13.1 and 13.2 are given using the functional notation. The standard infix representations of the binary arithmetic functions, given below the function

TABLE 13.1

Description

Addition

Multiplication

Predecessor

Proper sub

Exponentia

names, are used in the table. The symbol $\bar{+}$ denotes the successor function.

A primitive recursive function is a function whose domain is a subset of $\{0, 1\}^n$. Zero and one are represented by $\bar{0}$ and $\bar{1}$. In Table 13.2, the sign of a number is indicated when the argument is positive or negative. The sign of zero is zero. Binary predicate functions and the sign of a product are also defined.

TABLE

Description

Sign

Sign c

Less th

Greater

Equal

Not eq

pe con-

leted by
position**TABLE 13.1** Primitive Recursive Arithmetic Functions

Description	Function	Definition
Addition	$add(x, 0)$	$add(x, 0) = x$
	$x + y$	$add(x, y + 1) = add(x, y) + 1$
Multiplication	$mult(x, 0)$	$mult(x, 0) = 0$
	$x \cdot y$	$mult(x, y + 1) = mult(x, y) + x$
Predecessor	$pred(0)$	$pred(0) = 0$
	$pred(y + 1)$	$pred(y + 1) = y$
Proper subtraction	$sub(x, 0)$	$sub(x, 0) = x$
	$x \dot{-} y$	$sub(x, y + 1) = pred(sub(x, y))$
Exponentiation	$exp(x, 0)$	$exp(x, 0) = 1$
	x^y	$exp(x, y + 1) = exp(x, y) \cdot x$

names, are used in the arithmetic expressions throughout the chapter. The notation “+ 1” denotes the successor operator.

A primitive recursive predicate is a primitive recursive function whose range is the set {0, 1}. Zero and one are interpreted as false and true, respectively. The first two predicates in Table 13.2, the sign predicates, specify the sign of the argument. The function sg is true when the argument is positive. The complement of sg , denoted $cosg$, is true when the input is zero. Binary predicates that compare the input can be constructed from the arithmetic functions and the sign predicates using composition.

TABLE 13.2 Primitive Recursive Predicates

Description	Predicate	Definition
Sign	$sg(x)$	$sg(0) = 0$
		$sg(y + 1) = 1$
Sign complement	$cosg(x)$	$cosg(0) = 1$
		$cosg(y + 1) = 0$
Less than	$lt(x, y)$	$sg(y \dot{-} x)$
Greater than	$gt(x, y)$	$sg(x \dot{-} y)$
Equal to	$eq(x, y)$	$cosg(lt(x, y) + gt(x, y))$
Not equal to	$ne(x, y)$	$cosg(eq(x, y))$

Predicates are functions that exhibit the truth or falsity of a proposition. The logical operations negation, conjunction, and disjunction can be constructed using the arithmetic functions and the sign predicates. Let p_1 and p_2 be two primitive recursive predicates. Logical operations on p_1 and p_2 can be defined as follows:

Predicate	Interpretation
$\text{cosg}(p_1)$	not p_1
$p_1 \cdot p_2$	p_1 and p_2
$\text{sg}(p_1 + p_2)$	p_1 or p_2

Applying cosg to the result of a predicate interchanges the values, yielding the negation of the predicate. This technique was used to define the predicate ne from the predicate eq . Determining the value of a disjunction begins by adding the truth values of the component predicates. Since the sum is 2 when both of the predicates are true, the disjunction is obtained by composing the addition with sg . The resulting predicates are primitive recursive since the components of the composition are primitive recursive.

Example 13.2.1

The equality predicates can be used to explicitly specify the value of a function for a finite set of arguments. For example, f is the identity function for all input values other than 0, 1, and 2:

$$f(x) = \begin{cases} 2 & \text{if } x = 0 \\ 5 & \text{if } x = 1 \\ 4 & \text{if } x = 2 \\ x & \text{otherwise} \end{cases} \quad f(x) = \text{eq}(x, 0) \cdot 2 + \text{eq}(x, 1) \cdot 5 + \text{eq}(x, 2) \cdot 4 + \text{gt}(x, 2) \cdot x.$$

The function f is primitive recursive since it can be written as the composition of primitive recursive functions eq , gt , \cdot , and $+$. The four predicates in f are exhaustive and mutually exclusive; that is, one and only one of them is true for any natural number. The value of f is determined by the single predicate that holds for the input. \square

The technique presented in the previous example, constructing a function from exhaustive and mutually exclusive primitive recursive predicates, is used to establish the following theorem.

Theorem 13.2.1

Let g be a primitive recursive function and f a total function that is identical to g for all but a finite number of input values. Then f is primitive recursive.

Proof. Let g be primi

The equality predicate
input values, $f(x) = g$

is true whenever the va

$f(x) =$

Thus f is also primiti

The order of the v
The initial variables a
Combining composition
in specifying the num
flexibility is demonstr
function.

Theorem 13.2.2

Let $g(x, y)$ be a primitive

- i) (adding dummy va
 - ii) (permuting variab
 - iii) (identifying variab
- are primitive recursive.

Proof. Each of the fu
projections by composi

- i) $f = g \circ (p_1^{(n+2)}, p_2^{(n+1)})$
- ii) $f = g \circ (p_2^{(2)}, p_1^{(2)})$
- iii) $f = g \circ (p_1^{(1)}, p_2^{(1)})$

Dummy variables
compatible for compos

the logical arithmetic predicates.

Proof. Let g be primitive recursive and let f be defined by

$$f(x) = \begin{cases} y_1 & \text{if } x = n_1 \\ y_2 & \text{if } x = n_2 \\ \vdots & \\ y_k & \text{if } x = n_k \\ g(x) & \text{otherwise.} \end{cases}$$

The equality predicate is used to specify the values of f for input n_1, \dots, n_k . For all other input values, $f(x) = g(x)$. The predicate obtained by the product

$$ne(x, n_1) \cdot ne(x, n_2) \cdot \dots \cdot ne(x, n_k)$$

is true whenever the value of f is determined by g . Using these predicates, f can be written

$$\begin{aligned} f(x) = & eq(x, n_1) \cdot y_1 + eq(x, n_2) \cdot y_2 + \dots + eq(x, n_k) \cdot y_k \\ & + ne(x, n_1) \cdot ne(x, n_2) \cdot \dots \cdot ne(x, n_k) \cdot g(x). \end{aligned}$$

Thus f is also primitive recursive. ■

for a finite number greater than 0,

The order of the variables is an essential feature of a definition by primitive recursion. The initial variables are the parameters and the final variable is the recursive variable. Combining composition and the projection functions permits a great deal of flexibility in specifying the number and order of variables in a primitive recursive function. This flexibility is demonstrated by considering alterations to the variables in a two-variable function.

Theorem 13.2.2

Let $g(x, y)$ be a primitive recursive function. Then the functions obtained by

- i) (adding dummy variables) $f(x, y, z_1, z_2, \dots, z_n) = g(x, y)$
- ii) (permuting variables) $f(x, y) = g(y, x)$
- iii) (identifying variables) $f(x) = g(x, x)$

are primitive recursive.

Proof. Each of the functions is primitive recursive since it can be obtained from g and the projections by composition as follows:

- i) $f = g \circ (p_1^{(n+2)}, p_2^{(n+2)})$
- ii) $f = g \circ (p_2^{(2)}, p_1^{(2)})$
- iii) $f = g \circ (p_1^{(1)}, p_1^{(1)})$ ■

of primitive and mutually different value of f

□

from exhaustion following

g for all but

Dummy variables are used to make functions with different numbers of variables compatible for composition. The definition of the composition $h \circ (g_1, g_2)$ requires that g_1

and g_2 have the same number of variables. Consider the two-variable function f defined by $f(x, y) = (x \cdot y) + x!$. The constituents of the addition are obtained from a multiplication and a factorial operation. The former function has two variables and the latter has one. Adding a dummy variable to the function fact produces a two-variable function fact' satisfying $\text{fact}'(x, y) = \text{fact}(x) = x!$. Finally, we note that $f = \text{add} \circ (\text{mult}, \text{fact}')$ so that f is also primitive recursive.

13.3 Bounded Operators

The sum of a sequence of natural numbers can be obtained by repeated applications of the binary operation of addition. Addition and projection can be combined to construct a function that adds a fixed number of arguments. For example, the primitive recursive function

$$\text{add} \circ (p_1^{(4)}, \text{add} \circ (p_2^{(4)}, \text{add} \circ (p_3^{(4)}, p_4^{(4)})))$$

returns the sum of its four arguments. This approach cannot be used when the number of summands is variable. Consider the function

$$f(y) = \sum_{i=0}^y g(i) = g(0) + g(1) + \cdots + g(y).$$

The number of additions is determined by the input variable y . The function f is called the *bounded sum* of g . The variable i is the index of the summation. Computing a bounded sum consists of three actions: the generation of the summands, binary addition, and the comparison of the index with the input y .

We will prove that the bounded sum of a primitive recursive function is primitive recursive. The technique presented can be used to show that repeated applications of any binary primitive recursive operation is also primitive recursive.

Theorem 13.3.1

Let $g(x_1, \dots, x_n, y)$ be a primitive recursive function. Then the functions

- i) (bounded sum) $f(x_1, \dots, x_n, y) = \sum_{i=0}^y g(x_1, \dots, x_n, i)$
- ii) (bounded product) $f(x_1, \dots, x_n, y) = \prod_{i=0}^y g(x_1, \dots, x_n, i)$

are primitive recursive.

Proof. The sum

$$\sum_{i=0}^y g(x_1, \dots, x_n, i)$$

is obtained by adding

$$f(x_1, \dots, x_n, 0)$$

$$f(x_1, \dots, x_n, 1)$$

The bounded operator f is primitive recursive because its index reaches the value y in at most $y+1$ steps. By having the range of the function f bounded, the functions l and u are primitive recursive.

Theorem 13.3.2

Let g be an $n+1$ -variable primitive recursive function. Then

i) $f(x_1, \dots, x_n) =$

ii) $f(x_1, \dots, x_n) =$

are primitive recursive.

Proof. Since the lower bound l and upper bound u of the index i are primitive recursive, the result of the bounded sum or product is primitive recursive.

is true in precisely the same way.

If the lower bound $l(x_1, \dots, x_n)$ and upper bound $u(x_1, \dots, x_n)$ of the index i are primitive recursive functions, then

$$g'(x_1, \dots, x_n)$$

The values of g' are obtained by

$$g'(x_1, \dots, x_n)$$

$$g'(x_1, \dots, x_n)$$

$$g'(x_1, \dots, x_n)$$

defined by implication has one. tion fact' t' so that

is obtained by adding $g(x_1, \dots, x_n, y)$ to

$$\sum_{i=0}^{y-1} g(x_1, \dots, x_n, i).$$

Translating this into the language of primitive recursion, we get

$$\begin{aligned} f(x_1, \dots, x_n, 0) &= g(x_1, \dots, x_n, 0) \\ f(x_1, \dots, x_n, y+1) &= f(x_1, \dots, x_n, y) + g(x_1, \dots, x_n, y+1). \end{aligned}$$

cations of construct recursive

number of

f is called a bounded on, and the s primitive ions of any

The bounded operations just introduced begin with index zero and terminate when the index reaches the value specified by the argument y . Bounded operations can be generalized by having the range of the index variable determined by two computable functions. The functions l and u are used to determine the lower and upper bounds of the index.

Theorem 13.3.2

Let g be an $n+1$ -variable primitive recursive function and let l and u be n -variable primitive recursive functions. Then the functions

$$\begin{aligned} \text{i)} \quad f(x_1, \dots, x_n) &= \sum_{i=l(x_1, \dots, x_n)}^{u(x_1, \dots, x_n)} g(x_1, \dots, x_n, i) \\ \text{ii)} \quad f(x_1, \dots, x_n) &= \prod_{i=l(x_1, \dots, x_n)}^{u(x_1, \dots, x_n)} g(x_1, \dots, x_n, i) \end{aligned}$$

are primitive recursive.

Proof. Since the lower and upper bounds of the summation are determined by the functions l and u , it is possible that the lower bound may be greater than the upper bound. When this occurs, the result of the summation is assigned the default value zero. The predicate

$$gt(l(x_1, \dots, x_n), u(x_1, \dots, x_n))$$

is true in precisely these instances.

If the lower bound is less than or equal to the upper bound, the summation begins with index $l(x_1, \dots, x_n)$ and terminates when the index reaches $u(x_1, \dots, x_n)$. Let g' be the primitive recursive function defined by

$$g'(x_1, \dots, x_n, y) = g(x_1, \dots, x_n, y + l(x_1, \dots, x_n)).$$

The values of g' are obtained from those of g and $l(x_1, \dots, x_n)$:

$$\begin{aligned} g'(x_1, \dots, x_n, 0) &= g(x_1, \dots, x_n, l(x_1, \dots, x_n)) \\ g'(x_1, \dots, x_n, 1) &= g(x_1, \dots, x_n, 1 + l(x_1, \dots, x_n)) \\ &\vdots \\ g'(x_1, \dots, x_n, y) &= g(x_1, \dots, x_n, y + l(x_1, \dots, x_n)). \end{aligned}$$

By Theorem 13.3.1, the function

$$\begin{aligned} f'(x_1, \dots, x_n, y) &= \sum_{i=0}^y g'(x_1, \dots, x_n, i) \\ &= \sum_{i=l(x_1, \dots, x_n)}^{y+l(x_1, \dots, x_n)} g(x_1, \dots, x_n, i) \end{aligned}$$

is primitive recursive. The generalized bounded sum can be obtained by composing f' with the functions u and l :

$$f'(x_1, \dots, x_n, (u(x_1, \dots, x_n) \dot{-} l(x_1, \dots, x_n))) = \sum_{i=l(x_1, \dots, x_n)}^{u(x_1, \dots, x_n)} g(x_1, \dots, x_n, i).$$

Multiplying this function by the predicate that compares the upper and lower bounds ensures that the bounded sum returns the default value whenever the lower bound exceeds the upper bound. Thus

$$\begin{aligned} f(x_1, \dots, x_n) &= \text{cosg}(gt(l(x_1, \dots, x_n), u(x_1, \dots, x_n))) \\ &\quad \cdot f'(x_1, \dots, x_n, (u(x_1, \dots, x_n) \dot{-} l(x_1, \dots, x_n))). \end{aligned}$$

Since each of the constituent functions is primitive recursive, it follows that f is also primitive recursive.

A similar argument can be used to show that the generalized bounded product is primitive recursive. When the lower bound is greater than the upper, the bounded product defaults to one. ■

The value returned by a predicate p designates whether the input satisfies the property represented by p . For fixed values x_1, \dots, x_n ,

$$\mu z[p(x_1, \dots, x_n, z)]$$

is defined to be the smallest natural number z such that $p(x_1, \dots, x_n, z) = 1$. The notation $\mu z[p(x_1, \dots, x_n, z)]$ is read “the least z satisfying $p(x_1, \dots, x_n, z)$.” This construction is called the *minimization* of p , and μz is called the μ -operator. The minimization of an $n + 1$ -variable predicate defines an n -variable function

$$f(x_1, \dots, x_n) = \mu z[p(x_1, \dots, x_n, z)].$$

An intuitive interpretation of minimization is that it performs a search over the natural numbers. Initially, the variable z is set to zero. The search sequentially examines the natural numbers until a value of z for which $p(x_1, \dots, x_n, z) = 1$ is encountered.

Unfortunately, the predicate need not be

Consider the function

Using the characteriz
 $z^2 = x$. If x is a per
undefined.

By restricting the
minimalization opera

$$f(x_1, \dots, x_n)$$

The bounded μ -opera
 $p(x_1, \dots, x_n, z) = 1$
the search to the rang
function

In fact, the bounded n
ever the predicate is p

Theorem 13.3.3

Let $p(x_1, \dots, x_n, y)$

is primitive recursive.

Proof. The proof is
n-variable predicates.

This predicate is prim
predicate $\text{cosg} \circ p$.

Unfortunately, the function obtained by the minimization of a primitive recursive predicate need not be primitive recursive. In fact, such a function may not even be total. Consider the function

$$f(x) = \mu z[eq(x, z \cdot z)].$$

Using the characterization of minimization as search, f searches for the first z such that $z^2 = x$. If x is a perfect square, then $f(x)$ returns the square root of x . Otherwise, f is undefined.

By restricting the range over which the minimization occurs, we obtain a bounded minimization operator. An $n + 1$ -variable predicate defines an $n + 1$ -variable function

$$\begin{aligned} f(x_1, \dots, x_n, y) &= \mu z[p(x_1, \dots, x_n, z)] \\ &= \begin{cases} z & \text{if } p(x_1, \dots, x_n, i) = 0 \text{ for } 0 \leq i < z \leq y \\ & \quad \text{and } p(x_1, \dots, x_n, z) = 1 \\ y+1 & \text{otherwise.} \end{cases} \end{aligned}$$

The bounded μ -operator returns the first natural number z less than or equal to y for which $p(x_1, \dots, x_n, z) = 1$. If no such value exists, the default value of $y + 1$ is assigned. Limiting the search to the range of natural numbers between zero and y ensures the totality of the function

$$f(x_1, \dots, x_n, y) = \mu z[p(x_1, \dots, x_n, z)].$$

In fact, the bounded minimization operator defines a primitive recursive function whenever the predicate is primitive recursive.

Theorem 13.3.3

Let $p(x_1, \dots, x_n, y)$ be a primitive recursive predicate. Then the function

$$f(x_1, \dots, x_n, y) = \mu z[p(x_1, \dots, x_n, z)]$$

is primitive recursive.

Proof. The proof is given for a two-variable predicate $p(x, y)$ and easily generalizes to n -variable predicates. We begin by defining an auxiliary predicate

$$\begin{aligned} g(x, y) &= \begin{cases} 1 & \text{if } p(x, i) = 0 \text{ for } 0 \leq i \leq y \\ 0 & \text{otherwise} \end{cases} \\ &= \prod_{i=0}^y \cosg(p(x, i)). \end{aligned}$$

This predicate is primitive recursive since it is a bounded product of the primitive recursive predicate $\cosg \circ p$.

The bounded sum of the predicate g produces the bounded μ -operator. To illustrate the use of g in constructing the minimalization operator, consider a two-variable predicate p with argument n whose values are given in the left column:

$p(n, 0) = 0$	$g(n, 0) = 1$	$\sum_{i=0}^0 g(n, i) = 1$
$p(n, 1) = 0$	$g(n, 1) = 1$	$\sum_{i=0}^1 g(n, i) = 2$
$p(n, 2) = 0$	$g(n, 2) = 1$	$\sum_{i=0}^2 g(n, i) = 3$
$p(n, 3) = 1$	$g(n, 3) = 0$	$\sum_{i=0}^3 g(n, i) = 3$
$p(n, 4) = 0$	$g(n, 4) = 0$	$\sum_{i=0}^4 g(n, i) = 3$
$p(n, 5) = 1$	$g(n, 5) = 0$	$\sum_{i=0}^5 g(n, i) = 3$
⋮	⋮	⋮

The value of g is one until the first number z with $p(n, z) = 1$ is encountered. All subsequent values of g are zero. The bounded sum adds the results generated by g . Thus

$$\sum_{i=0}^y g(n, i) = \begin{cases} y+1 & \text{if } z > y \\ z & \text{otherwise.} \end{cases}$$

The first condition also includes the possibility that there is no z satisfying $p(n, z) = 1$. In this case the default value is returned regardless of the specified range.

By the preceding argument, we see that the bounded minimalization of a primitive recursive predicate p is given by the function

$$f(x, y) = \mu z[p(x, z)] = \sum_{i=0}^y g(x, i),$$

and consequently is primitive recursive. ■

Bounded minimalization $f(y) = \mu z[p(x, z)]$ can be thought of as a search for the first value of z in the range 0 to y that makes p true. Example 13.3.1 shows that minimalization can also be used to find first value in a subrange or the largest value z in a specified range that satisfies p .

Example 13.3.1

Let $p(x, z)$ be a primitive recursive predicate.

- i) $f_1(x, y_0, y) = \mu z[p(x, z)]$
- ii) $f_2(x, y) = \text{the first } z \text{ such that } p(x, z) \text{ is true}$
- iii) $f_3(x, y) = \text{the largest } z \text{ such that } p(x, z) \text{ is true}$

are also primitive recursive. Find the first value of z that satisfies p .

To show that f_1 is primitive recursive, note that f_1 is equal to $\mu z[p(x, z) \cdot ge(z, y_0)]$. This is the bounded minimalization of the predicate $p(x, z) \cdot ge(z, y_0)$.

returns the first value of z that satisfies p .

The minimalization operator μ is primitive recursive. The second value that satisfies p is returned by f_2 .

returns the second value of z that satisfies p .

A search for the first value of z in the range $0, 1, \dots, y-2, \dots, 1, 0$ that makes p true is primitive recursive. The desired order; when $z = 0$, $z = 1$, \dots , $z = y-2$, $z = y-1$, $z = y$. The function $f'(x, y) = \mu z[p(x, z)]$ is primitive recursive. How does it work? The comparison is used to determine if $p(x, z)$ is true.

$$f_3(x, y) = eq(y, 0)$$

returns the default value if no such value is returned.

Bounded minimalization is similar to bounded search with a function. The search is similar to that of Theorem 13.3.1.

rate the
icate p

Example 13.3.1

Let $p(x, z)$ be a primitive recursive predicate. Then the functions

- i) $f_1(x, y_0, y) =$ the first value in the range $[y_0, y]$ for which $p(x, z)$ is true,
- ii) $f_2(x, y) =$ the second value in the range $[0, y]$ for which $p(x, z)$ is true, and
- iii) $f_3(x, y) =$ the largest value in the range $[0, y]$ for which $p(x, z)$ is true

are also primitive recursive. For each of these functions, the default is $y + 1$ if there is no value of z that satisfies the specified condition.

To show that f_1 is primitive recursive, the primitive recursive function ge , greater than or equal to, is used to enforce a lower bound on the value of the function. The predicate $p(x, z) \cdot ge(z, y_0)$ is true whenever $p(x, z)$ is true and z is greater than or equal to y_0 . The bounded minimalization

$$f_1(x, y_0, y) = \mu z^y [p(x, z) \cdot ge(z, y_0)],$$

returns the first value in the range $[y_0, y]$ for which $p(x, z)$ is true.

The minimalization $\mu z' [p(x, z')]$ is the first value in $[0, y]$ for which $p(x, z)$ is true. The second value that makes $p(x, z)$ true is the first value greater than $\mu z' [p(x, z')]$ that satisfies p . Using the preceding technique, the function

$$f_2(x, y) = \mu z^y [p(x, z) \cdot gt(z, \mu z' [p(x, z')])]$$

returns the second value in the range $[0, y]$ for which p is true.

A search for the largest value in the range $[0, y]$ must sequentially examine $y, y - 1, y - 2, \dots, 1, 0$. The bounded minimalization $\mu z^y [p(x, y - z)]$ examines the values in the desired order; when $z = 0$, $p(x, y)$ is tested, when $z = 1$, $p(x, y - 1)$ is tested, and so on. The function $f'(x, y) = y - \mu z^y [p(x, y - z)]$ returns the largest value less than or equal to y that satisfies p . However, the result of f' is $y - (y + 1) = 0$ when no such value exists. A comparison is used to produce the proper default value. The first condition in the function

$$f_3(x, y) = eq(y + 1, \mu z^y [p(x, z)]) \cdot (y + 1) + neq(y + 1, \mu z^y [p(x, z)]) \cdot f'(x, y)$$

returns the default $y + 1$ if there is no value in $[0, y]$ that satisfies p . Otherwise, the largest such value is returned. \square

bsequent

$z) = 1$. In
primitive

or the first
malization
ified range

Bounded minimalization can be generalized by computing the upper bound of the search with a function u . If u is primitive recursive, so is the resulting function. The proof is similar to that of Theorem 13.3.2 and is left as an exercise.

Theorem 13.3.4

Let p be an $n + 1$ -variable primitive recursive predicate and let u be an n -variable primitive recursive function. Then the function

$$f(x_1, \dots, x_n) = \frac{u(x_1, \dots, x_n)}{\mu z} [p(x_1, \dots, x_n, z)]$$

is primitive recursive.

13.4 Division Functions

The fundamental operation of integer division, *div*, is not total. The function $\text{div}(x, y)$ returns the quotient, the integer part of the division of x by y , when the second argument is nonzero. The function is undefined when y is zero. Since all primitive recursive functions are total, it follows that *div* is not primitive recursive. A primitive recursive division function *quo* is defined by assigning a default value when the denominator is zero:

$$quo(x, y) = \begin{cases} 0 & \text{if } y = 0 \\ div(x, y) & \text{otherwise.} \end{cases}$$

The division function *quo* is constructed using the primitive recursive operation of multiplication. For values of y other than zero, $\text{quo}(x, y) = z$ implies that z satisfies $z \cdot y \leq x < (z + 1) \cdot y$. That is, $\text{quo}(x, y)$ is the smallest natural number z such that $(z + 1) \cdot y$ is greater than x . The search for the value of z that satisfies the inequality succeeds before z reaches x since $(x + 1) \cdot y$ is greater than x . The function

$$\mu z[gt((z+1) \cdot y, x)]$$

determines the quotient of x and y whenever the division is defined. The default value is obtained by multiplying the minimalization by $sg(y)$. Thus

$$quo(x, y) = sg(y) \cdot \mu z[gt((z + 1) \cdot y, x)],$$

where the bound is determined by the primitive recursive function $p_1^{(2)}$. The previous definition demonstrates that quo is primitive recursive since it has the form prescribed by Theorem 13.3.4.

The quotient function can be used to define a number of division-related functions and predicates including those given in Table 13.3. The function *rem* returns the remainder of the division of x by y whenever the division is defined. Otherwise, $\text{rem}(x, 0) = x$. The predicate *divides* defined by

$$\text{divides}(x, y) = \begin{cases} 1 & \text{if } x > 0, y > 0, \text{ and } y \text{ is a divisor of } x \\ 0 & \text{otherwise} \end{cases}$$

is true whenever y divides x . By convention, zero is not considered to be divisible by any number. The multiplication by $sg(x)$ in the definition of *divides* in Table 13.3 enforces this condition. The default value of the remainder function guarantees that $\text{divides}(x, 0) = 0$.

TABLE 13.3

Description
Quotient
Remainder
Divides
Number of div.
Prime

The generalized bound of the number of divisors is a number who

The predicate prime recursive function p_n that $p_n(0) = 2$, $p_n(1) = 3$, number greater than $p_n(n)$ type of search. To employ for the minimization. If value x ,

Lemma 13.4.1

Let $p_n(x)$ denote the x th

Proof. Each of the primes divide two consecutive numbers that contains a prime other than p .

The bound provided by the function $\text{fact}(x) + 1$. The

nu(0)

$$pn(x)$$

Let us take a moment to consider the family of primitive recursive functions.

TABLE 13.3 Primitive Recursive Division Functions

Description	Function	Definition
Quotient	$quo(x, y)$	$sg(y) \cdot \mu z[gt((z + 1) \cdot y, x)]$
Remainder	$rem(x, y)$	$x - (y \cdot quo(x, y))$
Divides	$divides(x, y)$	$eq(rem(x, y), 0) \cdot sg(x)$
Number of divisors	$ndivisors(x, y)$	$\sum_{i=0}^x divides(x, i)$
Prime	$prime(x)$	$eq(ndivisors(x), 2)$

The generalized bounded sum can be used to count the number of divisors of a number. The upper bound of the sum is obtained from the input by the primitive recursive function $p_1^{(1)}$. This bound is satisfactory since no number greater than x is a divisor of x . A prime number is a number whose only divisors are 1 and itself. The predicate *prime* simply checks if the number of divisors is two.

The predicate *prime* and bounded minimalization can be used to construct a primitive recursive function *pn* that enumerates the primes. The value of *pn*(i) is the i th prime. Thus, $pn(0) = 2$, $pn(1) = 3$, $pn(2) = 5$, $pn(3) = 7$, The $x + 1$ st prime is the first prime number greater than $pn(x)$. Bounded minimalization is ideally suited for performing this type of search. To employ the bounded μ -operator, we must determine an upper bound for the minimalization. By Theorem 13.3.4, the bound may be calculated using the input value x .

Lemma 13.4.1

Let *pn*(x) denote the x th prime. Then $pn(x + 1) \leq pn(x)! + 1$.

Proof. Each of the primes $pn(i)$, $i = 0, 1, \dots, x$, divides $pn(x)!$. Since a prime cannot divide two consecutive numbers, either $pn(x)! + 1$ is prime or its prime decomposition contains a prime other than $pn(0), pn(1), \dots, pn(x)$. In either case, $pn(x + 1) \leq pn(x)! + 1$. ■

The bound provided by the preceding lemma is computed by the primitive recursive function *fact*(x) + 1. The x th prime function is obtained by primitive recursion as follows:

$$\begin{aligned} pn(0) &= 2 \\ pn(x + 1) &= \mu z^{fact(pn(x)) + 1} [prime(z) \cdot gt(z, pn(x))]. \end{aligned}$$

Let us take a moment to reflect on the consequences of the relationship between the family of primitive recursive functions and Turing computability. By Theorem 13.1.3, every

primitive recursive function is Turing computable. Designing Turing machines that explicitly compute functions such as pn or $n\text{divisors}$ would require a large number of states and a complicated transition function. Using the macroscopic approach to computation, these functions are easily shown to be computable. Without the tedium inherent in constructing complicated Turing machines, we have shown that many useful functions and predicates are Turing computable.

A decoding function

The function

returns the i th element of a sequence. The μ -operator is used to implement minimization returning the smallest index for which a given condition is true. The λ -operator is used to encode an element in an encoding function. The λ -operator is used to decode a value from its encoding. The λ -operator is used to implement prime decomposition by finding the largest power of a prime that divides a given number.

When a computation needs to be repeated, the gn_{n-1} function can be used to encode a sequence of numbers. When they are needed, the dec function can be used to decode the elements of the sequence.

13.5 Gödel Numbering and Course-of-Values Recursion

Many common computations involving natural numbers are not number-theoretic functions. Sorting a sequence of numbers returns a sequence, not a single number. However, there are many sorting algorithms that we consider effective procedures. We now introduce primitive recursive constructions that allow us to perform this type of operation. The essential feature is the ability to encode a sequence of numbers in a single value. The coding scheme utilizes the unique decomposition of a natural number into a product of primes. Such codes are called *Gödel numberings* after German logician Kurt Gödel, who developed the technique.

A sequence x_0, x_1, \dots, x_{n-1} of n natural numbers is encoded by

$$pn(0)^{x_0+1} \cdot pn(1)^{x_1+1} \cdots \cdot pn(n)^{x_n+1} = 2^{x_0+1} \cdot 3^{x_1+1} \cdots \cdot pn(n)^{x_n+1}.$$

Since our numbering begins with zero, the elements of a sequence of length n are numbered $0, 1, \dots, n - 1$. Examples of the Gödel numbering of several sequences are

Sequence	Encoding
1, 2	$2^2 3^3 = 108$
0, 1, 3	$2^1 3^2 5^4 = 11,250$
0, 1, 0, 1	$2^1 3^2 5^1 7^2 = 4,410$

An encoded sequence of length n is a product of powers of the first n primes. The choice of the exponent $x_i + 1$ guarantees that $pn(i)$ occurs in the encoding even when x_i is zero.

The definition of a function that encodes a fixed number of inputs can be obtained directly from the definition of the Gödel numbering. We let

$$gn_n(x_0, \dots, x_n) = pn(0)^{x_0+1} \cdots \cdot pn(n)^{x_n+1} = \prod_{i=0}^n pn(i)^{x_i+1}$$

be the $n + 1$ -variable function that encodes a sequence x_0, x_1, \dots, x_n . The function gn_{n-1} can be used to encode the components of an ordered n -tuple. The Gödel number associated with the ordered pair $[x_0, x_1]$ is $gn_1(x_0, x_1)$.

generates the Fibonacci sequence. The computation of $f(y)$ for a given y is obtained by summing the previous two terms. The gn_1 function is used to store the current term and the previous term in an ordered pair with first component y .

$h(0) = 0$
 $h(y + 1) = h(y) + gn_1(h(y), y)$

The initial value of h is 0. Subsequent values are produced by producing the code for the next term in the sequence.

$[dec(1, h(y)), h(y + 1)]$

Encoding the pair works by summing the previous two terms in the sequence of Gödel numbers. The primitive recursive function gn_1 encodes the first components of the pairs.

The Gödel numbering function gn_n encodes the sequence of Gödel numbers.

chines that explicitly number states and computation, these are used in constructing functions and predicates

Section

theoretic functions. However, there are ways to introduce primitive recursive functions. The essential feature of this coding scheme utilizes Gödel numbers. Such codes are generated by applying the technique.

$$m(n)^{x_n+1}$$

th \$n\$ are numbered

times. The choice of \$x_i\$ when \$x_i\$ is zero. This can be obtained

$$x_i+1$$

he function \$gn_{n-1}\$

A decoding function is constructed to retrieve the components of an encoded sequence. The function

$$dec(i, x) = \mu z [cosg(divides(x, pn(i)^{z+1}))] - 1$$

returns the \$i\$th element of the sequence encoded in the Gödel number \$x\$. The bounded \$\mu\$-operator is used to find the power of \$pn(i)\$ in the prime decomposition of \$x\$. The minimalization returns the first value of \$z\$ for which \$pn(i)^{z+1}\$ does not divide \$x\$. The \$i\$th element in an encoded sequence is one less than the power of \$pn(i)\$ in the encoding. The decoding function \$dec(x, i)\$ returns zero for every prime \$pn(i)\$ that does not occur in the prime decomposition of \$x\$.

When a computation requires \$n\$ previously computed values, the Gödel encoding function \$gn_{n-1}\$ can be used to encode the values. The encoded values can be retrieved when they are needed by the computation.

Example 13.5.1

The Fibonacci numbers are defined as the sequence \$0, 1, 1, 2, 3, 5, 8, 13, \dots\$, where an element in the sequence is the sum of its two predecessors. The function

$$f(0) = 0$$

$$f(1) = 1$$

$$f(y + 1) = f(y) + f(y - 1) \text{ for } y > 1$$

generates the Fibonacci numbers. This is not a definition by primitive recursion since the computation of \$f(y + 1)\$ utilizes both \$f(y)\$ and \$f(y - 1)\$. To show that the Fibonacci numbers are generated by a primitive recursive function, the Gödel numbering function \$gn_1\$ is used to store the two values as a single number. An auxiliary function \$h\$ encodes the ordered pair with first component \$f(y - 1)\$ and second component \$f(y)\$:

$$h(0) = gn_1(0, 1) = 2^1 3^2 = 18$$

$$h(y + 1) = gn_1(dec(1, h(y)), dec(0, h(y)) + dec(1, h(y))).$$

The initial value of \$h\$ is the encoded pair \$[f(0), f(1)]\$. The calculation of \$h(y + 1)\$ begins by producing the components of the subsequent ordered pair

$$[dec(1, h(y)), dec(0, h(y)) + dec(1, h(y))] = [f(y), f(y - 1) + f(y)].$$

Encoding the pair with \$gn_1\$ completes the evaluation of \$h(y + 1)\$. This process constructs the sequence of Gödel numbers of the pairs \$[f(0), f(1)], [f(1), f(2)], [f(2), f(3)], \dots\$. The primitive recursive function \$f(y) = dec(0, h(y))\$ extracts the Fibonacci numbers from the first components of the ordered pairs. □

The Gödel numbering functions \$gn_i\$ encode a fixed number of arguments. A Gödel numbering function can be constructed in which the number of elements to be encoded

is computed from the arguments of the function. The approach is similar to that taken in constructing the bounded sum and product operations. The values of a one-variable primitive recursive function f with input $0, 1, \dots, n$ define a sequence $f(0), f(1), \dots, f(n)$ of length $n + 1$. Using the bounded product, the Gödel numbering function

$$gn_f(x_1, \dots, x_n, y) = \prod_{i=0}^y pn(i)^{f(i)+1}$$

encodes the first $y + 1$ values of f . The relationship between a function f and its encoding function gn_f is established in Theorem 13.5.1.

Theorem 13.5.1

Let f be an $n + 1$ -variable function and gn_f the encoding function defined from f . Then f is primitive recursive if, and only if, gn_f is primitive recursive.

Proof. If $f(x_1, \dots, x_n, y)$ is primitive recursive, then the bounded product

$$gn_f(x_1, \dots, x_n, y) = \prod_{i=0}^y pn(i)^{f(x_1, \dots, x_n, i)+1}$$

computes the Gödel encoding function. On the other hand, the decoding function can be used to recover the values of f from the Gödel number generated by gn_f :

$$f(x_1, \dots, x_n, y) = dec(y, gn_f(x_1, \dots, x_n, y)).$$

Thus f is primitive recursive whenever gn_f is. ■

The primitive recursive functions have been introduced because of their intuitive computability. In a definition by primitive recursion, the computation is permitted to use the result of the function with the previous value of the recursive variable. Consider the function defined by

$$\begin{aligned} f(0) &= 1 \\ f(1) &= f(0) \cdot 1 = 1 \\ f(2) &= f(0) \cdot 2 + f(1) \cdot 1 = 3 \\ f(3) &= f(0) \cdot 3 + f(1) \cdot 2 + f(2) \cdot 1 = 8 \\ f(4) &= f(0) \cdot 4 + f(1) \cdot 3 + f(2) \cdot 2 + f(3) \cdot 1 = 21 \\ &\vdots \end{aligned}$$

The function f can be written as

$$\begin{aligned} f(0) &= 1 \\ f(y+1) &= \sum_{i=0}^y f(i) \cdot (y+1-i). \end{aligned}$$

The definition, as $gn_f(y+1)$ utilizes all previously computable; it is calculated.

When the result $f(0), f(1), \dots, f(y)$ is determined, a different number of example, $f(2)$ requires $f(3)$. No single function preceding values since

Regardless of the encoded in the Gödel definition of course-of-

Definition 13.5.2

Let g and h be n and $n + 1$ -variable functions

- i) $f(x_1, \dots, x_n, 0)$
 - ii) $f(x_1, \dots, x_n, y)$
- is said to be obtained

Theorem 13.5.3

Let f be an $n + 1$ -variable recursive functions g and h .

Proof. We begin by recursive functions g and h .

$$gn_f(x_1, \dots, x_n, y)$$

$$gn_f(x_1, \dots, x_n, y)$$

The evaluation of gn_f depends on:

- i) the parameters x_0, x_1, \dots, x_n
- ii) y , the previous value of f
- iii) $gn_f(x_1, \dots, x_n, 0)$
- iv) the primitive recursive function h .

Thus, the function gn_f is primitive recursive.

aken in
imitive
 $f(n)$ of

ncoding

f. Then

can be

ive com-
use the
function

The definition, as formulated, is not primitive recursive since the computation of $f(y+1)$ utilizes all of the previously computed values. The function, however, is intuitively computable; the definition itself outlines an algorithm by which any value can be calculated.

When the result of a function with recursive variable $y+1$ is defined in terms of $f(0), f(1), \dots, f(y)$, the function f is said to be defined by course-of-values recursion. Determining the result of a function defined by course-of-values recursion appears to utilize a different number of inputs for each value of the recursive variable. In the preceding example, $f(2)$ requires only $f(0)$ and $f(1)$, while $f(4)$ requires $f(0), f(1), f(2)$, and $f(3)$. No single function can be used to compute both $f(2)$ and $f(4)$ directly from the preceding values since a function is required to have a fixed number of arguments.

Regardless of the value of the recursive variable $y+1$, the preceding results can be encoded in the Gödel number $gn_f(y)$. This observation provides the framework for a formal definition of course-of-values recursion.

Definition 13.5.2

Let g and h be $n+2$ -variable total number-theoretic functions, respectively. The $n+1$ -variable function f defined by

- i) $f(x_1, \dots, x_n, 0) = g(x_1, \dots, x_n)$
- ii) $f(x_1, \dots, x_n, y+1) = h(x_1, \dots, x_n, y, gn_f(x_1, \dots, x_n, y))$

is said to be obtained from g and h by **course-of-values recursion**.

Theorem 13.5.3

Let f be an $n+1$ -variable function defined by course-of-values recursion from primitive recursive functions g and h . Then f is primitive recursive.

Proof. We begin by defining gn_f by primitive recursion directly from the primitive recursive functions g and h .

$$\begin{aligned} gn_f(x_1, \dots, x_n, 0) &= 2^{f(x_1, \dots, x_n, 0)+1} \\ &= 2^{g(x_1, \dots, x_n)+1} \\ gn_f(x_1, \dots, x_n, y+1) &= gn_f(x_1, \dots, x_n, y) \cdot pn(y+1)^{f(x_1, \dots, x_n, y+1)+1} \\ &= gn_f(x_1, \dots, x_n, y) \cdot pn(y+1)^{h(x_1, \dots, x_n, y, gn_f(x_1, \dots, x_n, y))+1} \end{aligned}$$

The evaluation of $gn_f(x_1, \dots, x_n, y+1)$ uses only

- i) the parameters x_0, \dots, x_n ,
- ii) y , the previous value of the recursive variable,
- iii) $gn_f(x_1, \dots, x_n, y)$, the previous value of gn_f , and
- iv) the primitive recursive functions h , pn , \cdot , $+$, and exponentiation.

Thus, the function gn_f is primitive recursive. By Theorem 13.5.1, it follows that f is also primitive recursive. ■

In mechanical terms, the Gödel numbering gives computation the equivalent of unlimited memory. A single Gödel number is capable of storing any number of preliminary results. The Gödel numbering encodes the values $f(x_0, \dots, x_n, 0)$, $f(x_0, \dots, x_n, 1), \dots, f(x_0, \dots, x_n, y)$ that are required for the computation of $f(x_0, \dots, x_n, y+1)$. The decoding function provides the connection between the memory and the computation. Whenever a stored value is needed by the computation, the decoding function makes it available.

Example 13.5.2

Let h be the primitive recursive function

$$h(x, y) = \sum_{i=0}^x dec(i, y) \cdot (x + 1 - i).$$

The function f , which was defined earlier to introduce course-of-values computation, can be defined by course-of-values recursion from h .

$$\begin{aligned} f(0) &= 1 \\ f(y+1) &= h(y, gn_f(y)) = \sum_{i=0}^y dec(i, gn_f(y)) \cdot (y + 1 - i) \\ &= \sum_{i=0}^y f(i) \cdot (y + 1 - i) \quad \square \end{aligned}$$

13.6 Computable Partial Functions

The primitive recursive functions were defined as a family of intuitively computable functions. We have established that all primitive recursive functions are total. Conversely, are all computable total functions primitive recursive? Moreover, should we restrict our analysis of computability to total functions? In this section we will present arguments for a negative response to both of these questions.

We will use a diagonalization argument to establish the existence of a total computable function that is not primitive recursive. The first step is to show that the syntactic structure of the primitive recursive functions allows them to be effectively enumerated. The ability to list the primitive recursive functions permits the construction of a computable function that differs from every function in the list.

Theorem 13.6.1

The set of primitive recursive functions is a proper subset of the set of effectively computable total number-theoretic functions.

Proof. The primitive recursive functions can be represented as strings over the alphabet $\Sigma = \{s, p, z, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, (,), \circ, :, \langle, \rangle\}$. The basic functions s , z , and $p_i^{(j)}$

are represented by $\langle s \rangle$, $\langle \langle h \rangle \circ \langle \langle g_1 \rangle, \dots, \langle g_n \rangle \rangle \rangle$, and $\langle \langle g \rangle : \langle h \rangle \rangle$.

The strings in Σ^* of length one, length two, and so on, determine whether a string is correctly formed, enumerate the primitive recursive functions. A function g is represented by $\langle \langle g \rangle : \langle h \rangle \rangle$.

The total one-variable

is effectively computable. This establishes the theorem.

- i) determining the type of f
- ii) computing $f_i^{(1)}(i)$
- iii) adding one to $f_i^{(1)}$

Since each of these steps is effectively computable, the diagonalization argument

for any i . Consequently,

Theorem 13.6.1 uses a diagonalization argument to show that there are total computable functions that are not primitive recursive. A computable function that is not primitive recursive is called a partial function, known as A .

- i) $A(0, y) = y + 1$
- ii) $A(x + 1, 0) = A(x)$
- iii) $A(x + 1, y + 1) = A(x, y) + 1$

is one such function. The proof is as follows:

- (i). A proof by induction on x shows that $A(x, y)$ has the same values as $f(x, y)$ for all y (Exercise 22).
- (ii). By Theorem 13.6.1, A is not primitive recursive. Since A is total, it must be Ackermann's function.

of unlim-
inary re-
, 1), . . . ,
The decod-
Whenever
ilable.

tation, can

are represented by $\langle s \rangle$, $\langle z \rangle$, and $\langle pi(j) \rangle$. The composition $h \circ (g_1, \dots, g_n)$ is encoded $\langle \langle h \rangle \circ (\langle g_1 \rangle, \dots, \langle g_n \rangle) \rangle$, where $\langle h \rangle$ and $\langle g_i \rangle$ are the representations of the constituent functions. A function defined by primitive recursion from functions g and h is represented by $\langle \langle g \rangle : \langle h \rangle \rangle$.

The strings in Σ^* can be generated by length: first the null string, followed by strings of length one, length two, and so on. A straightforward mechanical process can be designed to determine whether a string represents a correctly formed primitive recursive function. The enumeration of the primitive recursive functions is accomplished by repeatedly generating a string and determining if it is a syntactically correct representation of a function. The first correctly formed string is denoted f_0 , the next f_1 , and so on. In the same manner, we can enumerate the one-variable primitive recursive functions. This is accomplished by deleting all n -variable functions, $n > 1$, from the previously generated list. This sequence is denoted $f_0^{(1)}, f_1^{(1)}, f_2^{(1)}, \dots$

The total one-variable function

$$g(i) = f_i^{(1)}(i) + 1$$

is effectively computable. The effective enumeration of the one-variable primitive recursive functions establishes the computability of g . The value $g(i)$ is obtained by

- i) determining the i th one-variable primitive recursive function $f_i^{(1)}$,
- ii) computing $f_i^{(1)}(i)$, and
- iii) adding one to $f_i^{(1)}(i)$.

Since each of these steps is effective, we conclude that g is computable. By the familiar diagonalization argument,

$$g(i) \neq f_i^{(1)}(i)$$

for any i . Consequently, g is total and computable but not primitive recursive. ■

Theorem 13.6.1 used diagonalization to demonstrate the existence of computable functions that are not primitive recursive. This can also be accomplished directly by constructing a computable function that is not primitive recursive. The two-variable number-theoretic function, known as *Ackermann's function*, defined by

- i) $A(0, y) = y + 1$
- ii) $A(x + 1, 0) = A(x, 1)$
- iii) $A(x + 1, y + 1) = A(x, A(x + 1, y))$

is one such function. The values of A are defined recursively with the basis given in condition (i). A proof by induction on x establishes that A is uniquely defined for every pair of input values (Exercise 22). The computations in Example 13.6.1 illustrate the computability of Ackermann's function.

Example 13.6.1

The values $A(1, 1)$ and $A(3, 0)$ are constructed from the definition of Ackermann's function. The column on the right gives the justification for the substitution.

$$\begin{aligned}
 \text{a) } A(1, 1) &= A(0, A(1, 0)) && \text{(iii)} \\
 &= A(0, A(0, 1)) && \text{(ii)} \\
 &= A(0, 2) && \text{(i)} \\
 &= 3
 \end{aligned}$$

$$\begin{aligned}
 \text{b) } A(2, 1) &= A(1, A(2, 0)) && \text{(iii)} \\
 &= A(1, A(1, 1)) && \text{(ii)} \\
 &= A(1, 3) && \text{(a)} \\
 &= A(0, A(1, 2)) && \text{(iii)} \\
 &= A(0, A(0, A(1, 1))) && \text{(iii)} \\
 &= A(0, A(0, 3)) && \text{(a)} \\
 &= A(0, 4) && \text{(i)} \\
 &= 5 && \text{(i)}
 \end{aligned}$$

□

The values of Ackermann's function exhibit a remarkable rate of growth. By fixing the first variable, Ackermann's function generates the one-variable functions

$$A(1, y) = y + 2$$

$$A(2, y) = 2y + 3$$

$$A(3, y) = 2^{y+3} - 3$$

$$A(4, y) = 2^{2^{\dots^{2^{16}}}} - 3.$$

The number of 2's in the exponential chain in $A(4, y)$ is y . For example, $A(4, 0) = 16 - 3$, $A(4, 1) = 2^{16} - 3$, and $A(4, 2) = 2^{2^{16}} - 3$. The first variable of Ackermann's function determines the rate of growth of the function values. We state, without proof, the following theorem that compares the rate of growth of Ackermann's function with that of the primitive recursive functions.

Theorem 13.6.2

For every one-variable primitive recursive function f , there is some $i \in \mathbb{N}$ such that $f(i) < A(i, i)$.

Clearly, the one-variable function $A(i, i)$ obtained by identifying the variables of A is not primitive recursive. It follows that Ackermann's function is not primitive recursive. If it

were, then $A(0, 0)$ would be primitive recursive.

Is it possible to find some new basic functions that are computable, that show that there must conclude that there are computable functions from the diagonal? The answer is that $g(i) \neq A(i, i)$ for all i . We be able to effect this by defining functions in the style of the previous section.

We now consider primitive recursive functions that are obtained from the basic functions by the bounded minimization operator. Place the bounded minimization operator is obtained by the procedure. Place the bounded minimization operator to be considered.

defined by unbounded minimization. Otherwise, we have a square. Otherwise, we have ad infinitum. A function f is primitive recursive if and only if $f(3) \uparrow$. A function f is primitive recursive if and only if the search fails to find a value for $f(3)$.

The introduction of the bounded minimization operator and primitive recursive functions is based on the definition of primitive recursive functions if either

- i) $g_i(x_1, \dots, x_n)$
- ii) $g_i(x_1, \dots, x_n)$

An undefined value is called a non-computable function.

The operator μ is called the bounded minimization operator. This operator is total. This means that it always returns a value. Let f be a partial function. Then $\mu x f(x) = \mu x f(x)$.

were, then $A(i, i)$, which can be obtained by the composition $A \circ (p_1^{(1)}, p_1^{(1)})$, would also be primitive recursive.

s function.

Is it possible to increase the set of primitive recursive functions, possibly by adding some new basic functions or additional operations, to include all total computable functions? Unfortunately, the answer is no. Regardless of the set of total functions that we consider computable, the diagonalization argument in the proof of Theorem 13.6.1 can be used to show that there is no effective enumeration of all total computable functions. Therefore, we must conclude that the computable functions cannot be effectively generated or that there are computable nontotal functions. If we accept the latter proposition, the contradiction from the diagonalization disappears. The reason we can claim that g is not one of the f_i 's is that $g(i) \neq f_i^{(1)}(i)$. If $f_i^{(1)}(i) \uparrow$, then $g(i) = f_i^{(1)}(i) + 1$ is also undefined. If we wish to be able to effectively enumerate the computable functions, it is necessary to include partial functions in the enumeration.

We now consider the computability of partial functions. Since composition and primitive recursion preserve totality, an additional operation is needed to construct partial functions from the basic functions. Minimalization has been informally described as a search procedure. Placing a bound on the range of the natural numbers to be examined ensures that the bounded minimalization operation produces total functions. *Unbounded minimalization* is obtained by performing the search without an upper limit on the set of natural numbers to be considered. The function

$$f(x) = \mu z[eq(x, z \cdot z)]$$

defined by unbounded minimalization returns the square root of x whenever x is a perfect square. Otherwise, the search for the first natural number satisfying the predicate continues ad infinitum. Although eq is a total function, the resulting function f is not. For example, $f(3) \uparrow$. A function defined by unbounded minimalization is undefined for input x whenever the search fails to return a value.

The introduction of partial functions forces us to reexamine the operations of composition and primitive recursion. The possibility of undefined values was considered in the definition of composition. The function $h \circ (g_1, \dots, g_n)$ is undefined for input x_1, \dots, x_k if either

- i) $g_i(x_1, \dots, x_k) \uparrow$ for some $1 \leq i \leq n$; or
- ii) $g_i(x_1, \dots, x_k) \downarrow$ for all $1 \leq i \leq n$ and $h(g_1(x_1, \dots, x_k), \dots, g_n(x_1, \dots, x_k)) \uparrow$.

An undefined value propagates from any of the g_i 's to the composite function.

The operation of primitive recursion required both of the defining functions g and h to be total. This restriction is relaxed to permit definitions by primitive recursion using partial functions. Let f be defined by primitive recursion from partial functions g and h .

$$f(x_1, \dots, x_n, 0) = g(x_1, \dots, x_n)$$

$$f(x_1, \dots, x_n, y+1) = h(x_1, \dots, x_n, y, f(x_1, \dots, x_n, y))$$

Determining the value of a function defined by primitive recursion is an iterative process. The function f is defined for recursive variable y only if the following conditions are satisfied:

- i) $f(x_1, \dots, x_n, 0) \downarrow$ if $g(x_1, \dots, x_n) \downarrow$
- ii) $f(x_1, \dots, x_n, y+1) \downarrow$ if $f(x_1, \dots, x_n, i) \downarrow$ for $0 \leq i \leq y$
and $h(x_1, \dots, x_n, y, f(x_1, \dots, x_n, y)) \downarrow$.

An undefined value for the recursive variable causes f to be undefined for all the subsequent values of the recursive variable.

With the conventions established for definitions with partial functions, a family of computable partial functions can be defined using the operations composition, primitive recursion, and unbounded minimalization.

Definition 13.6.3

The family of μ -recursive functions is defined as follows:

- i) The successor, zero, and projection functions are μ -recursive.
- ii) If h is an n -variable μ -recursive function and g_1, \dots, g_n are k -variable μ -recursive functions, then $f = h \circ (g_1, \dots, g_n)$ is μ -recursive.
- iii) If g and h are n and $n+2$ -variable μ -recursive functions, then the function f defined from g and h by primitive recursion is μ -recursive.
- iv) If $p(x_1, \dots, x_n, y)$ is a total μ -recursive predicate, then $f = \mu z[p(x_1, \dots, x_n, z)]$ is μ -recursive.
- v) A function is μ -recursive only if it can be obtained from condition (i) by a finite number of applications of the rules in (ii), (iii), and (iv).

Conditions (i), (ii), and (iii) imply that all primitive recursive functions are μ -recursive. Notice that unbounded minimization is not defined for all predicates, but only for total μ -recursive predicates.

The notion of Turing computability encompasses partial functions in a natural way. A Turing machine computes a partial number-theoretic function f if

- i) the computation terminates with result $f(x_1, \dots, x_n)$ whenever $f(x_1, \dots, x_n) \downarrow$, and
- ii) the computation does not terminate whenever $f(x_1, \dots, x_n) \uparrow$.

The Turing machine computes the value of the function whenever possible. Otherwise, the computation continues indefinitely.

We will now establish the relationship between the μ -recursive and Turing computable functions. The first step is to show that every μ -recursive function is Turing computable. This is not a surprising result; it simply extends Theorem 13.1.3 to partial functions.

Theorem 13.6.4

Every μ -recursive function is Turing computable.

Proof. Since the basic operations are computable, we can use Theorems 9.4.3 and 9.4.4 to show that the Turing computable total predicates where $p(x_1, \dots, x_n)$ is μ -recursive are closed under composition and unbounded minimization. This shows that every μ -recursive function is Turing computable.

The proof is completed by showing that the Turing computable total predicates where $p(x_1, \dots, x_n)$ is μ -recursive can be constructed by applying the rules of Definition 13.6.3 to the configuration of the Turing machine.

1. The representation of the function specified by the predicate p .

The number to be computed is the value of the operator.

2. A working copy of the predicate p .

3. The machine P for computing the value of the predicate p .

4. If $p(x_1, x_2, \dots, x_n) \downarrow$, then the value of $p(x_1, x_2, \dots, x_n)$ is the result of step 2.

A computation terminates if no further applications of the rules of Definition 13.6.3 are encountered. If no such applications are found, the computation continues indefinitely. If no such applications are found, the function f is undefined.

13.7 Turing Computability

It has already been established that every μ -recursive function is Turing computable. We now turn to the question of whether every Turing computable function is μ -recursive. We will show that the computations of a Turing machine can be simulated by the computations of a μ -recursive function, moving from the domain of the function to the domain of the computation.

translating machine computations to functions is known as the *arithmetization* of Turing machines.

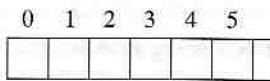
The arithmetization begins by assigning a number to a Turing machine configuration. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_n)$ be a standard Turing machine that computes a one-variable number-theoretic function f . We will construct a μ -recursive function to numerically simulate the computations of M . The construction easily generalizes to functions of more than one variable.

A configuration of the Turing machine M consists of the state, the position of the tape head, and the segment of the tape from the left boundary to the rightmost nonblank symbol. Each of these components must be represented by a natural number. We will denote the states and tape alphabet by

$$Q = \{q_0, q_1, \dots, q_n\}$$

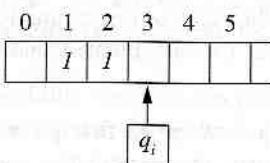
$$\Gamma = \{B = a_0, I = a_1, a_2, \dots, a_k\}$$

and the numbering will be obtained from the subscripts. Using this numbering, the tape symbols B and I are assigned zero and one, respectively. The location of the tape head can be encoded using the numbering of the tape positions.



The symbols on the tape to the rightmost nonblank square form a string over Σ^* . Encoding the tape uses the numeric representation of the elements of the tape alphabet. The string $a_{i_0}a_{i_1}\dots a_{i_n}$ is encoded by the Gödel number associated with the sequence i_0, i_1, \dots, i_n . The number representing the nonblank tape segment is called the *tape number*.

The tape number of the nonblank segment of the machine configuration



is $2^13^25^2 = 450$. Explicitly encoding the blank in position three produces $2^13^25^27^1 = 3150$, another tape number representing the tape. Any number of blanks to the right of the rightmost nonblank square may be included in the tape number.

Representing the blank by the number zero permits the correct decoding of any tape position regardless of the segment of the tape encoded in the tape number. If $dec(i, z) = 0$ and $pn(i)$ divides z , then the blank is specifically encoded in the tape number z . On the other hand, if $dec(i, z) = 0$ and $pn(i)$ does not divide z , then position i is to the right of the encoded segment of the tape. Since the tape number encodes the entire nonblank segment of the tape, it follows that position i must be blank.

A Turing machine tape number. The co

where gn_2 is the Gö

Example 13.7.1

The Turing machine

The configuration n of the successor of 1 and one, respectively,

State	Tape Number
$q_0 B I B$	0
$\vdash B q_1 I I B$	1
$\vdash B I q_1 I B$	1
$\vdash B I I q_1 B$	1
$\vdash B I q_2 I I B$	2
$\vdash B q_2 I I I B$	2
$\vdash q_2 B I I I B$	2

A transition of a must move the tape Gödel numbering en are identical.

A function tr_M is a computation means the machine config number of the config configuration of M is

of Turing configuration. The variable numerically encodes more than one symbol.

of the tape ink symbol. denote the

ing, the tape head can

Encoding
The string
 i_1, \dots, i_n .

$2^7 = 3150$,
right of the

of any tape
 $ec(i, z) = 0$
er z . On the
right of the
ink segment

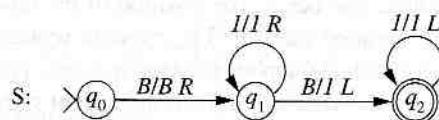
A Turing machine configuration is defined by the state number, tape head position, and tape number. The configuration number incorporates these values into the single number

$$gn_2(\text{state number}, \text{tape head position}, \text{tape number}),$$

where gn_2 is the Gödel numbering function that encodes ordered triples.

Example 13.7.1

The Turing machine S computes the successor function.



The configuration numbers are given for each configuration produced by the computation of the successor of 1. Recall that the tape symbols B and I are assigned the numbers zero and one, respectively.

	State	Position	Tape Number	Configuration Number
	$q_0 B I I B$	0	$2^1 3^2 5^2 = 450$	$gn_2(0, 0, 450)$
\vdash	$B q_1 I I B$	1	$2^1 3^2 5^2 = 450$	$gn_2(1, 1, 450)$
\vdash	$B I q_1 I B$	1	$2^1 3^2 5^2 = 450$	$gn_2(1, 2, 450)$
\vdash	$B I I q_1 B$	1	$2^1 3^2 5^2 7^1 = 3150$	$gn_2(1, 3, 3150)$
\vdash	$B I q_2 I I B$	2	$2^1 3^2 5^2 7^2 11^1 = 242550$	$gn_2(2, 2, 242550)$
\vdash	$B q_2 I I I B$	2	$2^1 3^2 5^2 7^2 11^1 = 242550$	$gn_2(2, 1, 242550)$
	$q_2 B I I I B$	2	$2^1 3^2 5^2 7^2 11^1 = 242550$	$gn_2(2, 0, 242550)$

A transition of a standard Turing machine need not alter the tape or the state, but it must move the tape head. The change in the tape head position and the uniqueness of the Gödel numbering ensure that no two consecutive configuration numbers of a computation are identical.

A function tr_M is constructed to trace the computations of a Turing machine M. Tracing a computation means generating the sequence of configuration numbers that correspond to the machine configurations produced by the computation. The value of $tr_M(x, i)$ is the number of the configuration after i transitions when M is run with input x . Since the initial configuration of M is $q_0 B \bar{x} B$,

$$tr_M(x, 0) = gn_2(0, 0, 2^1 \cdot \prod_{i=1}^{x+1} pn(i)^2).$$

The value of $tr_M(x, y + 1)$ is obtained by manipulating the configuration number $tr_M(x, y)$ to construct the encoding of the subsequent machine configuration.

The state and symbol in the position scanned by the tape head determine the transition to be applied by the machine M. The primitive recursive functions

$$\begin{aligned} cs(z) &= dec(0, z) \\ ctp(z) &= dec(1, z) \\ cts(z) &= dec(ctp(z), dec(2, z)) \end{aligned}$$

return the state number, tape head position, and the number of the symbol scanned by the tape head from a configuration number z . The position of the tape head is obtained by a direct decoding of the configuration number. The numeric representation of the scanned symbol is encoded as the $ctp(z)$ th element of the tape number. The c 's in cs , ctp , and cts stand for the components of the current configuration: current state, current tape position, and current tape symbol.

A transition specifies the alterations to the machine configuration and, hence, the configuration number. A transition of M is written

$$\delta(q_i, b) = [q_j, c, d],$$

where $q_i, q_j \in Q$; $b, c \in \Gamma$; and $d \in \{R, L\}$. Functions are defined to simulate the effects of a transition of M. We begin by listing the transitions of M:

$$\begin{aligned} \delta(q_{i_0}, b_0) &= [q_{j_0}, c_0, d_0] \\ \delta(q_{i_1}, b_1) &= [q_{j_1}, c_1, d_1] \\ &\vdots \\ \delta(q_{i_m}, b_m) &= [q_{j_m}, c_m, d_m]. \end{aligned}$$

The determinism of the machine ensures that the arguments of the transitions are distinct.

The "new state" function

$$ns(z) = \begin{cases} j_0 & \text{if } cs(z) = i_0 \text{ and } cts(z) = n(b_0) \\ j_1 & \text{if } cs(z) = i_1 \text{ and } cts(z) = n(b_1) \\ \vdots & \vdots \\ j_m & \text{if } cs(z) = i_m \text{ and } cts(z) = n(b_m) \\ cs(z) & \text{otherwise} \end{cases}$$

returns the number of the state entered by a transition from a configuration with configuration number z . The conditions on the right indicate the appropriate transition. Letting $n(b)$ denote the number of the tape symbol b , the first condition can be interpreted, "If the number of the current state is i_0 (state q_{i_0}) and the current tape symbol is b_0 (number $n(b_0)$), then the new state number has number j_0 (state q_{j_0}). This is a direct translation of the initial transition into the numeric representation. Each transition of M defines one condition in ns .

The final condition is a transition that matches a set of exhaustive primitive recursive, be defined in a comp

A function that position as specified tions as L (left) or R number and a move use the notation

The new tape position

$$ntp(z) =$$

The addition of $n(d_i)$ the transition moves left.

We have almost Given a machine con head position of the n

A transition repl In our functional app number z by the fun sented numerically by $p_n(ctp(z))$ in the cur $p_n(ctp(z))^{cts(z)+1}$, enter the transition, pos recursive function

$$ntn(z)$$

makes the desired su symbol at position ct $p_n(ctp(z))^{nts(z)+1}$, er

The trace function the effects of a transiti

$tr_M(x, y)$

transition

defined by the
tained by a
e scanned
 p , and cts
e position,

hence, the

e effects of

re distinct.

with config-
ion. Letting
eted, "If the
nber $n(b_0)$,
of the initial
dition in ns .

The final condition indicates that the new state is the same as the current state if there is no transition that matches the state and input symbol, that is, if M halts. The conditions define a set of exhaustive and mutually exclusive primitive recursive predicates. Thus, $ns(z)$ is primitive recursive. A function nts that computes the number of the new tape symbol can be defined in a completely analogous manner.

A function that computes the new tape head position alters the number of the current position as specified by the direction in the transition. The transitions designate the directions as L (left) or R (right). A movement to the left subtracts one from the current position number and a movement to the right adds one. To numerically represent the direction we use the notation

$$n(d) = \begin{cases} 0 & \text{if } d = L \\ 2 & \text{if } d = R. \end{cases}$$

The new tape position is computed by

$$ntp(z) = \begin{cases} ctp(z) + n(d_0) - 1 & \text{if } cs(z) = i_0 \text{ and } cts(z) = n(b_0) \\ ctp(z) + n(d_1) - 1 & \text{if } cs(z) = i_1 \text{ and } cts(z) = n(b_1) \\ \vdots & \vdots \\ ctp(z) + n(d_m) - 1 & \text{if } cs(z) = i_m \text{ and } cts(z) = n(b_m) \\ ctp(z) & \text{otherwise.} \end{cases}$$

The addition of $n(d_i) - 1$ to the current position number increments the value by one when the transition moves the tape head to the right. Similarly, one is subtracted on a move to the left.

We have almost completed the construction of the components of the trace function. Given a machine configuration, the functions ns and ntp compute the state number and tape head position of the new configuration. All that remains is to compute the new tape number.

A transition replaces the tape symbol occupying the position scanned by the tape head. In our functional approach, the location of the tape head is obtained from the configuration number z by the function ctp . The tape symbol to be written at position $ctp(z)$ is represented numerically by $nts(z)$. The new tape number is obtained by changing the power of $pn(ctp(z))$ in the current tape number. Before the transition, the decomposition of z contains $pn(ctp(z))^{nts(z)+1}$, encoding the value of the current tape symbol at position $ctp(z)$. After the transition, position $ctp(z)$ contains the symbol represented by $nts(z)$. The primitive recursive function

$$ntn(z) = quo(ctn(z), pn(ctp(z))^{nts(z)+1}) \cdot pn(ctp(z))^{nts(z)+1}$$

makes the desired substitution. The division removes the factor that encodes the current symbol at position $ctp(z)$ from the tape number $ctn(z)$. The result is then multiplied by $pn(ctp(z))^{nts(z)+1}$, encoding the new tape symbol.

The trace function tr_M is defined by primitive recursion from the functions that simulate the effects of a transition of M on the components of the configuration. As noted previously,

M is in state q_0 , the tape head is at position zero, and the tape has 1 's in positions one to $x + 1$ at the start of a computation with input x . This machine configuration is encoded in $tr_M(x, 0)$:

$$tr_M(x, 0) = gn_2(0, 0, 2^1 \cdot \prod_{i=1}^{x+1} pn(i)^2).$$

The subsequent machine configurations are obtained using the new state, new tape position, and new tape number functions with the previous configuration as input:

$$tr_M(x, y + 1) = gn_2(ns(tr_M(x, y)), ntp(tr_M(x, y)), ntn(tr_M(x, y))).$$

Since each of the functions in tr_M has been shown to be primitive recursive, we conclude that the tr_M is not only μ -recursive but also primitive recursive. The trace function, however, is not the culmination of our functional simulation of a Turing machine; it does not return the result of a computation but rather a sequence of configuration numbers.

The result of the computation of the Turing machine M that computes the number-theoretic function f with input x may be obtained from the function tr_M . We first note that the computation of M may never terminate; $f(x)$ may be undefined. The question of termination can be determined from the values of tr_M . If M specifies a transition for configuration $tr_M(x, i)$, then $tr_M(x, i) \neq tr_M(x, i + 1)$ since the movement of the head changes the Gödel number. On the other hand, if M halts after transition i , then $tr_M(x, i) = tr_M(x, i + 1)$ since the functions nts , ntp , and ntn return the preceding value when the configuration number represents a halting configuration. Consequently, the machine halts after the z th transition, where z is the first number that satisfies $tr_M(x, z) = tr_M(x, z + 1)$.

Since no bound can be placed on the number of transitions that occur before an arbitrary Turing machine computation terminates, unbounded minimalization is required to determine this value. The μ -recursive function

$$term(x) = \mu z[eq(tr_M(x, z), tr_M(x, z + 1))]$$

computes the number of the transition after which the computation of M with input x terminates. When a computation terminates, the halting configuration of the machine is encoded in the value $tr_M(x, term(x))$. Upon termination, the tape has the form $B\overline{f(x)}B$. The terminal tape number, ttn , is obtained from the terminal configuration number by

$$ttn(x) = dec(2, tr_M(x, term(x))).$$

The result of the computation is obtained by counting the number of 1 's on the tape or, equivalently, determining the number of primes that are raised to the power of 2 in the terminal tape number. The latter computation is performed by the bounded sum

$$sim_M(x) = \left(\sum_{i=0}^y eq(1, dec(i, ttn(x))) \right) - 1,$$

where y is the length of the tape, computed by the primitive recursive function nts from the bounded sum $tr_M(x, y)$.

Whenever f is defined, both compute the $f(x)$. A value and $sim_M(x)$ is given by the following theorem.

Theorem 13.7.1

Every Turing computable function is μ -recursive.

Theorems 13.6.4 and 13.7.1 give two approaches to computing a function.

Corollary 13.7.2

A function is Turing computable if and only if it is μ -recursive.

13.8 The Church-Turing Thesis

In its functional form, the Church-Turing thesis states that every function with Turing computable values is μ -recursive. This statement can be restated in terms of the Church-Turing thesis.

The Church-Turing Thesis Every μ -recursive function is computable by a Turing machine.

As before, no proof is given. Instead, the thesis is supported by the community of mathematicians by the accumulation of evidence supporting the thesis. The thesis is often referred to as the Church-Turing thesis, bestowing the title "most important" on the thesis. The thesis implies that any number-theoretic function that can be computed by a computer or technique can also be computed by a Turing machine. In particular, the thesis implies that any nonnumeric computation can be simulated by a Turing machine.

We begin by observing that the thesis is true for numeric computations. Consider a numeric computation that is performed by a computer, but this is only a consequence of the thesis. The input is in some encoding, such as ASCII or EBCDIC encoding scheme. The output is in some encoding, such as binary representation of a character string. The thesis implies that any computation that can be performed by a computer can be simulated by a Turing machine. This is because any computation that can be performed by a computer can be simulated by a Turing machine. The thesis implies that any computation that can be performed by a computer can be simulated by a Turing machine. This is because any computation that can be performed by a computer can be simulated by a Turing machine.

one to
ded in

osition,

include
ever,
t return

umber-
rst note
question
tion for
he head
(x, i) =
hen the
ne halts
, $z + 1$).
efore an
required

input x
achine is
 $Bf(x)B$.
er by

e tape or,
f 2 in the

where y is the length of the tape segment encoded in the terminal tape number. The bound y is computed by the primitive recursive function $gdln(ttn(x))$ (Exercise 17). One is subtracted from the bounded sum since the tape contains the unary representation of $f(x)$.

Whenever f is defined for input x , the computation of M and the simulation of M both compute the $f(x)$. If $f(x)$ is undefined, the unbounded minimalization fails to return a value and $sim_M(x)$ is undefined. The construction of sim_M completes the proof of the following theorem.

Theorem 13.7.1

Every Turing computable function is μ -recursive.

Theorems 13.6.4 and 13.7.1 establish the equivalence of the microscopic and macroscopic approaches to computation.

Corollary 13.7.2

A function is Turing computable if, and only if, it is μ -recursive.

13.8 The Church-Turing Thesis Revisited

In its functional form, the Church-Turing Thesis associates the effective computation of functions with Turing computability. Utilizing Theorem 13.7.2, the Church-Turing Thesis can be restated in terms of μ -recursive functions.

The Church-Turing Thesis (Revisited) A number-theoretic function is computable if, and only if, it is μ -recursive.

As before, no proof can be put forward for the Church-Turing Thesis. It is accepted by the community of mathematicians and computer scientists because of the accumulation of evidence supporting the claim. Accepting the Church-Turing Thesis is tantamount to bestowing the title “most general computing device” on the Turing machine. The thesis implies that any number-theoretic function that can be effectively computed by any machine or technique can also be computed by a Turing machine. This contention extends to nonnumeric computation as well.

We begin by observing that the computation of any digital computer can be interpreted as a numeric computation. Character strings are often used to communicate with the computer, but this is only a convenience to facilitate the input of the data and the interpretation of the output. The input is immediately translated to a string over $\{0, 1\}$ using either the ASCII or EBCDIC encoding schemes. After the translation, the input string can be considered the binary representation of a natural number. The computation progresses, generating another sequence of 0 's and 1 's, again a binary natural number. The output is then translated back to character data because of our inability to interpret and appreciate the output in its internal representation.

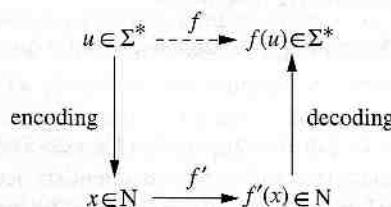
Following this example, we can design effective procedures that transform a string computation to a number-theoretic computation. The Gödel encoding can be used to translate strings to numbers. Let $\Sigma = \{a_0, a_1, \dots, a_n\}$ be an alphabet and f be a function from Σ^* to Σ^* . The generation of a Gödel number from a string begins by assigning a unique number to each element in the alphabet. For simplicity we will define the numbering of the elements of Σ by their subscripts. The encoding of a string $a_{i_0}a_{i_1}\dots a_{i_n}$ is generated by the bounded product

$$pn(0)^{i_0+1} \cdot pn(1)^{i_1+1} \cdot \dots \cdot pn(n)^{i_n+1} = \prod_{j=0}^y pn(j)^{i_j+1},$$

where y is the length of the string to be encoded.

The decoding function retrieves the exponent of each prime in the prime decomposition of the Gödel number. A string can be reconstructed using the decoding function and the numbering of the alphabet. If x is the encoding of a string $a_{i_0}a_{i_1}\dots a_{i_n}$ over Σ , then $dec(j, x) = i_j$. The original string can be obtained by concatenating the results of the decoding. Once the elements of the alphabet have been identified with natural numbers, the encoding and decoding are primitive recursive and therefore Turing computable.

The transformation of a string function f to a numeric function is obtained using character to number encoding and number to character decoding:



With the help of the Church-Turing Thesis, we will argue that a string function f is algorithmically computable if, and only if, the associated numeric function f' is Turing computable. We begin by noting that there is an effective procedure to obtain the values of f whenever f' is Turing computable. An algorithm to compute f consists of three steps:

- i) encoding the input string u to a number x ,
 - ii) computing $f'(x)$, and
 - iii) decoding $f'(x)$ to produce $f(u)$,

each of which can be performed by a Turing machine.

Now assume that there is an effective procedure to compute f . Using the reversibility of the encoding and decoding functions, we will outline an effective procedure to compute f' .

The value $f'(x)$ can be
 $f(u)$, and then transform
compute f' , the Church

The preceding argument shows the universality of Turing machines for decision problems. A standard example of a universal Turing machine is given in Example 13.8.1.

Example 13.8.1

Let Σ be the alphabet of a 's and the b 's in the input string, when combined with the symbols $\{ \}, \{ \}, \{ \}$. The elements of the alphabet, or the string $u = u_0 u_1 \dots u_n$, is

The power of $pn(i)$ in the string is a or b , re-

Let x be the encoding of the sequence encoded by y .

$$f'(x) = \prod_{i=0}^{g \ln(x)}$$

generates the encoding c of μ , where $c_i = 1$, the i th symbol in μ is 1.

contributes the factor p_n . The i th element of u is b_i , obtained from that of x by a string generates $f(u)$.

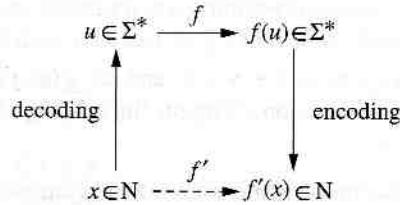
string
trans-
from
nique
of the
by the

osition
nd the
, then
of the
nbers,

using

on f is
Turing
lues of
steps:

ability of
pute f' .



The value $f'(x)$ can be generated by transforming the input x into a string u , computing $f(u)$, and then transforming $f(u)$ to obtain $f'(x)$. Since there is an effective procedure to compute f' , the Church-Turing Thesis allows us to conclude that f' is Turing computable.

The preceding argument shows that the implications of the Church-Turing Thesis and universality of Turing machine computation are not limited to numeric computation or decision problems. A string function is computable only if it can be realized by a suitably defined Turing machine combined with a Turing computable encoding and decoding. Example 13.8.1 exhibits the correspondence between string and numeric functions.

Example 13.8.1

Let Σ be the alphabet $\{a, b\}$. Consider the function $f : \Sigma^* \rightarrow \Sigma^*$ that interchanges the a 's and the b 's in the input string. A number-theoretic function f' is constructed which, when combined with the functions that encode and decode strings over Σ , computes f . The elements of the alphabet are numbered by the function n : $n(a) = 0$ and $n(b) = 1$. A string $u = u_0u_1\dots u_n$ is encoded as the number

$$pn(0)^{n(u_0)+1} \cdot pn(1)^{n(u_1)+1} \cdots \cdot pn(n)^{n(u_n)+1}.$$

The power of $pn(i)$ in the encoding is one or two depending upon whether the i th element of the string is a or b , respectively.

Let x be the encoding of a string u over Σ . Recall that $gdl(x)$ returns the length of the sequence encoded by x . The bounded product

$$f'(x) = \prod_{i=0}^{gdl(x)} (eq(dec(i, x), 0) \cdot pn(i) \cdot pn(i) + eq(dec(i, x), 1) \cdot pn(i))$$

generates the encoding of a string of the same length as the string u . When $eq(dec(i, x), 0) = 1$, the i th symbol in u is a . This is represented by $pn(i)^2$ in the encoding of u . The product

$$eq(dec(i, x), 0) \cdot pn(i) \cdot pn(i)$$

contributes the factor $pn(i)^2$ to $f'(x)$. Similarly, the power of $pn(i)$ in $f'(x)$ is one whenever the i th element of u is b . Thus f' constructs a number whose prime decomposition can be obtained from that of x by interchanging the exponents 1 and 2. The translation of $f'(x)$ to a string generates $f(u)$. \square

Exercises

1. Let $g(x) = x^2$ and $h(x, y, z) = x + y + z$, and let $f(x, y)$ be the function defined from g and h by primitive recursion. Compute the values $f(1, 0)$, $f(1, 1)$, $f(1, 2)$ and $f(5, 0)$, $f(5, 1)$, $f(5, 2)$.
2. Using only the basic functions, composition, and primitive recursion, show that the following functions are primitive recursive. When using primitive recursion, give the functions g and h .
 - a) $c_2^{(3)}$
 - b) pred
 - c) $f(x) = 2x + 2$
3. The functions below were defined by primitive recursion in Table 13.1. Explicitly, give the functions g and h that constitute the definition by primitive recursion.
 - a) sg
 - b) sub
 - c) exp
4. a) Prove that a function f defined by the composition of total functions h and g_1, \dots, g_n is total.
 b) Prove that a function f defined by primitive recursion from total functions g and h is total.
 c) Conclude that all primitive recursive functions are total.
5. Let $g = \text{id}$, $h = p_1^{(3)} + p_3^{(3)}$, and let f be defined from g and h by primitive recursion.
 - a) Compute the values $f(3, 0)$, $f(3, 1)$, and $f(3, 2)$.
 - b) Give a closed-form (nonrecursive) definition of the function f .
6. Let $g(x, y, z)$ be a primitive recursive function. Show that each of the following functions is primitive recursive.
 - a) $f(x, y) = g(x, y, x)$
 - b) $f(x, y, z, w) = g(x, y, x)$
 - c) $f(x) = g(1, 2, x)$
7. Let f be the function

$$f(x) = \begin{cases} x & \text{if } x > 2 \\ 0 & \text{otherwise.} \end{cases}$$
 - a) Give the state diagram of a Turing machine that computes f .
 - b) Show that f is primitive recursive.
8. Show that the following functions are primitive recursive. When using primitive recursion, give the functions g and h .
 - a) $\text{max}(x, y) = \dots$
 - b) $\text{min}(x, y) = \dots$
 - c) $\text{min}_3(x, y, z) = \dots$
 - d) $\text{even}(x) = \begin{cases} 1 & \text{if } x \text{ is even} \\ 0 & \text{if } x \text{ is odd} \end{cases}$
 - e) $\text{half}(x) = \dots$
 - * f) $\text{sqrt}(x) = \lfloor \sqrt{x} \rfloor$
9. Show that the following functions are primitive recursive. When using primitive recursion, give the functions g and h .
 - a) $\text{le}(x, y) = \begin{cases} 1 & \text{if } x \leq y \\ 0 & \text{if } x > y \end{cases}$
 - b) $\text{ge}(x, y) = \begin{cases} 1 & \text{if } x \geq y \\ 0 & \text{if } x < y \end{cases}$
 - c) $\text{btw}(x, y, z) = \begin{cases} 1 & \text{if } x < y < z \\ 0 & \text{otherwise} \end{cases}$
 - d) $\text{prsq}(x) = \begin{cases} 1 & \text{if } x \text{ is a perfect square} \\ 0 & \text{otherwise} \end{cases}$
10. Let t be a two-variable function. Explicitly give the following functions.
 - a) $f(x, y) = \begin{cases} 1 & \text{if } t(x) = t(y) \\ 0 & \text{otherwise} \end{cases}$
11. Let g and h be primitive recursive functions. Show that each of the following functions is primitive recursive.
 - a) $f(x, y) = \begin{cases} 1 & \text{if } g(x) = h(y) \\ 0 & \text{otherwise} \end{cases}$

8. Show that the following functions are primitive recursive. You may use the functions and predicates from Tables 13.1 and 13.2. Do not use the bounded operations.

a) $\max(x, y) = \begin{cases} x & \text{if } x \geq y \\ y & \text{otherwise} \end{cases}$

b) $\min(x, y) = \begin{cases} x & \text{if } x \leq y \\ y & \text{otherwise} \end{cases}$

c) $\min_3(x, y, z) = \begin{cases} x & \text{if } x \leq y \text{ and } x \leq z \\ y & \text{if } y \leq x \text{ and } y \leq z \\ z & \text{if } z \leq x \text{ and } z \leq y \end{cases}$

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

e) $\text{half}(x) = \text{div}(x, 2)$

*f) $\text{sqrt}(x) = \lfloor \sqrt{x} \rfloor$

9. Show that the following predicates are primitive recursive. You may use the functions and predicates from Tables 13.1 and 13.2 and Exercise 8. Do not use the bounded operators.

a) $\text{le}(x, y) = \begin{cases} 1 & \text{if } x \leq y \\ 0 & \text{otherwise} \end{cases}$

b) $\text{ge}(x, y) = \begin{cases} 1 & \text{if } x \geq y \\ 0 & \text{otherwise} \end{cases}$

c) $\text{btw}(x, y, z) = \begin{cases} 1 & \text{if } y < x < z \\ 0 & \text{otherwise} \end{cases}$

d) $\text{prsq}(x) = \begin{cases} 1 & \text{if } x \text{ is a perfect square} \\ 0 & \text{otherwise} \end{cases}$

10. Let t be a two-variable primitive recursive function and define f as follows:

$$f(x, 0) = t(x, 0)$$

$$f(x, y + 1) = f(x, y) + t(x, y + 1)$$

Explicitly give the functions g and h that define f by primitive recursion.

11. Let g and h be primitive recursive functions. Use bounded operators to show that the following functions are primitive recursive. You may use any functions and predicates that have been shown to be primitive recursive.

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

b) $f(x, y) = \begin{cases} 1 & \text{if } g(i) = x \text{ for some } 0 \leq i \leq y \\ 0 & \text{otherwise} \end{cases}$

c) $f(y) = \begin{cases} 1 & \text{if } g(i) = h(j) \text{ for some } 0 \leq i, j \leq y \\ 0 & \text{otherwise} \end{cases}$

d) $f(y) = \begin{cases} 1 & \text{if } g(i) < g(i+1) \text{ for all } 0 \leq i \leq y \\ 0 & \text{otherwise} \end{cases}$

e) $nt(x, y) = \text{the number of times } g(i) = x \text{ in the range } 0 \leq i \leq y$

f) $thrd(x, y) = \begin{cases} 0 & \text{if } g(i) \text{ does not assume the value } x \text{ at least} \\ & \text{three times in the range } 0 \leq i \leq y \\ j & \text{if } j \text{ is the third integer in the range } 0 \leq i \leq y \\ & \text{for which } g(i) = x \end{cases}$

g) $lrg(x, y) = \text{the largest value in the range } 0 \leq i \leq y \text{ for which } g(i) = x$

12. Show that the following functions are primitive recursive.

a) $gcd(x, y) = \text{the greatest common divisor of } x \text{ and } y$

b) $lcm(x, y) = \text{the least common multiple of } x \text{ and } y$

c) $pw2(x) = \begin{cases} 1 & \text{if } x = 2^n \text{ for some } n \\ 0 & \text{otherwise} \end{cases}$

d) $twopr(x) = \begin{cases} 1 & \text{if } x \text{ is the product of exactly two primes} \\ 0 & \text{otherwise} \end{cases}$

* 13. Let g be a one-variable primitive recursive function. Prove that the function

$$\begin{aligned} f(x) &= \min_{i=0}^x(g(i)) \\ &= \min\{g(0), \dots, g(x)\} \end{aligned}$$

is primitive recursive.

14. Prove that the function

$$f(x_1, \dots, x_n) = \mu z^{u(x_1, \dots, x_n)}[p(x_1, \dots, x_n, z)]$$

is primitive recursive whenever p and u are primitive recursive.

15. Compute the Gödel number for the following sequence:

a) 3, 0

b) 0, 0, 1

c) 1, 0, 1, 2

d) 0, 1, 1, 2, 0

16. Determine the se

a) 18,000

b) 131,072

c) 2,286,900

d) 510,510

17. Prove that the fo

a) $gdn(x) = \begin{cases} 1 & \text{if } x \text{ is even} \\ 0 & \text{if } x \text{ is odd} \end{cases}$

b) $gdln(x) = \begin{cases} 1 & \text{if } x \text{ is odd} \\ 0 & \text{if } x \text{ is even} \end{cases}$

c) $g(x, y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{if } x \neq y \end{cases}$

18. Construct a prim whose output is t have been swapped is the encoding o

19. Let f be the func

Give the values f

* 20. Let g_1 and g_2 be c variable primitive

are said to be co values $f_1(x, y + both of the functio$

16. Determine the sequences encoded by the following Gödel numbers:
- 18,000
 - 131,072
 - 2,286,900
 - 510,510
17. Prove that the following functions are primitive recursive:
- $gdn(x) = \begin{cases} 1 & \text{if } x \text{ is the Gödel number of some sequence} \\ 0 & \text{otherwise} \end{cases}$
 - $gdln(x) = \begin{cases} n & \text{if } x \text{ is the Gödel number of a sequence of length } n \\ 0 & \text{otherwise} \end{cases}$
 - $g(x, y) = \begin{cases} 1 & \text{if } x \text{ is a Gödel number and } y \text{ occurs in the sequence encoded in } x \\ 0 & \text{otherwise} \end{cases}$
18. Construct a primitive recursive function whose input is an encoded ordered pair and whose output is the encoding of an ordered pair in which the positions of the elements have been swapped. For example, if the input is the encoding of $[x, y]$, then the output is the encoding of $[y, x]$.
19. Let f be the function defined by

$$f(x) = \begin{cases} 1 & \text{if } x = 0 \\ 2 & \text{if } x = 1 \\ 3 & \text{if } x = 2 \\ f(x - 3) + f(x - 1) & \text{otherwise.} \end{cases}$$

Give the values $f(4)$, $f(5)$, and $f(6)$. Prove that f is primitive recursive.

- *20. Let g_1 and g_2 be one-variable primitive recursive functions. Also let h_1 and h_2 be four-variable primitive recursive functions. The two functions f_1 and f_2 defined by

$$\begin{aligned} f_1(x, 0) &= g_1(x) \\ f_2(x, 0) &= g_2(x) \\ f_1(x, y + 1) &= h_1(x, y, f_1(x, y), f_2(x, y)) \\ f_2(x, y + 1) &= h_2(x, y, f_1(x, y), f_2(x, y)) \end{aligned}$$

are said to be constructed by *simultaneous recursion* from g_1 , g_2 , h_1 , and h_2 . The values $f_1(x, y + 1)$ and $f_2(x, y + 1)$ are defined in terms of the previous values of both of the functions. Prove that f_1 and f_2 are primitive recursive.

21. Let f be the function defined by

$$f(0) = 1$$

$$f(y+1) = \sum_{i=0}^y f(i)^y.$$

- a) Compute $f(1)$, $f(2)$, and $f(3)$.
 b) Use course-of-values recursion to show that f is primitive recursive.

22. Let A be Ackermann's function (see Section 13.6).

- a) Compute $A(2, 2)$.
 b) Prove that $A(x, y)$ has a unique value for every $x, y \in \mathbb{N}$.
 c) Prove that $A(1, y) = y + 2$.
 d) Prove that $A(2, y) = 2y + 3$.

23. Prove that the following functions are μ -recursive. The functions g and h are assumed to be primitive recursive.

- a) $\text{cube}(x) = \begin{cases} 1 & \text{if } x \text{ is a perfect cube} \\ \uparrow & \text{otherwise} \end{cases}$
 b) $\text{root}(c_0, c_1, c_2) =$ the smallest natural number root of the quadratic polynomial $c_2 \cdot x^2 + c_1 \cdot x + c_0$
 c) $r(x) = \begin{cases} 1 & \text{if } g(i) = g(i+x) \text{ for some } i \geq 0 \\ \uparrow & \text{otherwise} \end{cases}$
 d) $l(x) = \begin{cases} \uparrow & \text{if } g(i) - h(i) < x \text{ for all } i \geq 0 \\ 0 & \text{otherwise} \end{cases}$
 e) $f(x) = \begin{cases} 1 & \text{if } g(i) + h(j) = x \text{ for some } i, j \in \mathbb{N} \\ \uparrow & \text{otherwise} \end{cases}$
 f) $f(x) = \begin{cases} 1 & \text{if } g(y) = h(z) \text{ for some } y > x, z > x \\ \uparrow & \text{otherwise.} \end{cases}$

- *24. The unbounded μ -operator can be defined for partial predicates as follows:

$$\mu z[p(x_1, \dots, x_n, z)] = \begin{cases} j & \text{if } p(x_1, \dots, x_n, i) = 0 \text{ for } 0 \leq i < j \\ & \quad \text{and } p(x_1, \dots, x_n, j) = 1 \\ \uparrow & \text{otherwise.} \end{cases}$$

That is, the value is undefined if $p(x_1, \dots, x_n, i) \uparrow$ for some i occurring before the first value j for which $p(x_1, \dots, x_n, j) = 1$. Prove that the family of functions obtained by replacing the unbounded minimalization operator in Definition 13.6.3 with the preceding μ -operator is the family of Turing computable functions.

25. Construct the function $\text{prt}(x)$ from Example 13.7.1.

26. Let M be the machine

M:

- a) What unary machine does M compute?
 b) Give the tape configuration of M with input $\overline{0}$.
 c) Give the tape configuration of M with input $\overline{2}$.

27. Let f be the function

- a) Give the state transition diagram of f .
 b) Trace the computation of f for each configuration in the computation.
 c) Show that f has been shown to be total.

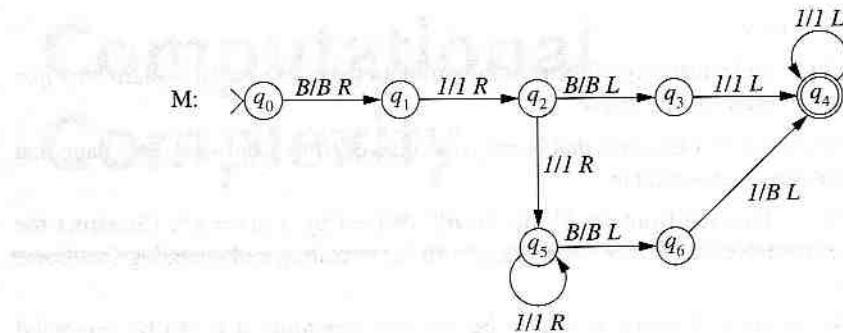
- *28. Let M be a Turing machine

- a) Show that the

$\text{prt}(x)$

is primitive recursive.

25. Construct the functions ns , ntp , and nts for the Turing machine S given in Example 13.7.1.
26. Let M be the machine



- a) What unary number-theoretic function does M compute?
- b) Give the tape numbers for each configuration that occurs in the computation of M with input $\overline{0}$.
- c) Give the tape numbers for each configuration that occurs in the computation of M with input $\overline{2}$.
27. Let f be the function defined by

$$f(x) = \begin{cases} x + 1 & \text{if } x \text{ even} \\ x - 1 & \text{otherwise.} \end{cases}$$

- a) Give the state diagram of a Turing machine M that computes f .
- b) Trace the computation of your machine for input 1 ($B11B$). Give the tape number for each configuration in the computation. Give the value of $tr_M(1, i)$ for each step in the computation.
- c) Show that f is primitive recursive. You may use the functions from the text that have been shown to be primitive recursive in Sections 13.1, 13.2, and 13.4.
- * 28. Let M be a Turing machine and tr_M the trace function of M .

- a) Show that the function

$$prt(x, y) = \begin{cases} 1 & \text{if the } y\text{th transition of } M \text{ with input } x \text{ prints} \\ & \text{a blank} \\ 0 & \text{otherwise} \end{cases}$$

is primitive recursive.

- b) Show that the function

$$lppt(x) = \begin{cases} 1 & \text{if the final transition of } M \\ & \text{with input } x \text{ that prints a } 1 \\ \uparrow & \text{otherwise} \end{cases}$$

is μ -recursive.

- c) In light of undecidability of the printing problem (Exercise 12.7), explain why $lppt$ cannot be primitive recursive.
29. Give an example of a function that is not μ -recursive. *Hint:* Consider a language that is not recursively enumerable.
30. Let f be the function from $\{a, b\}^*$ to $\{a, b\}^*$ defined by $f(u) = u^R$. Construct the primitive recursive function f' that, along with the encoding and decoding functions, computes f .
31. A number-theoretic function is said to be *macro-computable* if it can be computed by a Turing machine defined using only the machines S and D that compute the successor and predecessor functions and the macros from Section 9.3. Prove that every μ -recursive function is macro-computable. To do this you must show that
- i) The successor, zero, and projection functions are macro-computable.
 - ii) The macro-computable functions are closed under composition, primitive recursion, and unbounded minimization.
32. Prove that the programming language TM defined in Section 9.6 computes the entire set of μ -recursive functions.

Bibliographic Notes

The functional and mechanical development of computability flourished in the 1930s. Gödel [1931] defined a method of computation now referred to as Herbrand-Gödel computability. The properties of Herbrand-Gödel computability and μ -recursive functions were developed extensively by Kleene. The equivalence of μ -recursive functions and Turing computability was established in Kleene [1936]. Post machines [Post, 1936] provide an alternative mechanical approach to numeric computation. The classic book by Kleene [1952] presents computability, the Church-Turing Thesis, and recursive functions. A further examination of recursive function theory can be found in Hermes [1965], Péter [1967], and Rogers [1967]. Hennie [1977] develops computability from the notion of an abstract family of algorithms.

Ackermann's function was introduced in Ackermann [1928]. An excellent exposition of the features of Ackermann's function can be found in Hennie [1977].

PART

Com
Com

The objective of algorithmic computation is to find an algorithmic solution measured by the complexity of analysis of the quantity.

Complexity theory studies those that are solvable but not have a practical algorithm requiring an extra efficient algorithm.

Since it is the time that should be independent from those of the computational complexity as limiting the time are properties of the machine, which framework for the assures us that any.

The time and space and the amount of solvable in polynomial contain all efficient decision problems in time. Clearly, \mathcal{P} is identical.

PART IV

Computational Complexity

The objective of the preceding chapters was to characterize the set of solvable problems and computable functions. We now turn our attention from exhibiting the existence of algorithmic solutions of problems to analyzing their complexity, where the complexity is measured by the resources required in determining the solution. Thus we begin a formal analysis of the question *how much* first posed in the Introduction.

Complexity theory attempts to distinguish problems that are solvable in practice from those that are solvable in principle only. A problem that is theoretically solvable may not have a practical solution; there may be no algorithm that solves the problem without requiring an extraordinary amount of time or memory. Problems for which there are no efficient algorithms are said to be *intractable*.

Since it is the inherent complexity of a problem that is of interest to us, the analysis should be independent of any particular implementation. To isolate the features of a problem from those of the implementation, a single algorithmic system must be chosen for analyzing computational complexity. The choice should not place any unnecessary restrictions, such as limiting the time or memory available, upon the computation since these limitations are properties of the implementation and not of the algorithm itself. The standard Turing machine, which fulfills all of these requirements, provides the underlying computational framework for the analysis of problem complexity. Moreover, the Church-Turing Thesis assures us that any effective procedure can be implemented on such a machine.

The time and space complexities of a Turing machine measure the number of transitions and the amount of tape required in a computation, respectively. The class \mathcal{P} of problems solvable in polynomial time by a deterministic Turing machine is generally considered to contain all efficiently solvable problems. Another class of problems, \mathcal{NP} , consists of all decision problems that can be solved by a nondeterministic Turing machine in polynomial time. Clearly, \mathcal{P} is a subset of \mathcal{NP} . It is currently unknown if these two classes of problems are identical.

a why *lppt*

guage that

nstruct the
functions,

computed
mpute the
e that every

itive recur-
s the entire

930s. Gödel
imputability.
re developed
computabil-
n alternative
[52] presents
amination of
gers [1967].
f algorithms.
nt exposition

Using the guess-and-check strategy of nondeterministic solutions, the class NP consists of all problems for which solutions can be verified in polynomial time. Answering the $\mathcal{P} = \text{NP}$ question is equivalent to deciding whether constructing a solution to a problem is inherently more difficult than checking whether a single possibility is a solution. While it seems that this should be the case, as of yet it has not been formally proved.

A problem is NP-complete if every problem in the class NP can be reduced to it in polynomial time. Finding a polynomial time solution to one NP-complete problem is sufficient to establish that $\mathcal{P} = \text{NP}$, but no such algorithm has been discovered at this time. Moreover, the majority of computer scientists and mathematicians do not believe that such an algorithm exists. The examination of NP-completeness begins with showing that the Satisfiability Problem is NP-complete by explicitly constructing a reduction of any problem in NP to it. Polynomial-time reductions are then used to show that a number of additional problems are NP-complete.

Problems from many disciplines including pattern recognition, scheduling, decision analysis, combinatorics, network design, and graph theory have been shown to be NP-complete. Determining that a problem is NP-complete does not mean that solutions are no longer needed, only that it is quite unlikely that there is a polynomial-time algorithm that produces them. For NP-complete optimization problems, approximation algorithms are frequently used to produce near optimal solutions efficiently. To demonstrate the strategies employed in obtaining approximate solutions, we will examine algorithms that produce approximations within a predetermined accuracy bound for several well-known NP-complete problems.

We begin the study of a deterministic problem in a computational model initiated with a fixed time complexity. We then consider deterministic and nondeterministic implementations of their implementations.

The time complexity of the language is determined by the machine. First, we will consider a machine that accepts a language and produces a function that maps strings to values. This original machine has a fixed time complexity and is called a minimal asymptotic machine. It is able to compute any function that can be computed by any computational model. Finally, we will consider a machine that computes any computable function. The function is called a function of the function.

The Church-Turing thesis states that any function that can be computed by a computer is solvable by a Turing machine. This means that between computation and computation, a computer can be simulated by a Turing machine.

s NP consists
nswering the
to a problem
lution. While
l.

reduced to it
te problem is
d at this time.
ieve that such
wing that the
f any problem
of additional

ling, decision
vn to be NP-
lutions are no
lgorithm that
rithms are fre-
the strategies
t produce ap-
NP-complete

CHAPTER 14

Time Complexity

We begin the study of computational complexity with the analysis of the time complexity of a deterministic Turing machine, where time is measured by the number of transitions in a computation. Because of the variation in the number of transitions in computations initiated with strings of the same length, rates of growth are frequently used to describe time complexity. We will show that the time complexity of algorithms implemented on deterministic multitrack and multitape Turing machines differs only polynomially from their implementation on a standard Turing machine.

The time complexity of a language is determined by those of the machines that accept the language. Several important properties of the complexities of languages are established. First, we will see that there is no best Turing machine, in terms of time complexity, that accepts a language. A machine that accepts a language can be “sped up” to produce another machine whose complexity is reduced by any desired linear factor. The speedup theorem produces a faster machine but one whose time complexity has the same rate of growth as the original machine. We will also show that there is a language for which no Turing machine has minimal asymptotic time complexity. From any machine that accepts this language, we will be able to construct another that has a time complexity with a strictly smaller rate of growth. Finally, we will see that there is no upper bound on the time complexity of languages; for any computable function, there is a language whose complexity is not bounded by the values of the function.

The Church-Turing Thesis assures us that any problem solvable using a modern computer is solvable with a Turing machine, but this statement does not relate the complexity between computations in the two systems. We will show that the computation of a computer can be simulated by a Turing machine in which the number of transitions of the Turing

machine grows only polynomially with the number of instructions executed by the computer. Consequently, the resource bounds established for Turing machines provide practical information about the complexity of algorithms and computer programs.

14.1 Measurement of Complexity

Two main topics in the study of computational complexity are the assessment of algorithms that solve a particular problem and the comparison of the inherent difficulty of different problems. The focus of this presentation is the latter, but the comparison of problem complexity requires the ability to analyze the algorithms that solve each of the problems. To appreciate the issues involved in the analysis of algorithms, we will consider the measurement of the time complexity of the following four familiar problems:

Sort an Array of Integers

Input: Array $A[1..n]$

Output: Array $A'[1..n]$ with elements in sorted order

Square a Matrix

Input: An $n \times n$ matrix B with integral entries

Output: Matrix $C = B^2$

Path Problem for Directed Graphs

Input: Graph $G = (N, A)$, nodes $v_i, v_j \in N$

Output: yes; if there is a path from v_i to v_j in G

no; otherwise.

Acceptance by Turing Machine M (that halts for all inputs)

Input: string w

Output: yes; if M accepts w

no; otherwise.

Algorithms that perform the computations described in the first two problems compute functions. The sorting problem maps arrays to arrays and the squaring problem maps matrices to matrices. The path and acceptance problems are decision problems, with the result being either a yes or no response.

A complexity function describes the resources required or the number of steps involved in the solution of the problem. The items measured may vary based on the problem: number of data movements, number of arithmetic operations performed, number of instructions executed, the amount of space used, and so forth. The goal is not to calculate the exact resource requirement for every possible input but rather provide information that can be used to assure sufficient resources are available for each input.

TABLE 14.1 Complexity Functions

Problem	Input
Sort	Size n
Square	Dimension n
Path	Number of nodes n
Acceptance	Length of string n

The analysis of the complexity of an algorithm requires the resources to be considered and the construction of the algorithm.

After identifying the problem, the next step is to identify input instances. Each problem has an associated natural number input set. Table 14.1 gives some examples. For example, the input set for the path problem is the set of directed graphs. The resulting partition of the input set is $\{G_i\}_{i=1}^{\infty}$. The number i is the index of the graph G_i .

Let P be a problem with input sets I_1, I_2, \dots , where the i th input set is I_i . The complexity function c_P is a function that maps any problem instance i to a natural number (the cost of computation) that provides a measure of the utilization of the class I_n .

When comparing the complexity of different algorithms, the sources examined are the number of comparisons in the case of sorting algorithms and the number of insertions in the case of insertion sort. All other factors being equal, consequently, it is reasonable to assume that the complexity of the algorithms is proportional to the size of the input.

Problem-specific complexity functions are used to compare different problems. We have already seen that the complexity of the bubble sort algorithm and the complexity of the insertion sort algorithm are proportional to the size of the input. Consequently, it is reasonable to assume that the complexity of the bubble sort algorithm is even more problematic than the complexity of the insertion sort algorithm. If an array of size n should be sorted, the complexity of the bubble sort algorithm is $O(n^2)$ and the complexity of the insertion sort algorithm is $O(n^2)$.

l by the com-
vide practical

of algorithms
ty of different
problem com-
problems. To
r the measure-

blems compute
; problem maps
blems, with the

of steps involved
problem: number
r of instructions
lculate the exact
ation that can be

TABLE 14.1 Components of Complexity Functions

Problem	Input Complexity	Resource Usage Measured
Sort	Size of array	Number of data movements
Square	Dimension of matrix	Number of scalar multiplications
Path	Number of nodes in the graph	Number of nodes visited in search
Acceptance	Length of input string	Number of transitions in a computation

The analysis of the complexity of an algorithm requires three items: the identification of the resources to be considered, a partition of the input instances based upon their complexity, and the construction of a function that relates input complexity to the resource utilization.

After identifying the resources to be measured, the next step is to partition the set of input instances. Each set in the partition contains instances with similar characteristics and has an associated natural number that characterizes the complexity of the instances in the set. Table 14.1 gives standard partitions of the input domains of our four sample problems. For example, the input instances of the sorting problem are grouped by the size of the array. The resulting partition consists of sets A_0, A_1, A_2, \dots , where A_i contains all arrays of size i . The number i is the input complexity associated with the instances in the set A_i .

Let P be a problem whose input instances are partitioned into complexity classes I_0, I_1, I_2, \dots , where the subscript represents the numeric complexity assigned to each class. The complexity function for a solution to P specifies the maximum resource usage for any problem instance in a class I_i . That is, a complexity function is a mapping from the natural numbers (the complexity measure of the input) to the natural numbers (the resource utilization) that provides an upper bound on resource usage for each problem instance in the class I_n .

When comparing algorithms that solve the same problem, the input complexity and resources examined are frequently given in problem-specific terms. For example, the analysis of sorting algorithms uses measures similar to those in Table 14.1. Bubble sort, merge sort, and insertion sort all take an array as input and, in one manner or another, move data. Consequently, it is reasonable to use the number of data movements to compare the efficiency of the algorithms.

Problem-specific measures do not make sense when comparing algorithms that solve different problems. What is the relationship between the number of data movements of a sort algorithm and the number of nodes visited in a graph traversal? Even if we know the complexity functions of each algorithm, we are in no position to compare the efficiency of the algorithms or the relative difficulty of the problems. The assessment of input complexity is even more problematic; there is no reason to believe that the resources required for sorting an array of size n should in any way be related to those required for searching a graph with

n nodes. However, this is the information given by complexity functions defined in terms of the high-level components in these problems.

To be able to compare problems, the solutions must be implemented in a common algorithmic system so that the complexity can be analyzed in terms of the same input measure and resource utilization. The Turing machine provides the ideal algorithmic system for the study of problem complexity. A Turing machine has no artificial limitations on the memory or time available for a computation. Moreover, the Church-Turing Thesis assures us that any effective procedure can be implemented on a Turing machine. The common input measure for all problems is the length of the input string. The time and space complexity of the Turing machine describe the number of transitions and tape squares needed by a computation.

14.2 Rates of Growth

Obtaining the exact relationship between input complexity and resource utilization is sometimes quite difficult and almost always provides more information than we require. For this reason, time complexity is often represented by the rate of growth of the complexity function rather than by the function itself. Before continuing with our evaluation of the complexity of algorithms, we detour for a brief review of the mathematical analysis of the rate of growth of functions.

The rate of growth of a function measures the asymptotic performance of the function as the input gets arbitrarily large. Intuitively, the rate of growth is determined by the most significant contributor to the growth of the function. The contribution of the individual terms to the values of a function can be seen by examining the growth of the functions n^2 and $n^2 + 2n + 5$ in Table 14.2. The contribution of n^2 to $n^2 + 2n + 5$ is measured by the ratio of the function values in the bottom row. The linear and constant terms of the function $n^2 + 2n + 5$ are called the *lower-order terms*. Lower-order terms may exert undue influence on the initial values of the functions. As n gets large, it is clear that the lower-order terms do not significantly contribute to the growth of the function values. The order of a function and the “big oh” notation are introduced to describe the asymptotic growth of the values of a function.

Definition 14.2.1

Let $f : \mathbb{N} \rightarrow \mathbb{N}$ and $g : \mathbb{N} \rightarrow \mathbb{N}$ be one-variable number-theoretic functions.

- i) The function f is said to be of **order** g if there is a positive constant c and a natural number n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.
 - ii) The set of all functions of order g is denoted $O(g) = \{f \mid f \text{ is of order } g\}$ and called “big oh of g .”

A function f is of order g if the values of f are bounded by a constant multiple of the values of g . Because of the influence of the lower-order terms, the inequality $f(n) \leq c \cdot g(n)$

TABLE 14.2 Gravitational Potential Energy

$$\begin{array}{r} n \\ n^2 \\ \hline n^2 + 2n + 5 \\ n^2 / (n^2 + 2n + 5) \end{array}$$

is required to hold
order g , we say that

Traditionally, $O(g)$ is a set, it is unconventional use of arbitrary element f to indicate that f consists of n , without specifying f indicates that f is one of

Example 14.2.1

Let $f(n) = n^2$ and
 $n^2 \leq n^3$ for all natural numbers n .

Let us suppose

Choose n_1 to be the
contradicting the in-

Two functions
 $g \in O(f)$. When j
two inequalities

where c_1 and c_2 are these functions is by

These relationships bounds, it is clear t

in terms

common algorithmic measure for the memory required that any measure of time Turing computation.

is sometimes used. For this reason, the complexity of growth

of a function is the most individual distinction n^2 made by the function. The influence of lower-order terms on a function's values of

and a natural number called

multiple of the form $c \cdot g(n)$

TABLE 14.2 Growth of Functions

n	0	5	10	25	50	100	1,000
n^2	0	25	100	625	2,500	10,000	1,000,000
$n^2 + 2n + 5$	5	40	125	680	2,605	10,205	1,002,005
$n^2/(n^2 + 2n + 5)$	0	0.625	0.800	0.919	0.960	0.980	0.998

is required to hold only for input values greater than some specified number. When f is of order g , we say that g provides an *asymptotic upper bound* on f .

Traditionally, the notation $f = O(g)$ is used to indicate that f is of order g . Since $O(g)$ is a set, it is more mathematically precise to write $f \in O(g)$. The rationale for the unconventional use of “=” is that $O(g)$ is frequently used in an expression to denote an arbitrary element from the set. For example, a function may be written $f(n) = n^2 + O(n)$ to indicate that f consists of n^2 plus some lower-order terms that are asymptotically bounded by n , without specifically indicating the lower-order terms. We will write $f \in O(g)$ to indicate that f is of order g . This is frequently read “ f is big oh of g .”

Example 14.2.1

Let $f(n) = n^2$ and $g(n) = n^3$. Then $f \in O(g)$ and $g \notin O(f)$. Clearly, $n^2 \in O(n^3)$ since $n^2 \leq n^3$ for all natural numbers.

Let us suppose that $n^3 \in O(n^2)$. Then there are constants c and n_0 such that

$$n^3 \leq c \cdot n^2 \quad \text{for all } n \geq n_0.$$

Choose n_1 to be the maximum of $n_0 + 1$ and $c + 1$. Then $n_1^3 = n_1 \cdot n_1^2 > c \cdot n_1^2$ and $n_1 > n_0$, contradicting the inequality. Thus our assumption is false and $n^3 \notin O(n^2)$. \square

Two functions f and g are said to have the same rate of growth if $f \in O(g)$ and $g \in O(f)$. When f and g have the same rate of growth, Definition 14.2.1 provides the two inequalities

$$f(n) \leq c_1 \cdot g(n) \quad \text{for } n \geq n_1$$

$$g(n) \leq c_2 \cdot f(n) \quad \text{for } n \geq n_2,$$

where c_1 and c_2 are positive constants. Combining these inequalities, we see that each of these functions is bounded above and below by constant multiples of the other:

$$f(n)/c_1 \leq g(n) \leq c_2 \cdot f(n)$$

$$g(n)/c_2 \leq f(n) \leq c_1 \cdot g(n).$$

These relationships hold for all n greater than the maximum of n_1 and n_2 . Because of these bounds, it is clear that neither f nor g can grow faster than the other.

Example 14.2.2

Let $f(n) = n^2 + 2n + 5$ and $g(n) = n^2$. Then $f \in O(g)$ and $g \in O(f)$. Since

$$n^2 \leq n^2 + 2n + 5$$

for all natural numbers, setting c to 1 and n_0 to 0 satisfies the conditions of Definition 14.2.1. Consequently, $g \in O(f)$.

To establish the opposite relationship, we begin by noting that $2n \leq 2n^2$ and $5 \leq 5n^2$ for all $n \geq 1$. Then

$$\begin{aligned} f(n) &= n^2 + 2n + 5 \\ &\leq n^2 + 2n^2 + 5n^2 \\ &= 8n^2 \\ &= 8 \cdot g(n) \end{aligned}$$

whenever $n \geq 1$. In the big oh terminology, the preceding inequality shows that $n^2 + 2n + 5 \in O(n^2)$. \square

If f has the same rate of growth as g , g is said to be an *asymptotically tight bound* on f . The set

$$\Theta(g) = \{f \mid f \in O(g) \text{ and } g \in O(f)\}$$

consists of all functions for which g provides an asymptotically tight bound. Employing the same notation as used for the big oh, we write $f \in \Theta(g)$ to indicate that g is an asymptotically tight bound for f .

A *polynomial with integral coefficients* is a function of the form

$$f(n) = c_r \cdot n^r + c_{r-1} \cdot n^{r-1} + \cdots + c_1 \cdot n + c_0,$$

where the c_0, c_1, \dots, c_{r-1} are arbitrary integers, c_r is a nonzero integer, and r is a positive integer. The constants c_i are the coefficients of f , and r is the degree of the polynomial. A polynomial with integral coefficients defines a function from the natural numbers into the integers. The presence of negative coefficients may produce negative values. For example, if $f(n) = n^2 - 3n - 4$, then $f(0) = -4$, $f(1) = -6$, $f(2) = -6$, and $f(3) = -4$. The values of the polynomial $g(n) = -n^2 - 1$ are negative for all natural numbers n .

The rate of growth has been defined only for number-theoretic functions. The absolute value function can be used to transform an arbitrary polynomial into a number-theoretic function. The absolute value of an integer i is the nonnegative integer defined by

$$|i| = \begin{cases} i & \text{if } i \geq 0 \\ -i & \text{otherwise.} \end{cases}$$

Composing a polynomial f with the absolute value function produces a function $|f|$. The rate of growth of $|f|$ is the same as that of f .

The techniques presented here provide a general relationship between the rates of growth of two functions.

Theorem 14.2.2

Let f be a polynomial of degree r .

- i) $f \in \Theta(n^r)$

- ii) $f \in O(n^k)$ for all $k < r$

- iii) $f \notin O(n^k)$ for all $k \geq r$

One of the consequences of Theorem 14.2.2 is that a polynomial function of degree r has the same rate of growth as its absolute value function. This is not the same as that of the original function.

Other important functions such as logarithmic, exponential, and power functions with base a are defined by

Changing the base of a logarithm is done precisely,

This identity indicates that the logarithm of a product is the sum of the logarithms of the base.

Examples 14.2.1 and 14.2.2 illustrate the use of polynomial functions. We now turn to the use of limits to determine the rates of growth of number-theoretic functions.

1. If $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$, then $f \in O(g)$
2. If $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c$ where $c \neq 0$, then $f \in \Theta(g)$
3. If $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$, then $f \in \Omega(g)$

The determination of the limit of a function is the application of l'Hospital's rule.

Composing a polynomial f with the absolute value produces a number-theoretic function $|f|$. The rate of growth of a polynomial f is defined to be that of $|f|$.

The techniques presented in Examples 14.2.1 and 14.2.2 can be used to establish a general relationship between the degree of a polynomial and its rate of growth.

Theorem 14.2.2

Let f be a polynomial of degree r . Then

$$15 \leq 5n^2$$

- i) $f \in \Theta(n^r)$
- ii) $f \in O(n^k)$ for all $k > r$
- iii) $f \notin O(n^k)$ for all $k < r$.

One of the consequences of Theorem 14.2.2 is that the rate of growth of any polynomial can be characterized by a function of the form n^r . The first condition shows that a polynomial of degree r has the same rate of growth as n^r . Moreover, by conditions (ii) and (iii), its growth is not the same as that of n^k for any k other than r .

Other important functions used in measuring the performance of algorithms are the logarithmic, exponential, and factorial functions. A number-theoretic logarithmic function with base a is defined by

bound on

$$f(n) = \lfloor \log_a(n) \rfloor.$$

Changing the base of a logarithmic function alters the value by a constant multiple. More precisely,

Employing
at g is an

$$\log_a(n) = \log_a(b)\log_b(n).$$

This identity indicates that the rate of growth of the logarithmic functions is independent of the base.

Examples 14.2.1 and 14.2.2 used the definition of big oh to compare the rates of growth of polynomial functions. When the functions are more complicated, it is frequently easier to use limits to determine the asymptotic complexity of two functions. Let f and g be two number-theoretic functions, then

1. If $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$, then $f \in O(g)$ and $g \notin O(f)$.
2. If $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c$ with $0 < c < \infty$, then $f \in \Theta(g)$ and $g \in \Theta(f)$.
3. If $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$, then $f \notin O(g)$ and $g \in O(f)$.

The determination of the rate of growth of a function in this manner often requires the application of l'Hospital's Rule to obtain the limit.

The version of l'Hospital's Rule used in complexity analysis asserts that if f and g are functions from \mathbf{R}^+ to \mathbf{R}^+ that are continuous and differentiable as n approaches infinity, then

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{f'(n)}{g'(n)},$$

where f' and g' are the derivatives of f and g , respectively. Example 14.2.3 uses limits and l'Hospital's Rule to show that $n \log_a(n) \in O(n^2)$ for the logarithmic function with any base a .

Example 14.2.3

Let $f(n) = n \log_a(n)$ and $g(n) = n^2$. Two applications of l'Hospital's Rule to the ratio $f(n)/g(n)$ produce

$$\begin{aligned} \lim_{n \rightarrow \infty} \frac{n \log_a(n)}{n^2} &= \lim_{n \rightarrow \infty} \frac{\log_a(n) + n(\log_a(e)/n)}{2n} \\ &= \lim_{n \rightarrow \infty} \frac{\log_a(n)}{2n} + \lim_{n \rightarrow \infty} \frac{\log_a(e)}{2n} \\ &= \lim_{n \rightarrow \infty} \frac{\log_a(e)/n}{2} + 0 \\ &= \lim_{n \rightarrow \infty} \frac{\log_a(e)}{2n} \\ &= 0, \end{aligned}$$

where e is the base of the natural logarithm. Since the limit is 0, $f \in O(g)$. \square

Theorem 14.2.3 compares the growth of logarithmic, exponential, and factorial functions with each other and the polynomials. The proofs are left as exercises.

Theorem 14.2.3

Let r be a natural number and let a and b be real numbers greater than 1. Then

- i) $\log_a(n) \in O(n)$
- ii) $n \notin O(\log_a(n))$
- iii) $n^r \in O(b^n)$
- iv) $b^n \notin O(n^r)$
- v) $b^n \in O(n!)$
- vi) $n! \notin O(b^n)$.

A function f is said to *polynomially bounded* if $f \in O(n^r)$ for some natural number r . Although not a polynomial, it follows from Example 14.2.3 that $n \log_2(n)$ is bounded by the polynomial n^2 . The polynomially bounded functions, which include the polynomials,

TABLE 14.4 Growth of Functions

n	$\log_2(n)$
5	2
10	3
20	4
30	4
40	5
50	5
100	6
200	7

constitute an important class of efficient algorithms. Exponential functions are not polynomially bounded. Functions in increasing order outlined in Theorems 14.2.3 and 14.2.4, which $2^n \in O(f)$ as $n \rightarrow \infty$, are said to exhibit exponential growth.

The efficiency of polynomial algorithms is measured by the time complexity $O(n^r)$ for some $r \in \mathbb{N}$. This is apparent when considering Table 14.4. It illustrates that a polynomial algorithm is not polynomial.

and g are
infinity, then

es limits
with any

the ratio

orial func-

ral number
ounded by
ynomials,

TABLE 14.3 A Big Oh Hierarchy

Big Oh	Asymptotic Upper Bound
$O(1)$	Constant
$O(\log_a(n))$	Logarithmic
$O(n)$	Linear
$O(n \log_a(n))$	$n \log n$
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(n^r)$	Polynomial $r \geq 0$
$O(b^n)$	Exponential $b > 1$
$O(n!)$	Factorial

TABLE 14.4 Growth of Several Common Functions

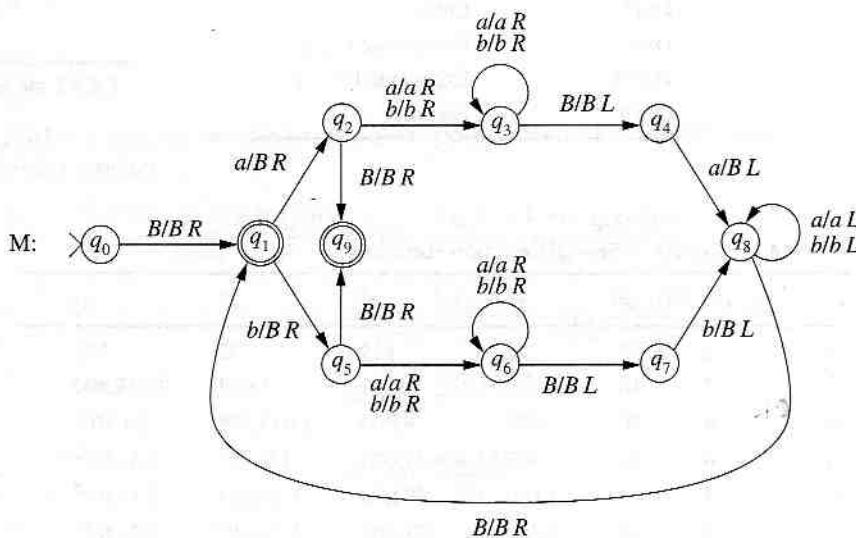
n	$\log_2(n)$	n	n^2	n^3	2^n	$n!$
5	2	5	25	125	32	120
10	3	10	100	1,000	1,024	3,628,800
20	4	20	400	8,000	1,048,576	$2.4 \cdot 10^{18}$
30	4	30	900	27,000	$1.0 \cdot 10^9$	$2.6 \cdot 10^{32}$
40	5	40	1,600	64,000	$1.1 \cdot 10^{12}$	$8.1 \cdot 10^{47}$
50	5	50	2,500	125,000	$1.1 \cdot 10^{15}$	$3.0 \cdot 10^{64}$
100	6	100	10,000	1,000,000	$1.2 \cdot 10^{30}$	$> 10^{157}$
200	7	200	40,000	8,000,000	$1.6 \cdot 10^{60}$	$> 10^{374}$

constitute an important family of functions that will be associated with the time complexity of efficient algorithms. Conditions (iv) and (vi) show that the exponential and factorial functions are not polynomially bounded. The big oh hierarchy in Table 14.3, which lists functions in increasing order of their rates of growth, is obtained from the relationships outlined in Theorems 14.2.2 and 14.2.3. It is standard practice to refer to a function f for which $2^n \in O(f)$ as having *exponential growth*. With this convention, n^n and $n!$ are both said to exhibit exponential growth.

The efficiency of an algorithm is commonly characterized by its rate of growth. A polynomial algorithm is one whose complexity is polynomially bounded. That is, $c(n) \in O(n^r)$ for some $r \in \mathbb{N}$. The distinction between polynomial and nonpolynomial algorithms is apparent when considering the growth of these functions as the size of the input increases. Table 14.4 illustrates the enormous resources required by an algorithm whose complexity is not polynomial.

14.3 Time Complexity of a Turing Machine

The time complexity of a computation measures the amount of work expended by the computation. The time of a computation of a Turing machine is quantified by the number of transitions processed. The issues involved in determining the time complexity of a Turing machine are presented by analyzing the computations of the machine M that accepts palindromes over the alphabet $\{a, b\}$.



A computation of M consists of a loop that compares the first nonblank symbol on the tape with the last. The first symbol is recorded and replaced with a blank by the transition from state q_1 . Depending upon the path taken from q_1 , the final nonblank symbol is checked for a match in state q_4 or q_7 . The machine then moves to the left through the nonblank segment of the tape and the comparison cycle is repeated. When a blank is read in states q_2 or q_5 , the string is an odd-length palindrome and is accepted in state q_9 . Even-length palindromes are accepted in state q_1 .

The computations of M are symmetric with respect to the symbols a and b . The upper path from q_1 to q_8 is traversed when processing an a and the lower path when processing a b . The computations in Table 14.5 contain all significant combinations of symbols in strings of length 0, 1, 2, and 3.

As expected, the computations show that the number of transitions in a computation depends upon the particular input string. Indeed, the amount of work may differ radically for strings of the same length. Rather than attempting to determine the exact number of transitions for each input string, the time complexity of a Turing machine measures the maximum amount of work required by the strings of a fixed length.

TABLE 14.5 Computational Methods

Length 0	Length 1
$q_0 B B$	$q_0 B a B$
$\vdash B q_1 B$	$\vdash B q_1 a B$
	$\vdash B B q_2 B$
	$\vdash B B B q_3$

Definition 14.3.1

Let M be a standard T
 $N \rightarrow N$ such that $t c_M(n)$
of M when initiated with

When evaluating the
tations terminate for ever
or more accurately the c

Definition 14.3.1 says that a function is defined in a similar manner in the next chapter.

Our definition of the learning machine. In analyzing reasons. The first is that the value $t_{\mathcal{C}_M}(n)$ specifying M terminates will be strictly pragmatic; the whole performance.

The computations demonstrate the process when the entire input string of symbols is discovered. Since we need only concern ourselves with the largest possible number of symbols, it is satisfied when the input string is fully processed.

TABLE 14.5 Computations of M

Length 0	Length 1	Length 2		Length 3	
q_0BB	q_0BaB	q_0BaaB	q_0BabB	q_0BabaB	q_0BaabB
$\vdash Bq_1B$	$\vdash Bq_1aB$	$\vdash Bq_1aaB$	$\vdash Bq_1abB$	$\vdash Bq_1abaB$	$\vdash Bq_1aabB$
		$\vdash BBq_2B$	$\vdash BBq_2aB$	$\vdash BBq_2bB$	$\vdash BBq_2baB$
		$\vdash BBBq_9B$	$\vdash BBaq_3B$	$\vdash BBbq_3B$	$\vdash BBaq_3bB$
			$\vdash BBq_4aB$	$\vdash BBq_4bB$	$\vdash BBbaq_3B$
			$\vdash Bq_8BBB$		$\vdash BBbq_4aB$
			$\vdash BBq_1BB$		$\vdash BBq_8BBB$
					$\vdash Bq_8BbBB$
					$\vdash BBq_1bBB$
					$\vdash BBBq_5BB$
					$\vdash BBBBq_9B$

Definition 14.3.1

Let M be a standard Turing machine. The **time complexity** of M is the function $tc_M : N \rightarrow N$ such that $tc_M(n)$ is the maximum number of transitions processed by a computation of M when initiated with an input string of length n.

When evaluating the time complexity of a Turing machine, we assume that the computations terminate for every input string. It makes no sense to attempt to discuss the efficiency, or more accurately the complete inefficiency, of a computation that continues indefinitely.

Definition 14.3.1 serves equally well for machines that accept languages and compute functions. The time complexity of deterministic multitrack and multitape machines is defined in a similar manner. The complexity of nondeterministic machines will be discussed in the next chapter.

Our definition of time complexity measures the *worst-case performance* of the Turing machine. In analyzing an algorithm, we choose the worst-case performance for two reasons. The first is that we are considering the limitations of algorithmic computation. The value $tc_M(n)$ specifies the minimum resources required to guarantee that the computation of M terminates when initiated with any input string of length n. The other reason is strictly pragmatic; the worst-case performance is often easier to evaluate than the average performance.

The computations of the machine M that accepts the palindromes over {a, b} is used to demonstrate the process of determining the time complexity. A computation of M terminates when the entire input string has been replaced with blanks or the first nonmatching pair of symbols is discovered. Since the time complexity measures the worst-case performance, we need only concern ourselves with the strings whose computations cause the machine to do the largest possible number of match-and-erase cycles. For the machine M, this condition is satisfied when the input is accepted.

Using these observations, we can obtain the initial values of the function tc_M from the computations in Table 14.5.

$$tc_M(0) = 1$$

$$tc_M(1) = 3$$

$$tc_M(2) = 6$$

$$tc_M(3) = 10$$

Determining the remainder of the values of tc_M requires a detailed analysis of the computations of M . Consider the actions of M when processing an even-length input string. The computation alternates between sequences of right and left movements of the machine. Initially, the tape head is positioned to the immediate left of the nonblank segment of the tape.

- *Rightward movement:* The tape head moves to the right, erasing the leftmost nonblank symbol. The remainder of the string is read and the machine enters state q_4 or q_7 . This requires $k + 1$ transitions, where k is the length of the nonblank portion of the tape.
- *Leftward movement:* M moves left, erasing the matching symbol, and continues through the nonblank portion of the tape. This requires k transitions.

The preceding actions reduce the length of the nonblank portion of the tape by two. The cycle of comparisons and erasures is repeated until the tape is completely blank. As previously noted, the worst-case performance for an even-length string occurs when M accepts the input. A computation accepting a string of length n requires $n/2$ iterations of the preceding loop.

Iteration	Direction	Transitions
1	Right	$n + 1$
	Left	n
2	Right	$n - 1$
	Left	$n - 2$
3	Right	$n - 3$
	Left	$n - 4$
⋮		
$n/2$	Right	1

The total number of transitions of a computation can be obtained by adding those of each iteration. As indicated by the preceding table, the maximum number of transitions in a computation of a string of even length n is the sum of the first $n + 1$ natural numbers. An analysis of odd-length strings produces the same result. Consequently, the time complexity of M is given by the function

$$tc_M(n) = \sum_{i=1}^{n+1} i = (n + 2)(n + 1)/2 \in O(n^2).$$

Example 14.3.1

The two-tape machine

M' : 

also accepts the set of palindromes. In fact, it makes a copy on tape 1. At some point, the heads move to the right and read the symbols on tape 2. If the input is not a palindrome, the computation ends. If the input is a palindrome, the computation continues until the two tapes 1 and 2.

For an input of length n , the computation of M' is a palindrome. An even-length string is a palindrome. An odd-length string is not a palindrome. A computation accepting a string of length n requires $n/2$ iterations of the preceding loop.

A transition of the form $[x/x]$ is a complicated operation. The time complexity of a transition is $O(n)$. The time complexities of the machine M' and M are the same. Between the time complexities of the machines M' and M , there is a difference of $O(1)$. See Section 14.4.

The first step in determining the time complexity of the computation of the strings that are accepted by M' is to determine the palindromes, these we will do by the following example.

Example 14.3.2

Let M be the two-tape machine

M : 

where the symbols x and y consist of all strings of length n and the k th to last position of the string is x . For example, if the string is $abcbcc$, and all odd length substrings are x .

from the

the comput-
ut string.
machine.
ent of the

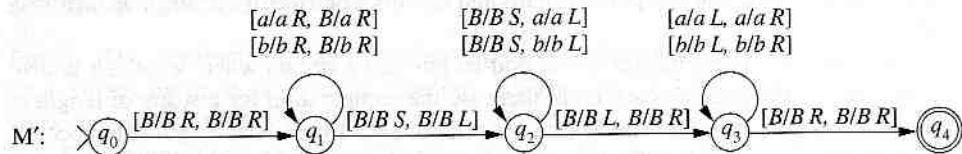
nonblank
or q_7 . This
he tape.
es through

be by two.
blank. As
s when M
erations of

ose of each
sitions in a
umbers. An
complexity

Example 14.3.1

The two-tape machine M'



also accepts the set of palindromes over $\{a, b\}$. A computation of M' traverses the input, making a copy on tape 2. The head on tape 2 is then moved back to tape position 0. At this point, the heads move across the input, tape 1 right to left and tape 2 left to right, comparing the symbols on tape 1 and tape 2. If the tape heads ever encounter different symbols, the input is not a palindrome and the computation halts and rejects the string. When the input is a palindrome, the computation halts and accepts when blanks are simultaneously read on tapes 1 and 2.

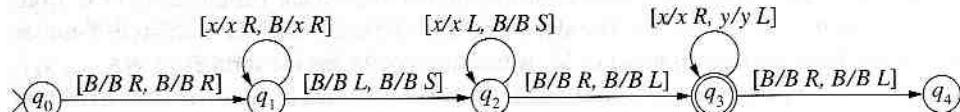
For an input of length n , the maximum number of transitions occurs when the string is a palindrome. An accepting computation requires three complete passes: the copy, the rewind, and the comparison. Counting the number of transitions in each pass, we see that the time complexity of M' is $tc_{M'}(n) = 3(n + 1) + 1$. \square

A transition of the two-tape machine utilizes more information and performs a more complicated operation than that of a one-tape machine. There is a trade-off between the complexity of a transition and the number that must be processed, as illustrated by the complexities of the machines M and M' that accept the palindromes. The precise relationship between the time complexity of one-tape and multitape Turing machines is established in Section 14.4.

The first step in determining the time complexity of a Turing machine is the identification of the strings that exhibit the worst-case behavior. In the machines that accepted the palindromes, these were the strings in the language. This is not always the case, as illustrated by the following example.

Example 14.3.2

Let M be the two-tape Turing machine



where the symbols x and y can be any symbol in $\{a, b, c\}$ and $x \neq y$. The language of M consists of all strings over $\{a, b, c\}$ in which there is at least one value k such that the k th and the k th to last position of the string have the same symbol. For example, $abaa$, $abccc$, $abcbbc$, and all odd length strings are in $L(M)$.

A computation of M employs the same strategy as the machine in Example 14.3.1. The input string is copied to tape 2 and the head on tape 1 is returned to the initial position. The symbols on tapes 1 and 2 are compared with tape head 1 moving left to right and tape head 2 moving right to left. The computation halts and accepts when the two heads scan identical symbols.

The worst-case performance for an odd-length string occurs when no match is discovered prior to the middle position. In this case the computation for a string of length n requires $\frac{5}{2}(n+1)$ transitions. The worst-case performance for an even length string occurs when the string is rejected by M . In a rejecting computation, tape head 1 scans the entire input three times. Thus

$$tc_M(n) = \begin{cases} \frac{5}{2}(n+1) & \text{if } n \text{ is odd} \\ 3(n+1) & \text{if } n \text{ is even.} \end{cases}$$

The acceptance of an even-length string takes at most $\frac{5}{2}n + 2$ transitions, which is always less than the worst-case performance. \square

Theorem 14.4.2

Let L be the language accepted by a k -track deterministic Turing machine M with time complexity $tc_M(n)$. Then L is accepted by a standard Turing machine M' with time complexity $tc_{M'}(n) = tc_M(n)$.

Proof. The construction of M' from M shows that $tc_{M'}(n) \leq tc_M(n)$.

The argument follows by showing that $tc_{M'}(n) \geq tc_M(n)$. The argument focuses on the actions of a multitape machine M' and the corresponding transitions of M .

Assume that we have a computation of M that accepts a string w . The head of M may be at any position on any track. The symbols on the odd-numbered tracks consist of the input string w , while the symbols on the even-numbered tracks consist of the tape alphabet.

14.4 Complexity and Turing Machine Variations

Several variations on the Turing machine model were presented in Chapter 8 to facilitate the design of machines that perform complex computations. In the study of decidability, the selection of the Turing machine model was irrelevant. We proved that any problem solvable using one Turing machine architecture was solvable using any of the others. In complexity theory, however, the choice matters. The machines in Section 14.3 that accept the palindromes over $\{a, b\}$ exhibit the potential differences in computational resources required by one-tape and two-tape machines. In this section we examine the relationship between the complexity of computations in various Turing machine models.

Theorem 14.4.1

Let L be the language accepted by a k -track deterministic Turing machine M with time complexity $tc_M(n)$. Then L is accepted by a standard Turing machine M' with time complexity $tc_{M'}(n) = tc_M(n)$.

Proof. This follows directly from the construction of a one-track Turing machine M' from a k -track machine in Section 8.4. The alphabet of the one-track machine consists of k -tuples of symbols from the tape alphabet of M . A transition of M has the form $\delta(q_i, x_1, \dots, x_k)$, where x_1, \dots, x_k are the symbols on track 1, track 2, \dots , track k . The associated transition of M' has the form $\delta(q_i, [x_1, \dots, x_k])$, where the k -tuple is the single alphabet symbol of M' . Thus the number of transitions processed by M and M' are identical for every input string and $tc_M = tc_{M'}$. \blacksquare

Action

Write symbol c

and return to q_i

Write symbol c

and return to q_i

⋮

Write symbol c

and return to q_i

1. The
on. The
e head
entical

is dis-
length n
occurs
e entire

always

□

facilitate
dability,
problem
thers. In
at accept
esources
ationship

ime com-
plexity

re M' from
f k -tuples
 \dots, x_k ,
transition
symbol of
every input

■

Theorem 14.4.2

Let L be the language accepted by a k -tape deterministic Turing machine M with time complexity $tc_M(n) = f(n)$. Then L is accepted by a standard Turing machine N with time complexity $tc_N(n) \in O(f(n)^2)$.

Proof. The construction of an equivalent one-tape machine from a k -tape machine uses a $2k + 1$ -track machine M' as an intermediary. By Theorem 14.4.1, all that is required is to show that $tc_{M'} \in O(f(n)^2)$.

The argument follows the construction of the multitrack machine M' that simulates the actions of a multitape machine described in Section 8.6. We begin by analyzing the number of transitions of M' that are required to simulate a single transition of M .

Assume that we are simulating the t th transition of M . The farthest right that a tape head of M may be at this time is tape position t . The first step in the simulation records the symbols on the odd-numbered tapes marked by the X 's on the even-numbered tapes. This consists of the following sequence of transitions of M' :

Action	Maximum Number of Transitions of M'
Find X on second track and return to tape position 0	$2t$
Find X on fourth track and return to tape position 0	$2t$
\vdots	\vdots
Find X on $2k$ th track and return to tape position 0	$2t$

After finding the symbol under each X , M' uses one transition to record the action taken by M . The simulation of the transition of M is completed by

Action	Maximum Number of Transitions of M'
Write symbol on track 1, reposition X on track 2, and return to tape position 0	$2(t + 1)$
Write symbol on track 3, reposition X on track 4, and return to tape position 0	$2(t + 1)$
\vdots	\vdots
Write symbol on track $2k - 1$, reposition X on track $2k$, and return to tape position 0	$2(t + 1)$

Consequently, the simulation of the t th transition of M requires at most $4kt + 2k + 1$ transitions of M' . The computation of M' begins with a single transition that places the markers on the even-numbered tracks and the $\#$ on track $2k + 1$. The remainder of the computation consists of the simulation of the transitions of M . An upper bound on the number of transitions of M' needed to simulate the computation of M with input of length n is

$$tc_{M'}(n) \leq 1 + \sum_{t=1}^{f(n)} (4kt + 2k + 1) \in O(f(n)^2). \blacksquare$$

14.5 Linear Speedup

The time complexity function $tc_M(n)$ of a Turing machine M gives the maximum number of transitions required for a computation with an input string of length n . In this section we show that a machine that accepts a language L can be “sped up” to produce another machine that accepts L in time that is faster by an arbitrary multiplicative constant.

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a k -tape Turing machine, $k > 1$, that accepts a language L . The underlying strategy involved in the speedup is to construct a machine N that accepts L in which a block of six transitions of N simulates m transitions of M , where the value of m is determined by the desired degree of speedup. For example, selecting $m = 12$ reduces the number of transitions by approximately one-half since six transitions of N achieve the same result as 12 of M . The word *approximately* is included in the previous sentence because of some initial overhead required by N prior to the simulation of the computation of M .

Since the machines M and N accept the same language, the input alphabet of N is also Σ . The tape alphabet of N includes that of M , as well as the symbol $\#$ and all ordered m -tuples of symbols of Γ . A computation of N consists of two phases, initialization and simulation. The initialization translates the input into a sequence of m -tuples. The remainder of the computation of N simulates the computation of M .

During the simulation of M , a tape symbol of N is an m -tuple of symbols of M , and the states of N are used to record the portion of the tapes of M that may be affected by the next m transitions of M . In this phase of the computation of N , a state of N consists of

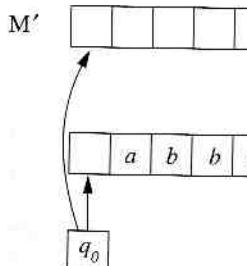
- i) the state of M ;
- ii) for $i = 1$ to k , the m -tuple currently scanned on tape i of N and the m -tuples to the immediate right and left; and
- iii) an ordered k -tuple $[i_1, \dots, i_k]$, where i_j is the position of the symbol on tape j being scanned by M in the m -tuple being scanned by N .

A sequence of six transitions of N uses the information in the state to simulate m transitions of M .

The process will be demonstrated using the two-tape machine M' from Example 14.3.1 with $m = 3$ and input $abbabba$. The input configuration of N is exactly that of M' , with the input string on tape 1 and tape 2 entirely blank. The first action of N is to encode the input

string into m -tuples. To every three consecutive ordered triple written evenly divisible by three original input string is N will simulate tape 1

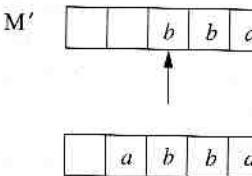
The next diagram shows the configuration of N after



After the initialization, the encoded triple [BBB] appears on tape 1. The remaining blanks of N will be written.

The m -tuples [BBB] are simulated. The ordered pair [aa] that occurs in the first cell of the input is simulated. The symbol ? is a placeholder for each ? is replaced with the symbol currently being scanned.

The simulation of M continues on tapes 1 and 2.



that would be obtained if the state of N is

ces the
of the
und on
nput of

number
ction we
machine

language
t accepts
alue of m
lutes the
the same
ecause of
M.

N is also
1 ordered
ation and
emainder

of M, and
ted by the
sts of

bles to the

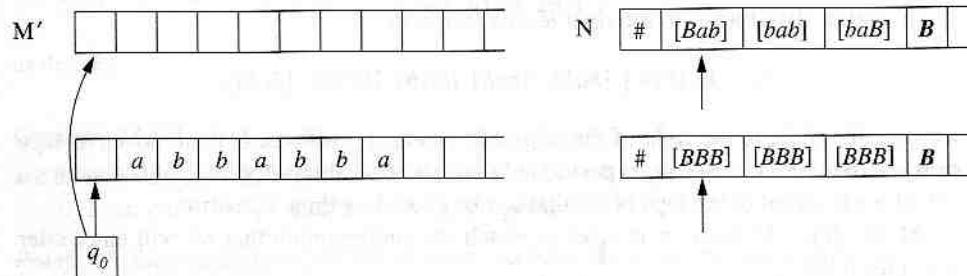
be j being

transitions

ple 14.3.1
I', with the
e the input

string into m -tuples. The process begins by writing # on position zero of both tapes. For every three consecutive symbols on tape 1, an ordered triple is written on tape 2. The final ordered triple written on tape 2 is padded with blanks since the length of *Babbabba* is not evenly divisible by three. The tape heads of N are repositioned at tape position one and the original input string is erased from tape 1. For the remainder of the computation, tape 2 of N will simulate tape 1 of M', and tape 1 of N will simulate tape 2 of M'.

The next diagram shows the initial configuration of M' with input $abbabba$ and the configuration of N after the encoding.

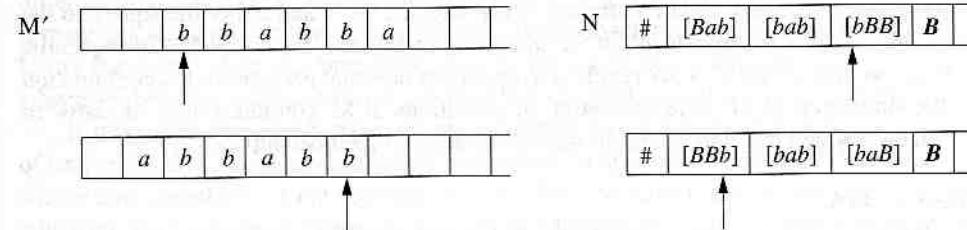


After the initialization, each blank on the tape of N will be considered to represent an encoded triple $[BBB]$ of blanks of M' . To illustrate the difference in the diagrams, the blanks of N will be written B . After the encoding of the input, N will enter the state

$(q_0; \text{ ?}, [BBB], \text{ ?}; \text{ ?}, [Bab], \text{ ?}; [1, 1]).$

The m -tuples [BBB] and [Bab] are those currently scanned by N on tapes 1 and 2, respectively. The ordered pair [1, 1] indicates that the computation of M' is scanning the symbol that occurs in the first position in each of the triples [BBB] and [Bab] in the state of N. The symbol ? is a placeholder; subsequent transitions will cause N to enter states in which each ? is replaced with information concerning the triples to the left and right of the position currently being scanned.

The simulation of m moves of M' is demonstrated by considering the configurations of M' and N



that would be obtained during the processing of *abbabba*. Upon entering this configuration, the state of N is

$$(q_3; \text{ ?}, [BBb], \text{ ?}; \text{ ?}, [bBB], \text{ ?}; \text{ [3, 1]}).$$

The ordered pair $[3, 1]$ in the state indicates that the computation of M' is reading the b in the triple $[BBb]$ on tape 1 and the b in the triple $[bBB]$ on tape 2.

The machine N then makes a move to the left on each tape, scans the squares, and enters state

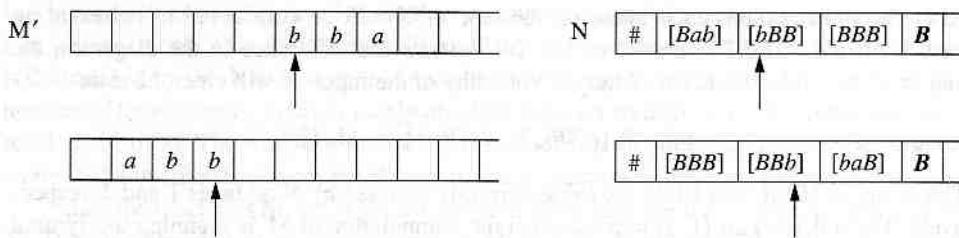
$$(q_3; \#, [BBb], ?; [bab], [bBB], ?; [3, 1]),$$

which records the triples to the left of the originally scanned squares in the state. The role of the marker $\#$ is to ensure that N will not cross a left-hand tape boundary in this phase of the simulation. Two moves to the right leaves N in state

$$(q_3; \#, [BBb], [bab]; [bab], [bBB], [BBB]; [3, 1]),$$

recording the triple to the right of the originally scanned positions. N then moves its tape heads left to return to the original position. After these transitions, the state of N contains a copy of the segment of the tape of M' that can be altered by three transitions.

At this point, N rewrites its tapes to match the configuration that M' will enter after three transitions



and enters state

$$(q_3; ?, [BBb], ?; ?, [bBB], ?; [3, 1])$$

to begin the simulation of the next three transitions of M' . Since each tape square of N has three symbols of M' , the portion of the tape of M' that can be altered by three transitions is contained in the tape square currently being scanned by N and either the square to the immediate right or immediate left of the square being scanned, but not both. Consequently, at most two transitions of N are required to update its tape and prepare for the continuation of the simulation of M' . The simulation of transitions of M' continues until M' halts, in which case N will halt and return the same indication of membership as M' .

Theorem 14.5.1

Let M be a k -tape Turing machine, $k > 1$, that accepts L with $tc_M(n) = f(n)$. For any constant $c > 0$, there is a k -tape machine N that accepts L with $tc_N(n) \leq \lceil cf(n) \rceil + 2n + 3$.

Proof. The construction of the machine N has just been described. Encoding an input string of length n as m -tuples and repositioning the tape heads require $2n + 3$ transitions.

The remainder of M . To obtain and N takes one move to position. At most two transitions of N are $m \geq 6/c$ produces

as desired.

Corollary 14.5.2

Let M be a one-tape machine in which $c > 0$, there is a two

Proof. In the standard machine in which the then be used to speed

The speedup in alphabet and vastly these sets is left as a

14.6 Properties

The definition of the machine M and not of different machines different time complexity if there is a standard Using the results whenever there is a

In this section we of languages. First, whose time complexity which no “best” accept that a machine accept not change the rate languages which, with time complexity greater

The remainder of the computation of N consists of the simulation of the computation of M . To obtain and record the information needed to simulate m transitions of M , machine N takes one move to the left, two to the right, and one to reposition the head at the original position. At most two transitions are then required to reconfigure the tapes of N . Thus six transitions of N are sufficient to produce the same result as m of the machine M . Choosing $m \geq 6/c$ produces

$$\begin{aligned} tc_N(n) &= \lceil (6/m)f(n) \rceil + 2n + 3 \\ &\leq \lceil cf(n) \rceil + 2n + 3, \end{aligned}$$

as desired. ■

Corollary 14.5.2

Let M be a one-tape Turing machine that accepts L with $tc_M(n) = f(n)$. For any constant $c > 0$, there is a two-tape machine N that accepts L with $tc_N(n) \leq \lceil cf(n) \rceil + 2n + 3$.

Proof. In the standard manner, the one-tape machine M can be considered to be a two-tape machine in which the second tape is not referenced in the computation. Theorem 14.5.1 can then be used to speed up the two-tape machine. ■

The speedup in Theorem 14.5.1 was obtained at the expense of creating a larger tape alphabet and vastly increasing the number of states. The exact determination of the size of these sets is left as an exercise.

14.6 Properties of Time Complexity of Languages

The definition of the time complexity function tc_M is predicated on the computations of the machine M and not on the underlying language accepted by the machine. We know that many different machines can be constructed to accept the same language, each with a possibly different time complexity. We say that a language L is accepted in deterministic time $f(n)$ if there is a standard (one-tape deterministic) Turing machine M with $tc_M(n) \in O(f(n))$. Using the results from the preceding section, we know that a language L is $O(f(n)^2)$ whenever there is a multitape Turing machine that accepts L with time complexity $O(f(n))$.

In this section we establish two interesting results on the bounds of the time complexity of languages. First, we show that for any computable total function $f(n)$, there is a language whose time complexity is not bounded by $f(n)$. We then show that there are languages for which no “best” accepting Turing machine exists. Theorem 14.5.1 has already demonstrated that a machine accepting a language can be sped up linearly. That process, however, does not change the rate of growth of the accepting machine. We will now show that there are languages which, when accepted by any machine, are also accepted by a machine whose time complexity grows at a strictly smaller rate than the original machine.

Both of these results utilize the ability to encode and enumerate all multitape Turing machines. An encoding of one-tape machines as strings over $\{0, 1\}$ was outlined in Section 11.5. This approach can be extended to an encoding of all multitape machines with input alphabet $\{0, 1\}$. The tape alphabet is assumed to consist of elements $\{0, 1, B, x_1, \dots, x_n\}$. The tape symbols are encoded as follows:

Symbol	Encoding
0	1
I	11
B	111
x_1	1111
\vdots	\vdots
x_n	1^{n+3}

As before, a number is encoded by its unary representation and a transition by its encoded components separated by 0's; encoded transitions are separated by 00. With these conventions, a k -tape machine may be encoded

$000\bar{k}000en$ (accepting states) $000en$ (transitions) 000 .

where \bar{k} is the unary representation of k and en denotes the encoding of the items in parentheses.

With this representation, every string $u \in \{0, 1\}^*$ can be considered to be the encoding of some multitape Turing machine. If u does not satisfy the syntactic conditions for the encoding of a multitape Turing machine, the string is interpreted as the representation of the one-tape, one-state Turing machine with no transitions.

In Exercise 8.32 a Turing machine E that enumerated all strings over $\{0, 1\}$ was constructed. Since every such string also represents a multitape Turing machine, the machine E can be equally well thought of as enumerating all Turing machines with input alphabet $\{0, 1\}$. The strings enumerated by E will be written u_0, u_1, u_2, \dots and the corresponding machines by M_0, M_1, M_2, \dots .

We will now show that there is no upper bound on the time complexity of languages. More precisely, for any computable function f we will build a recursive language L such that no Turing machine M with $tc_M(n) \leq f(n)$ accepts L . The proof uses diagonalization to obtain a contradiction from the assumption of the existence of such a machine.

Theorem 14.6.1

Let f be a total computable function. Then there is a language L such that tc_M is not bounded by f for any deterministic Turing machine M that accepts L .

Proof. Let F be a Turing machine that computes the function f . Consider the language $L = \{u; \mid M; \text{ does not accept } u; \text{ in } f(n) \text{ or fewer moves, where } n = \text{length}(u)\}$. First, we

show that L is recursive
 L is not bounded by

A machine M that
Recall that the string a
of all multitape Turing

1. determines the labels
 2. simulates the computation
 3. simulates M_1 on the first; and
 4. M accepts u_i if e has transitions. Otherwise

Clearly, the language is
terminable.

The language L used to produce a constant time complexity bound occurs somewhere in M is obtained by considering only if, M_j halts in the first $f(n)$ transitions.

The proof that M_j has complexity $\leq M_j$ is based on either $u_i \in L$ or $u_i \notin L$.

If $u_j \in L$, then M_j are assumed to be
if, M_i halts without a

If $u_j \notin L$, then the algorithm does not accept u_j . In this case,

In either case, this leads to a contradiction that accepts L is not true.

Next we show that how this might occur, language over \emptyset^* . The

- $$\text{i) } t(1) = 2$$

$$\text{iii) } t(n) = 2^{t(n-1)}$$

Thus $t(2) = 2^2$, $t(3)$ number of transitions

uring
Sec-
input
, $x_n\}$.

by its
n these

ems in

coding
for the
ition of

as con-
machine
lphabet
onding

guages.
L such
lization

bounded

anguage
First, we

show that L is recursive and then that the number of transitions of any machine that accepts L is not bounded by $f(n)$.

A machine M that accepts L is described below. The input to M is a string u_i in $\{0, 1\}^*$. Recall that the string u_i represents the encoding of the Turing machine M_i in the enumeration of all multtape Turing machines. A computation of M

1. determines the length of u_i , say, $\text{length}(u_i) = n$;
2. simulates the computation of F to determine $f(n)$;
3. simulates M_i on u_i until M_i either halts or completes $f(n)$ transitions, whichever comes first; and
4. M accepts u_i if either M_i halted without accepting u_i or M_i did not halt in the first $f(n)$ transitions. Otherwise, u_i is rejected by M .

Clearly, the language $L(M)$ is recursive, since step 3 ensures that each computation will terminate.

The language L has been designed so that diagonalization and self-reference can be used to produce a contradiction to the claim that L is accepted by a Turing machine with time complexity bounded by $f(n)$. Let M be any Turing machine that accepts L . Then M occurs somewhere in the enumeration of Turing machines, say $M = M_j$. The self-reference is obtained by considering the membership of u_j in L . Since $L(M_j) = L$, M_j accepts u_j if, and only if, M_j halts without accepting u_j in $f(n)$ or fewer transitions or M_j does not halt in the first $f(n)$ transitions.

The proof that M_j is not bounded by f is by contradiction. Assume that the time complexity of M_j is bounded by f and let $n = \text{length}(u_j)$. There are two cases to consider: either $u_j \in L$ or $u_j \notin L$.

If $u_j \in L$, then M_j accepts u_j in $f(n)$ or fewer transitions (since the computations of M_j are assumed to be bounded by f). But, as previously noted, M_j accepts u_j if, and only if, M_j halts without accepting u_j or M_j does not halt in the first $f(n)$ transitions.

If $u_j \notin L$, then the computation of M_j halts within the bound of $f(n)$ steps and does not accept u_j . In this case, $u_j \in L$ by the definition of L .

In either case, the assumption that the number of transitions of M_j is bounded by f leads to a contradiction. Consequently, we conclude that time complexity of any machine that accepts L is not bounded by f . ■

Next we show that there is a language that has no fastest accepting machine. To illustrate how this might occur, consider a sequence of machines N_0, N_1, \dots that all accept the same language over 0^* . The argument uses the function t that is defined recursively by

- i) $t(1) = 2$
- ii) $t(n) = 2^{t(n-1)}$.

Thus $t(2) = 2^2$, $t(3) = 2^{2^2}$, and $t(n)$ is a series of n 2's as a sequence of exponents. The number of transitions of machine N_i when run with input 0^j is given in the $[i, j]$ th position

TABLE 14.6 Machines N_i and Their Computations

	λ	0	00	0^3	0^4	0^5	0^6	\dots
N_0	*	2	4	$t(3)$	$t(4)$	$t(5)$	$t(6)$	
N_1	*	*	2	4	$t(3)$	$t(4)$	$t(5)$	
N_2	*	*	*	2	4	$t(3)$	$t(4)$	
N_3	*	*	*	*	2	4	$t(3)$	
N_4	*	*	*	*	*	2	4	
\vdots								

of Table 14.6. A * in the $[i, j]$ th position indicates that the number of transitions of this computation is irrelevant.

If such a sequence of machines exists, then

$$tc_{N_i}(n) = \log_2(tc_{N_{i-1}}(n))$$

for all $n \geq i + 1$. Consequently, we have a sequence of machines that accept the same language in which each machine has a strictly smaller rate of growth than its predecessor. A language that exhibits the “this can always be accepted more efficiently” property is constructed in Theorem 14.6.2.

The speedup in both the motivating discussion and in the construction in Theorem 14.6.2 uses the property that rates of growth measure the performance of the function as the input gets arbitrarily large. From the pattern in Table 14.6, we see that the computations of machines N_i and N_{i+1} are compared only on input strings of length $i + 2$ or greater.

Theorem 14.6.2

There is a language L such that, for any machine M that accepts L , there is another machine M' that accepts L with $tc_{M'}(n) \in O(\log_2(tc_M(n)))$.

Let t be the function defined recursively by $t(1) = 2$ and $t(n) = 2^{t(n-1)}$ for $n > 1$ as before. A recursive language $L \subseteq \{0\}^*$ is constructed that satisfies the following two conditions:

1. If M_i accepts L , then $tc_{M_i}(n) \geq t(n - i)$ for all n greater than some n_i .
2. For each k , there is a Turing machine M_j with $L(M_j) = L$ and $tc_{M_j}(n) \leq t(n - k)$ for all n greater than some n_k .

Assume that L has been constructed to satisfy the preceding conditions. For every machine M_i that accepts L there is an M_j that also accepts L with

$$tc_{M_j}(n) \in O(\log_2(tc_{M_i}(n))),$$

as desired. To see this, note that $tc_{M_j}(n) \leq t(n - i)$ since $tc_{M_i}(n) \geq t(n - i)$.

Combining the two inequalities, we get

$$tc_{M_j}(n) \geq \log_2(t(n - i))$$

That is, $tc_{M_j}(n) \leq \log_2(t(n - i))$.

We now define the language L as the union of the languages 0^n , $n = 0, 1, 2, \dots$. We will use an enumeration M_0, M_1, M_2, \dots of all machines to construct L . We will examine a machine $M_g(n)$ that accepts L for some n .

- M_j has not been previously examined.
- $tc_{M_j}(n) < t(n - j)$.

It is possible that no such machine M_j exists. Then $0^n \in L$ if, and only if, $g(n) \leq n$. In this case, $M_g(n)$ is marked as candidate for L . If $M_g(n)$ is not marked, then $L(M_g(n)) \neq L$. Consequently, $L(M_g(n)) \subseteq L$.

The proof of Theorem 14.6.2 first shows that the language L defined in Theorem 14.6.2 stated earlier are satisfied.

Lemma 14.6.3

The language L is recursive.

Proof. The definition of L shows that the computation of $tc_{M_g(n)}$ begins by examining the machines M_0, M_1, M_2, \dots until $M_g(n)$ is found. Let us determine the machine $M_g(n)$ that accepts L . We begin by examining the analysis of input λ , then 0 , then 00 , and so on. For each character 0 , there is a machine M_i that accepts L with input 0^i .

After the machines M_0, M_1, M_2, \dots are used to determine L , if $M_g(n)$ is not found, then $t(n - g(n))$ is cancelled, then $t(n - j)$, then $t(n - k)$, and so on until $tc_{M_j}(n) < t(n - j)$, then M_j is found. This continues until $g(n)$ is found or until $t(n - g(n))$ is cancelled.

If $g(n)$ exists, $M_g(n)$ accepts L . We can determine the membership of 0^n in L by examining the machines M_0, M_1, M_2, \dots until $M_g(n)$ is found or until $t(n - g(n))$ is cancelled.

as desired. To see this, set $k = i + 1$. By condition 2, there is a machine M_j that accepts L and $tc_{M_j}(n) \leq t(n - i - 1)$ for all $n \geq n_k$. However, by condition 1,

$$tc_{M_i}(n) \geq t(n - i) \text{ for all } n > n_i.$$

Combining the two inequalities with the definition of t yields

$$tc_{M_i}(n) \geq t(n - i) = 2^{t(n-i-1)} \geq 2^{tc_{M_j}(n)} \text{ for all } n > \max\{n_i, n_k\}.$$

That is, $tc_{M_j}(n) \leq \log_2(tc_{M_i}(n))$ for all $n > \max\{n_i, n_k\}$.

We now define the construction of the language L . Sequentially, we determine whether strings 0^n , $n = 0, 1, 2, \dots$, are in L . During this construction, Turing machines in the enumeration M_0, M_1, M_2, \dots are marked as cancelled. In determining whether $0^n \in L$, we examine a machine $M_{g(n)}$ where $g(n)$ is the least value j in the range $0, \dots, n$ such that

- i) M_j has not been previously cancelled, and
- ii) $tc_{M_j}(n) < t(n - j)$.

It is possible that no such value j may exist, in which case $g(n)$ is undefined. The string $0^n \in L$ if, and only if, $g(n)$ is defined and $M_{g(n)}$ does not accept 0^n . If $g(n)$ is defined, then $M_{g(n)}$ is marked as cancelled. The definition of L ensures that a cancelled machine cannot accept L . If $M_{g(n)}$ is cancelled, then $0^n \in L$ if, and only if, 0^n is not accepted by $M_{g(n)}$. Consequently, $L(M_{g(n)}) \neq L$.

The proof of Theorem 14.6.2 consists of establishing the following three lemmas. The first shows that the language L is recursive. The final two demonstrate that conditions 1 and 2 stated earlier are satisfied by L .

Lemma 14.6.3

The language L is recursive.

Proof. The definition of L provides a method for deciding whether $0^n \in L$. The decision process for 0^n begins by determining the index $g(n)$, if it exists, of the first machine in the sequence M_0, \dots, M_n that satisfies conditions 1 and 2. To accomplish this, it is necessary to determine the machines in the sequence M_0, M_1, \dots, M_{n-1} that have been cancelled in the analysis of input $\lambda, 0, \dots, 0^{n-1}$. This requires comparing the value of the complexity functions in Table 14.7 with the appropriate value of t . The input alphabet consists of the single character 0 , therefore $tc_{M_i}(m)$ can be determined by simulating the computation of M_i with input 0^m .

After the machines that have been cancelled are recorded, the computations with input 0^n are used to determine $g(n)$. Beginning with $j = 0$, if M_j has not previously been cancelled, then $t(n - j)$ is computed and the computation of M_j on 0^n is simulated. If $tc_{M_j}(n) < t(n - j)$, then $g(n) = j$. If not, j is incremented and the comparison is repeated until $g(n)$ is found or until all the machines M_0, \dots, M_n have been tested.

If $g(n)$ exists, $M_{g(n)}$ is run with input 0^n . The result of this computation determines the membership of 0^n in L : $0^n \in L$ if, and only if, $M_{g(n)}$ does not accept it. The preceding

TABLE 14.7 Computations to Determine Cancelled Machines

Input	m	Comparison $tc_{M_i}(m) \leq t(m - i)$
λ	0	$tc_{M_0}(0) \leq t(0 - 0) = t(0)$
0	1	$tc_{M_0}(1) \leq t(1 - 0) = t(1)$ $tc_{M_1}(1) \leq t(1 - 1) = t(0)$
00	2	$tc_{M_0}(2) \leq t(2 - 0) = t(2)$ $tc_{M_1}(2) \leq t(2 - 1) = t(1)$ $tc_{M_2}(2) \leq t(2 - 2) = t(0)$
:	:	:
0^{n-1}	$n - 1$	$tc_{M_0}(n - 1) \leq t(n - 1 - 0) = t(n - 1)$ $tc_{M_1}(n - 1) \leq t(n - 1 - 1) = t(n - 2)$ $tc_{M_2}(n - 1) \leq t(n - 1 - 2) = t(n - 3)$ ⋮ $tc_{M_{n-1}}(n - 1) \leq t(n - 1 - (n - 1)) = t(0)$

process describes a decision procedure that determines the membership of any string 0^n in L; hence, L is recursive. ■

Lemma 14.6.4

L satisfies condition 1.

Proof. Assume M_i accepts L. First note that there is some integer p_i such that if a machine M_0, M_1, \dots, M_i is ever cancelled, it is cancelled prior to examination of the string 0^{p_i} . Since the number of machines in the sequence M_0, M_1, \dots, M_i that are cancelled is finite, at some point in the generation of L all of those that are cancelled will be so marked. We may not know for what value of p_i this occurs, but it must occur sometime and that is all that we require.

For any 0^n with n greater than the maximum of p_i and i , no M_k with $k < i$ can be cancelled. Suppose $tc_{M_i}(n) < t(n - i)$. Then M_i would be cancelled in the examination of 0^n . However, a Turing machine that is cancelled cannot accept L. It follows that $tc_{M_i}(n) \geq t(n - i)$ for all $n > \max\{p_i, i\}$. ■

Lemma 14.6.5

L satisfies condition 2.

Proof. We must prove, for any integer k , that there is a machine M that accepts L and $tc_M(n) \leq t(n - k)$ for all n greater than some value n_k . We begin with the machine M that accepts L described in Lemma 14.6.3. To decide if 0^n is in L, machine M determines the value $g(n)$ and simulates $M_{g(n)}$ on 0^n . To establish $g(n)$, M must determine which of the M_i 's, $i < n$, have been cancelled during the analysis of strings $\lambda, 0, 00, \dots, 0^{n-1}$.

Unfortunately, a string may require more than $t(n - k)$ transitions.

As noted in Lemma 14.6.3, a machine is cancelled when $tc_{M_i}(n) < t(n - i)$. This value p_k can be determined by examining each $m \leq p_k$, the infinite sequence of strings accepted by M that accepts L.

The computation of p_k is as follows:

Case 1: $n \leq p_k$. The string 0^n is recorded in the states of M.

Case 2: $n > p_k$. The first step in the computation of Turing machine M is to see if M_i is cancelled for some $i < n$. If M_i is not cancelled, then M_{k+1}, \dots, M_n since M_i is the last machine to be cancelled and is greater than p_k .

The ability to skip over $n - p_k$ transitions needed to determine $g(n)$ is indicated in Table 14.7.

process describes a decision procedure that determines the membership of any string 0^n in L; hence, L is recursive. ■

Lemma 14.6.4

L satisfies condition 1.

Proof. Assume M_i accepts L. First note that there is some integer p_i such that if a machine M_0, M_1, \dots, M_i is ever cancelled, it is cancelled prior to examination of the string 0^{p_i} . Since the number of machines in the sequence M_0, M_1, \dots, M_i that are cancelled is finite, at some point in the generation of L all of those that are cancelled will be so marked. We may not know for what value of p_i this occurs, but it must occur sometime and that is all that we require.

For any 0^n with n greater than the maximum of p_i and i , no M_k with $k < i$ can be cancelled. Suppose $tc_{M_i}(n) < t(n - i)$. Then M_i would be cancelled in the examination of 0^n . However, a Turing machine that is cancelled cannot accept L. It follows that $tc_{M_i}(n) \geq t(n - i)$ for all $n > \max\{p_i, i\}$. ■

Lemma 14.6.5

L satisfies condition 2.

Proof. We must prove, for any integer k , that there is a machine M that accepts L and $tc_M(n) \leq t(n - k)$ for all n greater than some value n_k . We begin with the machine M that accepts L described in Lemma 14.6.3. To decide if 0^n is in L, machine M determines the value $g(n)$ and simulates $M_{g(n)}$ on 0^n . To establish $g(n)$, M must determine which of the M_i 's, $i < n$, have been cancelled during the analysis of strings $\lambda, 0, 00, \dots, 0^{n-1}$.

Checking whether a string requires more than $t(n - k)$ transitions. The maximum length of the preceding sequence is $t(n - k)$.

The machine M begins with $t(n - k - 1)$ transitions. It then examines the tape after the simulation of $M_{g(n)}$ and makes an additional $2t(n - k - 1)$ transitions. This is repeated for each machine M_i that is cancelled.

Unfortunately, a straightforward evaluation of these cases as illustrated in Table 14.7 may require more than $t(n - k)$ transitions.

As noted in Lemma 14.6.4, any Turing machine M_i , $i \leq k$, that is ever cancelled is cancelled when considering some initial sequence $\lambda, 0, 00, \dots, 0^{p_k}$ of input strings. This value p_k can be used to reduce the complexity of the preceding computation. For each $m \leq p_k$, the information on whether 0^m is accepted is stored in states of the machine M that accepts L .

The computation of machine M with input 0^n then can be split into two cases.

Case 1: $n \leq p_k$. The membership of 0^n in L is determined solely using the information recorded in the states of M .

Case 2: $n > p_k$. The first step is to determine $g(n)$. This is accomplished by simulating the computation of Turing machines M_i , $i = k + 1, \dots, n$ on inputs 0^m , $m = k + 1, \dots, n$ to see if M_i is cancelled on or before 0^n . We only need to check machines in the range M_{k+1}, \dots, M_n since no machine M_0, \dots, M_k will be cancelled by an input of length greater than p_k .

The ability to skip the simulations of machines M_0, \dots, M_k reduces the number of transitions needed to evaluate an input string 0^n with $n > p_k$. The number of simulations indicated in Table 14.7 is reduced to

ing 0^n in ■

Input	m	Comparison $tc_{M_i}(m) \leq t(m - i)$
0^{k+1}	$k + 1$	$tc_{M_{k+1}}(k + 1) \leq t(k + 1 - (k + 1)) = t(0)$
0^{k+2}	$k + 2$	$tc_{M_{k+1}}(k + 2) \leq t(k + 2 - (k + 1)) = t(1)$ $tc_{M_{k+2}}(k + 2) \leq t(k + 2 - (k + 2)) = t(0)$
\vdots	\vdots	\vdots
0^n	n	$tc_{M_{k+1}}(n) \leq t(n - (k + 1))$ $tc_{M_{k+2}}(n) \leq t(n - (k + 2))$ \vdots $tc_{M_n}(n) \leq t(n - n) = t(0)$

Checking whether machine M_i is cancelled with input 0^m requires at most $t(m - i)$ transitions. The maximum number of transitions required for any computation in the preceding sequence is $t(n - k - 1)$, which occurs for $i = k + 1$ and $m = n$.

The machine M must perform each of the indicated comparisons. At most $t(n - k - 1)$ transitions are required to simulate the computation of M_i on 0^m . Erasing the tape after the simulation and preparing the subsequent simulation can be accomplished in an additional $2t(n - k - 1)$ transitions. The simulation and comparison cycle must be repeated for each machine M_i , $i = k + 1, \dots, n$, and input 0^m , $m = k + 1, \dots, n$. Thus

epts L and
machine M
determines
e which of
 $\dots, 0^{n-1}$. ■

the process of simulation is repeated at most $(n - k)(n - k + 1)/2$ times. Consequently, the number of transitions required by M is less than $3(n - k)(n - k + 1)t(n - k - 1)/2$. That is,

$$tc_M(n) \leq 3(n-k)(n-k+1)t(n-k-1)/2.$$

However, the rate of growth of $3(n-k)(n-k+1)t(n-k-1)/2$ is less than that of $t(n-k) = 2^{t(n-k-1)}$. Consequently, $t_{CM}(n) \leq t(n-k)$ for all n greater than some n_+ . ■

The preceding proof demonstrated that for any machine M accepting L , there is a machine M' that accepts L more efficiently than M . Now, M' accepts L so, again by Theorem 14.6.2, there is a more efficient machine M'' that accepts L . This process can continue indefinitely, producing a sequence of machines each of which accepts L with strictly smaller rate of growth than its predecessor.

Theorem 14.6.2 reveals a rather nonintuitive property of algorithmic computation; there are decision problems that have no best solution. Given any algorithmic solution to such a problem, there is another solution that is significantly more efficient.

14.7 Simulation of Computer Computations

Our study of the complexity of an algorithm is based on the number of transitions in the computations of a Turing machine implementation of the algorithm. However, the vast majority of the computational work that we do is usually not done on a Turing machine but rather on a computer. To illustrate the practical application of the analysis of Turing machine computations, we will compare the time complexity of an algorithm run on a standard computer with the complexity of running the same algorithm on a Turing machine, where the time complexity of a computation on a computer is measured by the number of machine instructions executed during the computation.

We will not produce a theorem that precisely relates the number of instructions to the number of transitions. This is impossible since different computers have different architectures, instruction sets, memory sizes, and computational capabilities. What we will do, however, is define a general type of machine instruction that subsumes those of standard machine or assembly languages. In fact, the flexibility and computational power that we give to our instructions far surpass that found in typical computer architectures.

The first thing to note is that we are interested in comparing a real computer, not a theoretical machine, with a Turing machine. Thus our machine must have a finite memory. The memory can be as large as desired, but finite. The machine memory is divided into fixed-length addressable words. In practice, a word usually consists of 32 or 64 bits, but we will allow the length of the words in our machine to be of any fixed finite length. The sole restriction on the length of a word is that it be large enough to hold our machine instructions. Each word has an associated numeric address that is used to retrieve and store data.

A machine instruction may be performed, followed by Boolean calculations. Allocation may be required if memory available during execution of memory, say, m_a , would be exceeded. The number m_a , of course,

The operands designate the memory locations where the results are stored. If there are one or two operands, both are memory locations. Since t is the target of the instruction, we assume that it is a memory location. Our final result must be finite.

Summarizing these

Components

Memory:
Word size:
Instruction set:
Instruction:
Operation:

It should be clear that models satisfy these rudimentary requirements, and programs of interest to us. We are

We will now design sequence of instructions. Figure 14.1, where t is the number of words stored on tape 1. Like words: tape positions 0 to $t-1$, so on. Our memory allocation is such that it is never freed of memory on tape 1.

The program counter
gram control is sequential
as the value of one of its

A machine instruction consists of an operation code, which indicates the operation to be performed, followed by operands. An instruction may move data, perform arithmetic or Boolean calculations, adjust the program flow, or allocate additional memory. Memory allocation may be required for temporary calculations or to dynamically increase the amount of memory available during a computation. We will assume that there is a maximum amount of memory, say, m_a words, that can be allocated by the execution of a single instruction. The number m_a , of course, can be as large as we wish.

The operands designate locations from which to retrieve data, locations in which to store the results, or other addresses to be used in the operation. An instruction usually has one or two operands, but we will allow every instruction to have up to a fixed number t of operands. Since t is the maximal number of addresses that can be explicitly given in an instruction, we assume that the result of a single instruction can change at most t words in the memory. Our final restriction, if it can be called a restriction, is that the instruction set must be finite.

Summarizing these conditions, we will be considering the time complexity of a computer whose architecture and instruction set satisfy the following conditions:

Component	Conditions
Memory:	Finite
Word size:	Fixed word length, each word containing m_w bits
Instruction set:	Finite
Instruction:	Operation code and at most t operands, fits within a single word
Operation:	Changes at most t words, allocates at most m_a words of memory

It should be clear that most, if not all, standard computer architectures and instruction sets satisfy these rudimentary limitations. The details of how memory is accessed, an instruction is performed, and program flow is maintained in a particular computer architecture are not of interest to us. We are only concerned with the number of instructions that are executed.

We will now design a Turing machine to simulate a computation consisting of a sequence of instructions. We will use the $4 + t$ -tape Turing machine model depicted in Figure 14.1, where t is the number of operands in an instruction. The program and input are stored on tape 1. Like the computer memory, we will consider tape 1 to be divided into words: tape positions 0 to $m_w - 1$ constitute word 0, m_w to $2m_w - 1$ constitute word 1, and so on. Our memory allocation scheme is simple: Memory is allocated sequentially and once allocated it is never freed. The memory counter contains the address of the next free word of memory on tape 1.

The program counter contains the location of the next instruction to be executed. Program control is sequential unless an instruction specifies the location of the next instruction as the value of one of its operands. The input counter contains two addresses, the location

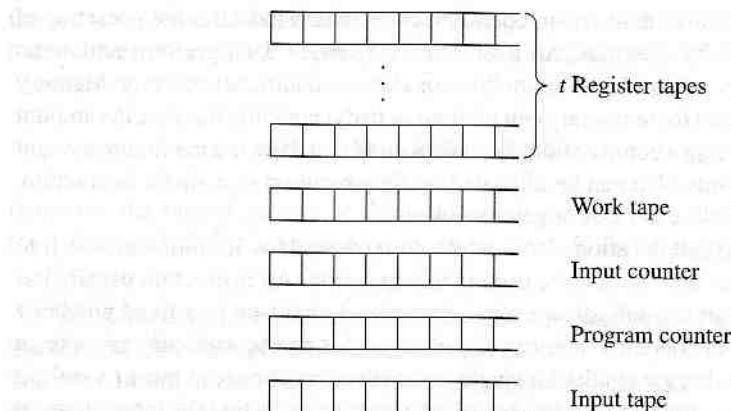


FIGURE 14.1 Turing machine architecture for computer simulation.

of the beginning of the input and the location of the next word to be read. There is an additional counter tape used in locating addresses on tape 1. Finally, there are t work tapes, one associated with each operand of an instruction. These tapes may be considered to be the Turing machine equivalent of registers; operations on data are performed only when the pertinent data have been moved to these tapes. Figure 14.1 shows the configuration of our Turing machine.

We now want to produce an upper bound on the number of transitions that are required for the Turing machine to simulate the execution of the k th instruction of a computation. An instruction may fetch data, store data, allocate memory, and perform a calculation. The simulation of an instruction by our Turing machine consists of the following actions:

- i) loading the data specified by operand i onto its associated tape (for each operand required by the operation),
- ii) performing the indicated operation, and
- iii) storing the result in the position indicated by operand i (for each operand required by the operation).

In the first step, the data to be processed may be in the instruction itself or the instruction may contain the address of the desired data.

To obtain an upper bound on the number of transitions needed to simulate the execution of an instruction, we will unrealistically assume that each instruction does the maximal amount of each type of action. That is, we will calculate the number of transitions as if each instruction fetches t words, performs an operation, stores t words, and allocates m_a words of memory. Thus we need to determine the number of transitions required for each of these actions.

Since there are only operands with the data in number of transitions needed, t_o , depends solely on the number of instructions in

The number of transitions depends upon the amount of memory, the number of bits used to store m_k the total memory “allocated” of instruction k . Thus

is the maximum amount of the k th transition.

During the simulation in m_k transitions. To find the tape. While the address is on the counter tape is decremented. The program tape head is read. There are fewer than m_w transitions since the bits in the last word

An upper bound on the k th instruction is

Since the operation may require m_{k+1} tape squares. Additionally, an instruction produces a

$$\begin{aligned}
 & (2t + 1)m_k \\
 & = (2t + 1)(m_{k+1}) \\
 & = (4t + 1)m_p
 \end{aligned}$$

Since there are only a finite number of instructions and each instruction uses at most t operands with the data in known locations on the register tapes, we can find the maximum number of transitions needed to perform any operation. This number, which we will call t_o , depends solely on the instruction set and is independent of the input, the data, and the number of instructions in a computation.

The number of transitions needed to load the operands and store the results depends upon the amount of memory that is being used by the Turing machine. We let m_p be the number of bits used to store the instructions, m_i be the number of bits to store the input, and m_k the total memory "allocated" by the Turing machine at the beginning of the simulation of instruction k . Thus

$$m_k = m_p + m_i + k \cdot m_a$$

is the maximum amount of memory allocated by the Turing machine prior to the simulation of the k th transition.

During the simulation of the k th transition, the Turing machine can locate any address in m_k transitions. To find the beginning of a word, the address is loaded onto the counter tape. While the address is not 0, the program tape head moves m_w squares to the right and the counter tape is decremented. This process halts when the counter tape is 0, in which case the program tape head is reading the first bit in the desired word. Copying the address requires fewer than m_w transitions. Finding the address requires fewer than $m_k - m_w$ transitions since the bits in the last word will not be read in this process.

An upper bound on the number of transitions required to simulate the execution of the k th instruction is

Action	Transitions
Find the instruction	m_k
Load the operands	$t \cdot m_k$
Return the register tape heads	$t \cdot m_k$
Perform the operation	t_o
Store the information	$t \cdot m_{k+1}$
Return the register tape heads	$t \cdot m_{k+1}$

Since the operation may allocate additional memory, the storing operation may access m_{k+1} tape squares. Adding the transitions associated with each step in the simulation of an instruction produces an upper bound of

$$\begin{aligned} & (2t + 1)m_k + 2tm_{k+1} + t_o \\ &= (2t + 1)(m_p + m_i + k \cdot m_a) + 2t(m_p + m_i + k \cdot m_a + m_o) + t_o \\ &= (4t + 1)m_p + (4t + 1)m_i + 2t \cdot m_a + t_o + (4t + 1)k \cdot m_a \end{aligned}$$

transitions to simulate the k th instruction. The values m_p , m_a , and t_o are constants independent of the input. If a computation of the computer with input length $m_i = n$ requires $f(n)$ steps, the simulation on our Turing machine requires

$$\begin{aligned} & \sum_{k=1}^{f(n)} ((4t+1)m_p + (4t+1)n + 2t \cdot m_a + t_o + (4t+1)k \cdot m_a) \\ &= f(n)((4t+1)m_p + (4t+1)n + 2t \cdot m_a + t_o) + \sum_{k=1}^{f(n)} (4t+1)k \cdot m_a \\ &= f(n)((4t+1)m_p + (4t+1)n + 2t \cdot m_a + t_o) + (4t+1)m_a \sum_{k=1}^{f(n)} k. \end{aligned}$$

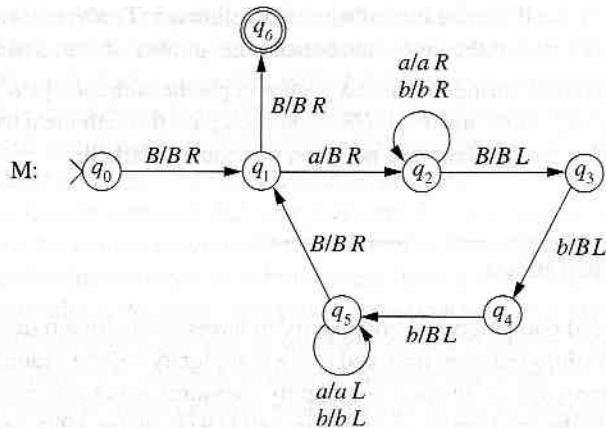
Thus the rate of growth is the larger of $O(nf(n))$ or $O(f(n)^2)$. The transition from computer to Turing machine simulation increases the order of the time complexity at most polynomially. In particular, any algorithm that runs in polynomial time on a computer can be simulated on a Turing machine in polynomial time.

Exercises

- For each of the functions below, choose the “best” big Oh from Table 14.3 that describes the rate of growth of the function.
 - $6n^2 + 500$
 - $2n^2 + n^2 \log_2(n)$
 - $\lfloor (n^3 + 2n)(n + 5)/n^2 \rfloor$
 - $n^2 \cdot 2^n + n!$
 - $25 \cdot n \cdot \sqrt{n} + 5n^2 + 23$
- Let f be a polynomial of degree r . Prove that f and n^r have the same rate of growth.
- Use Definition 14.2.1 or the limit rule to establish the following relationships.
 - $n \cdot \sqrt{n} \in O(n^2)$
 - $\log_2(n) \log_2(n) \in O(n)$
 - $n^r \in O(2^n)$
 - $2^n \notin O(n^r)$
 - $2^n \in O(n!)$
 - $n! \notin O(2^n)$
- Is $3^n \in O(2^n)$? Prove your answer.

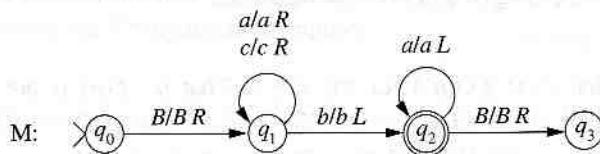
- Let a be a number such that $\log_a(n+c) \in O(1)$.
- Let $f(n) = n^{\frac{1}{n}}$.
 - Show that f is a polynomial function.
 - Show that f is decreasing.
- Let f and g be polynomials. Define the polynomial “ b ” by $b(n) = f(n)g(n)$. Prove your answer.
 - $f + g$
 - fg
 - f^2
 - $f \circ g$
- Determine the following.
 - Example 8.1
 - Example 8.2
 - Example 9.1
 - Example 9.2
- Let M be the function defined by $M(n) = \max\{m_i : i \in \{1, 2, \dots, n\}\}$.
 - Trace the computation of $M(10)$.
 - Describe the maximum number of transitions required to compute $M(n)$.
 - Give the function $M(n)$ in terms of big Oh notation.

- its independence requires $f(n)$.
5. Let a be a natural number greater than 1 and c be a constant greater than 0. Is $\log_a(n+c) \in O(\log_a(n))$? Prove your answer.
 6. Let $f(n) = n^{\log_2(n)}$.
 - a) Show that $f(n) \notin O(n^r)$ for any $r > 0$. That is, $f(n)$ is not bounded by a polynomial.
 - b) Show that $2^n \notin O(f(n))$. That is, $f(n)$ is not exponential.
 7. Let f and g be two unary functions such that $f \in \Theta(n^r)$ and $g \in \Theta(n^t)$. Give the polynomial “big theta” that has the same rate of growth as the following functions. Prove your answer.
 - a) $f + g$
 - b) fg
 - c) f^2
 - d) $f \circ g$
 8. Determine the time complexity of the following Turing machines.
 - a) Example 8.2.1, page 260
 - b) Example 8.6.3, page 274
 - c) Example 9.1.2, page 298
 - d) Example 9.2.1, page 301
 9. Let M be the Turing machine



- a) Trace the computation of M with input λ , a , and abb .
- b) Describe the string of length n for which the computation of M requires the maximum number of transitions.
- c) Give the function tc_M .

10. Let M be the Turing machine



- a) Trace the computation of M with input abc , aab , and cab .
 - b) Describe the string of length n for which the computation of M requires the maximum number of transitions.
 - c) Give a regular expression for $L(M)$.
 - d) Give the function tc_M .
11. Let L be the language over $\{a, b\}$ that contains a string u if it satisfies one of the following conditions:
- i) $u = a^i b^i$ and $\text{length}(u) \leq 100$, or
 - ii) $\text{length}(u) > 100$.
- a) Design a standard Turing machine M that accepts L .
 - b) Give the function tc_M .
 - c) What is the best polynomial rate of growth that describes the time complexity function tc_M ?
12. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a two-tape Turing machine that accepts a language L , and let N be the machine constructed following Theorem 14.4.2 with $m = 12$. Determine the size of the tape alphabet and the number of states of N .
- * 13. Design a standard Turing machine M that accepts the language $\{a^i b^i \mid i \geq 0\}$ with time complexity $tc_M \in O(n \log_2(n))$. Hint: On each pass through the data, mark half of the a 's and half of the b 's that have not been previously marked.

Bibliographic Notes

We have presented computational complexity in terms of the time required by a computation. The relationships between time and space complexity will be examined in Chapter 17. An axiomatic approach to abstract complexity measures was introduced in Blum [1967] and developed further by Hartmanis and Hopcroft [1971]. In the 1985 Association of Computing Machinery Turing Award Lecture, Richard Karp [1986] gave an interesting personal history of the initial development and directions of complexity theory.

Computability theory classifies problems into those that are computable and those that are not. A problem is computable if there is an algorithm that can solve it. A problem is decidable if there is a Turing machine that can solve it. There are many problems that are both computable and decidable. There are also many problems that are computable but not decidable. These are called semidecidable or recursively enumerable problems. There are also many problems that are neither computable nor decidable. These are called undecidable or noncomputable problems.

There are many problems that are both computable and decidable. We explore the relationship between these two classes of problems. We also look at polynomial-time algorithms and how they relate to other classes of problems. We also look at nondeterministic algorithms and how they relate to other classes of problems. Currently, the outstanding open problem in this area is whether P equals NP.

The duality between time and space complexity is a fundamental concept in computer science. Define complexity classes based on the length of an input string. The complexity of the instances of a problem is determined by the effect of the representation on the complexity. Some simple constructions can affect the complexity polynomially, while others can affect it exponentially.

CHAPTER 15

\mathcal{P} , \mathcal{NP} , and Cook's Theorem

es the maxi-

s one of the

e complexity

ts a language
with $m = 12$.

≥ 0 } with time
ark half of the

by a computa-
in Chapter 17.
n Blum [1967]
iation of Com-
esting personal

Computability theory is concerned with establishing whether decision problems are theoretically decidable. In complexity theory we further subdivide the solvable problems into those that have practical solutions and those that are solvable in principle only. A problem that is theoretically solvable may not have a practical solution; there may be no algorithm that solves the problem without requiring an extraordinary amount of time or memory. Problems for which there is no efficient algorithm are said to be *intractable*. Because of the rate of growth of the time complexity, nonpolynomial algorithms are not considered feasible for all but the simplest cases of the problem. The division of the class of solvable decision problems into polynomial and nonpolynomial problems is generally considered to distinguish the efficiently solvable problems from the intractable problems.

There are many famous problems that have polynomial-time nondeterministic solutions for which there are no known polynomial-time deterministic solutions. In this chapter we explore the relationship between solvability using deterministic and nondeterministic polynomial-time algorithms. Whether every problem that can be solved in polynomial time by a nondeterministic algorithm can also be solved deterministically in polynomial time is currently the outstanding open question of theoretical computer science.

The duality between solvable decision problems and recursive languages allows us to define complexity classes in terms of recursive languages. Because time complexity relates the length of an input string to the number of transitions, the selection of the representation of the instances of a decision problem may alter the complexity of the algorithm. To separate the effect of the representation from the inherent difficulty of the problem, we will impose some simple constraints on the representations so that a change in representation only polynomially affects the complexity of the solution.

15.1 Time Complexity of Nondeterministic Turing Machines

Nondeterministic computations are fundamentally different from their deterministic counterparts. A deterministic machine often generates and examines multiple possibilities in its search for a solution, while a nondeterministic machine employing a guess-and-check strategy need only determine if one of the possibilities provides the solution. Consider the problem of deciding whether a natural number k is a composite (not a prime). A constructive, deterministic solution can be obtained by sequentially examining every number in the interval from 2 to $\lfloor \sqrt{k} \rfloor$ to see if it is a factor of k . If a factor is discovered, then k is a composite. A nondeterministic computation begins by arbitrarily choosing a value in the designated range. A single division determines if the guess is a factor. If k is a composite, one of the nondeterministic choices will produce a factor and that computation returns the affirmative response.

A string is accepted by a nondeterministic machine if at least one computation terminates in an accepting state. The acceptance of the string is unaffected by the existence of other computations that halt in nonaccepting states or do not halt at all. The worst-case performance of the algorithm, however, measures the efficiency over all computations.

Definition 15.1.1

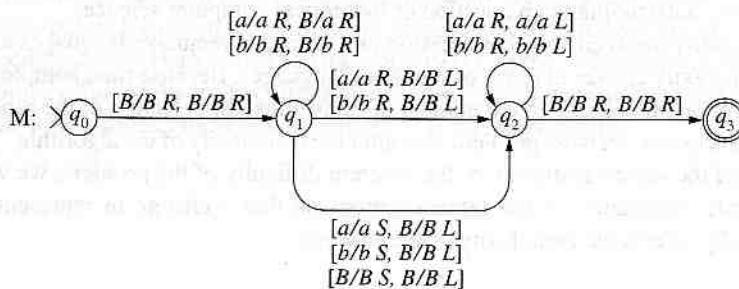
Let M be a nondeterministic Turing machine. The time complexity of M is the function $tc_M : N \rightarrow N$ such that $tc_M(n)$ is the maximum number of transitions processed by a computation, employing any choice of transitions, of an input string of length n .

The preceding definition is identical to that of the time complexity of a deterministic machine. It is included to emphasize that the nondeterministic analysis must consider all possible computations for an input string. As in the case of deterministic machines, our definition of time complexity assumes that every computation of M terminates.

Nondeterministic computations utilizing a guess-and-check strategy are generally simpler than their deterministic counterparts. The simplicity reduces the number of transitions required for a single computation. Employing this strategy, we can construct a nondeterministic machine to accept the palindromes over $\{a, b\}$.

Example 15.1.1

The two-tape nondeterministic machine M



accepts the palindromes copied on tape 2. The transition from q_1 that moves right for an odd-length palindrome location is checking that a move occurs in an accepting state between tape heads 1 and 2.

reflects the addition of the symbol pair.

The strategy ensures an equivalent deterministic solvability. It does, however, require a nondeterministic machine.

Theorem 15.1.2

Let L be the language of a nondeterministic machine M . Then the time complexity $tc_M(n) = O(n^2)$, where n is the time complexity $tc_{M'}(n)$ for any state, symbol pair of M .

Proof. Let $M = (Q, \Sigma, \Delta, \delta, q_0, F)$. Suppose M halts for all inputs, a symbol pair of M . Then δ is a mapping associating a unique transition with each symbol pair. The value m_i indicates the number of transitions in the i th step of the computation.

In Section 8.7, a nondeterministic computation with input w iteratively simulates the computation of M . For an input of length n , the computation of M is at most $f(n)$. Thus,

1. generates a sequence of transitions
2. simulates the computation of M
3. if the computation of M terminates in step 1.

In the worst case, the computation of M can take at most $f(n)$ steps. The time complexity of M' is $O(n^2)$.

The time complexity of a particular construction depends on the particular construction.

accepts the palindromes over $\{a, b\}$. Both tape heads move to the right with the input being copied on tape 2. The transition from state q_1 "guesses" the center of the string. A transition from q_1 that moves the tape head on tape 1 to the right and tape 2 to the left is checking for an odd-length palindrome, while a transition that leaves the head on tape 1 in the same location is checking for an even-length palindrome. The maximum number of transitions occurs in an accepting computation, which halts when a blank is simultaneously read by tape heads 1 and 2. The time complexity

$$tc_M(n) = \begin{cases} n+2 & \text{if } n \text{ is odd} \\ n+3 & \text{if } n \text{ is even} \end{cases}$$

reflects the additional transition required for the acceptance of an even-length string. \square

The strategy employed in the transformation of a nondeterministic machine to an equivalent deterministic machine given in Section 8.7 does not preserve polynomial time solvability. It does, however, provide an upper bound on the time complexity needed by a deterministic machine to accept the language of the original nondeterministic machine.

Theorem 15.1.2

Let L be the language accepted by a one-tape nondeterministic Turing machine M with time complexity $tc_M(n) = f(n)$. Then L is accepted by a deterministic Turing machine M' with time complexity $tc_{M'}(n) \in O(f(n)c^{f(n)})$, where c is the maximum number of transitions for any state, symbol pair of M .

Proof. Let $M = (Q, \Sigma, \Gamma, \delta, q_0)$ be a one-tape nondeterministic Turing machine that halts for all inputs, and let c be the maximum number of distinct transitions for any state, symbol pair of M . The transformation from nondeterminism to determinism is obtained by associating a unique computation of M with a sequence (m_1, \dots, m_n) , where $1 \leq m_i \leq c$. The value m_i indicates which of the c possible transitions of M should be executed on the i th step of the computation.

In Section 8.7, a three-tape deterministic machine M' was described whose computation with input w iteratively simulated all possible computations of M with input w . We will analyze the number of transitions required by the machine M' to simulate all computations of M . For an input of length n , the maximum number of transitions of any computation of M is at most $f(n)$. To simulate a single computation of M , machine M'

1. generates a sequence of integers (m_1, \dots, m_n) with $1 \leq m_i \leq c$;
2. simulates the computation of M specified by the sequence (m_1, \dots, m_n) ; and
3. if the computation does not accept the string, the computation of M' continues with step 1.

In the worst case, $c^{f(n)}$ sequences need to be examined. The simulation of a single computation of M can be performed using $O(f(n))$ transitions of M' . Thus, the time complexity of M' is $O(f(n)c^{f(n)})$. \blacksquare

The time complexity $O(f(n)c^{f(n)})$ produced in Theorem 15.1.2 is an artifact of the particular construction used to produce M' from M . Other approaches considering the

properties of the particular language in question may be used to design deterministic machines with time complexity significantly lower than the upper bound indicated by Theorem 15.1.2. For example, the nondeterministic machine in Example 8.7.1 that accepts $(a \cup b \cup c)^*(abc \cup cab)(a \cup b \cup c)^*$ uses at most $n + 3$ transitions when processing an input string of length n . The construction used in Theorem 15.1.2 produces a deterministic machine that accepts the language with time complexity $\Theta(n \cdot 3^n)$. However, this language is also accepted by a standard Turing machine with time complexity $n + 1$.

In the next several sections we will explore the relationship between the class of problems that can be solved deterministically in polynomial time and the class of problems that can be solved nondeterministically in polynomial time.

15.2 The Classes \mathcal{P} and \mathcal{NP}

A language L over Σ is **decidable in polynomial time**, or simply **polynomial**, if there is an algorithm that determines membership in L for which the growth in the time required by a computation increases at most polynomially with the length of the input string. The notion of polynomial time decidability is formally defined using transitions of the standard Turing machine to measure the time of a computation.

Definition 15.2.1

A language L is **decidable in polynomial time** if there is a standard Turing machine M that accepts L with $t_{CM} \in O(n^r)$, where r is a natural number independent of n . The family of languages decidable in polynomial time is denoted \mathcal{P} .

The class \mathcal{P} is defined in terms of the time complexity of an implementation of an algorithm on a standard Turing machine. We could just as easily have chosen a multitrack, multitape, or two-way deterministic machine as the computational model on which algorithms are evaluated. The class \mathcal{P} of polynomially decidable languages or solvable decision problems is invariant under the choice of the deterministic Turing machine model chosen for the analysis. In Section 14.4 it was shown that a language accepted by a multitrack machine in time $O(n^r)$ is also accepted by a standard Turing machine in time $O(n^r)$. The transition from multitape to standard machine also preserves polynomial solutions. A language accepted in time $O(n^r)$ by a multitape machine is accepted in $O(n^{2r})$ time by a standard machine.

The relationship between the complexity of running a program on a computer and its simulation on a Turing machine was analyzed in Section 14.7. The number of transitions in the Turing machine simulation increases only polynomially with the number of instructions executed by the computer. A consequence of this is that any problem that we would consider polynomially solvable on a standard computer is in \mathcal{P} . The robustness of the class \mathcal{P} under changes of machines and architectures provides support for its selection as defining the border between tractable and intractable problems.

The computation ines one of the po a single potential duces the comple completely analog accepted by nond

Definition 15.2.2

A language L is **s nondeterministic T number independe time is denoted \mathcal{NP}**

The family \mathcal{NP} number of transitions deterministic machine inclusion is the top

15.3 Problems

The development of the representation of that analyzes the sole concern was by a computation relates the length of the representation a computation.

In Chapter 11 we whether a natural number of the natural numbers

$M_1: \times q_0$

$M_2: \times q_0$

nistic
d by
cepts
ing an
nistic
guage

ss of
blems

e is an
d by a
notion
Turing

M that
nily of

f an al-
titrack,
h algo-
ecision
sen for
achine
transi-
nguage
standard

and its
tions in
uctions
consider
 \mathbb{P} under
ing the

The computation of a nondeterministic machine that solves a decision problem examines one of the possible solutions to the problem. The ability to nondeterministically select a single potential solution, rather than systematically examining all possible solutions, reduces the complexity of the computation of the nondeterministic machine. In a manner completely analogous to the definition of the class \mathcal{P} , we can define the family of languages accepted by nondeterministic Turing machines in polynomial time.

Definition 15.2.2

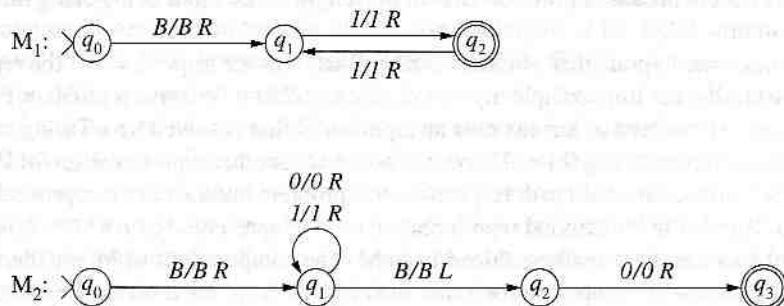
A language L is said to be accepted in **nondeterministic polynomial time** if there is a nondeterministic Turing machine M that accepts L with $tc_M \in O(n^r)$, where r is a natural number independent of n . The family of languages accepted in nondeterministic polynomial time is denoted \mathcal{NP} .

The family \mathcal{NP} is a subset of the recursive languages; the polynomial bound on the number of transitions ensures that all computations of M eventually terminate. Since every deterministic machine is also a nondeterministic machine, $\mathcal{P} \subseteq \mathcal{NP}$. The status of the reverse inclusion is the topic of the remainder of this chapter.

15.3 Problem Representation and Complexity

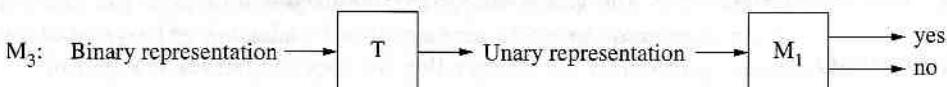
The development of a Turing machine solution to a decision problem consists of two steps: the representation of the problem instances as strings, followed by the design of the machine that analyzes the resulting strings and solves the problem. In the study of decidability, the sole concern was the discovery of an algorithm to solve a problem and the resources required by a computation were not considered. Since the time complexity of a Turing machine relates the length of the input to the number of transitions in the computations, the selection of the representation may have important consequences for the amount of work required by a computation.

In Chapter 11 we designed two simple Turing machines to solve the problem of deciding whether a natural number is even. The input to machine M_1 uses the unary representation of the natural numbers and M_2 the binary representation:



The time complexities of both of these machines is linear and the difference in representation does not significantly affect the complexity. Unfortunately, this is not always the case. A modification to the machine M_1 will have a considerable impact on the complexity.

A Turing machine T can be built to transform a natural number represented in binary to its unary representation (Exercise 6). The sequential operation of T with M_1 produces



which is another solution to the even number problem. Let us examine the complexity of this solution. The following table shows the increase in string length that results from the conversion of a binary to a unary representation. The second column gives the maximal binary number for the string length given in column one, and the final column has the corresponding unary representation.

String Length	Maximal Binary Number	Decimal Value	Unary Representation
1	1	1	$11 = 1^2$
2	11	3	$1111 = 1^4$
3	111	7	$11111111 = 1^8$
i	I^i	$2^i - 1$	I^{2^i}

The time complexity of M_3 is determined by the complexities of T and M_1 . For an input of length i , the string I^i requires the maximum number of transitions of M_3 . The time complexity of M_3 is

$$\begin{aligned} tc_{M_3}(n) &= tc_T(n) + tc_{M_1}(2^n) \\ &= tc_T(n) + 2(2^n) + 2, \end{aligned}$$

which is exponential even without adding the work required for the transformation. The strategy employed by M_1 for answering the problem is unchanged; the increase in time complexity occurs because of the decrease in the length of the input string using the binary representation.

The following hypothetical situation further illustrates the importance of the representation in assessing the time complexity of a decision problem. Imagine a problem P whose instances are represented by strings over an alphabet Σ that is solved by a Turing machine M with time complexity $tc_M(n) = 2^n$. We can construct another representation for P as follows: a new symbol $\#$ is added to the alphabet and a problem instance that is represented by a string w of length n in the original representation is now represented by $w\#\^{2^n-n}$. A machine M' that solves P can be trivially obtained from M . The computations of M' are identical to those of M , except M' treats $\#$ in the same manner that M treats a blank. Because of the increase in the length of the input string, $tc_{M'}(n) = n$.

The preceding section to artificially strings can be in responding decre

The dependence not every representation with complexity. Such introduce the not describe condition

A representation from problem instances of p_i . Let rep_1 and Representation rep_2 over Σ_2^* such that

- i) $t(rep_1(p_i)) = n$
- ii) if $u \in \Sigma_1^*$ is not a representation of a problem instance
- iii) t is computable

If rep_1 is transformed cannot increase many of symbols that can transitions of T .

Now, assume representation rep_2

produces a polynomial length of representation for the problem. Most length from the small representation of T exponentially from analysis the natural notation \bar{i} will be i .

Following the using the unary representation binary representation this property is some representation appears of the representation

tion e. A
nary es
yes no
ty of n the
timal s the
input e time

The preceding example provides a method for manipulating the time complexity function to artificially make inefficient algorithms appear efficient. If the length of the input strings can be increased without changing the underlying computation, there will be a corresponding decrease in the time complexity function.

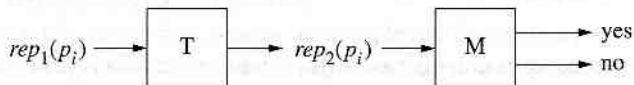
The dependence of the time complexity on the size of the representation shows that not every representation should be acceptable for complexity analysis. Using the smallest representation would avoid the possibility of the length of the representation affecting the complexity. Such a requirement, however, would be both too restrictive and unnecessary. We introduce the notion of a polynomial time transformation of representations to informally describe conditions for the suitability of a representation for complexity analysis.

A representation of a decision problem P with instances p_0, p_1, p_2, \dots is a mapping rep from problem instances to strings over an alphabet Σ , where $rep(p_i)$ is the representation of p_i . Let rep_1 and rep_2 be two representations of P over alphabets Σ_1 and Σ_2 , respectively. Representation rep_1 is polynomial-time transformable to rep_2 if there is a function $t : \Sigma_1^* \rightarrow \Sigma_2^*$ such that

- i) $t(rep_1(p_i)) = rep_2(p_i)$ for all i ;
- ii) if $u \in \Sigma_1^*$ is not the representation of a problem instance, then $t(u)$ is not the representation of a problem instance in Σ_2^* ; and
- iii) t is computable in polynomial time by a standard Turing machine T .

If rep_1 is transformable to rep_2 in polynomial time, the length of the string $t(rep_1(p_i))$ cannot increase more than polynomially with respect to the length of $rep_1(p_i)$; the number of symbols that can be added to the representation is necessarily less than the number of transitions of T .

Now, assume that P is solvable in polynomial time by a Turing machine M using representation rep_2 . The serial combination of T and M



produces a polynomial-time solution using representation rep_1 . Thus differences in the length of representations that differ only polynomially do not affect the tractability of the problem. Most reasonable representations of a problem differ only polynomially in length from the smallest representation. An obvious exception to this is the use of the unary representation of natural numbers, in which case the length of the input strings increases exponentially from their length in binary representation. For this reason, in complexity analysis the natural numbers will always be represented in binary. From this point on, the notation \bar{i} will be used to denote the binary representation of the number i .

Following the guidelines described, a decision problem that has a polynomial solution using the unary representation of natural numbers but no polynomial solution using the binary representation is not considered to be solvable in polynomial time. A problem with this property is sometimes called *pseudo-polynomial* because the solution with the unary representation appears to be a polynomial-time solution to someone not aware of the impact of the representation in the analysis of decision problem complexity.

15.4 Decision Problems and Complexity Classes

In this section we list several decision problems from \mathcal{P} and \mathcal{NP} . We will not describe details of algorithms that solve these problems since solutions have previously been presented or will be examined in detail in the next several chapters. The objective of this listing is to provide examples of familiar problems in each class in an attempt to identify properties shared by algorithms that solve the problems within a class.

Acceptance of Palindromes

Input: String u over alphabet Σ

Output: yes; u is a palindrome
no; otherwise.

Complexity: in \mathcal{P} —yes

Path Problem for Directed Graphs

Input: Graph $G = (N, A)$, nodes $v_i, v_j \in N$

Output: yes; if there is a path from v_i to v_j in G
no; otherwise.

Complexity: in \mathcal{P} —yes

Derivability in Chomsky Normal Form Grammar

Input: Chomsky normal form grammar G , string w

Output: yes; if there is a derivation $S \xrightarrow{*} w$
no; otherwise.

Complexity: in \mathcal{P} —yes

Each of the preceding problems has polynomial-time solutions. The palindromes are accepted by a standard Turing machine with time complexity $O(n^2)$ as demonstrated in Section 14.3. Dijkstra's algorithm can be used to discover if there is a path between two nodes in a directed graph in time $O(n^2)$, where n is the number of nodes in the graph. The CYK algorithm in Section 4.6 determines membership in a language defined by a Chomsky normal form grammar using $O(n^3)$ steps to complete the dynamic programming table.

Satisfiability

Input: Boolean formula u in conjunctive normal form

Output: yes; there is a truth assignment that satisfies u
no; otherwise.

Complexity: in \mathcal{P} —unknown
in \mathcal{NP} —yes

Each of these check strategy. The verification of a formula can be a guess for the Hamiltonian verification check. Subset-Sum Problem the subset.

For problems into the nature of demonstrated in the solution to the H

We add one p

The problem con that can contain strings that have

After introduc requires space an

Hamiltonian Circuit Problem**Input:** Directed graph $G = (N, A)$ **Output:** yes; if there is a simple cycle that visits all vertices of G exactly once
no; otherwise.**Complexity:** in \mathcal{P} —unknown
in \mathcal{NP} —yes**Subset Sum Problem****Input:** Set S , value function $v : S \rightarrow \mathbb{N}$, number k **Output:** yes; if there is a subset S' of S whose total value is k
no; otherwise.**Complexity:** in \mathcal{P} —unknown
in \mathcal{NP} —yes

Each of these problems can easily be solved nondeterministically using a guess-and-check strategy. The guess for the Satisfiability Problem is a single truth assignment. The verification of whether a particular truth assignment satisfies a conjunctive normal form formula can be accomplished in time polynomially related to the length of the formula. The guess for the Hamiltonian Circuit Problem produces a sequence of $n + 1$ vertices and the verification checks if the sequence defines a tour of the graph. Similarly, a guess for the Subset-Sum Problem is a subset and the check simply adds the values of the items in the subset.

For problems not known to be in \mathcal{P} , deterministic solutions often do not provide insight into the nature of the problem but rather have the flavor of an exhaustive search. This will be demonstrated in the next section where we present both a deterministic and nondeterministic solution to the Hamiltonian Circuit Problem.

We add one problem that is outside of the complexity classes that have been introduced. The problem considers the determination of the language described by a regular expression that can contain u^2 as an abbreviation of uu . For example, $(a^2)^*b(a \cup b)^*$ represents all strings that have an even number of a 's occurring before the first b .

Regular Expressions with Squaring**Input:** Regular expression α over an alphabet Σ **Output:** yes; if $\alpha \neq \Sigma^*$
no; otherwise.**Complexity:** in \mathcal{P} —no
in \mathcal{NP} —no

After introducing space complexity, we will show that any solution to this problem requires space and time that grows exponentially with the length of the regular expression.

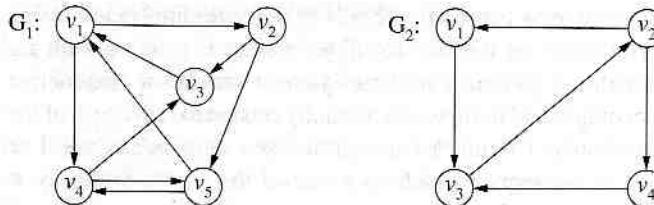
15.5 The Hamiltonian Circuit Problem

The Hamiltonian Circuit Problem is used to demonstrate the difference in both the strategy and the complexity of deterministic and nondeterministic solutions of decision problems. We begin by presenting a more detailed description of the problem than given in the preceding section.

Let G be a directed graph with n vertices numbered 1 to n . A Hamiltonian circuit is a path i_0, i_1, \dots, i_n in G that satisfies

- i) $i_0 = i_n$
- ii) $i_i \neq i_j$ whenever $i \neq j$ and $0 \leq i, j < n$.

That is, a Hamiltonian circuit is a path that visits every vertex exactly once and terminates at its starting point. A Hamiltonian circuit is frequently called a *tour*. For example, the graph G_1 has a tour $v_1, v_2, v_5, v_4, v_3, v_1$, and G_2 does not have a tour.



The Hamiltonian Circuit Problem is to determine whether a directed graph has a tour. Since each vertex is contained in a tour, we may assume that every tour begins and ends at vertex 1.

The deterministic solution in Example 15.5.1 performs an exhaustive search of sequences of vertices to determine if one is a tour. The sequences are systematically generated and tested until a tour is found or until all possibilities have been examined. The nondeterministic solution is obtained by eliminating the generate portion of the generate-and-test cycle. A nondeterministic guess produces a sequence of vertices, which is subsequently checked using the same procedure employed in the deterministic computation.

Example 15.5.1

We will describe the actions of a four-tape deterministic Turing machine that solves the Hamiltonian Circuit Problem. The first step is to design a representation for a directed graph with vertices numbered 1 to n . The alphabet of the representation is $\{0, 1, \#\}$ and a vertex of the graph is denoted by its binary representation. A graph with n vertices and m arcs is represented by the input string

$$\bar{x}_1 \# \bar{y}_1 \# \dots \# \# \bar{x}_m \# \bar{y}_m \# \# \# \bar{n},$$

where $[x_i, y_i]$ are the arcs of the graph and \bar{x} denotes the binary representation of the number x .

Throughout the computation generated determine if they The representation the halting condition to generate the sequence

A computation

1. generates a sequence
2. halts if tapes 2 and 3 are full
3. examines the sequence

If the computation contains a Hamiltonian circuit

The analysis is

Sequentially, the vertex tape 3 if

- i) $i_j \neq 1$
- ii) $i_j \neq i_k$ for $1 \leq k < j$
- iii) there is an arc from i_j to i_k

That is, i_j is added to the sequence, $j = i, \dots, n - 1$, i_{n-1} to 1, the path of the tour is found.

A computation of the input graph does not follow that sequences. Disregarding sequences grows exponentially. The representation is used by adding no arcs to the tour. Consequently, increasing the number of possible tours.

We have shown that the algorithm does not follow that sequences. The algorithm has been

Throughout the computation, tape 1 maintains the representation of the arcs. The computation generates and examines sequences of $n + 1$ vertices $1, i_1, \dots, i_{n-1}, 1$ to determine if they form a tour. The sequences are generated in numeric order on tape 2. The representation of the sequence $1, n, \dots, n, 1$ is written on tape 4 and used to trigger the halting condition. The techniques employed by the machine in Figure 8.1 can be used to generate the sequences on tape 2.

A computation is a loop that

1. generates a sequence $B\bar{1}B\bar{i}_1B\bar{i}_2B\dots B\bar{i}_{n-1}B\bar{1}B$ on tape 2,
2. halts if tapes 2 and 4 are identical, and
3. examines the sequence $1, i_1, \dots, i_{n-1}, 1$ and halts if it is a tour of the graph.

If the computation halts in step 2, all sequences have been examined and the graph does not contain a Hamiltonian circuit.

The analysis in step 3 begins with the machine configuration

Tape 4 $B\bar{1}(B\bar{n})^{n-1}B\bar{1}B$

Tape 3 $B\bar{1}B$

Tape 2 $B\bar{1}B\bar{i}_1B\dots B\bar{i}_{n-1}B\bar{1}B$

Tape 1 $B\bar{x}_1#\bar{y}_1##\dots##\bar{x}_m#\bar{y}_mB###\bar{n}B$.

Sequentially, the vertices i_1, \dots, i_{n-1} are examined. Vertex i_j is added to the sequence on tape 3 if

- i) $i_j \neq 1$;
- ii) $i_j \neq i_k$ for $1 \leq k \leq j - 1$; and
- iii) there is an arc $[i_{j-1}, i_j]$ represented on tape 1.

That is, i_j is added if $1, i_1, \dots, i_j$ is an acyclic path in the graph. If every vertex i_j , $j = i, \dots, n - 1$, in the sequence on tape 2 is added to tape 3 and there is an arc from i_{n-1} to 1, the path on tape 2 is a tour and the computation accepts the input.

A computation examines and rejects each sequence $1, i_1, i_2, \dots, i_{n-1}, 1$ when the input graph does not contain a tour. For a graph with n vertices, there are n^{n-1} such sequences. Disregarding the computations involved in checking a sequence, the number of sequences grows exponentially with the number of vertices of the graph. Since the binary representation is used to encode the vertices, increasing the number of vertices to $2n$ (but adding no arcs to the graph) increases the length of the input string by a single character. Consequently, incrementing the length of the input causes an exponential increase in the number of possible sequences that must be examined. \square

We have shown that the Hamiltonian Circuit Problem is solvable in exponential time. It does not follow that the problem cannot be solved in polynomial time. So far, no polynomial algorithm has been discovered. This may be because no such solution exists or maybe

we have just not been clever enough to find one! The likelihood and ramifications of the discovery of a polynomial-time solution are the topics of the remainder of the chapter.

Nondeterministic computations utilizing a guess-and-check strategy are generally simpler than their deterministic counterparts. The simplicity reduces the number of transitions required for a single computation. A nondeterministic machine employing this strategy is constructed that solves the Hamiltonian Circuit Problem in polynomial time.

Example 15.5.2

A three-tape nondeterministic machine that solves the Hamiltonian Circuit Problem in polynomial time is obtained by altering the deterministic machine from Example 15.5.1. The fourth tape, which is used to terminate the computation when the graph does not contain a tour, is not required in the nondeterministic machine. The computation

1. halts and rejects the input if the graph has fewer than $n + 1$ arcs,
2. nondeterministically generates a sequence $1, i_1, \dots, i_{n-1}, 1$ on tape 2, and
3. uses tapes 1 and 3 to determine whether the sequence on tape 2 defines a tour.

To show that the nondeterministic machine is polynomial, we construct an upper bound to the number of transitions in a computation. The maximum number of transitions occurs when the sequence of vertices defines a tour. Otherwise, the computation terminates examining fewer than $n + 1$ arcs on tape 2. Since the nodes are represented in binary, the maximum amount of tape needed to encode any node is $\lceil \log_2(n) \rceil + 1$.

The worst-case performance occurs for graphs with more than $n + 1$ arcs. The computations for graphs with fewer arcs halts in step 1 and avoids the transitions required by the check in step 3. Thus the length of the input for the worst-case performance of the algorithm depends upon the number of arcs in the graph. Let k be the number of arcs. We will show that the rate of growth of the number of transitions is polynomial in k . Since the length of the input cannot grow more slowly than k (each arc requires at least three tape positions), it follows that the time complexity is polynomial.

Rejecting the input in step 1 requires the computation to compare the number of arcs in the input with the number of nodes. This can be accomplished in time that grows polynomially with the number of arcs.

If the computation does not halt in step 1, we know that the number of arcs is greater than the number of nodes. Generating the guess on tape 2 and repositioning the tape head processes $O(n \log_2(n))$ transitions. Now assume that tape 3 contains the initial subsequence $B\bar{1}\#i_j\#\dots\#\bar{i}_{j-1}$ of the sequence on tape 2. The remainder of the computation consists of a loop that

1. moves tape heads 2 and 3 to the position of the first blank on tape 3 ($O(n \log_2(n))$ transitions),
2. checks if the encoded vertex on tape 2 is already on tape 3 ($O(n \log_2(n))$ transitions),

3. checks if repositioning i_j
4. writes i_j

A computation examination of $i_1, \dots, i_{n-1} 1$ computation

The rate determined by the in the arc list. of the entire se

15.6 Poly

A reduction of L to that of M decidability of their tractability by a machine M

- i) runs R on
- ii) runs M on

The string $r(w)$ complexity of the time required to Since we are e seems reasonable

Definition 15.6

Let L and Q be in polynomial such that $w \in L$

Polynomial times of the reduction. This property guarantees that the algorithm

Theorem 15.6.2

Let L be reducible to Q

3. checks if there is an arc from i_j to i_j ($O(k \log_2(n))$ transitions examining all arcs and repositioning the tape head), and
4. writes \bar{i}_j on tape 3 and repositions the tape heads ($O(n \log_2(n))$ transitions).

A computation consists of the generation of the sequence on tape 2 followed by examination of the sequence. The loop that checks the sequence is repeated for each vertex i_1, \dots, i_{n-1} on tape 2. The repetition of step 3 causes the number of transitions of the entire computation to grow at the rate $O(k^2 \log_2(k))$.

The rate of growth of the time complexity of the nondeterministic machine is determined by the portion of the computation that searches for the presence of a particular arc in the arc list. This differs from the deterministic machine in which the exhaustive search of the entire set of sequences of n vertices dominates the rate of growth. \square

15.6 Polynomial-Time Reduction

A reduction of a language L to a language Q transforms the question of membership in L to that of membership in Q . Reduction played an important role in establishing the decidability of languages and will play an equally important role in classifying problems by their tractability. Let r be a reduction of L to Q computed by a machine R . If Q is accepted by a machine M , then L is accepted by a machine that

- i) runs R on an input string $w \in \Sigma_1^*$, and
- ii) runs M on $r(w)$.

The string $r(w)$ is accepted by M if, and only if, $w \in L$. In complexity analysis, the time complexity of the composite solution to the question of membership in L includes both the time required to transform the instances of L and the time required by the solution to Q . Since we are equating efficiently solvable problems with polynomial time complexity, it seems reasonable to place the same conditions on the time complexity of a reduction.

Definition 15.6.1

Let L and Q be languages over alphabets Σ_1 and Σ_2 , respectively. We say that L is **reducible in polynomial time** to Q if there is a polynomial-time computable function $r : \Sigma_1^* \rightarrow \Sigma_2^*$ such that $w \in L$ if, and only if, $r(w) \in Q$.

Polynomial-time reductions are important because the bound on the number of transitions of the reduction limits the length of the string that is input to the subsequent machine. This property guarantees that the combination of a polynomial-time reduction and polynomial algorithm produces another polynomial algorithm.

Theorem 15.6.2

Let L be reducible to Q in polynomial time and let $Q \in \mathcal{P}$. Then $L \in \mathcal{P}$.

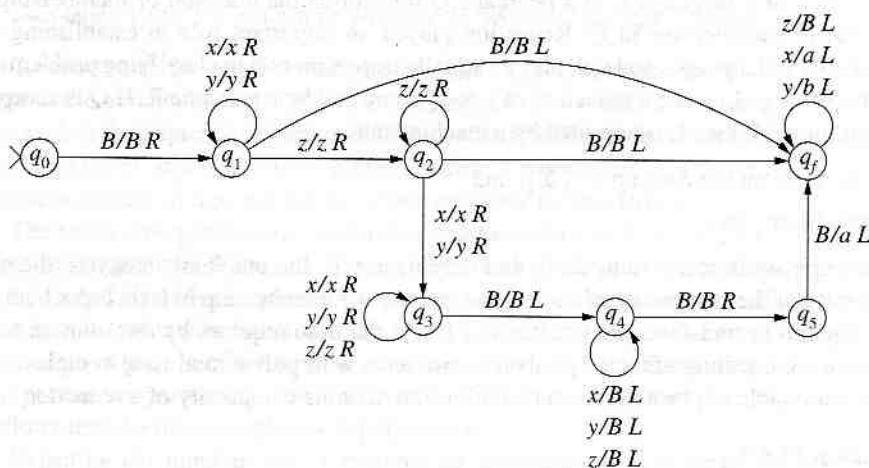
Proof. As before, we let R denote the machine that computes the reduction of L to Q and M the machine that decides Q. L is accepted by a machine that sequentially runs R and M. The time complexities tc_R and tc_M combine to produce an upper bound on the number of transitions of a computation of the composite machine. The computation of R with input string w generates the string $r(w) \in \Sigma_2^*$, which is the input to M. The function tc_R can be used to establish a bound on the length of $r(w)$. If the input string w to R has length n , then the length of $r(w)$ cannot exceed the maximum of n and $tc_R(n)$.

A computation of M processes at most $tc_M(k)$ transitions, where k is the length of its input string. The number of transitions of the composite machine is bounded by the sum of the estimates of the two separate computations. If $tc_R \in O(n^s)$ and $tc_M \in O(n^t)$, then

$$tc_R(n) + tc_M(tc_R(n)) \in O(n^{st}). \quad \blacksquare$$

Example 15.6.1

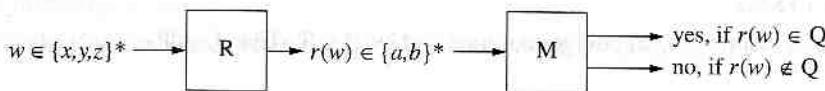
The Turing machine R



reduces the language $L = \{x^i y^j z^k \mid i \geq 0, j \geq 0, k \geq 0\}$ to $Q = \{a^i b^i \mid i \geq 0\}$. The motivation for this reduction was given in Section 11.3; here we are concerned with analyzing its time complexity.

For strings of length 0 and 1, $tc_R(0) = 2$ and $tc_R(1) = 4$. The worst-case computation for the remainder of the strings occurs when an x or y follows a z . In this case, the input string is read in states q_1, q_2 , and q_3 and erased in state q_4 . The computation is completed by writing an a in the input position. The time complexity is $tc_R(n) = 2n + 4$, for $n > 1$.

Consider the combination R with the machine M



that accepts Q with performance for the computation if n is odd. The computation

which is within the

Problem reduction problems. We begin by analyzing the time complexity of R correctly so, that $\Theta(n^2)$ is a true time and ingenuity problem and intractable problems are not

If L is reducible to Q as hard of a problem as the solution obtained by M. Moreover, the computation shows that if Q is tractable, then languages can be easily reduced to Q.

Definition 15.6.3

Let \mathcal{C} be a class of languages. \mathcal{C} is reducible to Q if

If Q is hard for \mathcal{C} , then Q is solvable in polynomial time.

15.7 P = NP?

A language accepted by a nondeterministic machine is in P. The construction of a deterministic machine that decides the language preserves polynomial time. Section 8.7. Unfortunately, this construction does not work as shown in Theorem 15.7.

The two solutions to the problem of whether P = NP differ between deterministic and nondeterministic machines. In the deterministic case, the solution is based on the fact that a nondeterministic machine can make many choices simultaneously, while a deterministic machine can only make one choice at a time. This allows a nondeterministic machine to solve certain problems much faster than a deterministic machine. However, it is not known whether this difference in speed is significant enough to make a practical difference in the real world.

Q and
and M.
number of
1 input
can be
n, then

h of its
sum of
en

■

that accepts Q with time complexity $tc_M(n) = (n^2 + 3n + 4)/2$. The worst-case performance for the composite machine occurs for strings $x^{n/2}y^{n/2}$ if n is even and $x^{(n+1)/2}y^{(n-1)/2}$ if n is odd. The complexity of the resulting solution to the membership problem of L is

$$tc_R(n) + tc_M(tc_R(n)) = 2n + 2 + (n^2 + 3n + 4)/2,$$

which is within the upper bound $O(n^3)$ given in the proof of Theorem 15.6.2. \square

Problem reduction gives us the basis for a comparison of the relative difficulty of two problems. We begin by noting that we will consider two problems to be of equal difficulty if the time complexity of their solutions differs only polynomially. It may be pointed out, and correctly so, that $\Theta(n^2)$ algorithms are preferred to $\Theta(n^3)$ algorithms and that considerable time and ingenuity has been spent to reduce the complexity of many algorithms. That is true (and a worthwhile endeavor), but our emphasis is on distinguishing between tractable and intractable problems. In this regard, polynomial differences between the complexity of algorithms are not significant.

If L is reducible to Q in polynomial time, then Q may be thought of as being at least as hard of a problem as L. Finding a solution to Q automatically yields a solution to L; the solution obtained by sequentially performing the reduction followed by the solution to Q. Moreover, the complexity of the composition of the reduction and the solution to Q shows that if Q is tractable, so is L. The relation between reduction and the relative hardness of languages can be extended to classes of languages.

Definition 15.6.3

Let \mathcal{C} be a class of languages. A language Q is **hard for the class \mathcal{C}** if every language in \mathcal{C} is reducible to Q in polynomial time.

If Q is hard for a class \mathcal{C} and is solvable in polynomial time, then every problem in \mathcal{C} is solvable in polynomial time and $\mathcal{C} \subseteq \mathcal{P}$.

15.7 $\mathcal{P} = \mathcal{NP}$?

ration for
g its time

putation
the input
completed
or $n > 1$.

$\equiv Q$
Q

A language accepted in polynomial time by a deterministic multitrack or multitape machine is in \mathcal{P} . The construction of an equivalent standard Turing machine from one of these alternatives preserves polynomial-time complexity. A technique for constructing an equivalent deterministic machine from the transitions of a nondeterministic machine was presented in Section 8.7. Unfortunately, this construction does not preserve polynomial-time complexity as shown in Theorem 15.1.2.

The two solutions to the Hamiltonian Circuit Problem dramatically illustrate the difference between deterministic and nondeterministic computations. To obtain an answer, the deterministic solution generates sequences of vertices in an attempt to discover a tour. In the worst case, this process requires the examination of all possible sequences of vertices that may constitute a tour of the graph. The nondeterministic machine avoided this

by “guessing” a single sequence of vertices and determining if this sequence forms a tour. The philosophic interpretation of the $\mathcal{P} = \mathcal{NP}$ question is whether constructing a solution to a problem is inherently more difficult than checking to see if a single possibility satisfies the conditions of the problem. Because of the additional complexity of currently known deterministic solutions over nondeterministic solutions across a wide range of important problems, it is generally believed that $\mathcal{P} \neq \mathcal{NP}$. The $\mathcal{P} = \mathcal{NP}$ question is, however, a precisely formulated mathematical problem and will be resolved only when the equality of the two classes or the proper inclusion of \mathcal{P} in \mathcal{NP} is proved.

One approach for determining whether $\mathcal{P} = \mathcal{NP}$ is to examine the properties of each language or decision problem on an individual basis. For example, considerable effort has been expended attempting to develop a deterministic polynomial algorithm to solve the Hamiltonian Circuit Problem. On the face of it, finding such a solution would resolve the question for only one language. What is needed is a universal approach that resolves the issue of deterministic polynomial solvability for all languages in \mathcal{NP} at once. The notion of a language being hard for the class \mathcal{NP} allows us to transform the question of polynomial-time solvability for all problems in \mathcal{NP} to that of a single problem.

Definition 15.7.1

A language Q is called **NP-hard** if for every $L \in \mathcal{NP}$, L is reducible to Q in polynomial time. An NP-hard language that is also in \mathcal{NP} is called **NP-complete**.

One can consider an NP-complete language as a universal language in the class \mathcal{NP} . The discovery of a polynomial-time machine that accepts an NP-complete language can be used to construct machines to accept every language in \mathcal{NP} in deterministic polynomial time. This, in turn, yields an affirmative answer to the $\mathcal{P} = \mathcal{NP}$ question.

Theorem 15.7.2

If there is an NP-complete language that is also in \mathcal{P} , then $\mathcal{P} = \mathcal{NP}$.

Proof. Assume that Q is an NP-complete language that is accepted in polynomial time by a deterministic Turing machine. Let L be any language in \mathcal{NP} . Since Q is NP-hard, there is a polynomial time reduction of L to Q . Now, by Theorem 15.6.2, L is also in \mathcal{P} . ■

The definition of NP-completeness utilized the terminology of recursive languages and Turing computable functions because of the precision afforded by the concepts and notation of Turing computability. The duality between recursive languages and solvable decision problems permits us to speak of NP-hard and NP-complete decision problems. It is worthwhile to reexamine these definitions in the context of decision problems.

Reducibility of languages using Turing computable functions is a formalization of the notion of reduction of decision problems that was developed in Chapter 11. A decision problem is NP-hard or NP-complete whenever the language accepted by a machine that solves the problem is. Utilizing the universal reducibility of problems in \mathcal{NP} to an NP-hard problem P , we can obtain a solution to any \mathcal{NP} problem by combining the reduction with the machine that solves P .

Regardless of the languages or decision problems involved, unfortunately, we have made only a substantial amount of progress.

15.8 The Satisfiability Problem

The Satisfiability Problem, also known as the Boolean satisfiability problem, is a decision problem in propositional logic, where the goal is to determine whether a formula is satisfiable. The objective of the problem is to find an assignment of truth values to propositions that makes a given formula true. The Satisfiability Problem is NP-complete.

A Boolean variable x is considered to be a propositional variable, the variable specifying the Boolean value of a truth assignment.

The logical connectives \neg , \wedge , and \vee are known as *well-formed formulas* (WFFs). x , y , and z to denote variables.

Definition 15.8.1

Let V be a set of variables.

- i) If $x \in V$, then x is a well-formed formula.
- ii) If u, v are well-formed formulas, then $u \wedge v$ and $u \vee v$ are well-formed formulas.
- iii) An expression of the form $\neg x$, where x is a Boolean variable, is a well-formed formula.

The expression $(x \wedge y) \vee (z \wedge \neg z)$ is a well-formed formula. The parentheses in a well-formed formula serve to indicate the scope of the logical operators. The logical operators \wedge and \vee are closed under conjunction and disjunction, respectively. These conventions, along with the rules of inference, permit us to prove theorems in propositional logic.

ms a tour.
a solution
y satisfies
ly known
important
ver, a pre-
dily of the

es of each
effort has
solve the
ld resolve
esolves the
notion of a
omial-time

polynomial

e class NP.
nguage can
polynomial

nial time by
ard, there is
P. ■

e languages
oncepts and
and solvable
problems. It
ns.
zation of the
. A decision
machine that
o an NP-hard
duction with

Regardless of whether we approach NP-completeness from the perspective of languages or decision problems, it is clear that this is an important class of problems. Unfortunately, we have not yet shown that such a universal problem exists. Although it requires a substantial amount of work, this omission is remedied in the next section.

15.8 The Satisfiability Problem

The Satisfiability Problem, which is concerned with the truth values of formulas in propositional logic, was the first decision problem shown to be NP-complete. The truth value of a formula is obtained from those of the elementary propositions occurring in the formula. The objective of the Satisfiability Problem is to determine whether there is an assignment of truth values to propositions that makes the formula true. Before demonstrating that the Satisfiability Problem is NP-complete, we will briefly review the fundamentals of propositional logic.

A *Boolean variable* is a variable that takes on values 0 and 1. Boolean variables are considered to be propositions, the elementary objects of propositional logic. The value of the variable specifies the truth or falsity of the proposition. The proposition x is true when the Boolean variable is assigned the value 1. The value 0 designates a false proposition. A *truth assignment* is a function that assigns a value 0 or 1 to every Boolean variable.

The logical connectives \wedge (and), \vee (or), and \neg (not) are used to construct propositions known as *well-formed formulas* from a set of Boolean variables. We will use the symbols x , y , and z to denote Boolean variables and u , v , and w to represent well-formed formulas.

Definition 15.8.1

Let V be a set of Boolean variables.

- i) If $x \in V$, then x is a well-formed formula.
- ii) If u , v are well-formed formulas, then (u) , $(\neg u)$, $(u \wedge v)$, and $(u \vee v)$ are well-formed formulas.
- iii) An expression is a well-formed formula over V only if it can be obtained from the Boolean variables in the set V by a finite number of applications of the operations in (ii).

The expressions $((\neg(x \vee y)) \wedge z)$, $((x \wedge y) \vee z) \vee \neg(x)$, and $((\neg x) \vee y) \wedge (x \vee z)$ are well-formed formulas over the Boolean variables x , y , and z . The number of parentheses in a well-formed formula can be reduced by defining a precedence relation on the logical operators. Negation is considered the most binding operation, followed by conjunction and then disjunction. Additionally, the associativity of conjunction and disjunction permits the parentheses in sequences of these operations to be omitted. Utilizing these conventions, we can rewrite the preceding formulas as $\neg(x \vee y) \wedge z$, $x \wedge y \vee z \vee \neg x$, and $\neg x \vee y \wedge (x \vee z)$.

The truth values of the variables are obtained directly from the truth assignment. The standard interpretation of the logical operations can be used to extend truth values from variables to the well-formed formulas. The truth values of formulas $\neg u$, $u \wedge v$, and $u \vee v$ are obtained from the values of u and v according to the rules given in the following tables.

u	$\neg u$	$u \quad v$	$u \wedge v$	$u \quad v$	$u \vee v$
0	1	0 0	0	0 0	0
1	0	0 1	0	0 1	1
		1 0	0	1 0	1
		1 1	1	1 1	1

A formula u is satisfied by a truth assignment if the values of the variables cause u to assume the value 1. Two well-formed formulas are equivalent if they are satisfied by the same truth assignments.

A *clause* is a well-formed formula that consists of a disjunction of variables or the negation of variables. An unnegated variable is called a *positive literal* and a negated variable a *negative literal*. Using this terminology, a clause is a *disjunction of literals*. The formulas $x \vee \neg y$, $\neg x \vee z \vee \neg y$, and $x \vee z \vee \neg x$ are clauses over the set of Boolean variables $\{x, y, z\}$. A formula is in *conjunctive normal form* if it has the form

$$u_1 \wedge u_2 \wedge \cdots \wedge u_n,$$

where each u_i is a clause. A classical theorem of propositional logic asserts that every well-formed formula can be transformed into an equivalent formula in conjunctive normal form.

Stated precisely, the Satisfiability Problem is the problem of deciding if a formula in conjunctive normal form is satisfied by some truth assignment. The formulas

$$u = (x \vee y) \wedge (\neg y \vee z)$$

$$v = (x \vee \neg y \vee \neg z) \wedge (x \vee z) \wedge (\neg x \vee \neg y)$$

built from the variables $\{x, y, z\}$ are satisfied by the truth assignment

t	
x	1
y	0
z	0

The first clause in u is satisfied by x and the second by $\neg y$. The first clause of v is satisfied by all three variables, the second by x , and the third by $\neg y$. The formula

$$w = \neg x \wedge (x \vee y) \wedge (\neg y \vee x)$$

is not satisfied by t . This assignment.

A deterministic algorithm for every truth assignment. Every number of Boolean variables. A method of constructing this exhaustive approach to an assignment, however, is to consider the formula. This observation is nondeterministic machine.

Theorem 15.8.2

The Satisfiability Problem

Proof. We begin by showing that every Boolean variables $\{x, y, z\}$ has a unique encoding. The encoding of a variable x is a binary string of length 2. The encoding of a positive literal is positive and the encoding of a negative literal is negative.

The number following the assignment is the number of variables in the literal.

A well-formed formula is represented by a sequence of symbols representing disjunctions and conjunctions.

is encoded as

Finally, the input to the function is followed by ## and the output is the preceding formula.

The representation of the alphabet $\Sigma = \{0, 1, \wedge, \vee, \neg\}$ is a satisfiable conjunctive formula.

The
om
/ v
les.

is not satisfied by t . Moreover, it is not difficult to see that w is not satisfied by any truth assignment.

A deterministic solution to the Satisfiability Problem can be obtained by checking every truth assignment. The number of possible truth assignments is 2^n , where n is the number of Boolean variables. An implementation of this strategy is essentially a mechanical method of constructing the complete truth table for the formula. Clearly, the complexity of this exhaustive approach is exponential. The work expended in checking a particular truth assignment, however, grows polynomially with the number of variables and the length of the formula. This observation provides the insight needed for designing a polynomial-time nondeterministic machine that solves the Satisfiability Problem.

Theorem 15.8.2

The Satisfiability Problem is in NP.

Proof. We begin by developing a representation for the well-formed formulas over a set of Boolean variables $\{x_1, \dots, x_n\}$. A variable is encoded by the binary representation of its subscript. The encoding of a literal consists of the encoded variable followed by #1 if the literal is positive and #0 if it is negative.

Literal	Encoding
x_i	$i\#1$
$\neg x_i$	$i\#0$

The number following the encoding of the variable specifies the Boolean value that satisfies the literal.

A well-formed formula is encoded by concatenating the literals with the symbols representing disjunction and conjunction. The conjunctive normal form formula

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3)$$

is encoded as

$$1\#1 \vee 10\#0 \wedge 1\#0 \vee 11\#1.$$

Finally, the input to the machine consists of the encoding of the variables in the formula followed by ## and then the encoding of the formula itself. The input string representing the preceding formula is

$$\begin{array}{c} 1\#10\#11\#\#1\#1 \vee 10\#0 \wedge 1\#0 \vee 11\#1 \\ \text{variables} \quad \text{formula} \end{array}$$

classified

The representation of an instance of the Satisfiability Problem is a string over the alphabet $\Sigma = \{0, 1, \wedge, \vee, \#\}$. The language L_{SAT} consists of all strings over Σ that represent satisfiable conjunctive normal form formulas.

A two-tape nondeterministic machine M that solves the Satisfiability Problem is described below. M employs the guess-and-check strategy; the guess nondeterministically generates a truth assignment. Configurations corresponding to the computation initiated with the input string representing the formula $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3)$ are given to illustrate the actions of the machine. The initial configuration of the tape contains the representation of the formula on tape 1 with tape 2 blank:

Tape 2 BB

1. If the input does not have the anticipated form, the computation halts and rejects the string.
 2. The encoding of the first variable on tape 1 is copied onto tape 2. This is followed by printing # and nondeterministically writing 0 or 1. If this is not the last variable, ## is written and the procedure is repeated for the next variable. Nondeterministically choosing a value for each variable defines a truth assignment t . The value assigned to variable x_i is denoted $t(x_i)$. Using this notation, the tapes have the form

Tape 1 $B1\#10\#11\#\#1\#1 \vee 10\#0 \wedge 1\#0 \vee 11\#1B$

The tape head on tape 2 is repositioned at the leftmost position. The head on tape 1 is moved past ## into a position to read the first variable of the formula.

The generation of the truth assignment is the only instance of nondeterminism of M . The remainder of the computation checks whether the formula is satisfied by the nondeterministically selected truth assignment.

3. Assume that the encoding of the variable x_i is scanned on tape 1. The encoding of x_i is found on tape 2. The subsequent actions of the machine are determined by the result of comparing the value $t(x_i)$ on tape 2 with the Boolean value following x_i on tape 1.
 4. If the values do not match, the current literal is not satisfied by the truth assignment. If the symbol following the literal is a B or \wedge , every literal in the current clause has been examined and failed. When this occurs, the truth assignment does not satisfy the formula and the computation halts in a nonaccepting state. If \vee is read, the tape heads are positioned to examine the next literal in the clause (step 3).
 5. If the values do match, the literal and current clause are satisfied by the truth assignment. The head on tape 1 moves to the right to the next \wedge or B . If a B is encountered, the computation halts and accepts the input. Otherwise, the next clause is processed by returning to step 3.

The matching procedure in step 3 determines the rate of growth of the time complexity of the computations. In the worst case, the matching requires comparing the variable on tape 1 with each of the variables on tape 2 to discover the match. This can be accomplished in $O(k \cdot n^2)$ time, where n is the number of variables and k the number of literals in the input. ■

We now polynomial-time. There are infinite languages that are not decidable by deterministic Turing machines, the most famous of which is the halting problem. In this manner, a generalization of the Church-Turing thesis is obtained.

Theorem 15.8.

The Satisfiability

Proof. Let L computations ; Problem is ach conjunctive no We then must s with the length

The states.

The blank is ass
elements of the
rejecting state is

Let $u \in \Sigma^*$ computations of transitions in that there is a tr

We now must show that L_{SAT} is NP-hard, that is, that every language in NP is polynomial-time reducible to L_{SAT} . At the outset, this may seem like an impossible task. There are infinitely many languages in NP, and they appear to have little in common. They are not even restricted to having the same alphabet. The lone universal feature of the languages in NP is that they are all accepted by a polynomial-time-bounded nondeterministic Turing machine. Fortunately, this is enough. Rather than concentrating on the languages, the proof will exploit the properties of the machines that accept the languages. In this manner, a general procedure is developed that can be used to reduce any language in NP to L_{SAT} .

Theorem 15.8.3 (Cook's Theorem)

The Satisfiability Problem is NP-hard.

Proof. Let L be a language accepted by a nondeterministic Turing machine M whose computations are bounded by a polynomial p. The reduction of L to the Satisfiability Problem is achieved by transforming the computations of M with an input string u into a conjunctive normal form formula $f(u)$ so that $u \in L(M)$ if, and only if, $f(u)$ is satisfiable. We then must show that the construction of $f(u)$ requires time that grows only polynomially with the length of u.

Without loss of generality, we assume that all computations of M halt in one of two states. All accepting computations terminate in state q_A and rejecting computations in q_R . Moreover, we assume that there are no transitions leaving these states. An arbitrary machine can be transformed into an equivalent one satisfying these restrictions by adding transitions from every accepting configuration to q_A and from every rejecting configuration to q_R . This alteration adds a single transition to every computation of the original machine. The transformation from computation to well-formed formula assumes that all computations with input of length n contain $p(n)$ configurations. The terminating configuration is repeated, if necessary, to ensure that the correct number of configurations are present.

The states, final state, and alphabets of M are denoted

$$\begin{aligned} Q &= \{q_0, q_1, \dots, q_m\} \\ \Gamma &= \{B = a_0, a_1, \dots, a_s, a_{s+1}, \dots, a_t\} \\ \Sigma &= \{a_{s+1}, a_{s+2}, \dots, a_t\} \\ F &= \{q_m\}. \end{aligned}$$

The blank is assumed to be the tape symbol numbered 0. The input alphabet consists of the elements of the tape alphabet numbered $s + 1$ to t . The lone accepting state is q_m and the rejecting state is q_{m-1} .

Let $u \in \Sigma^*$ be a string of length n. Our goal is to define a formula $f(u)$ that encodes the computations of M with input u. The length of $f(u)$ depends on $p(n)$, the maximum number of transitions in a computation of M with input of length n. The encoding is designed so that there is a truth assignment satisfying $f(u)$ if, and only if, $u \in L(M)$. The formulas are

built from three classes of variables; each class is introduced to represent a property of a machine configuration.

Variable	Interpretation (when satisfied)	
$Q_{i,k}$	$0 \leq i \leq m$	M is in state q_i at time k .
	$0 \leq k \leq p(n)$	
$P_{j,k}$	$0 \leq j \leq p(n)$	M is scanning position j at time k .
$S_{j,r,k}$	$0 \leq r \leq t$	
	$0 \leq j \leq p(n)$	Tape position j contains symbol a_r at time k .
	$0 \leq k \leq p(n)$	

Clause
i) State
$\bigvee_{i=0}^m Q_{i,k} \vee \neg Q_{i,k} \vee \neg Q_{i',k}$
ii) Tape head position
$\bigvee_{j=0}^{p(n)} P_{j,k} \vee \neg P_{j,k} \vee \neg P_{j',k}$
iii) Symbols on tape
$\bigvee_{r=0}^t S_{j,r,k} \vee \neg S_{j,r,k} \vee \neg S_{j,r',k}$
iv) Initial conditions
string $u = a_r Q_{0,0} P_{0,0} S_{0,0,0}$
$S_{1,r_1,0}$
$S_{2,r_2,0}$
\vdots
$S_{n,r_n,0}$
$S_{n+1,0,0}$
\vdots
$S_{p(n),0,0}$
v) Accepting configuration
$Q_{m,p(n)}$

The set of variables V is the union of the three sets defined in the table. A computation of M defines a truth assignment on V. For example, if tape position 3 initially contains symbol a_i , then $S_{3,i,0}$ is true. Necessarily, $S_{3,j,0}$ must be false for all $j \neq i$. A truth assignment obtained in this manner specifies the state, position of the tape head, and the symbols on the tape for each time k in the range $0 \leq k \leq p(n)$. This is precisely the information contained in the sequence of configurations produced by the computation.

An arbitrary assignment of truth values to the variables in V need not correspond to a computation of M. Assigning 1 to both $P_{0,0}$ and $P_{1,0}$ indicates that the tape head is at two distinct positions at time 0. Similarly, a truth assignment might specify that the machine is in several states at a given time or might designate the presence of multiple symbols in a single position.

The formula $f(u)$ should impose restrictions on the variables to ensure that the interpretations of the variables are identical with those generated by the truth assignment obtained from a computation. Eight sets of formulas are defined from the input string u and the transitions of M. Seven of the eight families of formulas are given directly in clause form. The clauses are accompanied by a brief description of their interpretation in terms of Turing machine configurations and computations. The notation

$$\bigwedge_{i=1}^k v_i \quad \bigvee_{i=i}^k v_i$$

represents the conjunction and disjunction of the literals v_1, v_2, \dots, v_k , respectively.

A truth assignment that satisfies the set of clauses defined in (i) in the following table indicates that the machine is in a unique state at each time. Satisfying the first disjunction guarantees that at least one of the variables $Q_{i,k}$ holds. The pairwise negations specify that no two states are satisfied at the same time. This is most easily seen using the tautological equivalence of the disjunction $\neg A \vee B$ to the implication $A \Rightarrow B$ to transform the clauses $\neg Q_{i,k} \vee \neg Q_{i',k}$ into implications. Writing $\neg Q_{i,k} \vee \neg Q_{i',k}$ as an implication produces $Q_{i,k} \Rightarrow \neg Q_{i',k}$, which can be interpreted as asserting that if the machine is in state q_i at time k , then it is not also in $q_{i'}$ for any $i' \neq i$.

roperty of a

	Clause	Conditions	Interpretation (when satisfied)
i) State	$\bigvee_{i=0}^m Q_{i,k}$	$0 \leq k \leq p(n)$	For each time k , M is in at least one state.
	$\neg Q_{i,k} \vee \neg Q_{i',k}$	$0 \leq i < i' \leq m$ $0 \leq k \leq p(n)$	M is in at most one state (not two different states at the same time).
ii) Tape head position	$\bigvee_{j=0}^{p(n)} P_{j,k}$	$0 \leq k \leq p(n)$	For each time k , the tape head is in at least one position.
	$\neg P_{j,k} \vee \neg P_{j',k}$	$0 \leq j < j' \leq p(n)$ $0 \leq k \leq p(n)$	At most one position.
iii) Symbols on tape	$\bigvee_{r=0}^t S_{j,r,k}$	$0 \leq j \leq p(n)$ $0 \leq k \leq p(n)$	For each time k and position j , position j contains at least one symbol.
	$\neg S_{j,r,k} \vee \neg S_{j,r',k}$	$0 \leq j \leq p(n)$ $0 \leq r < r' \leq t$ $0 \leq k \leq p(n)$	At most one symbol.
iv) Initial conditions for input string $u = a_{r_1}a_{r_2}\dots a_{r_n}$	$Q_{0,0}$ $P_{0,0}$ $S_{0,0,0}$ $S_{1,r_1,0}$ $S_{2,r_2,0}$ \vdots $S_{n,r_n,0}$ $S_{n+1,0,0}$ \vdots $S_{p(n),0,0}$	The computation begins reading the leftmost blank.	The string u is in the input position at time 0.
v) Accepting condition	$Q_{m,p(n)}$		The remainder of the tape is blank at time 0. The halting state of the computations is q_m .

Since the computation of M with input of length n cannot access the tape beyond position $p(n)$, a machine configuration is completely defined by the state, position of the tape head, and the contents of the initial $p(n)$ positions of the tape. A truth assignment that satisfies the clauses in (i), (ii), and (iii) defines a machine configuration for each time between 0 and $p(n)$. The conjunction of the clauses (i) and (ii) indicates that the machine is in a unique state scanning a single tape position at each time. The clauses in (iii) ensure that the tape is well-defined; that is, the tape contains precisely one symbol in each position that may be referenced during the computation.

A computation does not consist of a sequence of unrelated configurations but rather a sequence in which each configuration differs from its predecessor by the result of a single transition. We must add clauses whose satisfaction specifies the configuration at time 0 and links consecutive configurations. Initially, the machine is in state q_0 , the tape head scanning the leftmost position, the input on tape positions 1 to n , and the remaining tape squares blank. The satisfaction of the $p(n) + 2$ clauses in (iv) ensures the correct machine configuration at time 0.

Each subsequent configuration must be obtained from its successor by the application of a transition. Assume that the machine is in state q_i , scanning symbol a_j in position j at time k . The final three sets of formulas are introduced to generate the permissible configurations at time $k + 1$ based on the transitions of M and the variables that define the configuration at time k .

The effect of a transition on the tape is to rewrite the position scanned by the tape head. With the possible exception of position $P_{j,k}$, every tape position at time $k + 1$ contains the same symbol as at time k . Clauses must be added to the formula to ensure that the remainder of the tape is unaffected by a transition.

Clause	Conditions	Interpretation (when satisfied)
vi) Tape consistency	$\neg S_{j,r,k} \vee P_{j,k} \vee S_{j,r,k+1}$ $0 \leq j \leq p(n)$ $0 \leq r \leq t$ $0 \leq k \leq p(n)$	Symbols not at the position of the tape head are unchanged.

This clause is not satisfied if a change occurs to a tape position other than the one scanned by the tape head. This can be seen by noting that

$$\neg S_{j,r,k} \vee P_{j,k} \vee S_{j,r,k+1}$$

is equivalent to

$$\neg P_{j,k} \Rightarrow (S_{j,r,k} \Rightarrow S_{j,r,k+1}),$$

which clearly indicates that if the tape head is not at position j at time k , then the symbol at position j is the same at time $k + 1$ as it was at time k .

Now assume position j . These Boolean variables

a) $\neg Q_{i,k} \vee \neg P_j$
is satisfied only when entered state q_i and tape head position are

b) $\neg Q_{i,k} \vee \neg P_j$
c) $\neg Q_{i,k} \vee \neg P_j$

where $n(L) = -1$ only if the configuration application of the

The clausal representation guarantees configuration definition for states q_m and q_n for every state, sy

The conjunct

$$(\neg Q_{i,k}) \\ \wedge (\neg Q_{i,k}) \\ \wedge (\neg Q_{i,k})$$

is constructed for

where $[q_i, a_j, d]$ by the transition. The tape head to cross by having the success is encoded by the

$$(\neg Q_{i,k}) \\ \wedge (\neg Q_{i,k}) \\ \wedge (\neg Q_{i,k})$$

for all transitions [

be beyond
ion of the
ssignment
each time
e machine
iii) ensure
h position

ut rather a
of a single
time 0 and
d scanning
ares blank.
nfiguration

lication of
n j at time
figurations
nfiguration

e tape head.
ontains the
e remainder

n satisfied)

osition of
changed.

han the one

i the symbol

Now assume that for a given time k , the machine is in state q_i scanning symbol a_r in position j . These features of a configuration are designated by the assignment of 1 to the Boolean variables $Q_{i,k}$, $P_{j,k}$, and $S_{j,r,k}$. The clause

$$a) \neg Q_{i,k} \vee \neg P_{j,k} \vee \neg S_{j,r,k} \vee Q_{i',k+1}$$

is satisfied only when $Q_{i',k+1}$ is true. In terms of the computation, this signifies that M has entered state $q_{i'}$ at time $k + 1$. Similarly, the symbol in position j at time $k + 1$ and the tape head position are specified by the clauses

$$b) \neg Q_{i,k} \vee \neg P_{j,k} \vee \neg S_{j,r,k} \vee S_{j,r',k+1} \text{ and}$$

$$c) \neg Q_{i,k} \vee \neg P_{j,k} \vee \neg S_{j,r,k} \vee P_{j+n(d),k+1},$$

where $n(L) = -1$ and $n(R) = 1$. The conjunction of clauses of (a), (b), and (c) is satisfied only if the configuration at time $k + 1$ is obtained from the configuration at time k by the application of the transition $[q_{i'}, a_{r'}, d] \in \delta(q_i, a_r)$.

The clausal representation of transitions is used to construct a formula whose satisfaction guarantees that the time $k + 1$ variables define a configuration obtained from the configuration defined by the time k variables by the application of a transition of M. Except for states q_m and q_{m-1} , the restrictions on M ensure that at least one transition is defined for every state, symbol pair.

The conjunctive normal form formula

$$(\neg Q_{i,k} \vee \neg P_{j,k} \vee \neg S_{j,r,k} \vee Q_{i',k+1}) \quad (\text{new state})$$

$$\wedge (\neg Q_{i,k} \vee \neg P_{j,k} \vee \neg S_{j,r,k} \vee P_{j+n(d),k+1}) \quad (\text{new tape head position})$$

$$\wedge (\neg Q_{i,k} \vee \neg P_{j,k} \vee \neg S_{j,r,k} \vee S_{j,r',k+1}) \quad (\text{new symbol at position } r)$$

is constructed for every

$$0 \leq k \leq p(n) \quad (\text{time})$$

$$0 \leq i < m - 1 \quad (\text{nonhalting state})$$

$$0 \leq j \leq p(n) \quad (\text{tape head position})$$

$$0 \leq r \leq t \quad (\text{tape symbol})$$

where $[q_{i'}, a_{r'}, d] \in \delta(q_i, a_r)$ except when the position is 0 and the direction L is specified by the transition. The exception occurs when the application of a transition would cause the tape head to cross the left-hand boundary of the tape. In clausal form, this is represented by having the succeeding configuration contain the rejecting state q_{m-1} . This special case is encoded by the formulas

$$(\neg Q_{i,k} \vee \neg P_{0,k} \vee \neg S_{0,r,k} \vee Q_{m-1,k+1}) \quad (\text{entering the rejecting state})$$

$$\wedge (\neg Q_{i,k} \vee \neg P_{0,k} \vee \neg S_{0,r,k} \vee P_{0,k+1}) \quad (\text{same tape head position})$$

$$\wedge (\neg Q_{i,k} \vee \neg P_{0,k} \vee \neg S_{0,r,k} \vee S_{0,r,k+1}) \quad (\text{same symbol at position } r)$$

for all transitions $[q_{i'}, a_{r'}, L] \in \delta(q_i, a_r)$.

Since M is nondeterministic, there may be several transitions that can be applied to a given configuration. The result of the application of any of these alternatives is a permissible succeeding configuration in a computation. Let $\text{trans}(i, j, r, k)$ denote disjunction of the conjunctive normal form formulas that represent the alternative transitions for a configuration at time k in state q_i , tape head position j , and tape symbol r . The formula $\text{trans}(i, j, r, k)$ is satisfied only if the values of the variables encoding the configuration at time $k + 1$ represent a legitimate successor to the configuration encoded in the variables with time k .

Formula	Interpretation (when satisfied)
vii) Generation of successor configuration $\text{trans}(i, j, r, k)$	Configuration $k + 1$ follows from configuration k by the application of a transition.

The formulas $\text{trans}(i, j, r, k)$ do not specify the actions to be taken when the machine is in state q_m or q_{m-1} , the halting states of the machine. In this case the subsequent configuration is identical to its predecessor.

Clause	Interpretation (when satisfied)
viii) Halted computation	
$\neg Q_{i,k} \vee \neg P_{j,k} \vee \neg S_{j,r,k} \vee Q_{i,k+1}$	(same state)
$\neg Q_{i,k} \vee \neg P_{j,k} \vee \neg S_{j,r,k} \vee P_{j,k+1}$	(same tape head position)
$\neg Q_{i,k} \vee \neg P_{j,k} \vee \neg S_{j,r,k} \vee S_{j,r,k+1}$	(same symbol at position r)

These clauses are built for all j, r, k in the appropriate ranges and $i = q_{m-1}, q_m$.

Let $f'(u)$ be the conjunction of the formulas constructed in (i) through (viii). When $f'(u)$ is satisfied by a truth assignment on V , the variables define the configurations of a computation of M that accepts the input string u . The clauses in condition (iv) specify that the configuration at time 0 is the initial configuration of a computation of M with input u . Each subsequent configuration is obtained from its successor by the result of the application of a transition. The string u is accepted by M since the satisfaction of condition (v) indicates that the final configuration contains the accepting state q_m .

A conjunctive normal form formula $f(u)$ can be obtained from $f'(u)$ by converting each formula $\text{trans}(i, j, r, k)$ into conjunctive normal form using the technique presented in Lemma 15.8.4 that follows. All that remains is to show that the transformation of a string $u \in \Sigma^*$ to $f(u)$ can be done in polynomial time.

The transformation of u to $f(u)$ consists of the construction of the clauses and the conversion of trans to conjunctive normal form. The number of clauses is a function of

- i) the number of states m and the number of tape symbols t ,
- ii) the length n of the input string u , and
- iii) the bound $p(n)$ on the length of the computation of M .

The values m and t are constant. From the range of t and m , the development of the formula which, by Lemma 15.8.4, is in CNF, is in polynomial time. By Theorem 15.8.2, $f(u)$ is satisfiable.

We have shown that the number of steps that are needed is the representation of the machine which solves the Satisfiability problem. Theorem 15.8.2, reduces the machine representation to a CNF formula.

The one step transformation of $f'(u)$ to $f(u)$ is the conjunction of conjunctive normal forms in polynomial time.

Lemma 15.8.4

Let $u = w_1 \vee w_2 \vee \dots \vee w_n$ over the variables y_1, y_2, \dots, y_m where the variables y_1, y_2, \dots, y_m are in V' such that

- i) w' is in conjunctive normal form,
- ii) w' is satisfiable over V' ,
- iii) the transformation of w' to CNF is in polynomial time in the w 's.

Proof. The transformation of w' to CNF has been presented. This technique can be applied to each clause in $f'(u)$. Let $u = w_1 \vee w_2 \vee \dots \vee w_n$. Then w_1 and w_2 can be converted to CNF.

plied to a
missible
on of the
onfigura-
 i, j, r, k
 $+ 1$ rep-
me k .

iration k

machine
ibsequent

fied)

ii). When
tions of a
ecify that
h input u .
pplication
) indicates

converting
presented
of a string

es and the
tion of

The values m and t obtained from the Turing machine M are independent of the input string. From the range of the subscripts, we see that the number of clauses is polynomial in $p(n)$. The development of $f(u)$ is completed with the transformation into conjunctive normal form which, by Lemma 15.8.4, is polynomial in the number of clauses in the formulas $\text{trans}(i, j, r, k)$.

We have shown that the conjunctive normal form formula can be constructed in a number of steps that grows polynomially with the length of the input string. What is really needed is the representation of the formula that serves as input to a Turing machine that solves the Satisfiability Problem. Any reasonable encoding, including the one developed in Theorem 15.8.2, requires only polynomial time to convert the high-level representation to the machine representation. ■

The one step missing in the preceding proof is the conversion of the formulas $\text{trans}(i, j, r, k)$ to conjunctive normal form. The following lemma will show that any disjunction of conjunctive normal form formulas can be converted to conjunctive normal form in polynomial time.

Lemma 15.8.4

Let $u = w_1 \vee w_2 \vee \dots \vee w_n$ be the disjunction of conjunctive normal form formulas w_1, w_2, \dots, w_n over the set of Boolean variables V . Also let $V' = V \cup \{y_1, y_2, \dots, y_{n-1}\}$ where the variables y_i are not in V . The formula u can be transformed into a formula u' over V' such that

- i) u' is in conjunctive normal form;
- ii) u' is satisfiable over V' if, and only if, u is satisfiable over V ; and
- iii) the transformation can be accomplished in $O(m \cdot n^2)$, where m is the number of clauses in the w 's.

Proof. The transformation of the disjunction of two conjunctive normal form formulas is presented. This technique may be repeated $n - 1$ times to transform the disjunction of n formulas. Let $u = w_1 \vee w_2$ be the disjunction of two conjunctive normal form formulas. Then w_1 and w_2 can be written

$$w_1 = \bigwedge_{j=1}^{r_1} \left(\bigvee_{k=1}^{s_j} v_{j,k} \right)$$

$$w_2 = \bigwedge_{j=1}^{r_2} \left(\bigvee_{k=1}^{t_j} p_{j,k} \right),$$

where r_i is the number of clauses in w_i , s_j is the number of literals in the j th clause of w_1 , and t_j is the number of literals in the j th clause of w_2 . Define

$$u' = \bigwedge_{j=1}^{r_1} \left(y \vee \bigvee_{k=1}^{s_j} v_{j,k} \right) \wedge \bigwedge_{j=1}^{r_2} \left(\neg y \vee \bigvee_{k=1}^{t_j} p_{j,k} \right).$$

The formula u' is obtained by disjoining y to each clause in w_1 and $\neg y$ to each clause in w_2 .

We now show that u' is satisfiable whenever u is. Assume that w_1 is satisfied by a truth assignment t over V . Then the truth assignment t'

$$t'(x) = \begin{cases} t(x) & \text{if } x \in V \\ 0 & \text{if } x = y \end{cases}$$

satisfies u' . When w_2 is satisfied by t , the truth assignment t' may be obtained by extending t by setting $t'(y) = 1$.

Conversely, assume that u' is satisfied by the truth assignment t' . Then the restriction of t' to V satisfies u . If $t'(y) = 0$, then w_1 must be true. On the other hand, if $t'(y) = 1$, then w_2 is true.

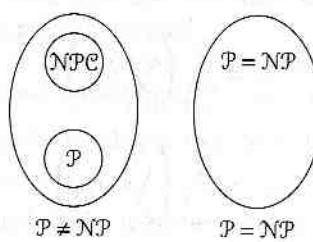
The transformation of

$$u = w_1 \vee w_2 \vee \dots \vee w_n$$

requires $n - 1$ iterations of the preceding process. The repetition adds $n - 1$ literals to each clause in w_1 and w_2 , $n - 2$ literals to each clause in w_3 , $n - 3$ literals to each clause in w_4 , and so on. The transformation requires fewer than $m \cdot n^2$ steps, where m is the total number of clauses in the formulas w_1, w_2, \dots, w_n . ■

15.9 Complexity Class Relations

We end this chapter with two diagrams that illustrate the possible relationships between the classes that have been introduced. The class consisting of all NP-complete problems, which the Satisfiability Problem ensures us is nonempty, is denoted NPC .



If $P \neq NP$, then P is not equal to NP . To be true by most researchers, it would be better if P were equal to NP , the sets of problems in P and NP would be the same.

Exercises

1. Let M be the Turing machine

- a) Trace all configurations of M starting from q_0 .
- b) Describe the language accepted by M in terms of transitions.
- c) Give the fundamental theorem of computation theory for M .

2. Let M be the Turing machine

$M: \boxed{q_0} [B/B R, B/B]$

where x represents the input string.

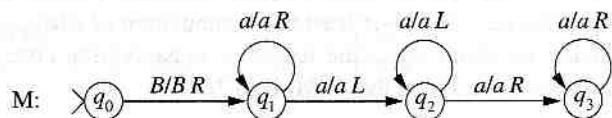
- a) Trace the computation of M on the input string x .
- b) Give a set-theoretic description of the language accepted by M .
- c) What strings are accepted by M ?
- d) Give the fundamental theorem of computation theory for M .
- 3. Show that the class P is closed under union.
- 4. Show that the class NP is closed under intersection.
- * 5. Let $L = \{R(M)u \mid M \in T\}$.
 - a) Prove that L is a regular language. Hint: conclude that L is a finite union of regular languages. The representation of $R(M)$ in $2^{\text{length}(R(M))}$ is finite.
 - b) Prove that L is a context-free language.

of w_1 ,

If $\mathcal{P} \neq \text{NP}$, then \mathcal{P} and NP-C are nonempty, disjoint subsets of NP . This scenario is believed to be true by most mathematicians and computer scientists. In the unlikely case that \mathcal{P} does equal NP , the sets collapse to a single class. Exercise 17 asks you to identify the set of NP-complete problems in this eventuality.

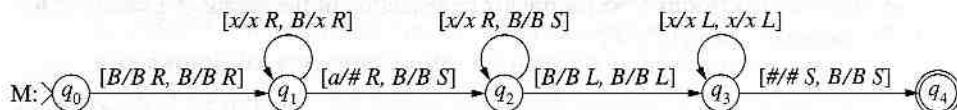
Exercises

1. Let M be the Turing machine



- a) Trace all computations of M with input λ , a , and aa .
- b) Describe the computation of M with input a^n that requires the maximum number of transitions.
- c) Give the function tc_M .

2. Let M be the Turing machine



where x represents either a or b .

- a) Trace the computations of M with input $bbabb$.
 - b) Give a set-theoretic definition of the language of M .
 - c) What strings of length n require the maximum number of transitions? Why?
 - d) Give the function tc_M .
3. Show that the class \mathcal{P} is closed under union, concatenation, and complementation.
4. Show that the class NP is closed under union, concatenation, and the Kleene star operation.
- * 5. Let $L = \{R(M)w \mid M \text{ accepts } w \text{ using at most } 2^{\text{length}(w)} \text{ transitions}\}$.
- a) Prove that L is not in \mathcal{P} . Hint: Use the closure of \mathcal{P} under complementation to conclude that if L is in \mathcal{P} , then there is a Turing machine M' that accepts all representations $R(M)$ of machines M that do not accept their own representations in $2^{\text{length}(R(M))}$ transitions. Then use self-reference to obtain a contradiction.
 - b) Prove that L is not in NP .

6. Design a two-tape Turing machine that transforms unary numbers to binary numbers. Determine the time complexity of your machine.
7. Design a two-tape Turing machine that transforms binary numbers to unary numbers. Explain why this transformation cannot be accomplished in polynomial time.
8. Let P be a decision problem whose input consists of a single natural number and let M be a Turing machine that solves P using the binary representation in polynomial time. Design a machine, using M , that solves P using the base 3 representation of natural numbers. Show that this solution is also polynomial.
- *9. Let M be a nondeterministic machine and p a polynomial. Assume that every string of length n in $L(M)$ is accepted by at least one computation of $p(n)$ or fewer transitions. Note this makes no claim about the length of nonaccepting computations or other accepting computations. Prove that $L(M)$ is in NP .
10. Construct a deterministic Turing machine that reduces the language L to Q in polynomial time. Using the big oh notation, give the time complexity of the machine that computes the reduction.
- $L = \{a^i b^j c^l \mid i \geq 0, j \geq 0\} \quad Q = \{a^i c^l \mid i \geq 0\}$
 - $L = \{a^i (bb)^j \mid i \geq 0\} \quad Q = \{a^i b^i \mid i \geq 0\}$
 - $L = \{a^i b^j a^l \mid i \geq 0\} \quad Q = \{c^i d^l \mid i \geq 0\}$
11. The machine R performs a polynomial-time reduction of the language $L = aa(a \cup b)^*$ to the language $Q = ccc(c \cup d)^*$.

12. The machine R $R: \times q_0$ q_f $x \in \{a\}$

computes a function

- Use the \vdash notation
- What string?
- Give $tc_R(n)$
- Does the machine accept $(c \cup d)cdcd$? Give a string.

13. For each of the following

- $(x \vee y \vee \neg z) \wedge (\neg x \vee y \vee \neg z)$
- $(\neg x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee z)$
- $(x \vee y) \wedge (\neg x \vee \neg y \vee z)$

14. Show that the following set of clauses is not satisfiable.

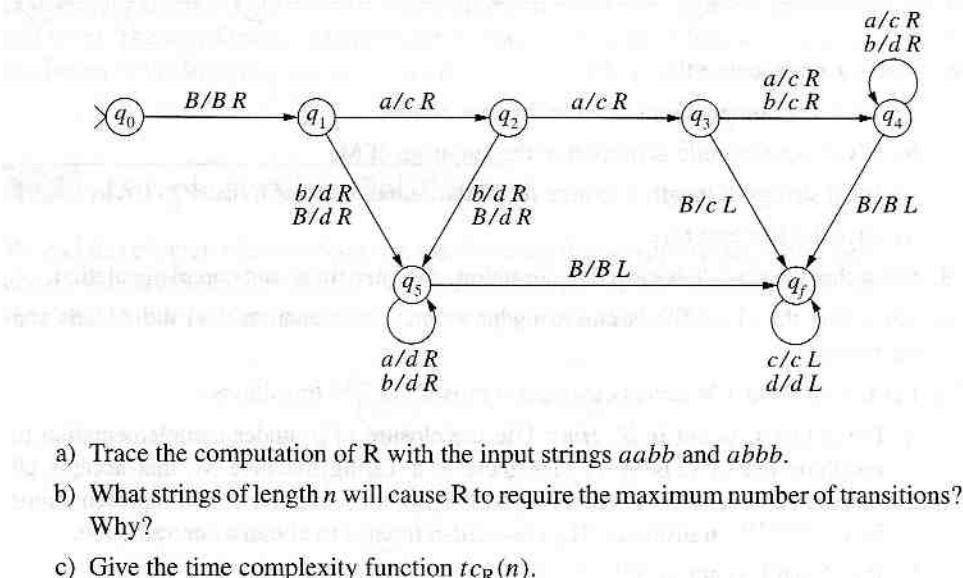
15. Construct four clauses such that their conjunction is unsatisfiable, but the conjunction of any three of them is satisfiable.

16. Prove that the following statements are true.

- $u = v, u' = v'$
- $u = v \vee w, u' = v \vee w'$

17. Assume that $\text{P} \neq \text{NP}$. Then

- Let L be a language in NP but not in P .
- Why is $\text{NP} \subseteq \text{P}$?



numbers.

numbers.

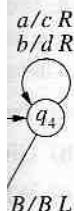
and let M

min time.
of natural

y string of
transitions.
s or other

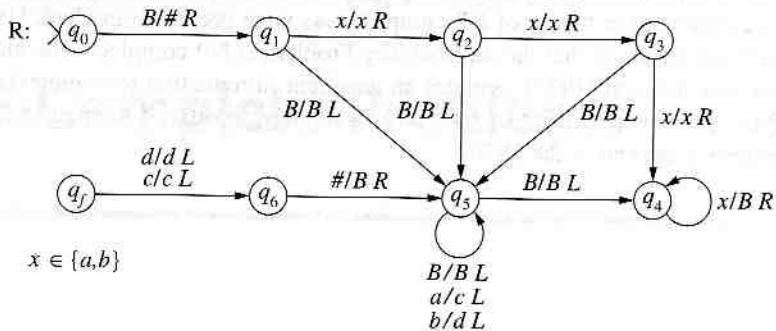
Q in poly-
chine that

$ia(a \cup b)^*$



transitions?

12. The machine R



computes a function from $\{a, b\}^*$ to $\{c, d\}^*$.

- Use the \vdash notation to trace the computation of R with input string $abba$.
 - What string of length n will cause R to use the greatest number of transitions? Why?
 - Give $t_{CR}(n)$. Give both a formula and an explanation of why your formula is correct.
 - Does the machine R reduce the language $L=abb(a \cup b)^*$ to the language $Q=(c \cup d)cdd^*$? If yes, prove that the function computed by R is a reduction. If no, give a string that demonstrates that the mapping is not a reduction.
13. For each of the formulas that follow, give a truth assignment that satisfies the formula.
- $(x \vee y \vee \neg z) \wedge (\neg x \vee y) \wedge (\neg x \vee \neg y \vee \neg z)$
 - $(\neg x \vee y \vee \neg z) \wedge (x \vee \neg y) \wedge (y \vee \neg z) \wedge (\neg x \vee \neg y \vee z)$
 - $(x \vee y) \wedge (\neg x \vee \neg y \vee z) \wedge (x \vee \neg z) \wedge (\neg y \vee \neg z)$
14. Show that the formula $(x \vee \neg y) \wedge (\neg x \vee z) \wedge (y \vee \neg z) \wedge (\neg x \vee \neg y) \wedge (y \vee z)$ is not satisfiable.
15. Construct four clauses over $\{x, y, z\}$ such that the conjunction of any three is satisfiable but the conjunction of all four is unsatisfiable.
16. Prove that the formula u' is satisfiable if, and only if, u is satisfiable.
- $u = v, u' = (v \vee y \vee z) \wedge (v \vee \neg y \vee z) \wedge (v \vee y \vee \neg z) \wedge (v \vee \neg y \vee \neg z)$
 - $u = v \vee w, u' = (v \vee w \vee y) \wedge (v \vee w \vee \neg y)$
17. Assume that $\mathcal{P} = \text{NP}$.
- Let L be a language in NP with $L \neq \emptyset$ and $\bar{L} \neq \emptyset$. Prove that L is NP-complete.
 - Why is NPC a proper subset of NP ?

Bibliographic Notes

The family \mathcal{P} was introduced in Cobham [1964]. NP was first studied by Edmonds [1965]. The foundations of the theory of NP-completeness were presented in Cook [1971]. This work includes the proof that the Satisfiability Problem is NP-complete. The classic book by Garey and Johnson [1979] provides an excellent introduction to complexity analysis and NP-completeness. In addition, it serves as an encyclopedia of problems known to be NP-complete at the end of the 1970s.

The Satisfiability problem is NP-complete. It requires exponential time computations with required the ingenious and frequently simple NP-complete problems. Using them a number of disciplines.

Once a problem has been shown to be NP-complete, a polynomial-time solution will not be found. To solve the problem, an NP-complete problem requires average time computations for optimization problems. In the case of an NP-complete problem,

16.1 Reductions

Two conditions are required for a problem to be NP and it must be NP-hard. It is simply to design a reduction from one NP-hard problem to another.

nds [1965],
1971]. This
lassic book
ty analysis
nown to be

CHAPTER 16

NP-Complete Problems

The Satisfiability Problem was shown to be NP-complete by associating Turing machine computations with conjunctive normal form formulas. If every proof of NP-completeness required the ingenuity of this transformation, the number of problems known to be NP-complete would not be very large. Fortunately, problem reduction provides an alternative and frequently simpler method for demonstrating that problems are NP-complete. Reducing an NP-complete problem to another problem in NP proves that the latter is also NP-complete. Using this technique we will obtain NP-completeness results for problems from a number of disciplines. We also extend the notion of NP-completeness to optimization problems.

Once a problem is shown to be NP-complete, attempting to discover a polynomial-time solution will most likely be unsuccessful. Instead of looking for efficient algorithms to solve the problem, it may be more profitable to adopt a different strategy when an NP-complete problem is encountered. One alternative is to design algorithms that have a good average time complexity, but have some cases that exhibit exponential performance. In optimization problems, accepting a near optimal solution may reduce the time complexity of the problem. In the final section we consider alternatives to be considered when confronting an NP-complete problem.

16.1 Reduction and NP-Complete Problems

Two conditions are required for a language to be NP-complete: the problem must be in NP and it must be NP-hard. The most common way of satisfying the former condition is simply to design a nondeterministic algorithm that solves the problem in polynomial time.

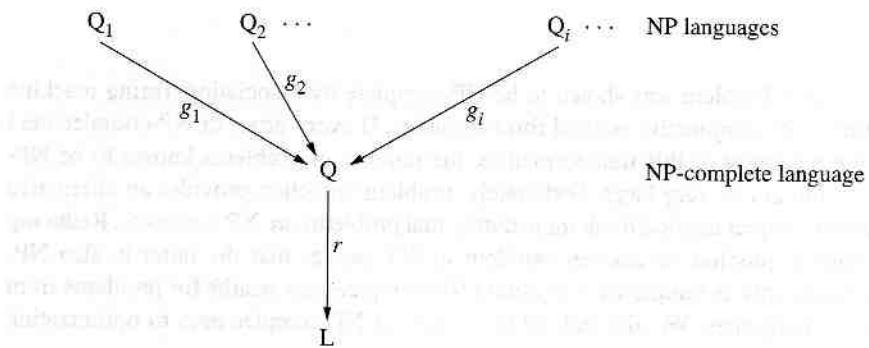
To prove that a language L is NP-hard, it is necessary to show that every language in NP is reducible to L in polynomial time. Rather than directly producing reductions to L , a known NP-complete problem can be used as an intermediate step. Theorem 16.1.1 shows that employing an intermediate step decreases the number of reductions needed to prove that a language is NP-hard from infinitely many to one.

Theorem 16.1.1

Let Q be an NP-complete language. If Q is reducible to L in polynomial time, then L is NP-hard.

Proof. Let r be the computable function that reduces Q to L in polynomial time and let Q_i be any language in NP. Since Q is NP-complete, there is a computable function g_i that reduces Q_i to Q . The composite function $r \circ g_i$ is a reduction of Q_i to L . A polynomial time-bound to the reduction can be obtained from the bounds on r and g_i . ■

The composition used to establish that a language is NP-hard by reduction can be represented pictorially as a two-step process:



The first level shows the polynomial-time reducibility of any language Q_i in NP to Q via a function g_i . Following the arrows from Q_i to L illustrates the reducibility of any NP language to L . If the time complexity of the machines that compute g_i and r are $O(n^s)$ and $O(n^t)$, respectively, the time complexity of the composite function $r \circ g_i$ is $O(n^{st})$ and the reduction of Q_i to L is accomplished in polynomial time. In the next three sections we will use Theorem 16.1.1 to show that several additional problems are NP-complete.

16.2 The 3-Satisfiability Problem

The 3-Satisfiability Problem is a subproblem of the Satisfiability Problem that is NP-complete in its own right. A formula is said to be in **3-conjunctive normal form** if it is in conjunctive normal form and each clause contains precisely three literals. The objective of the 3-Satisfiability Problem is to determine whether a 3-conjunctive normal form formula is satisfiable.

Using the de
establish that the

Reduct

Satisfiab

to

3-Satisfia

That is, the redu
3-conjunctive no
construction of u

Theorem 16.2.1

The 3-Satisfiabil

Proof. Clearly,
bility Problem fo
of 3-conjunctive

We must sho
transformed to a
if, u' is satisfiab
each clause w_i i
conjunction of th
be designed to e
satisfies the origi
assumed not to o

If w_i has th
clause of u that
normal form for

Length 1: w

w'

Length 2: w

w'

Length $n > 3$: w

w'

Establishing
two and their tra

is in NP
s to L, a
1 shows
to prove

then L is

e and let
n g_i that
ynomial

can be

Using the description of reductions introduced in Chapter 11, the condition needed to establish that the 3-Satisfiability Problem is NP-hard can be written

Reduction	Input	Condition
Satisfiability to 3-Satisfiability	conjunctive normal form formula u \downarrow 3-conjunctive normal form formula u'	u is satisfiable if, and only if, u' is satisfiable.

That is, the reduction must transform an arbitrary conjunctive normal form formula into a 3-conjunctive normal form formula that satisfies the prescribed condition. In addition, the construction of u' must be accomplished in time that is polynomial in the length of u .

Theorem 16.2.1

The 3-Satisfiability Problem is NP-complete.

Proof. Clearly, the 3-Satisfiability Problem is in NP. The machine that solves the Satisfiability Problem for arbitrary conjunctive normal form formulas also solves it for the subclass of 3-conjunctive normal form formulas.

We must show that every conjunctive normal form formula $u = w_1 \vee \dots \vee w_m$ can be transformed to a 3-conjunctive normal form formula u' such that u is satisfiable if, and only if, u' is satisfiable. The construction of u' is accomplished by independently transforming each clause w_i in u into a 3-conjunctive normal form formula w'_i . The formula u' is the conjunction of the resulting 3-conjunctive normal form formulas. The transformation must be designed to ensure that w'_i is satisfiable if, and only if, there is a truth assignment that satisfies the original clause w_i . The variables added in the transformation of a clause are assumed not to occur elsewhere in u' .

If w_i has three literals, then no transformation is required and $w'_i = w_i$. Let w be a clause of u that does not have three literals. The transformation of w into a 3-conjunctive normal form formula is based on the number of literals in w .

Length 1: $w = v_1$

$$w' = (v_1 \vee y \vee z) \wedge (v_1 \vee \neg y \vee z) \wedge (v_1 \vee y \vee \neg z) \wedge (v_1 \vee \neg y \vee \neg z)$$

Length 2: $w = v_1 \vee v_2$

$$w' = (v_1 \vee v_2 \vee y) \wedge (v_1 \vee v_2 \vee \neg y)$$

Length $n > 3$: $w = v_1 \vee v_2 \vee \dots \vee v_n$

$$\begin{aligned} w' = & (v_1 \vee v_2 \vee y_1) \wedge (v_3 \vee \neg y_1 \vee y_2) \wedge \dots \wedge (v_j \vee \neg y_{j-2} \vee y_{j-1}) \wedge \dots \\ & \wedge (v_{n-2} \vee \neg y_{n-4} \vee y_{n-3}) \wedge (v_{n-1} \vee v_n \vee \neg y_{n-3}) \end{aligned}$$

Establishing the relationship between the satisfiability of clauses of length one and two and their transformations is left as an exercise. Let V be the variables in the clause

$w = v_1 \vee v_2 \vee \dots \vee v_n$ and let t be a truth assignment that satisfies w . Since w is satisfied by t , there is at least one literal satisfied by t . Let v_j be the first such literal. Then the truth assignment

$$t'(x) = \begin{cases} t(x) & \text{if } x \in V \\ 1 & \text{if } x = y_1, \dots, y_{j-2} \\ 0 & \text{if } x = y_{j-1}, \dots, y_{n-3} \end{cases}$$

satisfies w' . The first $j - 2$ clauses are satisfied by literals y_1, \dots, y_{j-2} . The final $n - j + 1$ clauses are satisfied by $\neg y_{j-1}, \dots, \neg y_{n-3}$. The remaining clause, $v_j \vee \neg y_{j-2} \vee y_{j-1}$, is satisfied by v_j .

Conversely, let t' be a truth assignment that satisfies w' . The truth assignment t obtained by restricting t' to V satisfies w . The proof is by contradiction. Assume that t does not satisfy w . Then no literal v_j , $1 \leq j \leq n$, is satisfied by t . Since the first clause of w' has the value 1, it follows that $t'(y_1) = 1$. Now, $t'(y_2) = 1$ since the second clause also has the value 1. Employing the same reasoning, we conclude that $t'(y_k) = 1$ for all $1 \leq k \leq n - 3$. This implies that the final clause of w' has value 0, a contradiction since t' was assumed to satisfy w' .

The transformation of each clause into a 3-conjunctive normal form formula is clearly polynomial in the number of literals in the clause. The work required for the construction of the 3-conjunctive normal form formula is the sum of the work of the transformation of the individual clauses. Thus, the construction is polynomial in the number of clauses in the original form. ■

It is not the case that a subproblem of an NP-complete problem is automatically NP-complete. The 2-Satisfiability Problem, determining whether conjunctive normal form formulas with clauses containing exactly two literals, has a deterministic polynomial-time solution (Exercise 1). Thus 2-satisfiability is not NP-complete unless $P = NP$.

16.3 Reductions from 3-Satisfiability

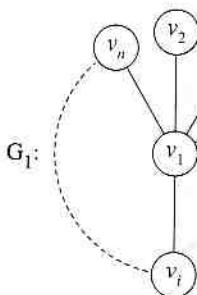
The two problems that we have shown to be NP-complete are both concerned with the satisfaction of logical formulas. In this section we expand the scope of our set of NP-complete problems to include questions about covering sets, paths in graphs, and the accumulation of values. The structure of 3-conjunctive normal form formulas makes them well suited for designing reductions to problems in other domains. In the remainder of this chapter, reductions will be described using high-level representations of the problem instances.

The first problem that we consider is the *Vertex Cover Problem*. A vertex cover of an undirected graph $G = (N, A)$ is a subset VC of N such that for every arc $[u, v]$ in A at least one of u or v is in the set VC . The Vertex Cover Problem can be stated as follows: For an undirected graph G and an integer k , is there a vertex cover of G containing k or fewer

vertices? Example

Example 16.3.1

The arcs of the graph G_2 requires $n/2$ vertices.



Theorem 16.3.1

The Vertex Cover Problem is NP-complete.

Proof. The Vertex Cover Problem is NP-hard. We show that the 3-Satisfiability Problem reduces to the Vertex Cover Problem.

Reduction

3-Satisfiability
to
Vertex Cover Problem

That is, for any 3-Satisfiability instance w , there is a graph G such that G has a vertex cover if and only if w is satisfiable.

Let

be a 3-conjunctive normal form formula over the set $V = \{x_1, x_2, \dots, x_n\}$. The position of a literal x_i in a clause C is the index of the clause and the position of a negated literal $\neg x_i$ is the index of the clause plus the size of the clause. The reduction consists of constructing a graph G such that G has a vertex cover if and only if w is satisfiable.

satisfied
the truth

$-j+1$
 y_{j-1} , is

btained
oes not
 w' has
has the
 $\leq n-3$.
imed to

clearly
truction
ation of
es in the

atically
al form
ial-time

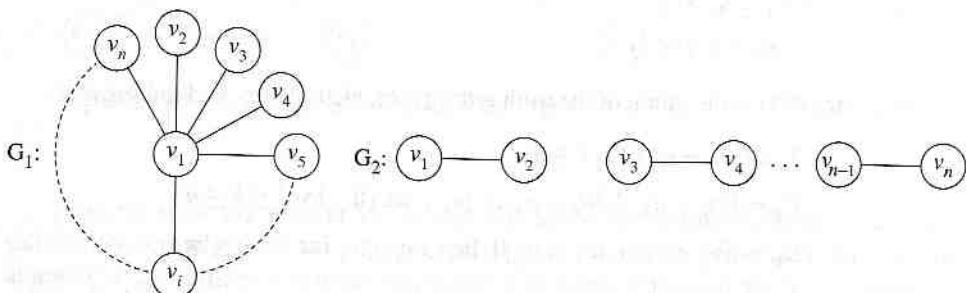
with the
of NP-
and the
es them
inder of
problem

er of an
at least
ws: For
or fewer

vertices? Example 16.3.1 shows that the size of a vertex cover is not necessarily related to the number of nodes or arcs in the graph.

Example 16.3.1

The arcs of the graph G_1 are covered by the single vertex v_1 . The smallest vertex cover of G_2 requires $n/2$ vertices, one for each arc in the graph.



Theorem 16.3.1

The Vertex Cover Problem is NP-complete.

Proof. The Vertex Cover Problem can easily be seen to be in NP . The nondeterministic solution strategy consists of choosing a set of k vertices and determining whether they cover the arcs of the graph. We show that the Vertex Cover Problem is NP-hard by reducing the 3-Satisfiability Problem to it:

Reduction	Input	Condition
3-Satisfiability to Vertex Cover Problem	3-conjunctive normal form formula u undirected graph $G = (N, A)$, integer k	u is satisfiable if, and only if, G has a vertex cover of size k

That is, for any 3-conjunctive normal form formula u , we must construct a graph G so that G has a vertex cover of some predetermined size k if, and only if, u is satisfiable.

Let

$$u = (u_{1,1} \vee u_{1,2} \vee u_{1,3}) \wedge \cdots \wedge (u_{m,1} \vee u_{m,2} \vee u_{m,3})$$

be a 3-conjunctive normal form formula where each $u_{i,j}$, $1 \leq i \leq m$ and $1 \leq j \leq 3$, is a literal over the set $V = \{x_1, \dots, x_n\}$ of Boolean variables. The symbol $u_{i,j}$ is used to indicate the position of a literal in a 3-conjunctive normal form formula; the first subscript indicates the clause and the second subscript the position of the literal in the clause. The reduction consists of constructing a graph G from the 3-conjunctive normal form formula in which the satisfiability of u is equivalent to the existence of a cover of G containing $n + 2m$ vertices.

To transform the question of the existence of a satisfying truth assignment into a question of a vertex cover, we must represent truth assignments and formulas as graphs. Three sets of arcs are introduced to build a graph from a 3-conjunctive normal form formula: the set T of truth setting arcs model truth assignments, the clausal graphs C_k represent the clauses of u , and the linking arcs L_k link the clause graphs with truth values.

The vertices of G consist of the sets

- i) $\{x_i, \neg x_i \mid 1 \leq i \leq n\}$, and
- ii) $\{u_{ij} \mid 1 \leq i \leq m, 1 \leq j \leq 3\}$.

The set of arcs of G is the union of the truth setting arcs, clausal arcs, and linking arcs:

$$\begin{aligned} T &= \{[x_i, \neg x_i] \mid 1 \leq i \leq n\} \\ C_k &= \{[u_{k,1}, u_{k,2}], [u_{k,2}, u_{k,3}], [u_{k,3}, u_{k,1}] \} \quad \text{for } 1 \leq k \leq m \\ L_k &= \{[u_{k,1}, v_{k,1}], [u_{k,2}, v_{k,2}], [u_{k,3}, v_{k,3}] \} \quad \text{for } 1 \leq k \leq m, \end{aligned}$$

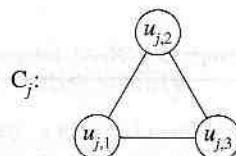
where $v_{k,j}$ is the literal from $\{x_i, \neg x_i \mid 1 \leq i \leq n\}$ that occurs in position $u_{k,j}$ of the formula. We begin by considering the form of the graphs defined by T and C_k and the size of sets needed to cover them.

An arc in T connects a positive literal x_i to its corresponding negative literal $\neg x_i$:



A vertex cover must include one vertex from each pair $x_i, \neg x_i$. At least n vertices are needed to cover the arcs in T . A vertex cover of T with n vertices selects exactly one of x_i or $\neg x_i$. This, in turn, can be considered to define a truth assignment on V .

Each clause $u_{j,1} \vee u_{j,2} \vee u_{j,3}$ generates a subgraph C_j of the form



The subgraph C_j connects the literals $u_{j,1}$, $u_{j,2}$, and $u_{j,3}$. A set of vertices that covers C_j must contain at least two vertices. Thus a cover of the arcs in the set T and the C_k 's must contain at least $n + 2m$ vertices.

The arcs in L_k link the symbols $u_{i,j}$ that indicate the positions of the literals to the corresponding literal x_k or $\neg x_k$ in the formula. Figure 16.1 gives the graph obtained from the formula $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_4)$. It is easy to see that the construction of the graph is polynomially dependent upon the number of variables and clauses in the formula. All that remains is to show that the formula u is satisfiable if, and only if, the associated graph has a cover of size $n + 2m$.

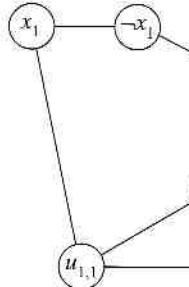


FIGURE 16.1

First, we show that if u is satisfiable, then the graph G satisfies the formula u . Since u is satisfiable, there is a truth assignment t that satisfies the formula. This assignment t selects exactly one vertex from each pair $x_i, \neg x_i$. At least n vertices are needed to cover the arcs in T . A vertex cover of T with n vertices selects exactly one of x_i or $\neg x_i$ from each pair. This, in turn, can be considered to define a truth assignment on V .

That is, the literal x_i is true if and only if $t(x_i) = 1$.

To see that t selects exactly one vertex from each pair $x_i, \neg x_i$, consider a vertex x_i . If $t(x_i) = 1$, then x_i is selected and $\neg x_i$ is not selected. If $t(x_i) = 0$, then $\neg x_i$ is selected and x_i is not selected. In either case, exactly one vertex from the pair $x_i, \neg x_i$ is selected. Therefore, the assignment t selects exactly one vertex from each pair $x_i, \neg x_i$.

Now assume that G has a vertex cover of size $n + 2m$. Let t be the truth assignment defined by the vertex cover. We claim that t is a vertex cover of u . To see this, let C_j be a clause in u . The subgraph C_j consists of three vertices $u_{j,1}, u_{j,2}, u_{j,3}$ arranged in a triangle. If all three vertices $u_{j,1}, u_{j,2}, u_{j,3}$ are covered by the vertex cover, then at least two vertices are covered. If only one vertex is covered, then it must be $u_{j,1}$ because $u_{j,2}$ and $u_{j,3}$ are connected to $u_{j,1}$. In either case, at least two vertices are covered. Therefore, t is a vertex cover of u .

We now return to the problem of determining whether a graph has a vertex cover of size $n + 2m$. This problem has already been shown to be NP-complete (Example 15.5.1) and is known as the 3-Satisfiability problem.

Reduction from 3-Satisfiability to Hamiltonian Circuit

Given a 3-CNF formula u , we want to construct a graph G such that u is satisfiable if and only if G has a Hamiltonian circuit.

ment into a
as graphs.
m formula:
present the

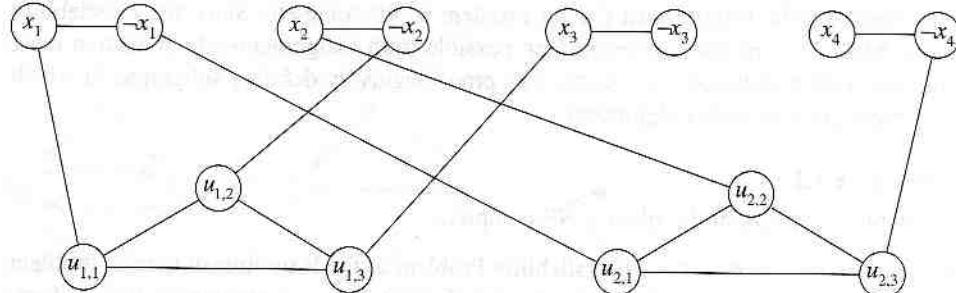


FIGURE 16.1 Graph representing reduction of $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_4)$.

First, we show that a cover VC of size $n + 2m$ defines a truth assignment on V that satisfies the formula u . By the previous remarks, we know that every cover must contain at least $n + 2m$ vertices. Consequently, exactly one vertex from each arc $[x_i, \neg x_i]$ and two vertices from each subgraph C_j are in VC. A truth assignment is obtained from VC by

$$t(x_i) = \begin{cases} 1 & \text{if } x_i \in \text{VC} \\ 0 & \text{otherwise.} \end{cases}$$

That is, the literal from the pair x_i or $\neg x_i$ in the vertex cover is assigned truth value 1 by t .

To see that t satisfies each clause, consider the covering of the subgraph C_j . Only two of the vertices $u_{j,1}, u_{j,2}$, and $u_{j,3}$ can be in VC. Assume $u_{j,k}$ is not in VC. Then the arc $[u_{j,k}, v_{j,k}]$ must be covered by $v_{j,k}$ in VC. This implies that $t(u_{j,k}) = 1$ and the clause is satisfied.

Now assume that $t : V \rightarrow \{0, 1\}$ is a truth assignment that satisfies u . A vertex cover VC of the associated graph can be constructed from the truth assignment. VC contains the vertex x_i if $t(x_i) = 1$ and $\neg x_i$ if $t(x_i) = 0$. Let $u_{j,k}$ be a literal in clause j that is satisfied by t . The arc $[u_{j,k}, v_{j,k}]$ is covered by $v_{j,k}$. Adding the two other vertices of C_j completes the cover. Clearly, $\text{card}(\text{VC}) = n + 2m$, as desired. ■

We now return to our old friend, the Hamiltonian Circuit Problem. This problem has already been shown to be solvable in exponential time by a deterministic machine (Example 15.5.1) and in polynomial time by a nondeterministic machine (Example 15.5.2). A reduction of the form

Reduction	Input	Condition
3-Satisfiability to Hamiltonian Circuit Problem	3-conjunctive normal form formula u ↓ directed graph $G = (N, A)$	u is satisfiable if, and only if, G has a tour

establishes that the Hamiltonian Circuit Problem is NP-complete. Since the satisfiability of a formula is determined by examining possible truth assignments, the reduction must represent truth assignments as graphs. The proof begins by defining subgraphs in which tours correspond to truth assignments.

Theorem 16.3.2

The Hamiltonian Circuit Problem is NP-complete.

Proof. The reduction of the 3-Satisfiability Problem to the Hamiltonian Circuit Problem is accomplished by constructing a directed graph $G(u)$ from a 3-conjunctive normal form formula u . The construction is designed so that the presence of a Hamiltonian circuit in $G(u)$ is equivalent to the satisfiability of u . Let $u = w_1 \wedge w_2 \wedge \dots \wedge w_m$ be a 3-conjunctive normal form formula and $V = \{x_1, x_2, \dots, x_n\}$ be the set of variables occurring in u . The j th clause of u is denoted $u_{j,1} \vee u_{j,2} \vee u_{j,3}$, where each $u_{j,k}$ is a literal over V .

For each variable x_i , let r_i be the larger of the number of occurrences of x_i in u or the number of occurrences of $\neg x_i$ in u . A graph V_i is constructed for each variable x_i as illustrated in Figure 16.2(a). Node e_i is considered the entrance to V_i and o_i the exit. There are precisely two paths through V_i that begin with e_i , end with o_i , and visit each vertex once. These are depicted in Figure 16.2(b) and (c). The arc from e_i to $t_{i,0}$ or $f_{i,0}$ determines the remainder of the path through V_i .

The subgraphs V_i are joined to construct the graph G' depicted in Figure 16.2(d). The two paths through each V_i combine to generate 2^n Hamiltonian circuits through the graph G' . A Hamiltonian circuit in G' represents a truth assignment on V . The value of x_i is specified by the arc from e_i . An arc from e_i to $t_{i,0}$ designates a truth assignment of 1 for x_i . Otherwise, x_i is assigned 0. The graph constructed from the formula

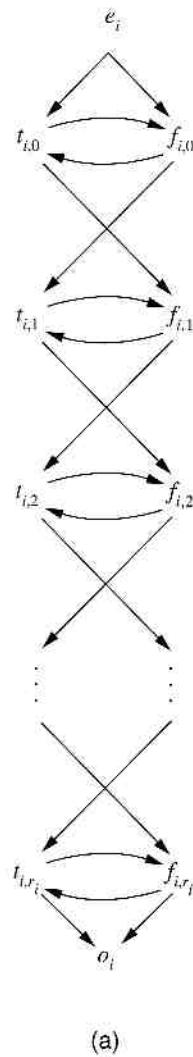
$$(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_4) \wedge (x_1 \vee \neg x_2 \vee x_4) \wedge (\neg x_1 \vee x_3 \vee x_4)$$

is given in Figure 16.3. The tour highlighted by bold arcs in the graph defines the truth assignment $t(x_1) = 1$, $t(x_2) = 0$, $t(x_3) = 0$, and $t(x_4) = 1$. The Hamiltonian circuits of G' encode the possible truth assignments of V . We now augment G' with subgraphs that encode the clauses of the 3-conjunctive form formula.

For each clause w_j , we construct a subgraph C_j that has the form shown in Figure 16.4. The graph $G(u)$ is constructed by connecting these subgraphs to G' as follows:

- i) If x_i is a literal in w_j , then pick some $f_{i,k}$ that has not previously been connected to a graph C. Add an arc from $f_{i,k}$ to a vertex $in_{j,m}$ of C_j that has not already been connected to G' . Then add an arc from $out_{j,m}$ to $t_{i,k+1}$.
 - ii) If $\neg x_i$ is a literal in w_j , then pick some $t_{i,k}$ that has not previously been connected to a graph C. Add an arc from $t_{i,k}$ to a vertex $in_{j,m}$ of C_j that has not already been connected to G' . Then add an arc from $out_{j,m}$ to $f_{i,k+1}$.

The graph in Figure 16.5 is obtained by connecting the subgraph representing the clause $(x_1 \vee x_2 \vee \neg x_3)$ to the graph G' from Figure 16.3.



(a)

A truth assignment σ has a positive literal in the i -th vertex of C_j . Similarly, the i -th vertex of C_j contains a positive literal in the j -th vertex of $G(u)$ when the assignment σ satisfies the condition.

sifiability
ion must
in which

Problem
mal form
circuit in
njunctive
in u . The

x_i in u or
able x_i as
xit. There
ch vertex
etermines

2(d). The
graph G' .
specified
otherwise,

(4)

the truth
uits of G'
at encode

figure 16.4.

ected to a
connected

ected to a
connected

the clause

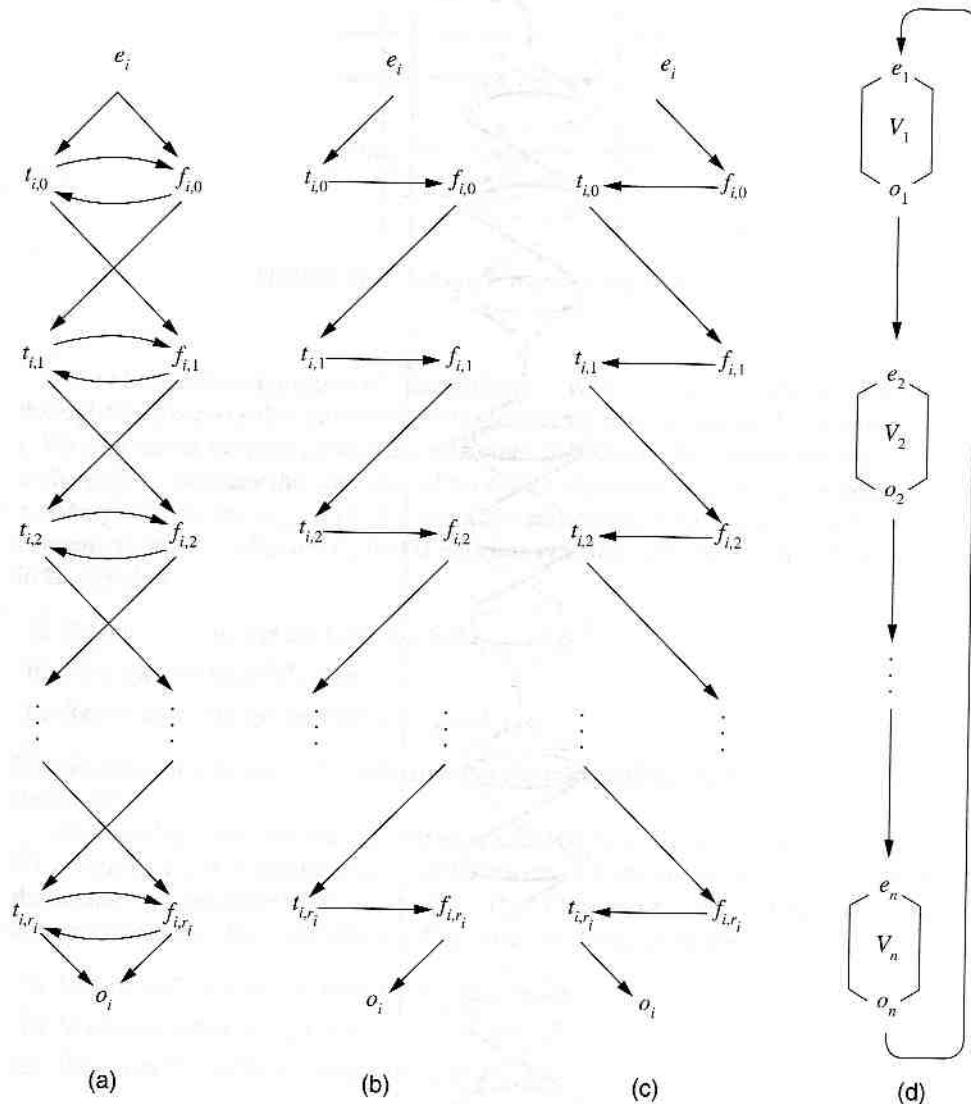


FIGURE 16.2 Subgraph for each variable x_i .

A truth assignment is represented by a Hamiltonian circuit in the graph G' . If x_i is a positive literal in the clause w_j , then there is an arc from some vertex $f_{i,k}$ to one of the m vertices of C_j . Similarly, if $\neg x_i$ is in w_j , then there is an arc from some vertex $t_{i,k}$ to one of the m vertices of C_j . These arcs are used to extend the Hamiltonian circuit in G' to a tour of $G(u)$ when the associated truth assignment satisfies u .

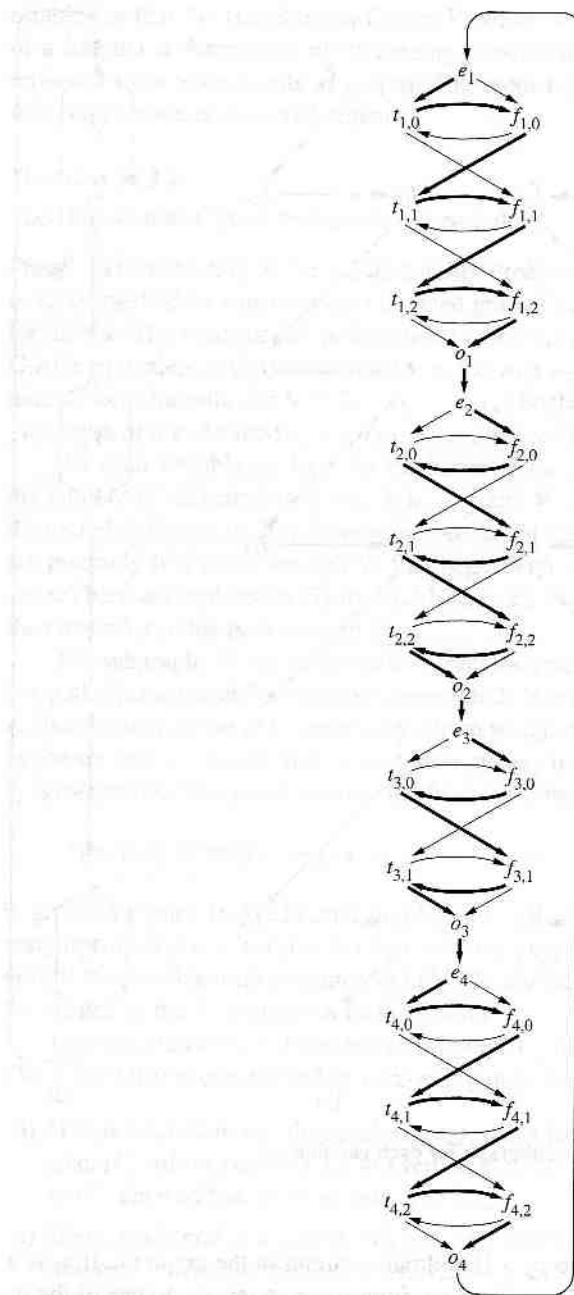


FIGURE 16.3 Truth assignment by Hamiltonian circuit.

Let t be a truth assignment through $G(u)$ based on t . We now detour the path V_i in the path V_i indicated by a node $f_{i,k}$ by an arc that contains an arc to a subgraph in G' as follows:

- i) Detour to C_j via $t_{i,k}$
- ii) Visit each vertex in C_j
- iii) Return to V_i via $t_{i,k}$

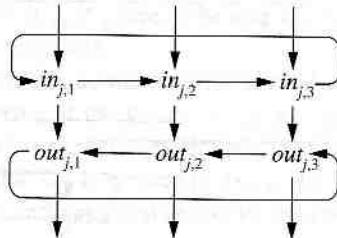
The presence of a detour indicates the clause w_j .

On the other hand, if $t(x_i) = 0$, all the vertices $t_{i,k}$ are either connected to one of the

- i) Detour to C_j via $t_{i,k}$
- ii) Visit each vertex in C_j
- iii) Return to V_i via $t_{i,k}$

Since each clause is satisfied, the tour visits each subgraph satisfying truth assignment t .

Now assume that G' is satisfiable. The Hamiltonian

FIGURE 16.4 Subgraph representing clause w_j .

Let t be a truth assignment on V that satisfies u . We will construct a Hamiltonian circuit through $G(u)$ based on the values of t . We begin with the tour through the V_i 's that represents t . We now detour the path through the subgraphs that encode the clauses. An arc $[t_{i,k}, f_{i,k}]$ in the path V_i indicates that the value of the truth assignment $t(x_i) = 1$. If the path reaches a node $f_{i,k}$ by an arc $[t_{i,k}, f_{i,k}]$, $f_{i,k}$ is not already connected to a clause graph, and $f_{i,k}$ contains an arc to a subgraph C_j that is not already in the path, then connect C_j to the tour in G' as follows:

- Detour to C_j via the arc from $f_{i,k}$ to $in_{j,m}$ in C_j .
- Visit each vertex of C_j once.
- Return to V_i via the arc from $out_{j,m}$ to $t_{i,k+1}$.

The presence of a detour to C_j indicates that the truth assignment encoded in G' satisfies the clause w_j .

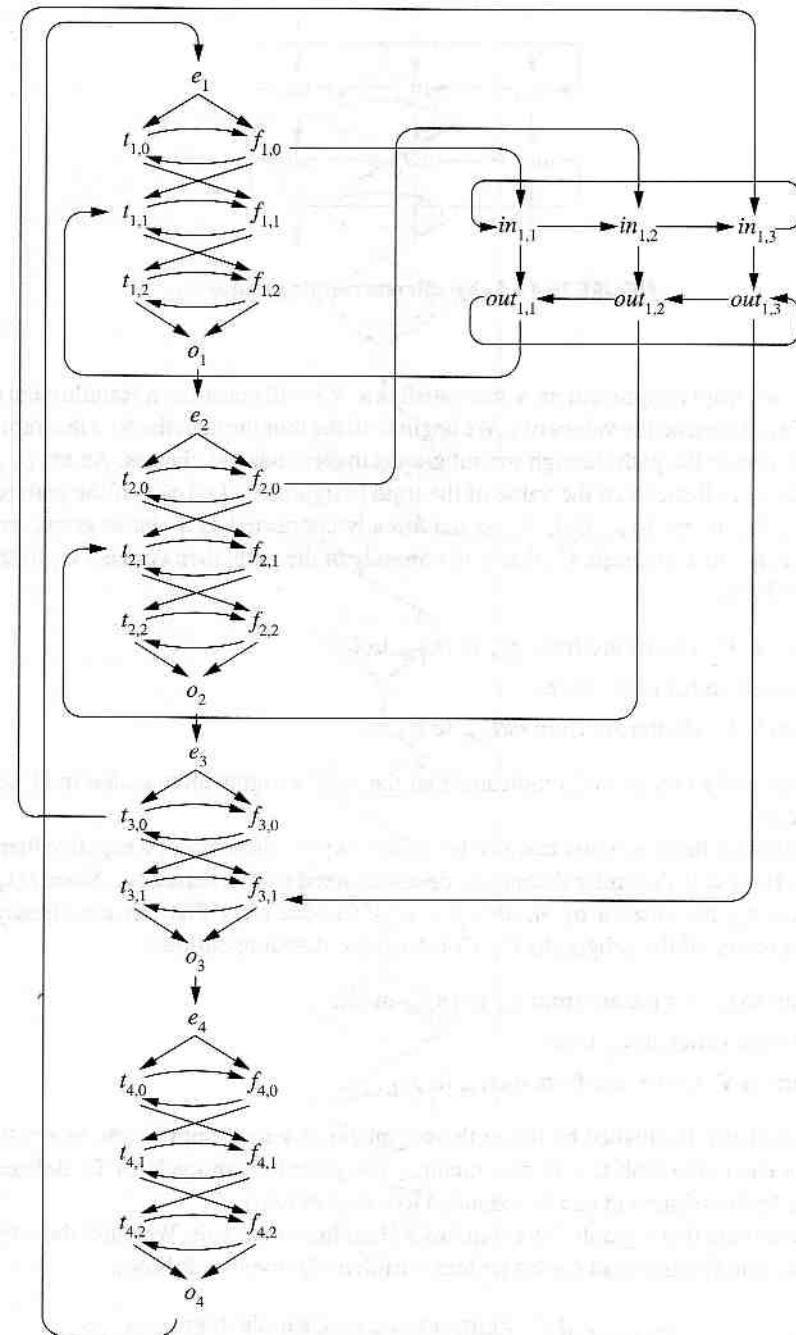
On the other hand, a clause can also be satisfied by the presence of a negative literal $\neg x_i$ for which $t(x_i) = 0$. A similar detour can be constructed from a vertex $t_{i,k}$. Since $t(x_i) = 0$, the vertices $t_{i,k}$ are entered by an arc $[f_{i,k}, t_{i,k}]$. Choose a $t_{i,k}$ that has not already been connected to one of the subgraphs C_j . Construct the detour as follows:

- Detour to C_j via the arc from $t_{i,k}$ to $in_{j,m}$ in C_j .
- Visit each vertex in C_j once.
- Return to V_i via the arc from $out_{j,m}$ to $f_{i,k+1}$.

Since each clause is satisfied by the truth assignment, a detour from G' can be constructed that visits each subgraph C_j . In this manner, the Hamiltonian cycle of G' defined by a satisfying truth assignment can be extended to a tour of $G(u)$.

Now assume that a graph $G(u)$ contains a Hamiltonian circuit. We must show that u is satisfiable. The Hamiltonian circuit defines a truth assignment as follows:

$$t(x_i) = \begin{cases} 1 & \text{if the arc } [e_i, t_{i,0}] \text{ is in the tour} \\ 0 & \text{if the arc } [e_i, f_{i,0}] \text{ is in the tour.} \end{cases}$$

FIGURE 16.5 Connection of C_1 to G' .

If $t(x_i) = 1$, then all the arcs $[f_{i,k}, t_{i,k}]$ v

Before proving the subgraph C_j . U six vertices in C_j . subpath of a tour. A subpaths of a tour be twice.

Pa

in_j
 in_j
 in_j
 in_j

Thus the only paths same property holds.

Each of the C_j 's from a vertex $f_{i,k}$, to the arc $[f_{i,k}, in_{j,m}]$, x_i . Moreover, when the arc $[t_{i,k}, f_{i,k}]$. O we conclude that $t([t_{i,k}, in_{j,m}])$, then \rightarrow

Combining the Hamiltonian circuit that remains is to show in the formula u . The number of occurrences adds 6 vertices and

Many problems forth. The final problem accumulation or assignment example of such a problem intention of spending total cost will be examined. S, a value function v

If $t(x_i) = 1$, then all of the arcs $[t_{i,k}, f_{i,k}]$ are in the tour. On the other hand, the tour contains the arcs $[f_{i,k}, t_{i,k}]$ whenever $t(x_i) = 0$.

Before proving that t satisfies u , we examine several properties of a tour that enters the subgraph C_j . Upon entering at the vertex $in_{j,m}$, the path may visit two, four, or all six vertices in C_j . A path that exits C_j at any position other than $out_{j,m}$ cannot be a subpath of a tour. Assume that C_j is entered at $in_{j,1}$; the following paths in C_j are not subpaths of a tour because the vertices listed cannot be reached without visiting some vertex twice.

Path	Unreachable Vertices
$in_{j,1}, in_{j,2}, in_{j,3}, out_{j,3}$	$out_{j,2}, out_{j,1}$
$in_{j,1}, in_{j,2}, in_{j,3}, out_{j,3}, out_{j,2}$	$out_{j,1}$
$in_{j,1}, in_{j,2}, out_{j,2}$	$out_{j,1}$
$in_{j,1}, in_{j,2}, out_{j,2}, out_{j,1}, out_{j,3}$	$in_{j,3}$

Thus the only paths entering C_j at $in_{j,1}$ that are subpaths of tours must exit at $out_{j,1}$. The same property holds for $in_{j,2}$ and $in_{j,3}$.

Each of the C_j 's must be entered by the tour. If C_j is entered at vertex $in_{j,m}$ by an arc from a vertex $f_{i,k}$, then the tour exits C_j via the arc from $out_{j,m}$ to $t_{i,k+1}$. The presence of the arc $[f_{i,k}, in_{j,m}]$ in $G(u)$ indicates that w_j , the clause encoded by C_j , contains the literal x_i . Moreover, when C_j is entered by an arc $[f_{i,k}, in_{j,m}]$, the vertex $f_{i,k}$ must be entered by the arc $[t_{i,k}, f_{i,k}]$. Otherwise, the vertex $t_{i,k}$ is not in the tour. Since $[t_{i,k}, f_{i,k}]$ is in the tour, we conclude that $t(x_i) = 1$. Thus, w_j is satisfied by t . Similarly, if C_j is entered by an arc $[t_{i,k}, in_{j,m}]$, then $\neg x_i$ is in w_j and $t(x_i) = 0$.

Combining the previous observations, we see that the truth assignment generated by a Hamiltonian circuit through $G(u)$ satisfies each of the clauses of u and hence u itself. All that remains is to show that the construction of $G(u)$ is polynomial in the number of literals in the formula u . The number of vertices and arcs in a subgraph V_i increases linearly with the number of occurrences of the variable x_i in u . For each clause, the construction of C_j adds 6 vertices and 15 arcs to $G(u)$. ■

Many problems associate numeric values with objects: costs, weights, worth, and so forth. The final problem that we consider in this section shows that problems dealing with the accumulation or assessment of a set of numeric values can be NP-complete. A whimsical example of such a problem is posed by a person who goes on shopping spree with the intention of spending every cent that he has. The question: Is there a set of objects whose total cost will be exactly the amount of money in his possession? The *Subset-Sum Problem* formalizes the preceding example. An instance of the Subset-Sum Problem consists of a set S , a value function $v : S \rightarrow \mathbb{N}$, and an integer k . The answer is positive if there is a subset

$S' \subseteq S$ such that the sum of the values of all the elements in S' is k . For simplicity, we will let $v(A)$ denote the total of the values of the elements in a set A .

The Subset-Sum Problem clearly is in NP . A nondeterministic guess selects a subset of S . The remainder of the computation adds the values of the items in the subset and compares the total with the value k given in the problem definition. All that remains is to show that the Subset-Sum Problem is NP-hard.

Theorem 16.3.3

The Subset-Sum Problem is NP-complete.

Proof. A reduction of the 3-Satisfiability Problem to the Subset-Sum Problem has the form

Reduction	Input	Condition
3-Satisfiability to Subset-Sum Problem	3-conjunctive normal form formula u \downarrow set S , function $v : S \rightarrow \mathbb{N}$, integer k	u is satisfiable if, and only if, there is a subset $S' \subseteq S$ with $v(S') = k$

We need to construct a set S , a value function v on S , and an integer k from a 3-conjunctive normal form formula u such that S has a subset with total value k if, and only if, u is satisfiable. As in the previous problems, we let $u = w_1 \wedge w_2 \wedge \dots \wedge w_m$ be a 3-conjunctive normal form formula with $V = \{x_1, x_2, \dots, x_n\}$ the set of variables in u .

The set S consists of the items

- i) $x_i, i = 1, \dots, n$,
- ii) $\neg x_i, i = 1, \dots, n$,
- iii) $y_j, j = 1, \dots, m$, and
- iv) $y'_j, j = 1, \dots, m$.

Thus S has $2n + 2m$ objects. We must now assign a value to every object in S . Each value will be an integer with $n + m$ digits. The rules for assigning the values are

- x_i : the i th digit from the right is 3,
if x_i is in clause w_j , then the $n + j$ th digit from the right is 1,
all other digits are 0,
- $\neg x_i$: the same construction as x_i ,
- y_j : the $n + j$ th digit from the right is 1,
all other digits are 0,
- $\neg y_j$: the same as w_j .

The integer k has no manner from a 3-c

To appreciate values assigned to

The $m + n$ positions in the first $2n$ rows are used to describe truth in a clause.

When x_i occurs the literals x_i and $\neg x_i$

$x_i :$ w_m
 $\neg x_i :$ $-$

The occurrence of the x_i indicates that x_i is true. The 0 in the $\neg x_i$, w_j

Before proving the 3-Satisfiability Problem to the Subset-Sum Problem generated from

$u :$

The integer k has $m + n$ digits all of which are 3. The Subset-Sum Problem obtained in this manner from a 3-conjunctive form formula u will be called $S(u)$.

To appreciate the motivation behind this construction we consider the digits in the values assigned to the objects to be entries in a $2n + 2m$ by $m + n$ table:

	w_m	...	w_1	x_n	...	x_2	x_1
x_1	-	...	-	0	...	0	3
$\neg x_1$	-	...	-	0	...	0	3
x_2	-	...	-	0	-	3	0
$\neg x_2$	-	...	-	0	...	3	0
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
x_n	-	...	-	3	...	0	0
$\neg x_n$	-	...	-	3	...	0	0
y_1	0	...	1	0	...	0	0
y'_1	0	...	1	0	...	0	0
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
y_2	1	...	0	0	...	0	0
y'_2	1	...	0	0	...	0	0

The $m + n$ positions in each value correspond to the $n + m$ columns of the table. The entries in the first $2n$ rows contain the values assigned to the literals. The rightmost n columns are used to describe truth assignments. The leftmost m columns indicate whether a literal occurs in a clause.

When x_i occurs in a clause w_j and $\neg x_i$ does not, the rows of the table associated with the literals x_i and $\neg x_i$ have the form

	w_m	...	w_j	...	w_1	x_n	...	x_i	...	x_1
$x_i :$	-	...	1	...	0	0	...	3	...	0
$\neg x_i :$	-	...	0	...	0	0	...	3	...	0

The occurrence of the 1 in the column associated with clause w_j and row corresponding to x_i indicates that x_i occurs in the clause and, consequently, that w_j is satisfied if $t(x_i) = 1$. The 0 in the $\neg x_i, w_j$ position indicates that $\neg x_i$ does not occur in w_j .

Before proving that the preceding construction is a reduction of the 3-Satisfiability Problem to the Subset-Sum Problem, we will consider the instance of the Subset-Sum Problem generated from the 3-conjunctive normal form formula

$$u = w_1 \wedge w_2 = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee \neg x_4).$$

The corresponding set S is $\{x_1, x_2, x_3, x_4, -x_1, -x_2, -x_3, -x_4, y_1, y_1', y_2, y_2'\}$ and the values assigned to each of the objects of S , given in tabular form, are

	w_2	w_1	x_4	x_3	x_2	x_1	Values
x_1	0	1	0	0	0	3	$v(x_1) = 010003$
$\neg x_1$	1	0	0	0	0	3	$v(\neg x_1) = 100003$
x_2	0	1	0	0	3	0	$v(x_2) = 010030$
$\neg x_2$	0	0	0	0	3	0	$v(\neg x_2) = 000030$
x_3	1	0	0	3	0	0	$v(x_3) = 100300$
$\neg x_3$	0	1	0	3	0	0	$v(\neg x_3) = 010300$
x_4	0	0	3	0	0	0	$v(x_4) = 003000$
$\neg x_4$	1	0	3	0	0	0	$v(\neg x_4) = 103000$
y_1	0	1	0	0	0	0	$v(y_1) = 010000$
y'_1	0	1	0	0	0	0	$v(y'_1) = 010000$
y_2	1	0	0	0	0	0	$v(y_2) = 100000$
y'_2	1	0	0	0	0	0	$v(y'_2) = 100000$

By our definition of $S(u)$, $k = 333333$.

The formula u is satisfied by the truth assignment $t(x_1) = 1, t(x_2) = 1, t(x_3) = 0$, and $t(x_4) = 0$. The literals satisfied by the truth assignment, $x_1, x_2, \neg x_3$, and $\neg x_4$, along with y_2 and y'_2 , form a subset that affirmatively answers the Subset-Sum Problem. That is, the sum of the values of these elements is k . The example exhibits the role of the y_i 's and y'_i 's in the set S . When a clause w_j is satisfied by only one or two of its literals, these objects can be added to the set to bring the sum of the column associated with w_j to 3.

The values in the table show that the sum of the digits in a column labeled by a clause is five and in a column labeled by a variable is six. Thus there are no carries and no interaction between columns when adding the values of the objects in any subset of S .

First, we show that if a 3-conjunctive normal form formula u is satisfiable, then $S(u)$ has a subset whose objects have a total value of k . Let $t : V \rightarrow \{0, 1\}$ be a truth assignment that satisfies u . We will build the subset S' from the truth assignment. Initially, S' contains one of x_i or $\neg x_i$ for each variable x_i ; x_i if $t(x_i) = 1$ and $\neg x_i$ if $t(x_i) = 0$. Since each x_i occurs exactly once in this set, either as a positive or a negative literal, the rightmost n digits in the sum of the values of these objects are all 3.

Each clause w_j must be satisfied by some literal. This literal has a 1 in the column associated with w_j . Thus the sum of the digits in the w_j column from the rows corresponding to literals in the truth assignment is at least 1 and at most 3. If the sum is 1, we add y_j and y'_j to S' . If the sum is 2, we simply add y'_j to S' . After the potential addition of y_j and y'_j , the sum of the digits in the w_j column becomes 3, as desired.

Now let $S(u)$ be an instance of the Subset-Sum Problem obtained by the preceding construction that has a subset S' whose sum is k . We must show that u is satisfiable. First note that one, but not both, of x or $\neg x$, is in S' . If neither are in the set, the sum of the values

of the objects in S ,
the set, the sum has
truth assignment:

For each clause w_j ,
total can include a
1 in the w_j column

The construction
and each clause get

16.4 Reduct

Each of the reductions maps an unrelated domain into one related to the analysis, and to the analysis of the second. Such a transformation has been seen one example of satisfiability. The domain and the reduction itself

The most common similar problem among them is that the more similar the two problems are, ideally we show that one can be reduced to the other. This is a subproblem of the general problem of reduction of one problem to another. We have previously shown that the reduction can be carried out by using an essential component of the problem and the

The Partition P

Partition
Input:
Output

of the objects in S' has a 0 in the i th position from the right digit. If both x_i and $\neg x_i$ are in the set, the sum has a 6 in that position. Thus the occurrences of the literals in S' define a truth assignment:

$$t(x_i) = \begin{cases} 1 & \text{if } x_i \in S' \\ 0 & \text{otherwise.} \end{cases}$$

For each clause w_j , the sum of the values of the objects in S' in the w_j column is three. This total can include a maximum of two from y_j and y'_j . Thus there must be a literal that has a 1 in the w_j column and this literal satisfies the clause w_j .

The construction of $S(u)$ is clearly polynomial in the length of u since each variable and each clause generate two objects of S . ■

16.4 Reduction and Subproblems

Each of the reductions in the previous section transformed problems from one domain to an unrelated domain: 3-conjunctive form formulas to vertex covers, to path generation, and to the analysis of the values of sets of objects. A reduction between domains requires the ability to reconfigure problems from the first domain as equivalent problems in the second. Such a transformation is not always obvious or straightforward. Fortunately, the vast majority of NP-completeness proofs do not require a change in domains. We have already seen one example of a reduction between problems in the same domain—satisfiability to 3-satisfiability. The domain of both of these problems is the satisfaction of Boolean formulas and the reduction simply transformed formulas to formulas.

The most common technique for showing that a problem is NP-complete is to find a similar problem among the thousands of known NP-complete problems. The rule of thumb is that the more similar the problems, the less work that is likely to be involved in the reduction. Ideally we show that a problem P is NP-hard by finding an NP-complete problem Q that is a subproblem of P or one in which the instances of Q can easily be transformed into instances of P . This strategy will be demonstrated using reductions from problems that we have previously shown to be NP-complete. The proofs will include neither the design of a nondeterministic algorithm that solves the problem in polynomial time nor an argument that the reduction can be accomplished in polynomial time. The satisfaction of both of these essential components of an NP-completeness proof will be obvious from the definition of the problem and the transformation involved in the reduction.

The Partition Problem

Partition Problem

Input: Set A , value function $v : A \rightarrow \mathbb{N}$

Output: yes; if there is a subset A' of A such that $v(A') = v(A - A')$
no; otherwise

asks if the elements of a set can be divided into two disjoint subsets of equal value. The result of both the Partition Problem and the Subset-Sum Problem are determined by the existence of a set of objects with a predetermined total value. Using this similarity, we will show that the Partition Problem is NP-complete by reducing the Subset-Sum Problem to it.

Theorem 16.4.1

The Partition Problem is NP-complete.

Proof. A reduction of the Subset-Sum Problem to the Partition Problem

Reduction	Input	Condition
Subset-Sum Problem	set S , function $v : S \rightarrow N$, integer k	there is a subset $S' \subseteq S$ with $v(S') = k$
to Partition Problem	set A , function $v' : A \rightarrow N$	if, and only if, there is a subset $A' \subseteq A$ with $v'(A') = v'(A - A')$

requires the construction of a set A and a value function v' from the components S , v , and k of an instance of the Subset-Sum Problem. The set A and value function v' are defined by

$$\begin{aligned} A &= S \cup \{y, z\} \\ v'(x) &= 2v(x) \text{ for all } x \in S \\ v'(y) &= 3t - 2k \\ v'(z) &= t + 2k, \end{aligned}$$

where $t = v(S)$ is the sum of the values of all the elements in the set S . The sole reason for the multiplication of $v(x)$ by two is to ensure that the total value of the set A is even and consequently a partition is possible. The total value of all the elements in the set A , using the value function v' , is $2t + (3t - 2k) + (t + 2k) = 6t$.

First, we show that we can construct a solution to the Partition Problem from a solution S' to the Subset-Sum Problem. Since S' is a solution, we know that

$$v(S') = \sum_{x \in S'} v(x) = k.$$

Defining A' to

which is one-h

Now assu
partitioning sub
 S' whose elem

The elem
 $v'(y) + v'(z) =$
sets, assume tha

Now $X - \{y\}$ is
the Subset-Sum

Consider th
representative o
to any number o
form a council t
Set Problem. Fo
collection $C = \{$
if $C \cap C_i \neq \emptyset$ fo
an affirmative an

Instead of a
This interpretation
We will reduce t
latter is NP-comp

Theorem 16.4.2

The Hitting Set P

Proof. An insta
A), k of the Verte
of G . Each arc $[n]$

value. The
ined by the
nilarity, we
m Problem

Defining A' to be the set $S' \cup \{y\}$, we get

$$\begin{aligned} v'(A') &= \sum_{a \in A'} v'(a) \\ &= v'(y) + \sum_{x \in S'} v'(x) \\ &= 3t - 2k + 2k \\ &= 3t, \end{aligned}$$

which is one-half of the total value of A . Thus A' is a solution to the Partition Problem.

Now assume that A and v' are obtained by a reduction from S , v , and k and that A has partitioning subsets X and Y with $v'(X) = v'(Y) = 3t$. We must show that there is a subset S' whose elements have total value k .

The elements y and z cannot belong to the same set in the partition of A , since the value $v'(y) + v'(z) = 4t$ is greater than half of the total value of A . The element y is in one of the sets, assume that it is X . Then

$$\begin{aligned} v'(X - \{y\}) &= v'(X) - v'(y) \\ &= 3t - (3t - 2k) \\ &= 2k. \end{aligned}$$

Now $X - \{y\}$ is a subset of S and its value $v(X - \{y\}) = k$. Thus $X - \{y\}$ is a solution to the Subset-Sum Problem. ■

Consider the dilemma of a school principal who wants to form a council with a representative of every club in the school. There are 15 clubs and a student may belong to any number of clubs. The principal wants the council to have only 10 members. Can he form a council that satisfies his requirements? This question is an example of the *Hitting Set Problem*. Formally, an instance of the Hitting Set Problem consists of a set S , a finite collection $\mathcal{C} = \{C_1, \dots, C_n\}$ of subsets of S , and an integer k . A set C is a hitting set of \mathcal{C} if $C \cap C_i \neq \emptyset$ for each C_i . That is, every set C_i is hit by an element of C . The problem has an affirmative answer if there is a hitting set of size k or less.

Instead of an element of C hitting a set C_i , we may think of the element as covering C_i . This interpretation reveals the similarity between the Vertex Cover and Hitting Set problems. We will reduce the Vertex Cover Problem to the Hitting Set Problem and conclude that the latter is NP-complete.

Theorem 16.4.2

The Hitting Set Problem is NP-complete.

Proof. An instance of the Hitting Set Problem can be obtained from an instance $G = (N, A)$, k of the Vertex Cover Problem in the following manner. The elements of S are the nodes of G . Each arc $[n_i, n_j]$ defines a two element set $\{n_i, n_j\}$. The class \mathcal{C} consists of all of the

two element sets obtained from the arcs of G . Finally, the integer k is the same for both problems. Now we show that G has a vertex cover of size k if, and only if, there is a hitting set of the associated class \mathcal{C} of size k or less.

Assume that there is a vertex cover VC of size k . This set is a hitting set of \mathcal{C} of the appropriate size. Conversely, assume that \mathcal{C} has a hitting set C of size k or less. Then each set $\{n_i, n_j\} \in \mathcal{C}$ is hit by an element of C . In terms of the graph G , every arc $[n_i, n_j]$ is covered by a vertex from C . Thus C is a vertex cover of size at most k . ■

With the interpretation of arcs as two element sets and covering as hitting, the Vertex Cover Problem becomes a subproblem of the Hitting Set Problem. The ability to interpret a known NP-complete problem as a subproblem of the problem under consideration often makes the ensuing NP-completeness proof almost trivial. The *Bin-Packing Problem*

Bin Packing Problem

Input: Set A , a size function $s : A \rightarrow \mathbb{N}$, positive integers k and m

Output: yes; if there is a partition A_1, A_2, \dots, A_k of A such that $s(A_i) \leq m$ for $1 \leq i \leq k$
no; otherwise

provides another example of this phenomenon. We show that the Partition Problem can be easily transformed into a subproblem of bin packing.

Theorem 16.4.3

The Bin Packing Problem is NP-complete.

Proof. The reduction has the form

Reduction	Input	Condition
Partition Problem	set A , function $v : A \rightarrow \mathbb{N}$	there is a subset $A' \subseteq A$ with $v(A') = v(A - A')$ if, and only if,
to Bin Packing Problem	set A , function $s = v$ integers k and m	there is a partition A_1, A_2, \dots, A_k with $s(A_i) \leq m$ for all i

As indicated in the description of the reduction, the same set and function are used for both problems. What remains is to select integers k and m for the Bin Packing Problem. Since the Partition Problem attempts to divide A into two equally valued subsets, we let $k = 2$. The reduction is complete by setting $m = s(A)/2$, one half of the total value of all the elements in A .

This reduction identifies the Partition Problem as bin packing limited to two bins with maximum capacity $s(A)/2$. If a set $A' \subseteq A$ satisfies the Partition Problem, then A' and $A - A'$ constitute a partition that satisfies the Bin Packing Problem. Conversely, a solution A_1, A_2 to the Bin Packing Problem with capacity bound $s(A)/2$ is a solution to the Partition Problem. ■

16.5 Optimization Problems

There are many problems in computer science that require us to find an optimal solution. For example, given a set of cities, what is the shortest distance from one city to another? Given a set of cities and a set of roads connecting them, what is the minimal cost tour that visits every city? Given a set of cities and the weights of the roads between them, what is the shortest distance from one city to another?

We will use the Traveling Salesman Problem as a general example of an optimization problem. The Traveling Salesman Problem is a generalization of the minimal cost tour problem. Given a set of cities and the weights of the roads between them, what is the shortest distance from one city to another?

The Traveling Salesman Problem is NP-hard. The Traveling Salesman Problem is a generalization of the minimal cost tour problem. Given a set of cities and the weights of the roads between them, what is the shortest distance from one city to another?

Placing the bound on the number of cities or no response.

A solution to the Traveling Salesman Problem is a tour that visits every city exactly once and returns to the starting city. The cost of the tour is the sum of the weights of the edges in the tour. The Traveling Salesman Problem is NP-hard because it is a generalization of the minimal cost tour problem. Given a set of cities and the weights of the roads between them, what is the shortest distance from one city to another?

until an affirmative response. In the last section, we saw how to solve the Traveling Salesman Problem using dynamic programming. The Traveling Salesman Problem is NP-hard because it is a generalization of the minimal cost tour problem. Given a set of cities and the weights of the roads between them, what is the shortest distance from one city to another?

Theorem 16.5.1

The Traveling Salesman Problem is NP-hard because it is a generalization of the minimal cost tour problem. Given a set of cities and the weights of the roads between them, what is the shortest distance from one city to another?

16.5 Optimization Problems

There are many problems in which the goal is not just to determine whether a solution exists, but to find an optimal solution. An optimal solution may minimize the cost, maximize the value, most efficiently utilize resources, and so forth. Since the result is not a yes or no answer, an optimization problem does not match our definition of a decision problem. However, the complexity issues that we have considered for decision problems are equally pertinent to optimization problems.

We will use the Traveling Salesman Problem to illustrate the technique employed for establishing the NP-completeness of an optimization problem. The Traveling Salesman Problem is a generalization of the Hamiltonian Circuit Problem that seeks to find the minimal cost tour of a weighted directed graph, where the cost of a path is the sum of the weights of the arcs in the path. The name of the problem describes the situation of a salesman who wishes to visit every town on his route exactly once, and do so while traveling the shortest distance possible.

The Traveling Salesman Problem can be converted to a decision problem by adding a distance bound to the problem instances:

Traveling Salesman Decision Problem

Input: Weighted directed graph $G = (N, A, w)$, integer k

Output: yes; if G has a tour of cost less than or equal to k
no; otherwise.

Placing the bound k on the cost of the tour changes the desired answer from a path to a yes or no response.

A solution to the decision problem can be iteratively employed to produce a solution to the original optimization problem. Let n be the number of nodes of G , l be the sum of the cost of the n arcs with the least cost, and u the sum of the n highest cost arcs. The cost of any tour of G must be between l and u . The cost of the least-cost tour can be obtained by iteratively solving the sequence of decision problems

$$G = (N, A, w), k = l$$

$$G = (N, A, w), k = l + 1$$

$$G = (N, A, w), k = l + 2$$

⋮

$$G = (N, A, w), k = u$$

until an affirmative answer is produced or all the problem instances have returned negative responses. In the latter case, there is no tour of the graph.

Theorem 16.5.1

The Traveling Salesman Problem is NP-complete.

Proof. The Hamiltonian Circuit Problem can be considered to be a subproblem of the Traveling Salesman Problem. Let $G = (N, A)$ be an instance of the Hamiltonian Circuit Problem. To obtain an instance of the Traveling Salesman Problem we need only define a weight function w and bound k for G . Let w assign the value 1 to each arc and let k be the number of nodes of G . The graph G has a tour if, and only if, the corresponding weighted directed graph (N, A, w) has a tour of cost k . ■

The *Knapsack Problem* is a classic optimization problem concerned with selecting a set of objects of maximal value subject to a size constraint. The most colorful description of this problem describes the plight of a burglar who must decide which items to put in his knapsack. His objective is to maximize the value of the objects, but his selection is constrained by the size of the knapsack. The decision problem version of the Knapsack Problem is

Knapsack Decision Problem

Input: Set S , size function $s : S \rightarrow N$, value function $v : S \rightarrow N$,

size bound b , minimal value m

Output: yes; if there is a subset of $S' \subseteq S$ with $s(S') \leq b$ and $v(S') \geq m$,
no; otherwise.

Theorem 16.5.2

The Knapsack Problem is NP-complete.

Proof. The reduction

Reduction	Input	Condition
Partition Problem to Knapsack Problem	set A , function $v : A \rightarrow N$ ↓ set A , function $s = v$, v , integers b and m	there is a subset $A' \subseteq A$ with $v(A') = v(A - A')$ if, and only if, there is a subset A' with $s(A') \leq b$, $v(A') \geq m$

creates a Knapsack Problem with the same domain as the Partition Problem. The value and size functions of the Knapsack Problem are both set to the value function of the Partition Problem. The reduction is completed by defining b and m as $s(A)/2$. Because of the identification of the size and value functions of the Knapsack Problem with the size function of the Partition Problem, a set A' satisfies the requirement of the Partition Problem if, and only if, it satisfies the requirements of the corresponding Knapsack Problem. ■

16.6 Appro

The significance arise naturally in combinatorics, no complete does not that a polynomial-

We will consider the deliberations of his route. The cities graph $G = (N, A)$, city x to city y . The time he must produce once and return home the Traveling Salesman problem of determining

The first step is to consider his particular situation algorithms that solve the problem instance directly the application of the

If the algorithm be worthwhile to implement the search techniques by the Turing machine requires examining all algorithms can be used to be considered. A time of $O(n^2^n)$. Although the exhaustive search

The next step, is another problem that may not be optimal approach, the salesmen a route that begins west in an east-to-west direction strictly west-to-east

The motivation should not contain 1 good approximation but the two-directional a_4 to a_{12} can be con-

16.6 Approximation Algorithms

The significance of the class NP is not theoretical, but practical. NP-complete problems arise naturally in many areas including pattern recognition, scheduling, decision analysis, combinatorics, network design, and graph theory. Determining that a problem is NP-complete does not mean that solutions are no longer needed, only that it is quite unlikely that a polynomial-time algorithm will be found to produce them.

We will consider the process of dealing with a problem that is NP-complete through the deliberations of a salesman who wishes to automate the process by which he determines his route. The cities and roads are represented by the nodes and arcs of a weighted directed graph $G = (N, A, w)$, and the weight function $w(x, y)$ gives the distance of the road from city x to city y . The cities that the salesman must visit are subject to change, at which time he must produce a new route. His objective, of course, is to visit every city exactly once and return home while spending as little time traveling as possible. Knowing that the Traveling Salesman Problem is NP-complete, how should the salesman approach the problem of determining his route?

The first step is to decide whether the NP-completeness of the problem is relevant for his particular situation. If the route contains only a few cities, the asymptotic performance of algorithms that solve the problem is immaterial. The number of nodes that constitute a small problem instance depends upon the computational resources available and the frequency of the application of the algorithm.

If the algorithm is used frequently, even with a relatively small number of nodes, it may be worthwhile to investigate the use of techniques from the theory of algorithms to refine the search technique. Exhaustive testing of all sequences of nodes, the strategy employed by the Turing machine in Example 15.5.1 that solves the Hamiltonian Circuit Problem, requires examining n^{n-1} potential paths where n is the number of cities. Branch-and-bound algorithms can be used to prune the search tree and reduce the number of paths that need to be considered. A dynamic programming algorithm produces minimum-distance tours in $O(n2^n)$ time. Although still exponential, this is a considerable reduction in complexity from the exhaustive search strategy.

The next step, if needed, is for the salesman to consider reformulating the problem as another problem that can be solved in polynomial time. The solutions to the new problem may not be optimal tours, but they may be acceptable for his purposes. Following this approach, the salesman marks all the cities that he must visit on a map and decides to design a route that begins with the farthest east city and goes to the farthest west city traveling solely in an east-to-west direction. The tour is completed by returning to the original city using a strictly west-to-east route.

The motivation for an east-to-west strategy is that a short route from the two cities should not contain legs that move away from the goal. While this method often produces good approximations, Figure 16.6 gives a graph in which the optimal tour has distance 82, but the two-directional solution has distance 140. The pattern in the graph formed by nodes a_4 to a_{12} can be continued by adding more "switchbacks" to get from a_4 to a_{12} . This will

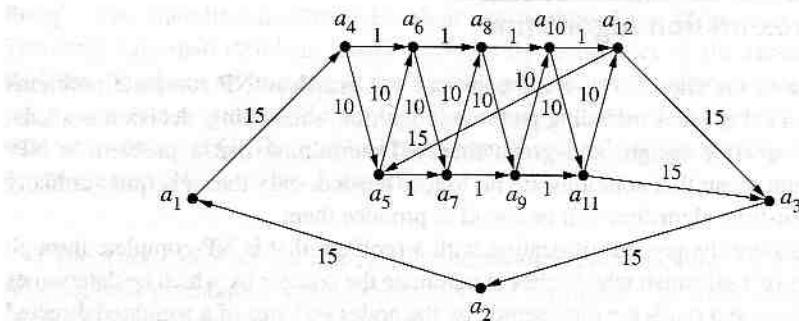


FIGURE 16.6 Solution to two-directional Traveling Salesman Problem.

not increase the minimal-cost tour, but the cost of the least-cost two-directional tour can be made as large as desired.

Realizing that his two-direction strategy may produce excessively long tours, the salesman asks the following two questions:

1. Is there a polynomial-time algorithm that solves the two-direction problem without the tour becoming arbitrarily longer than the optimal tour?
2. If the answer to the preceding question is no, what other conditions could be added to obtain an approximate solution in polynomial time?

These questions will be answered after introducing measures to characterize the performance of an approximation algorithm.

The solution to an optimization problem includes a numeric value that we will generically refer to as the cost of the solution. For example, the solution to an instance of the Traveling Salesman Problem consists a tour with the total distance being the cost of the tour. A solution to the Knapsack Problem consists of a set of objects and the associated cost is the total value of the objects in the set. An approximation algorithm produces a solution that may not have the optimal cost. The error of an approximation is the difference between the costs of the optimal and approximate solutions.

Let $c(p_i)$ denote the cost of the solution produced by an approximation algorithm and $c^*(p_i)$ be the optimal cost for a problem instance p_i of an optimization problem P . The quality of an approximation algorithm is measured by a comparison of the cost of the approximate solution to that of an optimal solution.

Definition 16.6.1

An algorithm that produces approximate solutions to an optimization problem P is said to be an **α -approximation algorithm** if

- i) the problem is a minimization problem and $c(p_i) \leq \alpha \cdot c^*(p_i)$, or
- ii) the problem is a maximization problem and $c^*(p_i) \leq \alpha \cdot c(p_i)$

for a constant $\alpha \geq 1$ and all instances p_i of P .

A 2-approximation is a cost at most twice the 2-approximation.

The salesman approximation algorithms for the problem are not necessarily an answer to the first question. Approximation algorithms that satisfy the triangle inequality

Theorem 16.6.2

If $P \neq NP$, then the Traveling Salesman Problem is NP-hard.

Proof. We will reduce the Traveling Salesman Problem to the Hamiltonian Circuit Problem. Since the latter can be solved in polynomial time, the algorithm under consideration is also polynomial.

We begin by reducing instances of the Traveling Salesman Problem to instances of the Hamiltonian Circuit Problem. The Traveling Salesman Problem is a total problem, so it has a solution if and only if it has a tour.

Clearly, the constant factor in the length of a representation of a tour is 2.

If G has a tour, then the cost of a tour of G' has cost at most twice the former case, since G' has n edges and only if, the α -approximation algorithm finds a tour.

The preceding construction shows that if G has a tour, then G' has a tour. Construct G' from G using the preceding observation. Then G' has a tour if and only if, G has a tour, so is the corollary.

We can easily show that the Traveling Salesman Problem is NP-hard when the graph is complete and satisfies the triangle inequality.

A 2-approximation algorithm for a minimization problem produces solutions that have a cost at most twice that of an optimal solution. For a maximization problem, the cost of the 2-approximate solution is at least half of the optimal cost.

The salesman's questions can now be restated as: "Is there a polynomial-time α -approximation algorithm for the Traveling Salesman Problem?" and "What changes in the problem are necessary to obtain a polynomial-time α -approximation algorithm?" The answer to the first question is no unless $P = NP$. One answer to the second is that a 2-approximation algorithm can be obtained if the graph is totally connected and the distances satisfy the triangle inequality.

Theorem 16.6.2

If $P \neq NP$, there is no polynomial-time α -approximation algorithm for the Traveling Salesman Problem.

Proof. We will prove that a polynomial-time α -approximation algorithm to the Traveling Salesman Problem can be used to solve the Hamiltonian Circuit Problem in polynomial time. Since the latter cannot be done if $P \neq NP$, it follows that there can be no such approximation algorithm under the same assumption.

We begin by defining a transformation of instances of the Hamiltonian Circuit Problem to instances of the Traveling Salesman Problem. Let $G = (N, A)$ be an instance of the Hamiltonian Circuit Problem with $n = \text{card}(N)$. The corresponding Traveling Salesman Problem is a totally connected graph $G' = (N, A', w)$, where w is defined by

$$w(x, y) = \begin{cases} 1 & \text{if } [x, y] \in A \\ \alpha \cdot n + 1 & \text{otherwise.} \end{cases}$$

Clearly, the construction of G' from G can be accomplished in time polynomial with the length of a representation of G .

If G has a tour, the corresponding tour in G' has cost n . If G does not have a tour, every tour of G' has cost greater than $\alpha \cdot n$ since it must contain at least one arc that is not in A . In the former case, running an α -approximation algorithm on G' must produce a tour of cost n because all other tours exceed the approximation bound. Consequently, G has a tour if, and only if, the α -approximation algorithm returns a tour of cost n .

The preceding equivalence describes a solution to the Hamiltonian Circuit Problem: Construct G' from G and obtain a tour of G' using the approximation algorithm. By the preceding observation, the tour returned by the approximation algorithm has length n if, and only if, G has a tour. If the α -approximation algorithm is computable in polynomial time, so is the corresponding solution to the Hamiltonian Circuit Problem. ■

We can easily produce a 2-approximation algorithm for the Traveling Salesman Problem when the graph $G = (N, A, w)$ is totally connected and the distance function is commutative and satisfies the triangle inequality. That is,

$$\begin{aligned} w(x, y) &= w(y, x) \text{ and,} \\ w(x, y) &\leq w(x, z) + w(z, y) \end{aligned}$$

for all $x, y, z \in N$. The Traveling Salesman Problem with these added conditions is sometimes called the *Euclidean Traveling Salesman Problem*.

The approximation algorithm first constructs a minimum cost spanning tree of G . A spanning tree of an undirected connected graph is a connected acyclic subgraph that contains all nodes of the graph. The cost of a spanning tree is the sum of the weights of the arcs in the tree. A weighted directed graph G that is totally connected with a commutative distance function can be considered to be an undirected graph. For each arc $[x, y]$, there is an arc $[y, x]$ with the same weight.

With the interpretation of G as an undirected graph, Prim's algorithm can be used to generate a minimum cost spanning tree in time $O(n^2)$, where n is the number of nodes of G . The following four-step procedure defines a 2-approximation algorithm for the Euclidean Traveling Salesman Problem:

1. Select a node $x \in N$ to be the root of the spanning tree.
2. Build the minimum-cost spanning tree of G .
3. Construct the sequence of nodes visited by a preorder traversal of the spanning tree.
4. Delete nodes that occur more than once in the sequence.

Figure 16.7 illustrates the process of obtaining a tour from a spanning tree. A preorder traversal begins with the root c , visits all nodes (many, several times), and finishes at c . To obtain a tour from the path produced by the traversal, we sequentially delete multiple visits to the same node. In the sequence in Figure 16.7, the node c is revisited after a and before d . Deleting this occurrence of c may be thought of as taking a direct road from a to d that bypasses c . The total connectivity of the graph assures us of the presence of an arc from a to d , and the triangle inequality guarantees that the alternative route is no longer than the original.

This process can be repeated to remove multiple occurrences of all nodes except for the occurrence of the root at the beginning and ending of the path. The resulting path is a tour. To analyze the cost of the tour, we let t^* , m^* , p , and t be the costs of the minimum-cost tour, the minimum-cost spanning tree, the path generated by the preorder traversal, and the tour obtained using the node removal strategy, respectively.

We can obtain a spanning tree by deleting any single arc from a minimal-cost tour of the graph. The cost of the resulting spanning tree is an upper bound on the cost of the minimum-cost spanning tree M . Consequently,

$$m^* \leq t^*.$$

The path generated by the preorder traversal contains each arc of the spanning tree twice, so

$$p = 2m^*.$$

The cost of the tour since the node the inequalities

yields the 2-approximation

In this section we

a) determine

b) reformulate

c) develop alg

These steps provide

16.7 Approximation Algorithms

An ideal system would specify the degree of approximation in which extreme values of the necessary parameters are accurate approximations to the NP-complete problems.

is some
tree of G . A
at contains
the arcs in
ve distance
re is an arc

be used to
nodes of G .
Euclidean

ning tree.

A preorder
hes at c . To
liple visits
and before
 a to d that
arc from a
ger than the

s except for
ng path is a
nimum-cost
rsal, and the

al-cost tour
e cost of the

spanning tree

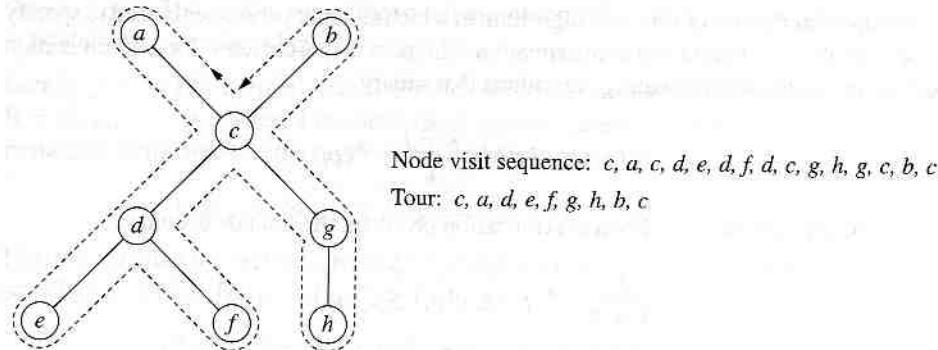


FIGURE 16.7 Spanning tree to tour.

The cost of the tour produced by the algorithm is bounded by the cost of the preorder path, since the node deletion process cannot increase the cost of the resulting path. Combining the inequalities,

$$t \leq p \leq 2t^*,$$

yields the 2-approximation bound on the tours constructed in this manner.

In this section we outlined a strategy for constructing solutions when confronted with an NP-complete problem. The steps employed by our mythical salesman were

- determine whether the asymptotic complexity is relevant to the problem,
- reformulate the problem into an efficiently solvable problem, or
- develop algorithms that produce approximate solutions.

These steps provide a good starting place for obtaining suitable solutions to NP-complete optimization problems.

16.7 Approximation Schemes

An ideal system for approximating an NP-complete problem would allow the user to specify the degree of error that is permissible for a particular application. For problems in which extremely high accuracy is critical, an error bound would be selected to achieve the necessary precision. For problems that do not require a high degree of precision, less accurate approximate solutions could be produced in a more efficient manner. For a number of NP-complete problems, this ideal can be realized.

An *approximation scheme* is an algorithm in which an input parameter is used to specify the acceptable error bound. An approximation scheme with parameter k for a minimization problem generates approximating algorithms that satisfy

$$c^*(p_i) \leq c(p_i) \leq \frac{k+1}{k} \cdot c^*(p_i)$$

for all problem instances p_i . For a maximization problem, the bounds become

$$\frac{k}{k+1} \cdot c^*(p_i) \leq c(p_i) \leq c^*(p_i).$$

In either case, increasing the value of k increases the precision of the approximations. A polynomial-time approximation scheme is an approximation scheme in which the time complexity is polynomial for all values of the parameter k .

We will use the Knapsack Problem to demonstrate the properties of an approximation scheme. The simplest approximation scheme for the Knapsack Problem initially places a number of items in the knapsack and completes the selection using a greedy algorithm. An instance of the optimization version of the Knapsack Problem consists of a set $S = \{a_1, \dots, a_n\}$, size function $s : S \rightarrow \mathbb{N}$, value function $v : S \rightarrow \mathbb{N}$, and size bound b . We let c^* denote the optimal value of a solution of a Knapsack Problem and c the value of an approximation, respectively.

A greedy strategy for the Knapsack Problem is to select the item a_i with highest relative value $v(a_i)/s(a_i)$ that fits into the knapsack. The process is repeated until no additional items can be put into the knapsack. Unfortunately, there is no bound on the error that may be produced using this approach (Exercise 14).

The approximation scheme with parameter k selects a set in the following manner:

1. All subsets $I_i \subseteq S$ of cardinality k or less are generated.
2. For each subset with size $s(I_i) \leq b$, a set G_i is generated using the greedy algorithm on the set $S - I_i$ with the original value and size functions and bound $b - s(I_i)$. The sets I_i and G_i are combined to produce the set $T_i = I_i \cup G_i$.
3. The result is a set T_i that has maximum value.

Generating all subsets with k or fewer elements and testing to determine if they satisfy the size bound produces a family of initial sets I_i . Each initial set is completed by producing a set G_i using the greedy algorithm. The total set T_i for the initial set I_i is the union $I_i \cup G_i$. We need to show that, for every problem instance and every $k \geq 1$, the algorithm produces an approximation that satisfies

$$\frac{k}{k+1} \cdot c^* \leq c.$$

Assume that an optimal solution is given by a set T with j elements. We consider two cases: $j \leq k$ and $j > k$.

Case 1: $j \leq k$. The set T is produced by the greedy algorithm.

Case 2: $j > k$. The set T is produced by the greedy algorithm.

First we note that $v(T) \geq v(I)$ so $v(I) \geq k \cdot v(\hat{a}_m)$

c^*

and the inequality holds.

Consider the set I_i . If the greedy algorithm produces an optimal set I_i ,

If not, let G_m be the greedy set containing the first item in the set $S - I_i$. Since there is insufficient space in the knapsack for all objects that have been selected so far, this set contains at least one item \hat{a}_m than \hat{a}_m . We now show that the items in G_m have greater relative value than \hat{a}_m . This follows since G_m is the greedy set and all other elements in $S - I_i$ have less relative value than \hat{a}_m . All the other elements in $S - I_i$ have less relative value than \hat{a}_m . Therefore, the items in R need to be combined with the items in I_i to form a subset of R of size k .

The maximum relative value of an item in R is $v(\hat{a}_m)$, since less than k items in its relative value is less than $v(\hat{a}_m)$. Therefore, the remaining space in the knapsack is less than $v(\hat{a}_m)/s(\hat{a}_m)$. Put

$c^* = v(\hat{a}_m)$

Using the inequality $v(T) \geq k \cdot v(\hat{a}_m)$,

or

as desired.

specify
ization

Case 1: $j \leq k$. The set T is one of the initial sets generated in step 1, and an optimal solution is produced by the algorithm.

Case 2: $j > k$. The optimal solution T can be split into two sets $I = \{\hat{a}_1, \dots, \hat{a}_k\}$ and $R = \{\hat{a}_{k+1}, \dots, \hat{a}_j\}$ where I contains the k highest valued items of T and R contains the remaining items listed in the order of their relative value:

$$v(\hat{a}_{k+1})/s(\hat{a}_{k+1}) \geq v(\hat{a}_{k+2})/s(\hat{a}_{k+2}) \geq \dots \geq v(\hat{a}_j)/s(\hat{a}_j).$$

First we note that for each $\hat{a}_t \in R$, $v(\hat{a}_t) \leq c^*/(k+1)$. Each $\hat{a}_t \in I$ has value greater than \hat{a}_t , so $v(I) \geq k \cdot v(\hat{a}_t)$. Thus

$$c^* = v(T) = v(I) + v(R) \geq k \cdot v(\hat{a}_t) + v(\hat{a}_t) \geq (k+1)v(\hat{a}_t)$$

and the inequality follows.

Consider the approximate solution generated from the set I using the greedy algorithm. If the greedy algorithm selects all the items in the optimal solution R, then the algorithm produces an optimal solution.

If not, let G be the extension of I produced by the greedy algorithm and let \hat{a}_m be the first item in the set R that is not selected by the greedy algorithm. This occurs only if there is insufficient space remaining in the knapsack when \hat{a}_m is considered. Now, let G_m be the set of objects that have been selected by the greedy algorithm at the time when \hat{a}_m is not taken. This set contains $\hat{a}_{k+1}, \hat{a}_{k+2}, \dots, \hat{a}_{m-1}$ from R and other items with relative value greater than \hat{a}_m . We now use G_m to produce an upper bound on the value of the set R. The elements in G_m have greater relative value than the initial items in R whose size totals $s(G_m)$. This follows since G_m contains all the objects of relative value greater than $v(\hat{a}_m)/s(\hat{a}_m)$ that are in R. All the other objects in G_m have relative value greater than $v(\hat{a}_m)/s(\hat{a}_m)$, whereas all other elements in R have relative value less than $v(\hat{a}_m)/s(\hat{a}_m)$. Note that the size of the items in R need not add exactly to $s(G_m)$. We may consider dividing an item to obtain a subset of R of size $s(G_m)$.

The maximum possible value that can be added to G_m to fill the knapsack is less than $v(\hat{a}_m)$, since less than $s(\hat{a}_m)$ space remains and the greedy algorithm has already passed \hat{a}_m in its relative value ordered search. This is also an upper bound on the value of filling the remaining space in R since the items $\hat{a}_m, \dots, \hat{a}_j$ in R all have relative values of at most $v(\hat{a}_m)/s(\hat{a}_m)$. Putting these observations together, we see that

$$c^* = v(I) + v(R) \leq v(I) + v(G_m) + v(\hat{a}_m) \leq v(I) + v(G) + v(\hat{a}_m).$$

Using the inequality $v(\hat{a}_m) \leq c^*/(k+1)$, we get

$$c^* \leq v(I) + v(G) + v(\hat{a}_m) \leq c + c^*/(k+1)$$

or

$$\frac{k}{k+1} \cdot c^* \leq c$$

as desired.

We also need to show that the approximating algorithm produced for every value $k \geq 1$ is polynomial in the size of the instance of the Knapsack Problem. Letting $C(n, i)$ be the number of combinations of n things taken i at a time, the number of subsets of cardinality at most k of a set of n objects is

$$\begin{aligned} \sum_{i=0}^k C(n, i) &= 1 + \sum_{i=1}^k \frac{n(n-1)\cdots(n-i+1)}{i!} \\ &\leq 1 + \sum_{i=1}^k n^i \\ &\leq 1 + \sum_{i=1}^k n^k \\ &= 1 + k \cdot n^k. \end{aligned}$$

Extending each of these with the greedy algorithm requires time $O(n)$. Thus the time complexity is $O(k \cdot n^{k+1})$.

Although the preceding approximation algorithm is polynomial for each k , the time complexity grows exponentially with the parameter k . Thus decreasing the error is accompanied by an exponential growth in the time needed to produce approximations. An approximation scheme that is polynomial in both n and k is called *fully polynomial*. There is an $O(k \cdot n^2)$ fully polynomial approximation scheme for the Knapsack Problem that combines the greedy algorithm with dynamic programming to reduce the time complexity.

Exercises

- * 1. A formula is in 2-conjunctive normal form if it is the conjunction of clauses consisting of the disjunction of two literals. Prove that the Satisfiability Problem for 2-conjunctive normal form formulas is in \mathcal{P} .
- 2. A formula is in 4-conjunctive normal form if it is the conjunction of clauses consisting of the disjunction of four literals. Prove that the Satisfiability Problem for 4-conjunctive normal form formulas is NP-complete.
- 3. Design a string representation for the Subset-Sum Problem and describe the computations of a nondeterministic Turing machine that solves the problem in polynomial time.
- 4. Design a polynomial-time reduction of the Partition Problem to the Subset-Sum Problem. A polynomial-time reduction of the Subset-Sum Problem to the Partition Problem was given in Theorem 16.4.1.
- 5. A clique in an undirected graph G is a subgraph of G in which every two vertices are connected by an arc. The Clique Problem is to determine, for an arbitrary graph G

and integer complete. I
between cl
an arc betw
vertices in V

- * 6. Let $\mathcal{C} = \{C_1, C_2, \dots, C_m\}$ be a collection of sets of integers. Prove that \mathcal{C} is said to cover S if

The Minim
k or less tha

- 7. Let \mathcal{C} be a collection of sets of integers. Prove that \mathcal{C} is complete.

- * 8. An instance of the Traveling Salesman Problem consists of a set of cities and edges between them. Show that the Traveling Salesman Problem is NP-hard.

- * 9. The input to the Knapsack Problem consists of a set of items, each with a weight and a value, and a knapsack with a maximum weight capacity. The goal is to find a subset of items that minimizes the total weight while maximizing the total value. max $\{l(A_i) | A_i \in \mathcal{A}\}$

- a) Formulate the Knapsack Problem as a linear programming problem.
- b) Show that the Knapsack Problem is NP-hard.

- * 10. The Integer Linear Programming Problem consists of a set of variables, a vector b of bounds, and a set of constraints. The goal is to find a vector x that minimizes the objective function $c^T x$ subject to the constraints $Ax \leq b$. Prove that the Integer Linear Programming Problem is NP-hard. (The knowledge of linear algebra is not required.)

- 11. Show that the Partition Problem is NP-complete.

- 12. The objective of the Minimum Size Vertex Cover Problem is to construct a vertex cover of minimum size for a graph G . A vertex cover is a set of vertices such that every edge in G has at least one endpoint in the set. Prove that the Minimum Size Vertex Cover Problem is NP-hard. (The knowledge of graph theory is not required.)

- * 13. The input to the Minimum Size Vertex Cover Problem consists of a graph G and a size function s that maps each vertex to a nonnegative integer. The goal is to find a vertex cover of minimum size such that the size of the vertex cover is at least $s(v)$ for every vertex v in the graph. Prove that the Minimum Size Vertex Cover Problem is NP-hard. (The knowledge of graph theory is not required.)

and integer k , whether G has a clique of size k . Prove that the Clique Problem is NP-complete. Hint: To show that the Clique Problem is NP-hard, establish a relationship between cliques in a graph G and vertex covers in the complement graph \bar{G} . There is an arc between vertices x and y in \bar{G} if, and only if, there is no arc connecting these vertices in G .

- *6. Let $\mathcal{C} = \{C_1, \dots, C_n\}$ be a collection of subsets of a set S . A subcollection $\mathcal{C}' \subseteq \mathcal{C}$ is said to cover S if

$$S = \bigcup_{C_i \in \mathcal{C}'} C_i.$$

The Minimum Cover Problem asks whether a collection \mathcal{C} has a subcollection of size k or less that covers S . Prove that the Minimum Cover Problem is NP-complete.

7. Let \mathcal{C} be a collection of finite sets and k an integer less than or equal to the cardinality of \mathcal{C} . Prove that the problem of determining whether \mathcal{C} contains k disjoint sets is NP-complete.
- *8. An instance of the Longest Path Problem is a graph $G = (N, A)$ and an integer $k \leq |A|$. Show that the problem of determining whether G has an acyclic path with k or more edges is NP-complete.
- *9. The input to the Multiprocessor Scheduling Problem consists of a set A of tasks, a length function $l : A \rightarrow N$ that describes the running time of each task, and the number k of available processors. The objective is to find a partition A_1, A_2, \dots, A_k of A that minimizes the time needed to complete all the tasks, that is, that minimizes $\max\{l(A_i) \mid i = 1, \dots, n\}$ over all partitions.
- Formulate the Multiprocessor Scheduling Problem as a decision problem.
 - Show that the associated decision problem is NP-complete.
- *10. The Integer Linear Programming Problem is: Given an n by m matrix A and a column vector b of length n , does there exist a column vector x such that $Ax \geq b$? Use a reduction of 3-satisfiability to prove that the integer linear programming problem is NP-hard. (The Integer Linear Programming Problem is also in NP; the proof requires knowledge of some of the elementary properties of linear algebra.)
11. Show that the Traveling Salesman Decision Problem for undirected graphs is NP-complete.
12. The objective of the optimization version of the Vertex Cover Problem is to find a minimum size vertex cover of an undirected graph G . An approximation strategy constructs a cover VC by selecting an arbitrary arc $[x, y]$ from G and adding it to VC, removing $[x, y]$ and all arcs incident to $[x, y]$ from G , and repeating the selection and deletion cycle until VC covers the original graph. Prove that this strategy yields a polynomial-time 2-approximation algorithm for the Vertex Cover Problem.
- *13. The input to the optimization version of the Bin Packing Problem consists of a set A , a size function $s : A \rightarrow N$, and a bin size n greater than the maximum size of any object.

The objective is to determine the minimum number of bins needed to store the objects in A , where the bin size n is an upper bound on the total size of the objects that can be placed in a single bin. A first-fit algorithm takes an object and places it in the first bin in which it fits. If it does not fit in any of the current bins, the object is placed in a new bin. This process is repeated until all the objects have been stored. Show that the first-fit strategy produces a polynomial-time 2-approximation algorithm for the Bin Packing Problem.

14. A greedy strategy for the Knapsack Problem is to select the item a with highest relative value $v(a)/s(a)$ that fits into the knapsack. The process is repeated until no additional items can fit into the knapsack. Show that there is no upper bound on the possible error using the greedy choice strategy.
- * 15. An approximation algorithm for the Knapsack Problem can be obtained by modifying the greedy strategy as follows: The algorithm returns either the solution produced by the greedy algorithm or the solution that consists of the single item with largest value that fits into the knapsack. Prove that the modification produces a 2-approximation algorithm for the Knapsack Problem.

Bibliographic Notes

Karp's [1972] seminal paper proved the NP-completeness of the 3-Satisfiability Problem, the Vertex Cover Problem, and the Hamiltonian Circuit Problem. All of the problems in this chapter, and many more, are examined in Garey and Johnson's [1979] classic book on NP-completeness. This book also includes a description of the reductions that are required for most of the exercises.

Because of the importance of NP-complete problems, an extensive literature has been developed on the topic of approximation algorithms. An introduction to the field of approximation algorithms can be found in the previously mentioned book by Garey and Johnson and in Papadimitriou and Steiglitz [1982] and Hochbaum [1997]. Christofides [1976] designed a polynomial-time 1.5-approximation algorithm for the classic Traveling Salesman Problem. The approximation scheme for the Knapsack Problem given in Section 16.6 is from Sahni [1975]. Ibarra and Kim [1975] used dynamic programming to develop an $O(k \cdot n^2)$ fully polynomial approximation scheme for the Knapsack Problem.

There are a number of excellent books on the general theory of algorithms including Cormen, Leiserson, Rivest, and Stein [2001], Levitin [2003], and Brassard and Bratley [1996]. In addition to NP-complete problems and approximation algorithms, these books cover the graph algorithms, greedy algorithms, and dynamic programming strategies used in the approximation algorithms mentioned in this chapter.

Complexity theorists study the relationship between problems based on their complexity in a language. They are interested in the time complexity of algorithms and the existence of efficient algorithms for solving problems. They also examine the relationship between different complexity classes, such as P and NP. This is a fundamental question in computer science, as it is required for a complete understanding of the nature of computation and the existence of problems that cannot be solved efficiently by any algorithm.

17.1 Derivation

Our study of tractable problems has focused on polynomial time algorithms. These are deterministic algorithms that always produce the correct result given enough time. We now turn our attention to the P = NP question, which asks whether there is a polynomial-time algorithm that can solve any problem in NP. This question is one of the most important open problems in computer science, and its resolution would have profound implications for many fields.

The classes P and NP are defined based on the complexity of the algorithms that can solve them. Between these two classes, there are many other complexity classes, such as NP-hard and NP-complete. These classes are related to each other through reductions, which are mappings that transform one problem into another. If a reduction exists from a problem A to a problem B, then any algorithm that solves problem B can be used to solve problem A. This means that if problem B is NP-hard, then problem A is also NP-hard. Similarly, if problem A is NP-complete, then any problem that reduces to it is also NP-complete.

te objects
at can be
e first bin
in a new
the first-
Packing

t relative
dditional
ble error

odifying
uced by
est value
imation

roblem,
is in this
on NP-
ired for

as been
pproxima-
son and
esigned
n Prob-
is from
 $(k \cdot n^2)$

luding
Bratley
books
es used

CHAPTER 17

Additional Complexity Classes

Complexity theory is concerned with assessing the resources required to determine membership in a language, to solve a decision problem, or to compute a function. The study of time complexity has identified the class \mathcal{P} of problems that can be solved by polynomial-time algorithms as comprising the efficiently solvable problems. We begin this chapter by examining the properties of several complexity classes that can be derived from the classes \mathcal{P} and NP . This is followed by developing relationships between the amount of time and space required for a computation. Finally, properties of space complexity are used to demonstrate the existence of problems that are not solvable by any polynomial-time or polynomial-space algorithm.

17.1 Derivative Complexity Classes

Our study of tractability introduced the class \mathcal{P} of languages decidable deterministically in polynomial time and the class NP of languages decidable in polynomial time by nondeterministic computations. The question of whether these classes are identical is currently unknown. We now consider several additional classes of languages that provide insight into the $\mathcal{P} = \text{NP}$ question. Interestingly enough, properties of these classes are often dependent upon the relationship between \mathcal{P} and NP . The majority of the following discussion will proceed under the assumption that $\mathcal{P} \neq \text{NP}$. However, this condition will be explicitly stated in any results that utilize the assumption.

The classes \mathcal{P} and NP-C are both nonempty subsets of NP , but what is the relationship between these two classes? By Theorem 15.6.2, if $\mathcal{P} \cap \text{NP-C}$ is nonempty, then $\mathcal{P} = \text{NP}$.

Consequently, under the assumption $\mathcal{P} \neq \mathcal{NP}$, \mathcal{P} and \mathcal{NPC} must be disjoint. The diagram in Section 15.9 shows the inclusions of \mathcal{P} and \mathcal{NPC} in \mathcal{NP} if $\mathcal{P} \neq \mathcal{NP}$. One question immediately arises when looking at this diagram: Are there languages in \mathcal{NP} that are not in either \mathcal{P} or \mathcal{NPC} ?

We define the family of languages \mathcal{NPJ} , where the letter J represents intermediate, to consist of all languages that are in \mathcal{NP} but in neither \mathcal{NPC} nor \mathcal{P} . The use of the word *intermediate* in this context is best explained in terms of solving decision problems. A problem in \mathcal{NPJ} is not NP-hard and therefore not considered to be as difficult as the problems in \mathcal{NPC} . On the other hand, since it is not in \mathcal{P} , it is considered to be more difficult than problems in that class. The term *intermediate* comes from this interpretation of problems in \mathcal{NPJ} being harder than the problems in \mathcal{P} and not as hard as problems in \mathcal{NPC} .

If $\mathcal{P} = \mathcal{NP}$, the class \mathcal{NPJ} is empty. Theorem 17.1.1, stated without proof, guarantees the existence of intermediate problems if $\mathcal{P} \neq \mathcal{NP}$.

Theorem 17.1.1

If $\mathcal{P} \neq \mathcal{NP}$, then \mathcal{NPJ} is not empty.

Recall that the complement of a language L over an alphabet Σ , denoted \bar{L} , consists of all strings not in L ; that is, $\bar{L} = \Sigma^* - L$. A family of languages \mathcal{F} is closed under complementation if $\bar{L} \in \mathcal{F}$ whenever $L \in \mathcal{F}$. The family \mathcal{P} is closed under complementation; a deterministic Turing machine that accepts a language in polynomial time can be transformed to accept the complement with the same polynomial bound. The transformation simply consists of interchanging the accepting and rejecting states of the Turing machine.

The asymmetry of nondeterminism has a dramatic impact on the complexity of machines that accept a language and those that accept its complement. To obtain an affirmative answer, a single nondeterministic “guess” that can verify the affirmative answer is all that is required. A negative answer is obtained only if all possible guesses fail. The Satisfiability Problem is used to demonstrate the asymmetry of the complexity of nondeterministic acceptance of a language and its complement.

The input to the Satisfiability Problem is a conjunctive normal form formula u over a set of Boolean variables V , and the output is yes if u is satisfiable and no otherwise. Theorem 15.5.2 described the computation of a nondeterministic machine that solves the Satisfiability Problem in polynomial time. This was accomplished by guessing a truth assignment on V . Checking whether a truth assignment satisfies a formula u is a straightforward process that can be accomplished in time polynomial in the length of u .

The complement of the Satisfiability Problem is to determine whether a conjunctive normal form formula is unsatisfiable; that is, it is not satisfied by any truth assignment. An affirmative answer is obtained for a formula u if u is false for every possible truth assignment. A nondeterministic strategy to solve the “unsatisfiability problem” requires a guess that can verify the unsatisfiability of u . The guess cannot be a single truth assignment since discovering that one truth assignment does not satisfy u is not sufficient to conclude that u is unsatisfiable. Intuitively, the truth values of u under all possible truth assignments are required. If $\text{card}(V) = n$, there are 2^n truth assignments to be examined. It seems reasonable

to conclude
reasonable
known wh

Rather
examine t
The family

Theorem

If $\mathcal{NP} \neq c$

Proof. A
compleme

Theor
sufficient t
answer the
 $\mathcal{NP} = \text{co-3}$
theoretical
deciding m
Theorem 1

Theorem

If there is a

Proof. As
under these
NP-comple
a reduction

By our
Turing ma
machine th
time. Thus,

To con
inclusion. I
The comple

The Sa
of the fami
complement
if, and only
are shown i

e diagram
question
are not in
ermediate,
f the word
problems. A
e problems
ficult than
problems
2.
guarantees

\bar{L} , consists
under comple-
mentation;
n be trans-
formation
g machine.
city of ma-
affirmative
r is all that
Satisfiabil-
terministic

ila u over a
wise. Theo-
the Satisfi-
assignment
ard process

conjunctive
gnment. An
assignment.
a guess that
ment since
conclude that
gnments are
s reasonable

to conclude that this problem is not in NP . Note the use of the terms *intuitively* and *it seems reasonable* in the previous sentences. These hedges have been included because it is not known whether the unsatisfiability problem is in NP .

Rather than considering only the complement of the Satisfiability Problem, we will examine the family of languages consisting of the complements of all languages in NP . The family $\text{co-NP} = \{\bar{L} \mid L \in \text{NP}\}$.

Theorem 17.1.2

If $\text{NP} \neq \text{co-NP}$, then $\text{P} \neq \text{NP}$.

Proof. As noted previously, P is closed under complementation. If NP is not closed under complementation, the two classes of languages cannot be identical. ■

Theorem 17.1.2 provides another method for answering the $\text{P} = \text{NP}$ question. It is sufficient to find a language $L \in \text{NP}$ with $\bar{L} \notin \text{NP}$. Proving that $\text{NP} = \text{co-NP}$ does not answer the question of the identity of P and NP . At this time, it is unknown whether $\text{NP} = \text{co-NP}$. Just as it is generally believed that $\text{P} \neq \text{NP}$, it is also the consensus of theoretical computer scientists that $\text{NP} \neq \text{co-NP}$. However, the majority does not rule in deciding mathematical properties and the search for a proof of these inequalities continues. Theorem 17.1.3 provides one approach for establishing the equality of NP and co-NP .

Theorem 17.1.3

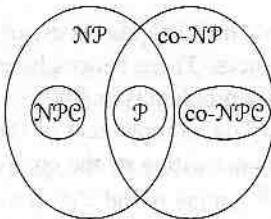
If there is an NP-complete language L with $\bar{L} \in \text{NP}$, then $\text{NP} = \text{co-NP}$.

Proof. Assume that L is a language that satisfies the above conditions. We first show that, under these conditions, the complement of any language Q in NP is also in NP . Since L is NP-complete, there is a polynomial-time reduction of Q to L . This reduction also serves as a reduction of \bar{Q} to \bar{L} .

By our assumption that $\bar{L} \in \text{NP}$, \bar{L} is accepted in polynomial time by a nondeterministic Turing machine. Combining the machine that performs the reduction of \bar{Q} to \bar{L} with the machine that accepts \bar{L} produces a nondeterministic machine that accepts \bar{Q} in polynomial time. Thus, $\text{co-NP} \subseteq \text{NP}$.

To complete the proof that $\text{NP} = \text{co-NP}$, it is necessary to establish the opposite inclusion. Let Q be any language in NP . By the preceding argument, \bar{Q} is also in NP . The complement of \bar{Q} , which is Q itself, is then in co-NP . ■

The Satisfiability Problem and its complement were used to initiate the examination of the family co-NP . At that point we said that it seems reasonable to believe that the complement of the Satisfiability Problem is not in NP . By Theorem 17.1.2, \bar{L}_{SAT} is in NP if, and only if, $\text{NP} = \text{co-NP}$. The presumed relationships between P , NP , NPC , and co-NP are shown in Figure 17.1.

FIGURE 17.1 Inclusions if $P \neq NP$ and $NP \neq co\text{-}NP$.

17.2 Space Complexity

The focus of the preceding chapters has been the time complexity of Turing machines and decision problems. We could have equally well chosen to analyze the space required by a computation. In high-level algorithmic problem solving, the amount of time and memory required by a program are often related. We will show that the time complexity of a Turing machine provides an upper bound on the space required and vice versa. Unless otherwise stated, the properties of space complexity that we present hold for both deterministic and nondeterministic Turing machines. The effect of limiting the space available for a computation on the acceptance of languages will be examined in Section 17.3.

The Turing machine architecture depicted in Figure 17.2 is used for measuring the space required by a computation. Tape 1, which contains the input, is read-only. With an input string of length n , the head on the input tape must remain within tape positions 0 through $n + 1$. The Turing machine reads the input tape but performs its work on the remaining tapes. Providing a read-only input tape separates the amount of space required for the input from the work space needed by the computation. The space complexity provides an upper bound on the amount of space used on the work tapes. A Turing machine that satisfies the preceding conditions is sometimes referred to as an *off-line Turing machine*, since the input may be considered to be provided off-line prior to the computation and is not included in the assessment of resource utilization. Unless otherwise specified, for the remainder of the chapter all Turing machines are assumed to be designed for the analysis of space complexity.

Definition 17.2.1

The **space complexity** of a $k + 1$ -tape Turing machine M is the function $sc_M : N \rightarrow N$ such that $sc_M(n)$ is the maximum number of tape squares read on any work tape by a computation of M when initiated with an input string of length n .

This definition serves equally well for deterministic and nondeterministic Turing machines. For nondeterministic machines, the maximum is taken over every possible computation for each string of length n . Unlike time complexity, we do not assume that the computations of a Turing machine terminate for every input. The tape heads of a machine

may remain within
terminates.

The space com
take any transition
determination. Since
 $sc_M(n) < n$. That i
input. In Example
demonstrate compu

Example 17.2.1

A two-tape Turing
Example 14.3.1. The
space complexity a
input string and the
a computation repre
reading the strings i

We now design
 $O(\log_2(n))$. The w
natural number. The
of the string with th
and the $i + 1$ st elem
discovered that do n
the string is rejected

A computation

1. A single 1 is w
2. Tape 3 is copie
3. The input tape

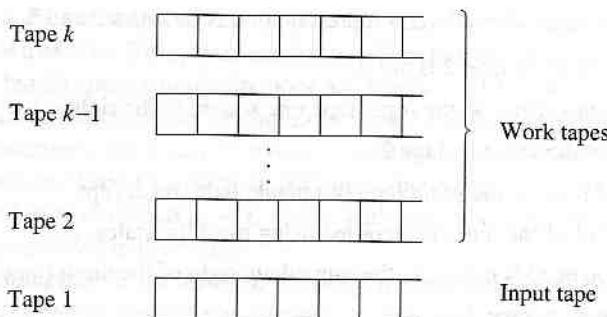


FIGURE 17.2 Turing machine architecture for space complexity.

hines and
quired by
d memory
f a Turing
otherwise
erministic
able for a

g the space
h an input
0 through
remaining
r the input
s an upper
atisfies the
e the input
t included
remainder
s of space

$\rightarrow N$ such
omputation

Turing
mable
com
ne that the
a machine

may remain within a finite length initial segment of the tape even though computation never terminates.

The space complexity is always greater than 0. Even if a Turing machine does not take any transitions, the leftmost position on the work tapes must be read to make this determination. Since space complexity measures only the work tapes, it is possible that $sc_M(n) < n$. That is, the space needed for computation may be less than the length of the input. In Example 17.2.1 we design yet another machine that accepts the palindromes to demonstrate computations with space complexity $O(\log_2(n))$.

Example 17.2.1

A two-tape Turing machine that accepts the palindromes over $\{a, b\}$ was constructed in Example 14.3.1. This machine conforms to the specifications of a machine designed for space complexity analysis. The input tape is read-only and the tape head reads only the input string and the blanks on either side of the input. The space complexity of M' is $n + 2$; a computation reproduces the input on tape 2 and compares the strings on tapes 1 and 2 by reading the strings in opposite directions.

We now design a three-tape machine M that accepts the palindromes with $sc_M(n) = O(\log_2(n))$. The work tapes are used as counters and hold the binary representation of a natural number. The strategy is to use the counters to identify and compare the i th element of the string with the i th element from the right. If they match, the counter is incremented and the $i + 1$ st elements are compared. This process continues until a pair of elements is discovered that do not match or until all elements have been compared. In the former case the string is rejected, and in the latter it is accepted.

A computation of M with input u of length n consists of the following steps:

1. A single 1 is written in tape position 1 on tape 3.
2. Tape 3 is copied to tape 2.
3. The input tape head is positioned at the leftmost square.

Let i be the integer whose binary representation is on tapes 2 and 3.

4. While the number on tape 2 is not 0,
 - a) Move the tape head on the input tape one square to the right.
 - b) Decrement the value on tape 2.
5. If the symbol read on the input tape is a blank, halt and accept.
6. The i th symbol of the input is recorded using machine states.
7. The input tape head is moved to the immediate right of the input (tape position $n + 1$).
8. Tape 3 is copied to tape 2.
9. While the number on tape 2 is not 0,
 - a) Move the tape head of the input tape one square to the left.
 - b) Decrement the value on tape 2.
10. If the $(n - i + 1)$ st symbol matches the i th symbol, the value on tape 3 is incremented, the tape heads are returned at their initial positions, and the computation continues with step 2. Otherwise the computation rejects the input.

The operations on tapes 2 and 3 increment and decrement the binary representation of a natural number. Since $n + 1$ is the largest number written on either of these tapes, each tape uses at most $\lceil \log_2(n + 1) \rceil + 2$ tape squares. \square

An off-line Turing machine is said to be $s(n)$ space-bounded if the maximum number of tape squares used on a work tape during a computation with an input of length n is at most $\max\{1, s(n)\}$. The space complexity function $sc_M(n)$ specifies the maximum space actually required by a computation of M with input n , while a space bound provides an upper bound that may not be achieved. As previously noted about space complexity, a Turing machine may be space-bounded even though it has computations that do not terminate.

The computations of a $k + 1$ -tape Turing machine M with space bound $s(n) \geq n$ can be simulated by a machine with one work tape that is also $s(n)$ space-bounded. This differs from measurement of time complexity where a reduction in the number of tapes produces an increase in the time complexity. The proof utilizes the construction of a $2k + 1$ -track machine from a k -tape Turing machine presented in Section 8.6. The number of tape squares scanned by the resulting multitrack machine is exactly the maximum number read on any work tape of the original multitape machine. This observation is summarized in the following theorem.

Theorem 17.2.2

Let L be a language accepted by a $k + 1$ -tape Turing machine M with space bound $s(n) \geq n$. Then L is accepted by an $s(n)$ space-bounded Turing machine with one work tape.

As in Theorem 17.2.2, we will frequently use the assumption that a Turing machine has space complexity $sc_M(n) \geq n$ or has a space bound $s(n) \geq n$. These conditions are added to the statement of a theorem to ensure the availability of at least n tape squares. The first

condition implies
The reverse is no
space-bounded, b

Although ou
off-line Turing m
machines. A one
tape squares used
store the input is

With the ass
we can, when co
fact, any languag
deterministic Tur
uses the same red

The reason fo
ity is to have a si
Many interesting
of the input. In pa
machines has bee
that may require a
be at least the size

17.3 Relations

The time comple
space complexity.
computation is lin

Theorem 17.3.1

Let M be a $k + 1$ -ta
 $f(n) + 1$.

Proof. The maxi
on the work tapes
squares read on an

Obtaining the
complicated since
two-tape machine
deterministic mach
assume that M halts
generalization from

condition implies the second, since the space complexity function is itself a space bound. The reverse is not true. The Turing machine M described in Example 17.2.1 is $s(n) = n + 2$ space-bounded, but its space complexity does not satisfy $sc_M(n) \geq n$.

Although our definition of space complexity is based on the computations of multitape off-line Turing machines, the notion of a space bound is also applicable to one-tape Turing machines. A one-tape Turing machine is $s(n)$ space-bounded if the maximum number of tape squares used is at most $\max\{n + 1, s(n)\}$. With only one tape, the space required to store the input is included in the bound.

With the assumption that a machine is $s(n) \geq n$ space-bounded and Theorem 17.2.2 we can, when convenient, restrict our attention to machines with a single work tape. In fact, any language that is accepted with a space bound $s(n) \geq n$ is accepted by a one-tape deterministic Turing machine that satisfies the same space bound (Exercise 9). The argument uses the same reduction from multitrack to single-track machine.

The reason for the selection of the off-line Turing machine for studying space complexity is to have a single Turing machine model suitable for the analysis of all space bounds. Many interesting languages are accepted by machines with space bounds less than the length of the input. In particular, the class of languages accepted by $\log_2(n)$ space-bounded Turing machines has been extensively studied. Our attention, however, has focused on problems that may require a significant amount of resources and a restriction that the available space be at least the size of the input is reasonable for these problems.

17.3 Relations between Space and Time Complexity

The time complexity of a Turing machine can be used to obtain an upper bound on the space complexity. The number of tape squares that a single tape head can read during a computation is limited by the number of transitions in the computation.

Theorem 17.3.1

Let M be a $k + 1$ -tape Turing machine with time complexity $tc_M(n) = f(n)$. Then $sc_M(n) \leq f(n) + 1$.

Proof. The maximum amount of tape is used when each transition of M moves the heads on the work tapes to the right on each transition. In this case the maximum number of tape squares read on any work tape is $f(n) + 1$. ■

Obtaining the restriction on time complexity imposed by a known space bound is more complicated since a machine may read a particular segment of the tape multiple times. A two-tape machine M is used to demonstrate the bound on the time of a computation of a deterministic machine that can be obtained from the space complexity of the machine. We assume that M halts for all input strings since this is a requirement of time complexity. The generalization from two-tape to $k + 1$ -tape machines is straightforward.

Theorem 17.3.2

Let M be a two-tape deterministic Turing machine that halts for all inputs with space bound $s(n)$. Then $tc_M(n) \leq m \cdot s(n) \cdot (n+2) \cdot t^{s(n)}$, where m is the number of states and t the number of tape symbols of M .

Proof. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a two-tape Turing machine with $m = \text{card}(Q)$ and $t = \text{card}(\Gamma)$. For an input of length n , the space bound restricts the computation of M to at most $s(n)$ positions on tape 2. Limiting the computation to a finite length segment of the tape allows us to count the number of distinct machine configurations that M may enter.

The work tape may have any of the t symbols in each position, yielding $t^{s(n)}$ possible configurations. The head on tape 1 may read any of the first $n+2$ positions, while the head on tape 2 may read positions 0 through $s(n)-1$. Thus there are $s(n) \cdot (n+2) \cdot t^{s(n)}$ possible combinations of tape configurations and head positions. For any of these, the machine may be in one of m states, producing a total of $m \cdot s(n) \cdot (n+2) \cdot t^{s(n)}$ distinct configurations.

The repetition of a configuration by a deterministic machine indicates that the machine has entered an infinite loop. Since M halts for all computations, the computation must halt prior to $m \cdot s(n) \cdot (n+2) \cdot t^{s(n)}$ transitions. ■

For a nondeterministic machine, a terminating computation may have more transitions than the number of possible configurations. When a configuration is repeated, the computation may select a different transition. In Corollary 17.3.3 we use the limit on the number of configurations to produce an exponential bound on the number of transitions required for the acceptance of a string by any space-bounded Turing machine. The bound is given in exponential form to facilitate the comparison of the amount of space required by deterministic and nondeterministic computations in the next section. By Theorem 17.2.2, it is sufficient to consider Turing machines with one work tape.

Corollary 17.3.3

Let M be a Turing machine with space bound $s(n) \geq n$. There is a constant c that depends on number of states and tape symbols of M such that any string of length n accepted by M is accepted by a computation with at most $c^{s(n)}$ transitions.

Proof. Again we let $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a two-tape Turing machine with $m = \text{card}(Q)$ and $t = \text{card}(\Gamma)$. By the argument in Theorem 17.3.2, there are $m \cdot s(n) \cdot (n+2) \cdot t^{s(n)}$ possible configurations for a computation with input of length n . The derivation of an exponential bound on the number of machine configurations uses the inequality

$$(n+2)s(n) \leq 3^{s(n)},$$

which holds whenever $n \leq s(n)$ and $s(n) > 0$. The exponential bound on the number of transitions is obtained by replacing the terms in $m \cdot s(n) \cdot (n+2) \cdot t^{s(n)}$ with functions that have $s(n)$ as an exponent:

and the constant c Turing machine M

Any computation A computation of

where the first string 2, and the dot indicates. Removing the portion another accepting config

of strictly smaller length than $c^{s(n)}$ is produc

The upper bound on the number of transitions a string can be used to accept a string of same space complexity as the string M that is used to compute the string. The bound provided by the theorem is exponential computation halts at some point. This construction is an exponential bound on the space bound of M . The numbers on the left are the numbers on the right.

Corollary 17.3.4

Let L be a language accepted by a Turing machine M with space bound $s(n)$.

$$\begin{aligned}
 m \cdot s(n) \cdot (n+2) \cdot t^{s(n)} &\leq m^{s(n)} \cdot s(n) \cdot (n+2) \cdot t^{s(n)} \\
 &\leq m^{s(n)} \cdot 3^{s(n)} \cdot t^{s(n)} \\
 &= (3mt)^{s(n)} \\
 &= c^{s(n)}
 \end{aligned}$$

and the constant c is obtained directly from the number of states and tape symbols of the Turing machine M .

Any computation of M that has more than $c^{s(n)}$ transitions must repeat a configuration. A computation of this form that accepts a string w can be written

$$\begin{aligned}
 q_0 : .BwB, .BB \\
 \vdash q_i : Bu.vB, x.y \\
 \vdash q_i : Bu.vB, x.y \\
 \vdash q_j : Bu'.v'B, x'.y',
 \end{aligned}$$

where the first string after the semicolon represents tape 1, the second string represents tape 2, and the dot indicates that the tape head is reading the symbol to the immediate right. Removing the portion of the computation between the repeating configuration produces another accepting computation

$$\begin{aligned}
 q_0 : .BwB, .BB \\
 \vdash q_i : Bu.vB, x.y \\
 \vdash q_j : Bu'.v'B, x'.y'
 \end{aligned}$$

of strictly smaller length. This process can be repeated until a computation of length less than $c^{s(n)}$ is produced. ■

The upper bound on the number of transitions needed by a Turing machine M to accept a string can be used to construct a machine that accepts the same language as M , has the same space complexity, and halts for all input strings. The idea is to add another tape to M that is used to count the number of transitions. The counter tape is initialized to the bound provided by Corollary 17.3.3. With each transition, the counter is decremented. The computation halts and rejects the input if the counter reaches zero. The sole concern with this construction is to ensure that the counter tape uses no more tape than permitted by the space bound of M . This can be accomplished by selecting a suitable base b and representing the numbers on the counter tape in the base b system.

Corollary 17.3.4

Let L be a language accepted by a Turing machine with space bound $s(n) \geq n$. Then L is accepted by a Turing machine M' with space bound $s(n)$ that halts for all inputs.

A space bound $s(n)$ is *fully space constructible* if there is a Turing machine M for which the computation of every string of length n accesses exactly $s(n)$ tape squares. If M is a Turing machine with space complexity $sc_M(n) = s(n) \geq n$, then $s(n)$ is fully space constructible (Exercise 5). The set of fully space constructible functions includes n^r , 2^n , and $n!$ and most common number-theoretic functions. In addition, if $s_1(n)$ and $s_2(n)$ are functions that are fully space constructible, so are $s_1(n)s_2(n)$, $2^{s_1(n)}$, and $s_2(n)^{s_1(n)}$. The preceding observations allow us to conclude that the function

$$s(n) = 2^{2^{\dots^{2^n}}}$$

is fully space constructible for any number of 2's in the exponential chain. Thus there is no limit on the amount of space required for Turing machine computations. Theorem 17.3.5 gives conditions under which increasing the space available for a computation increases the family of languages that can be accepted.

Theorem 17.3.5

Let $s_1(n) \geq n$ and $s_2(n) \geq n$ be functions from \mathbb{N} to \mathbb{N} such that

$$\inf_{n \rightarrow \infty} \frac{s_1(n)}{s_2(n)} = 0$$

and s_2 is fully space constructible. Then there is a language L accepted by an $s_2(n)$ space-bounded Turing machine that is not accepted by any $s_1(n)$ space-bounded Turing machine.

Proof. We will construct a five-tape $s_2(n)$ space-bounded Turing machine M whose language is not accepted by any $s_1(n)$ space-bounded machine. The input to M is a string over $\{0, 1\}$ and the computation uses the interpretation of such a string as a two-tape Turing machine. The computation of M when run with an input string w consists of the simulation of a computation of two Turing machines. The first configures a tape of M to enforce the $s_2(n)$ space bound. The second simulates the computation of the machine encoded by the string w , which we will call M_w , when run with input w . A diagonalization argument is given to show that the language of M is not accepted by any $s_1(n)$ space-bounded Turing machine.

We use the encoding of multitape Turing machines described in Section 14.6, but we allow any number of 1 's to precede the string 000 that begins the encoding. Thus if $w \in \{0, 1\}^*$ is the encoding of a Turing machine, the strings $1w, 11w, 111w, \dots$ are encodings of the same machine. With this modification, an enumeration of the strings in $\{0, 1\}^*$ contains an infinite number of encodings of each Turing machine. Any string w that does not satisfy the requirements for an encoding of a two-tape machine is considered to represent the two-tape, one-state machine with no transitions.

The computation of M with input w begins by marking the $s_2(n) - 1$ st position of tape 5 with a 1 . Since $s_2(n)$ is fully space constructible, there is a Turing machine that will use exactly $s_2(n)$ squares when run with input w . The computation of this machine with

input w can be simulated. The computation is recorded.

After establishing that the computation with input w on tape 5 is recorded, it is pictured as

During the simulation, the heads attempt to move. The machine rejects the input. The string input w is accepted because the machine halts without accepting.

We now show that M is not accepted by any $s_1(n)$ space-bounded Turing machine. The proof is by contradiction.

Assume that M is accepted by some $s_1(n)$ space-bounded machine. By Corollary 17.3.4 we know that the computation of M with input w on tape 5 occurs an infinite number of times.

there is some $n \geq n_0$ such that the string w has at least n leading 1 's to produce the string $1w$.

Now consider the computation of M with input w . The computation has sufficient space to simulate the computation of M_w . However, M' does not accept the string w . Consider the computation of M with input w . The machine that can accept w is M' .

The space complexity of M is bounded by $s_2(n)$. This shows the existence of a lower bound for the space complexity of M . The latter bound is $s_2(n)$. The machine with polynomial space complexity $s_2(n)$ is M' . It is a lower bound for the space complexity of M . The membership problem for M is decidable.

M for
es. If
space
 2^n ,
) are
The

is no
7.3.5
es the

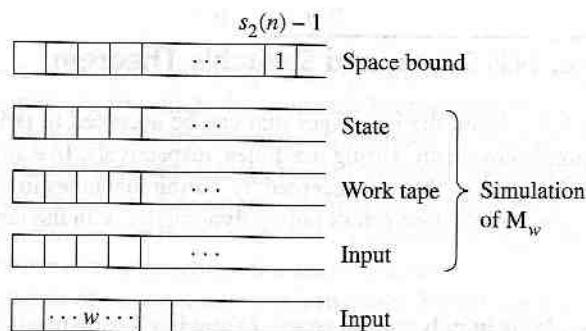
space-
chine.
e lan-
g over
Turing
ilation
ce the
by the
ment is
Turing

.6, but
Thus if
.. are
ings in
w that
ered to

tion of
at will
ie with

input w can be simulated on tapes 2 through 4 and the furthest right square accessed in the computation is recorded on tape 5.

After establishing the tape bound, M simulates the computation of the machine M_w with input w on tapes 2 through 4. At the beginning of this phase, the machine M can be pictured as



During the simulation of M_w , the heads on tapes 3 and 5 move synchronously. If the tape heads attempt to move to the right of the marker on tape 5, the computation of M halts and rejects the input. Thus M is guaranteed to be $s_2(n)$ space-bounded. The machine M accepts the string input w only if the computation is not terminated by the space bound and M_w halts without accepting w .

We now show that $L(M)$ cannot be accepted by any $s_1(n)$ space-bounded Turing machine. The proof is by contradiction.

Assume that $L(M)$ is accepted by an $s_1(n)$ space-bounded Turing machine M' . By Corollary 17.3.4 we may assume that M' halts for all inputs. Recall that the encoding of M' occurs an infinite number of times in the enumeration of $\{0, 1\}^*$. Since

$$\inf_{n \rightarrow \infty} \frac{s_1(n)}{s_2(n)} = 0,$$

there is some $n \geq \text{length}(w)$ such that $s_1(n) < s_2(n)$. The string w can be padded with leading 1's to produce an encoding w' of M' with length exactly n .

Now consider the computation of M when run with input w' . Since $s_1(n) < s_2(n)$, M has sufficient space to simulate the computation of M' . Thus M accepts w' if, and only if, M' does not. Consequently, $L(M) \neq L(M')$. It follows that there is no $s_1(n)$ space-bounded machine that can accept $L(M)$. ■

The space constructibility of 2^n and 2^{2^n} combine with Theorem 17.3.5 to guarantee the existence of a language L that is not accepted by any machine with space bound 2^n . The latter bound has a rate of growth greater than any polynomial. Thus there is no Turing machine with polynomial space complexity that accepts L. Since space complexity provides a lower bound for time complexity, L cannot be accepted in polynomial time. Consequently, the membership problem for the language L is intractable.

The preceding argument establishes the existence of intractable languages without identifying a particular language whose space or time complexity is not polynomially bounded. In Section 17.5 we will show that a question concerning the language described by a regular expression requires exponential space.

Let $M = (Q,$
A computation of

17.4 P-Space, NP-Space, and Savitch's Theorem

The classes P and NP contain the languages that can be accepted in polynomial time by deterministic and nondeterministic Turing machines, respectively. In a similar manner, we can define classes of languages that are accepted by Turing machines in which the amount of space required for a computation grows only polynomially with the length of the input.

Definition 17.4.1

A language L is **decidable in polynomial space** if there is a Turing machine M that accepts L with $s_{CM} \in O(n^r)$, where r is a natural number independent of n . The family of languages decidable in polynomial space by a deterministic Turing machine is denoted P -Space. Similarly, the family of languages decidable in polynomial space by a nondeterministic Turing machine is denoted NP -Space.

There are some obvious inclusions concerning these new complexity classes. Clearly, P -Space \subseteq NP -Space. Moreover, by Theorem 17.3.1, $P \subseteq P$ -Space and $NP \subseteq NP$ -Space. The surprising relation is that between P -Space and NP -Space. Whether P is a proper subset of NP is an open question that has defied all attempts at a solution since it was posed in the 1960s. The answer to the analogous question for space complexity is known, P -Space = NP -Space. The fundamental difference between time and space complexity is that space can be reused during a computation.

We will show that every language accepted by a nondeterministic $s(n)$ space-bounded Turing machine is accepted deterministically with an $O(s(n)^2)$ space bound. It follows immediately that a language accepted in polynomial space by a nondeterministic Turing machine is also accepted in polynomial space by a deterministic machine. As usual, we will limit ourselves to the consideration of two-tape machines.

The construction of an equivalent deterministic machine from a nondeterministic Turing machine must specify a method for systematically examining all alternative computations of the nondeterministic machine. We begin by considering the potential space requirements of a standard approach for constructing the alternative computations of a nondeterministic Turing machine for an input string w . The critical feature for the space analysis of this approach is the need to store each machine configuration in the current computation to be able to generate successive computations. The configurations are maintained and accessed through a stack, producing a depth-first analysis of the nondeterministic computations.

If there is no transiting state, or all ap "back up" to q_i : Two questions mu much space is req the maximum num

The represent $s(n)$ requires enco only tape, the loca on the work tape. squares for the sta squares for the wor configuration can b

The answer to forming a nondeter desired bound on the the number of con with $s(n)$. Another

The critical ob sitions,

can be broken into

each with $k/2$ trans the first computation

We will employ a two-tape nondeter

ges without polynomially e described

ial time by manner, we the amount f the input.

that accepts f languages d P-Space. deterministic

ses. Clearly, NP-Space. is a proper since it was y is known, mplexity is

ce-bounded l. It follows istic Turing s usual, we

nistic Tur nitive com ential space ons of a non ace analysis nt computa intiated and stic compu

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a two-tape Turing machine with space bound $s(n)$. A computation of M with input w has the form

$$\begin{aligned} & q_0 : .BwB, .BB \\ \vdash & q_i : Bu.vB, x.y \\ \vdash & q_j : Bu'.v'B, x'.y'. \end{aligned}$$

If there is no transition for machine configuration $q_j : Bu'.v'B, x'.y'$ and q_j is not an accepting state, or all applicable transitions have already been examined, the computation must "back up" to $q_i : Bu.vB, x.y$ to try alternative transitions. A stack of machine configurations provides the last-in first-out strategy needed to test all the alternative computations. Two questions must be answered to determine the space complexity of this strategy: "How much space is required for the representation of a machine configuration?" and "What is the maximum number of configurations that may be stored?"

The representation of a configuration of a two-tape Turing machine with space bound $s(n)$ requires encoding the state of the machine, the location of the tape head on the read-only tape, the location of the tape head on the work tape, and the first $s(n)$ tape squares on the work tape. For an input string of length n , the space required is $\lceil \log_2(\text{card}(Q)) \rceil$ squares for the state, $\lceil \log_2(n+2) \rceil$ squares for the input tape head position, $\lceil \log_2(s(n)) \rceil$ squares for the work tape head position, and $s(n)$ squares for the work tape. Thus the entire configuration can be encoded in $O(s(n))$ space.

The answer to the second question shows that this straightforward approach to transforming a nondeterministic machine into a deterministic machine will not produce the desired bound on the space complexity of the deterministic computation. By Theorem 17.3.2, the number of configurations that need to be stored on the stack may grow exponentially with $s(n)$. Another approach is needed.

The critical observation for effectively reusing space is that a computation of k transitions,

$$\begin{aligned} & q_0 : .BwB, .BB \\ \vdash & q_j : Bu.vB, x.y, \end{aligned}$$

can be broken into two computations

$$\begin{aligned} & q_0 : .BwB, .BB \\ \vdash & q_j : Bu'.v'B, x'.y' \\ \vdash & q_i : Bu.vB, x.y, \end{aligned}$$

each with $k/2$ transitions. If the two computations are done sequentially, the space used in the first computation will be available for the second.

We will employ this memory reuse strategy to determine if a string w is accepted by a two-tape nondeterministic Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ with space bound

$s(n)$. Let cf_1, cf_2, \dots, cf_p be a listing of all possible machine configurations with w on the input tape, where cf_1 is the representation of the initial configuration $q_0 : .BwB, .BB$. The space bound $s(n)$ ensures us that the number of configurations is finite (Theorem 17.3.2).

The algorithm uses a divide-and-conquer technique to determine if a configuration cf_{i_2} is derivable from a configuration cf_{i_1} in k or fewer transitions. To answer this question, it suffices to find a configuration cf_{i_3} such that

1. $cf_{i_1} \vdash^* cf_{i_3}$ in $k/2$ transitions or fewer, and
2. $cf_{i_3} \vdash^* cf_{i_2}$ in $k/2$ transitions or fewer.

Similarly, to discover if $cf_{i_1} \vdash^* cf_{i_3}$ in $k/2$ transitions or fewer, it suffices to find configuration cf_{i_4} such that

1. $cf_{i_1} \vdash^* cf_{i_4}$ in $k/4$ transitions or fewer, and
2. $cf_{i_4} \vdash^* cf_{i_3}$ in $k/4$ transitions or fewer.

The procedure *Derive* in Algorithm 17.4.4 uses recursion to perform this search. The recursion tree associated with a call to *Derive*(cf_{i_1}, cf_{i_2}, k) is pictured in Figure 17.3. The node $[m, n]$ represents a call to determine if cf_{i_n} is derivable from cf_{i_m} . As illustrated in the figure, the evaluation of *Derive*(cf_{i_1}, cf_{i_2}, k) has at most $\lceil \log_2(k) \rceil$ nested recursive calls. The preceding observations are now used to produce a space bound for the deterministic algorithm that accepts the language defined by a nondeterministic machine.

Theorem 17.4.2 (Savitch's Theorem)

Let M be a two-tape nondeterministic Turing machine with space bound $s(n)$. Then $L(M)$ is accepted by a deterministic Turing machine with space bound $O(s(n)^2)$.

Proof. Algorithm 17.4.4 describes a recursive search for a derivation of string w . By Corollary 17.3.3, every string $w \in L(M)$ is accepted by a computation with at most $c^{s(n)}$ transitions. The machine configurations are sequentially examined and the recursive search procedure *Derive* is called for each accepting configuration of M in step 3.2. The parameters in the call are the initial configuration of M , an accepting configuration, and the transition bound $c^{s(n)}$. If one of the calls to *Derive* discovers a derivation, the algorithm halts and accepts the string. If all of the calls fail, then w is not derivable and the string is rejected.

All that remains is to determine the amount of memory required for this approach. On a recursive call, the calling procedure *Derive* stores an *activation record* that contains the values of its parameters and local variables. When the call is completed, activation record is used to restore the values. The activation record for *Derive* consists of the two machine configurations and the integral valued transition bound.

A Turing machine implementation of this algorithm must store the activation records on a tape. As previously noted, a machine configuration requires only $O(s(n))$ tape squares and consequently the space required by an activation record is also $O(s(n))$. The maximum number of nested calls is

$$\lceil \log_2(c^{s(n)}) \rceil = \lceil s(n) \log_2(c) \rceil \in O(s(n)).$$

Thus the total space for the $O(s(n))$ activation records is $O(s(n)^2)$. ■

[1, (k/2) + 2]

The bound
 $= \mathcal{NP}$ -Space.

Corollary 17.4.3

If L is in \mathcal{NP} -Space,

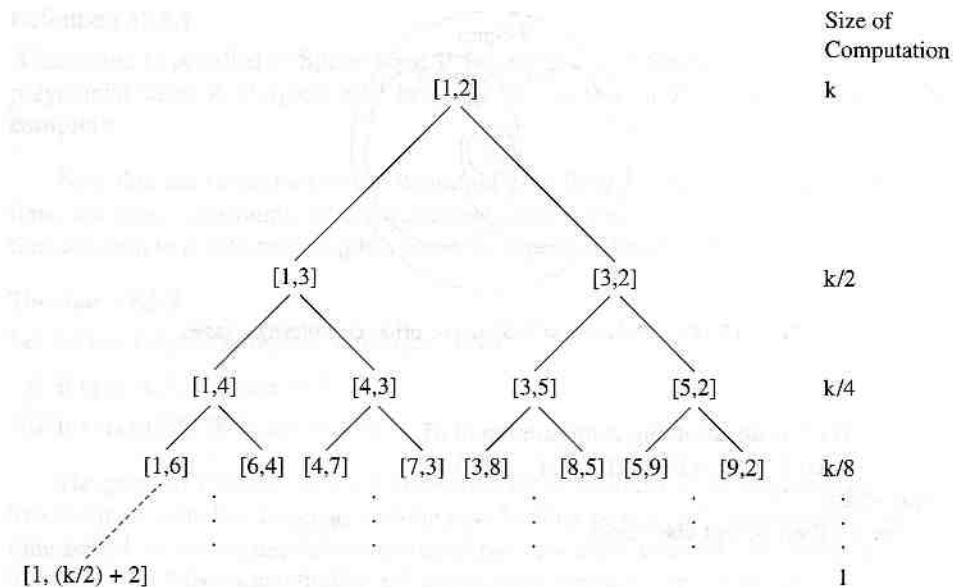
Proof. If L is in \mathcal{NP} -Space, then there is a polynomial space algorithm for L .

Algorithm 17.4.4 Recursive Simulation

input: Turing Machine M , string w , configuration cf_i , constant c , space bound $s(n)$

1. $found = \text{false}$
2. $i = 1$
3. **while** not $found$ **do**

3.1 $i := i + 1$

FIGURE 17.3 Recursion tree for $\text{Derive}(cf_{i_1}, cf_{i_2}, k)$.

The bound on the space complexity in Theorem 17.4.2 can be used to show that \mathcal{P} -Space = \mathcal{NP} -Space.

Corollary 17.4.3

If L is in \mathcal{NP} -Space, then L is in \mathcal{P} -Space.

Proof. If L is in \mathcal{NP} -Space, it is accepted by a nondeterministic Turing machine with a polynomial space bound $p(n)$. By Theorem 17.4.2, L is accepted by a deterministic Turing machine with space bound $O(p(n)^2)$ and consequently is in \mathcal{P} -Space. ■

Algorithm 17.4.4

Recursive Simulation of Nondeterministic Turing Machine

input: Turing Machine $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$

string $w \in \Sigma^*$

configurations cf_1, cf_2, \dots, cf_p of M

constant $c = 3 \cdot \text{card}(Q) \cdot \text{card}(\Gamma)$

space bound $s(n)$

1. $found = \text{false}$
 2. $i = 1$
 3. while not $found$ and $i < p$ do
 - 3.1 $i := i + 1$
- (check all accepting configurations)

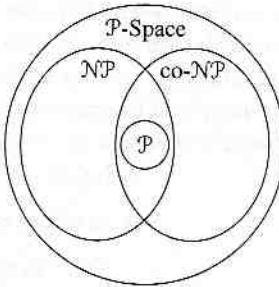


FIGURE 17.4 Relation of P-Space to other complexity classes.

```

3.2 if  $cf_i$  is an accepting configuration of M
    then  $found = \text{Derive}(cf_1, cf_i, c^{s(n)})$ 
end while
4. if  $found$  then accept else reject

 $\text{Derive}(cfs, cfe, k);$ 
begin
   $Derive = \text{false}$ 
  if  $k = 0$  and  $cfs = cfe$  then  $Derive = \text{true}$ 
  if  $k = 1$  and  $cfs \vdash cfe$  then  $Derive = \text{true}$ 
  if  $k > 1$  then do
     $i = 1$ 
    while not  $Derive$  and  $i < p$  do      (check all intermediate configurations)
       $i := i + 1$ 
       $Derive = \text{Derive}(cfs, cf_i, [k/2]) \text{ and } \text{Derive}(cf_i, cfe, [k/2])$ 
    end while
  end if
end.

```

Figure 17.4 shows the relationships between P-Space and the other complexity classes. It is not known whether $P\text{-Space} = NP$ or $P\text{-Space} = P$. However, it is believed that all of the inclusions in Figure 17.4 are proper.

17.5 P-Space Completeness

The notion of P-Space completeness is introduced to characterize the universal problems of the class P-Space and to provide a method for determining which, if any, of the inclusions $P \subseteq P\text{-Space}$ or $NP \subseteq P\text{-Space}$ are equalities.

Definition 17.5.1

A language Q is **P-Space complete** if it is in P-Space and every language in P-Space can be reduced to it in polynomial time.

Note that the term "complete" refers to the time, not space complexity. A language is said to be complete if it has a polynomial time solution to a problem.

Theorem 17.5.2

Let Q be a P-Space complete language.

- i) If Q is in P , $P = P\text{-Space}$
- ii) If Q is in NP , $NP = P\text{-Space}$

The proof of Theorem 17.5.2 is based on the fact that every language in P-Space can be reduced to a P-Space complete language in polynomial time. This reduces the question of the complexity of the language to the complexity of the reduction.

The remainder of this section is devoted to the proof of Theorem 17.5.2.

is P-Space complete. First, we must design a reduction that solves the problem. This is done if, and only if, it is the regular expression of all strings over its alphabet. This is left as an exercise. We do not describe all steps of the proof.

The second step is to show that the reduction runs in polynomial time. This is done by showing that the reduction of the Satisfiability Problem to the language accepts all strings in the language. For each string w , the reduction accepts w if, and only if,

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA with space bound $s(n)$. The reduction accepts a string w if, and only if, the regular expression R_w accepts w . We assume that there are no states in Q that are both accepting and non-accepting.

Definition 17.5.1

A language Q is called **P -Space hard** if for every $L \in P\text{-Space}$, L is reducible to Q in polynomial time. A P -Space hard language that is also in $P\text{-Space}$ is called **P -Space complete**.

Note that the reductions in the definition of P -Space completeness have polynomial time, not space constraints. This requirement ensures that the discovery of a polynomial-time solution to a P -Space complete problem implies $P\text{-Space} = P$.

Theorem 17.5.2

Let Q be a P -Space complete language. Then

- i) If Q is in P , $P\text{-Space} = P$.
- ii) If Q is in NP , $P\text{-Space} = NP$.

The proof of Theorem 17.5.2 follows from the reducibility of all languages in $P\text{-Space}$ to a P -Space complete language and the now familiar process of obtaining a polynomial-time bound on the sequential execution of two machines with polynomial-time bounds. Theorem 17.5.2 shows that finding a P -Space complete language in either P or NP answers the question of the proper inclusion of these classes in $P\text{-Space}$.

The remainder of this section is devoted to showing that the decision problem defined by

Input: Regular expression α over an alphabet Σ
Output: yes; if $\alpha \neq \Sigma^*$
 no; otherwise

is P -Space complete. Two steps are required to prove that this problem is P -Space complete. First, we must design a string representation for regular expressions and a Turing machine that solves the problem in polynomial space. That is, the Turing machine accepts a string if, and only if, it is the representation of a regular expression whose language does not consist of all strings over its alphabet. This step is done at the level of the acceptance of strings and is left as an exercise. The language consisting of representations of regular expressions that do not describe all strings will be denoted L_{REG} .

The second step is to show that every language in $P\text{-Space}$ is reducible to L_{REG} in polynomial time. The proof employs the strategy utilized in the proof of NP -completeness of the Satisfiability Problem. To show that a language L in $P\text{-Space}$ is reducible to L_{REG} , we transform computations of a space-bounded Turing machine M that accepts L into regular expressions. For each string $w \in \Sigma_M^*$, we construct a regular expression α_w such that M accepts w if, and only if, α_w does not contain all strings over its alphabet.

Let $M = (Q, \Sigma_M, \Gamma_M, \delta, q_0, F)$ be a one-tape deterministic Turing machine with space bound $s(n)$. The alphabets of M are subscripted to differentiate them from the alphabet of the regular expression that we will build from M and w . Without loss of generality, we assume that there are no transitions from the accepting states of M .

First, we define an alphabet Σ that allows us to represent computations of M as strings over Σ . The alphabet contains ordered pairs of the form $[q_i, a]$ and $[*, a]$ for each $q_i \in Q$ and $a \in \Gamma_M$. In addition to the ordered pairs, Σ contains the symbol \vdash . Intuitively, an ordered pair $[q_i, a]$ represents a tape position containing an a that is being scanned by the tape head. The asterisk in the first position, $[*, a]$, indicates that the tape head is not scanning this symbol. A sequence of $s(n)$ symbols can be used to represent any machine configuration of a computation of the machine M with an input of length n .

The initial configuration of M with input $w = a_1 \dots a_n$ is represented by the string

$$[q_0, B][*, a_1][*, a_2] \dots [*, a_n][*, B]^{s(n)-n-1},$$

where the exponent represents the concatenation of $s(n) - n - 1$ copies of $[*, B]$. The addition of the blanks following the input produces a representation of $s(n)$ tape squares, which is an upper bound on the space required by a computation. We will represent every configuration with exactly $s(n)$ symbols. The representation of a computation of M consists of a sequence of machine configurations separated by the symbol \vdash .

Now we design a regular expression α_w that contains all strings over Σ that are not the representation of a computation that accepts w . If we are successful in constructing such a regular expression,

$$\begin{aligned} \alpha_w \neq \Sigma^* &\text{ if, and only if, there is a computation of } M \text{ that accepts } w \\ &\text{if, and only if, } w \in L(M) \\ &\text{if, and only if, } w \in L. \end{aligned}$$

Consequently, an algorithm that decides L_{REG} will be able to determine whether a string w is in L . The construction of α_w utilizes the space bound on the computation of M .

Three conditions must be satisfied for a string over Σ to be the representation of an accepting computation of w :

1. The first $s(n)$ symbols must represent the initial configuration of M with input w .
2. The symbol \vdash separates configurations and each configuration must follow from the preceding configuration by a transition of M .
3. The final configuration must have an accepting state.

For each of the preceding conditions, we construct a regular expression that contains strings over Σ that do not satisfy the condition. The union of these expressions defines the set of all strings that are not the representation of an accepting computation of M with input w .

A string does not satisfy the first condition if its first symbol is not $[q_0, B]$, or if its first symbol matches $[q_0, B]$ but its second symbol is not $[*, a_1]$, or if its first two symbols match but its third is not $[*, a_2]$, and so on. Exactly $s(n)$ statements of the preceding form describe the strings that do not match the initial configuration. The language of the regular expression

$\alpha_1 = (\Sigma$

\cup

\cup

\cup

\cup

\cup

\cup

\cup

generates these expressions for

The regular

generates every

The second by a transition construct a reg agree with the to the right pro

in which $[q_i, a]$

For each tr

\cup

generates string expression has positions later. of a computation transition that expressions for

The transfo is used to show L is accepted by no transitions fr show that the expression α_1 is subexpressions

I as strings
 $q_i \in Q$ and
 an ordered
 tape head.
 inning this
 nfiguration

ie string

$[*, B]$. The
 pe squares,
 esent every
 M consists

are not the
 eting such a

r a string w
 M.
 tation of an
 input w .
 w from the

tains strings
 es the set of
 th input w .
 $B]$, or if its
 two symbols
 ceding form
 f the regular

$$\begin{aligned} \alpha_1 = & (\Sigma - \{[q_0, B]\})\Sigma^* \\ & \cup [q_0, B](\Sigma - \{[*, a_1]\})\Sigma^* \\ & \cup [q_0, B][*, a_1](\Sigma - \{[*, a_2]\})\Sigma^* \\ & \vdots \\ & \cup [q_0, B][*, a_1][*, a_2] \dots [*, a_{n-1}] (\Sigma - \{[*, a_n]\})\Sigma^* \\ & \cup [q_0, B][*, a_1][*, a_2] \dots [*, a_{n-1}][*, a_n] (\Sigma - \{[*, B]\})\Sigma^* \\ & \vdots \\ & \cup [q_0, B][*, a_1][*, a_2] \dots [*, a_{n-1}][*, a_n][*, B]^{s(n)-n-2} (\Sigma - \{[*, B]\})\Sigma^* \end{aligned}$$

generates these strings. The notation $(\Sigma - A)$ is used as an abbreviation of the regular expression for the subset of the alphabet obtained by deleting the elements in A .

The regular expression

$$\alpha_3 = (\Sigma - \{[q_i, a] \mid a \in \Gamma_M, q_i \in F\})^*$$

generates every string that does not contain a symbol with an accepting state.

The second condition requires that successive configurations be obtained as prescribed by a transition of M . Since each machine configuration has exactly $s(n)$ symbols, we construct a regular expression α_2 in which symbols $s(n) + 1$ tape positions apart do not agree with the result of a transition. A transition $\delta(q_i, a) = [b, q_j, R]$ that specifies a move to the right produces a substring in the representation

$$\dots [*, x][q_i, a][*, x] \dots \vdash \dots [*, x][*, b][q_j, x] \dots,$$

in which $[q_i, a]$ and $[*, b]$ are separated by exactly $s(n)$ symbols.

For each transition $\delta(q_i, a) = [b, q_j, R]$, the regular expression

$$\bigcup_{x \in \Gamma_M} \Sigma^*[q_i, a][*, x] \left(\Sigma^{s(n)}(\Sigma - [q_j, x])\Sigma^* \cup \Sigma^{s(n)-1}(\Sigma - [*, b])\Sigma^* \right)$$

generates strings that differ from the result of the transition. A string produced by this expression has an occurrence of $[q_i, a][*, x]$ and symbols other than $[*, b][q_j, x]$ $s(n) + 1$ positions later. Consequently, a string matching this condition cannot be the representation of a computation of M . In a similar manner, a regular expression is obtained for each transition that specifies a move to the left. The regular expression α_3 is the union of the expressions for each transition.

The transformation from space-bounded standard Turing machine to regular expression is used to show that L_{REG} is P-Space complete. Let L be any language in P-Space. Then L is accepted by a standard Turing machine M with a polynomial-space bound $p(n)$ with no transitions from the accepting states (Exercise 10). For a string w of length n , we must show that the size of the resulting regular expression grows polynomially in n . The regular expression α_1 is the union of $p(n)$ subexpressions, each of size $O(p(n))$. The size of the subexpressions in α_2 is also $O(p(n))$ and the number of subexpressions is independent of

the length of the input. Finally, the size of α_3 is a constant determined by the number of states and tape symbols of M . Thus the size of $\alpha = \alpha_1 \cup \alpha_2 \cup \alpha_3$ grows only polynomially with the length of a string w . The preceding argument demonstrates that L_{REG} is hard for the class P -Space. Combining this with a polynomial-space decision procedure for membership in L_{REG} , we conclude:

Theorem 17.5.3

The language L_{REG} is P -Space complete.

Because of the inclusion of NP in P -Space, every P -Space complete problem is also NP -hard. Thus L_{REG} is an example of an NP -hard problem for which there is no known nondeterministic polynomial-time solution.

17.6 An Intractable Problem

One measure of the importance of the class of NP -complete problems is the frequency with which they are encountered in diverse problem domains and applications. Even though there is no known polynomial-time algorithm that solves these problems, we cannot conclude that they are not in P . Generally speaking, showing that a language or a problem is in a complexity class is more easily accomplished than showing that it is outside of a class. Consider the ease in which we have been able to use a “guess-and-check” strategy to demonstrate that the Satisfiability Problem, the Hamiltonian Circuit Problem, and the Vertex Cover Problem are in NP . As of this time, no one has been able to prove that any of these problems are not in the class P .

The reason for the difference in difficulty is that producing one algorithm is sufficient to show that a problem is in P or NP or P -Space. Proving that a problem is not in one of these classes requires producing a lower bound on the time or space complexity of all algorithms that solve the problem. In this section we will see that a variation of the problem of recognizing L_{REG} is intractable, that is, that it is provably outside of P . In fact, we show that it is outside of P -Space and consequently not in either P or NP .

The family of regular expressions with squaring adds one more construction to the standard definition of regular expression given in Chapter 2. The *regular expressions with squaring* over an alphabet Σ are defined recursively from \emptyset , λ , and a , for every $a \in \Sigma$. If u and v are regular expressions with squaring over Σ , then so are $(u \cup v)$, (uv) , (u^*) , and (u^2) . As before, we can use associativity and operator precedence to reduce the number of parentheses.

Since the expression u^2 designates the same language as uu , the addition of squaring does not increase the languages that can be represented by regular expressions. However, the availability of the squaring operator reduces the length of expressions needed to describe a language. The squaring operation allows us to write an expression for the concatenation of 2^n copies of a regular expression u in $O(n)$ symbols,

$$(\dots ((u)^2)^2 \dots)^2,$$

applying the squaring operator to a regular expression with n symbols results in a regular expression with $2n$ symbols. This shows that regular expressions with squaring are at least as powerful as regular expressions.

We will show that the language L_{REG2} is P -Space complete. We begin by showing that L_{REG2} is in P -Space. We will show this by exhibiting a polynomial-space decision procedure for membership in L_{REG2} . We will do this by reducing the problem of membership in L_{REG2} to the problem of membership in L_{REG} . We will do this by showing that $L_{REG2} \leq_p L_{REG}$.

Let L be a language in P -Space. Let M be a polynomial-space Turing machine that decides L . We will show that there is a polynomial-space Turing machine M' that decides L_{REG2} such that $L_{REG2} \leq_p L$.

For each string $s \in \{0, 1\}^*$, let α_s be a regular expression with squaring that represents all strings that are in L and have length $|s|$. Let α_w be the same regular expression with squaring that represents all strings that are in L and have length $|w|$.

In Section 17.5, we showed that L_{REG} is P -Space complete. In particular, we showed that L_{REG} is in P -Space and that $L_{REG} \leq_p L$. We will use this result to show that L_{REG2} is in P -Space and that $L_{REG2} \leq_p L$.

followed by 2^n copies of the regular expression u . The regular expressions α_1 and α_3 require only $O(n)$ symbols. The regular expression α_2 requires $O(n)$ symbols because it contains $[*, B]$ and Σ . Consequently, L_{REG2} is in P -Space.

Let L_{REG2} denote the language $\{w \in \{0, 1\}^* \mid M' \text{ accepts } w\}$. We will show that the language L_{REG2} is P -Space complete.

Theorem 17.6.1

The language L_{REG2} is P -Space complete.

Proof. Assume that L_{REG2} is not P -Space complete. Then there is a polynomial-space Turing machine M' that decides L_{REG2} and there is a string $w \in \{0, 1\}^*$ such that $w \notin L_{REG2}$. Let M be a polynomial-space Turing machine that decides L . We will show that M decides L_{REG2} in polynomial space.

1. Input: a string $w \in \{0, 1\}^*$
2. Transformation: compute α_w
3. Computation: run M on α_w
4. Result: $w \in L$ if and only if $\alpha_w \in L$

of states
with the
he class
rship in

n is also
known

ency with
gh there
onclude
n is in a
a class.
ategy to
e Vertex
of these

sufficient
t in one
ity of all
problem
we show

on to the
ions with
 $z \in \Sigma$. If
 $[u^*]$, and
umber of
squaring
However,
describe
atenation

applying the squaring operation n times. Since complexity relates input length to time and space, a more compact representation of input may be accompanied by an increase in the complexity measures.

We will show that the problem of deciding whether the language of a regular expression with squaring does not contain all strings over its alphabet is not in P-Space. This is the same problem considered in the previous section; the sole difference is the presence of the squaring operator in regular expressions. The proof uses the representation of Turing machine computations as regular expressions, this time with squaring, developed in the preceding section.

Let L be a language accepted by a Turing machine with space bound 2^n but not by any Turing machine with space bound $2^{n/2}$. Theorem 17.5.3 assures us of the existence of such a language. Let M be a one-tape deterministic Turing machine with space complexity $sc_M(n) = 2^n$ that accepts L . As in the previous section, the computations of M can be represented as regular expressions over the alphabet $\Sigma = \{[q_i, a], [*], \vdash \mid q_i \in Q, a \in \Gamma\}$.

For each string $w = a_1 \dots a_n$ in Σ_M^* , we define a regular expression α_w whose language is all strings that do not represent a computation of M that accepts w . The construction of α_w uses the same approach as in Section 17.5, but we now use squaring to ensure that the length of α_w grows linearly with the length of w .

In Section 17.5, each machine configuration encoded in α_w had $s(n)$ tape squares where $s(n)$ was the space bound of the machine M . Here we choose $2^n + n + 1$ tape positions for the simplicity of the numeric manipulation. The string representation of the initial configuration of the computation of M with input w consists of

$$[q_0, B][*, a_1][*, a_2] \dots [*, a_n]$$

followed by 2^n copies of $[*, B]$. The squaring operation lets us describe this string with a regular expression of length $O(n)$. By examination, we see that subexpressions α_1 , α_2 , and α_3 require only $O(n)$ space when squaring is used to represent the exponential repetition of $[*, B]$ and Σ . Consequently, the length of α_w is $O(n)$.

Let L_{REG2} denote the set of all regular expressions with squaring of the form α_w such that the language of $\alpha_w \neq \Sigma^*$.

Theorem 17.6.1

The language L_{REG2} is intractable.

Proof. Assume that membership in L_{REG2} is decided by a polynomially space-bounded Turing machine M' . Combining the construction of α_w with the computation of M' produces the following sequence operations:

1. Input: a string $w \in \Sigma_M^*$ of length n
2. Transformation: construction of the regular expression α_w
3. Computation of M' : determination if $\alpha_w \neq \Sigma^*$
4. Result: $w \in L$ if, and only if, α_w is accepted by M' .

The entire process is completed in polynomial space and accepts the language $L(M)$. This is a contradiction since $L(M)$ is not accepted by any Turing machine with space complexity less than $2^{n/2}$. ■

The language L_{REG2} is clearly decidable. A simple strategy is to expand the occurrences of the squares in α_w to produce a standard regular expression for the same language. By Exercise 13, the question for the resulting expression can be answered in space that is polynomial to its length. Unfortunately, the space of the latter expression may grow exponentially with the length of w .

Exercises

1. Let Q be a language reducible to a language L in polynomial time. Prove that \bar{Q} is reducible to \bar{L} in polynomial time.
2. Design a two-tape Turing machine with space complexity $O(\log_2(n))$ that accepts $\{a^i b^i \mid i \geq 0\}$.
3. Let L be a language that is accepted by a Turing machine M whose computations with input of length n require at most $s(n)$ space. Note that we do not require that all computations of M terminate. Prove that L is recursive.
4. Show that \mathcal{P} -Space is closed under union and complementation.
5. For each space bound, design a Turing machine that shows that the function is fully space constructible:
 - a) $s(n) = n$
 - b) $s(n) = 3n$
 - c) $s(n) = n^2$
 - d) $s(n) = 2^n$
6. Let M be a Turing machine with space complexity $sc_M(n) = f(n) \geq n$. Recall that this means that there is some input of length n for which M uses exactly $sc_M(n)$ tape squares. Show that $f(n)$ is fully space constructible.
- *7. Design a one-tape deterministic Turing machine with input alphabet $\{1\}$ that uses exactly 2^n tape squares for input of length $n > 1$.
8. Let $s(n)$ be a fully space-constructible function with $s(n) \geq n$ and $s(0) > 0$. Show that there is a one-tape Turing machine that uses $s(n)$ tape squares for any input of length n .
9. Let M be an $s(n)$ space-bounded Turing machine with $s(n) \geq n$. Prove that there is a one-tape $s(n)$ space-bounded Turing machine that accepts $L(M)$.

10. Let L be a language that is accepted by a Turing machine with no transitions. Prove that L is polynomial-space complete.
11. Prove that the set of languages that are accepted by a Turing machine with space bounded by a polynomial is closed under union.
- *12. Is the set of languages that are accepted by a Turing machine with space bounded by a polynomial a proper subset of the set of languages that are accepted by a Turing machine with space bounded by a polynomial?
13. Show that the set of languages that are accepted by a Turing machine with space bounded by \mathcal{P} -Space is closed under union and complementation.
14. Prove Theorem 17.1.
15. Show that a language is in \mathcal{P} -Space if and only if it is in \mathcal{NP} -Space.

Bibliography

The existence of a universal Turing machine [1972] provided a theoretical basis for the notion of complexity follows from the work of Cook [1971]. The first proof that the class of NP-complete problems is not closed under complementation was given by Cook [1973].

- M). This complexity ■
- currences language. pace that may grow
- that \bar{Q} is
- at accepts
- putations require that
- on is fully
- Recall that $\Sigma_M(n)$ tape
- that uses
- Show that it of length
- there is a
10. Let L be a language in \mathcal{P} -Space. Prove that there is a one-tape Turing machine with no transitions from the accepting states that accepts L whose computations have a polynomial-space bound.
 11. Prove that the set of languages accepted by Turing machines with an $s(n) = \log_2(n)$ space bound is a proper subset of languages accepted with an $s(n) = n$ space bound.
 - * 12. Is the set of languages accepted by Turing machines with an $s(n) = n^r$ space bound a proper subset of languages accepted with a $s(n) = 2n^r$ space bound? Prove your answer.
 13. Show that the language L_{REG} is in \mathcal{P} -Space. *Hint:* Use the equivalence of \mathcal{P} -Space and NP -Space and design a nondeterministic polynomial space-bounded Turing machine that decides membership in L_{REG} .
 14. Prove Theorem 17.5.2.
 15. Show that any \mathcal{P} -Space complete language is NP-hard.

Bibliographic Notes

The existence of languages in $\text{NP} \setminus \mathcal{P}$, given $\mathcal{P} \neq \text{NP}$, was proved by Ladner [1975]. Karp [1972] provided the first proofs of \mathcal{P} -Space completeness. The presentation of space complexity follows that given in Hopcroft and Ullman [1979]. Additional results on space complexity can be found in that book and in Papdimitriou [1994]. The intractability of determining the language of extended regular expressions is from Meyer and Stockmeyer [1973].

PART V

Deterministic Parsing

Programming language definition and program compilation provide a direct link between the theory of formal languages and computer science applications. Compiling a program is a multistep process in which source code written in a high-level programming language is analyzed and transformed into executable machine or assembly language code. The two initial steps of the process, lexical analysis and parsing, check the syntactic correctness of the source code. Lexical analysis reads the characters in the source code and constructs a sequence of tokens (reserved word, identifiers, special symbols, and the like) of the programming language. A parser then determines whether the resulting token string satisfies the syntactic requirements specified in the programming language definition.

In 1960, ALGOL 60 became the first programming language to have its syntax formally defined using the rules of a grammar. Since that time, grammars have been the primary tool for defining the syntax of programming languages. The Backus-Naur form grammar for the programming language Java given in Appendix III defines the set of syntactically correct Java programs, but how can we determine whether a sequence of Java source code constitutes a syntactically correct program? The syntax is correct if the source code is derivable from the variable (*CompilationUnit*) using the rules of the grammar. To answer a question about the syntactic correctness of a Java program, or that of a program written in any language defined by a context-free grammar, parsing algorithms must be designed to generate derivations for strings in the language of a grammar. When a string is not in the language, these procedures should discover that no derivation exists.

In Chapter 18 we demonstrate the feasibility of algorithmic syntax checking. Both top-down and bottom-up parsing are introduced via searching a graph of possible derivations. The parsers perform an exhaustive search; the top-down parser examines all permissible rule applications and the bottom-up parser performs all possible reductions. In either case, the algorithms have the potential of examining many extraneous derivations. While these algorithms demonstrate the feasibility of algorithmic syntax analysis, their inefficiency makes them unacceptable for commercial compilers or interpreters.

In Chapters 19 and 20, we introduce two families of context-free grammars that can be parsed efficiently. To ensure the selection of the appropriate action, the parsers “look ahead”

in the string being analyzed. A parser is deterministic if at each step there is at most one rule that can successfully extend the current derivation. LL(k) grammars permit deterministic top-down parsing with a k symbol lookahead. LR(k) parsers use a finite automaton and k symbol lookahead to select a reduction or a shift in a bottom-up parse. The syntax of most modern programming languages is defined by LL or LR grammars, or variations of these, to permit efficient parsing. Throughout the introduction to parsing, we will assume that the grammars are unambiguous. This is a reasonable assumption for any grammar used to define a programming language.

In this chapter we will learn how to build a top-down and bottom-up parser for a grammar. We will also learn how to build a parser for a grammar whose non-terminals are strings over the alphabet of terminals. This is done by defining a new grammar whose non-terminals are strings over the alphabet of terminals. The parser then determines the string of terminals that corresponds to the string of non-terminals.

Top-down
applies rules
procedure; it
produce the st
form of the ru
However, using
halt and a nonc
of the bottom-

Grammars rules to efficient efficient parsing

18.1 The

In this intuitive
ions. Since any
will limit the se

one rule
ministic
on and k
of most
of these,
ime that
r used to

CHAPTER 18

Parsing: An Introduction

In this chapter we introduce two simple parsing algorithms to demonstrate the properties of top-down and bottom-up parsing. These algorithms are based on a breadth-first search of a graph whose paths represent derivations of the grammar. The input to a parser is a string over the alphabet of the grammar and the desired result is a derivation of the input string, if the string is in the language of the grammar. If not, the parser should indicate this by determining that no derivation is possible.

Top-down parsing begins with the start symbol of the grammar and systematically applies rules in an attempt to generate the input string. Bottom-up parsing reverses the procedure; it begins with the string itself and applies rules "backwards" in an attempt to produce the start symbol. These simple algorithms demonstrate the potential effect of the form of the rules on parsing. With an arbitrary grammar, the searches may not terminate. However, using a Greibach normal form grammar ensures that the top-down algorithm will halt and a noncontracting grammar without chain rules is sufficient to ensure the termination of the bottom-up parser.

Grammars that define programming languages require additional conditions on the rules to efficiently parse the strings of the language. Grammars specifically designed for efficient parsing are presented in Chapters 19 and 20.

18.1 The Graph of a Grammar

In this intuitive introduction, top-down parsing is described as searching a graph of derivations. Since any derivable terminal string has a leftmost derivation (Theorem 3.5.1), we will limit the search to leftmost derivations. If the grammar is unambiguous, the derivations

form a tree whose root is the start symbol of the grammar. It is important to note that for any interesting grammar, there are infinitely many derivations and the graph has infinitely many nodes.

Definition 18.1.1

Let $G = (V, \Sigma, P, S)$ be a context-free grammar. The **graph of the grammar** $g(G)$, denoted $g(G)$, is the labeled directed graph where the nodes and arcs are defined by

- $N = \{w \in (V \cup \Sigma)^* \mid S \xrightarrow[L]{*} w\}$
- $A = \{[v, w, k] \in N \times N \times N \mid v \xrightarrow[L]{k} w \text{ by application of rule } k\}$.

The nodes of the graph are the left sentential forms of the grammar, the strings derivable from the start symbol by a leftmost derivation. A string w is adjacent to v in $g(G)$ if $v \xrightarrow[L]{*} w$, that is, if w can be obtained from v by one leftmost rule application. The rules of the grammar are assigned numbers, which are used as the labels on the arcs of the graph and in the subsequent parsing algorithms. If the application of rule k is used to create an arc from v to w , the arc is labeled by k . A path from S to w in $g(G)$ represents a leftmost derivation of w from S .

The graph of a grammar is defined for an arbitrary context-free grammar. If the grammar is unambiguous, the resulting graph is a tree with the start symbol as the root. Since grammars used for deterministic parsing are unambiguous, we will feel free to use the terminology of trees and tree searching when describing the parsing strategies. In particular, we will call $g(G)$ the tree of derivations of the grammar G .

With the representation of derivations as paths in $g(G)$, the problem of deciding whether a string w is in the language of G is reduced to that of finding a path from S to w in $g(G)$. The representation of derivations as paths in a graph is illustrated in Figure 18.1 using the grammar AE (additive expressions):

1. $S \rightarrow A$
2. $A \rightarrow T \mid A + T$
3. $A \rightarrow A + T$
4. $T \rightarrow b$
5. $T \rightarrow (A)$.

The start symbol of AE is S and the language consists of arithmetic expressions constructed from the operator $+$, the single operand b , and parentheses. Strings generated by AE include b , $((b))$, $(b + b)$, and $(b) + b$. The grammar AE will be used throughout this chapter to demonstrate the properties of the parsing algorithms.

The number of rules that can be applied to the leftmost variable of a sentential form determines the number of children of the node. The presence of either direct or indirect recursion produces infinitely many nodes in the tree. Repeated applications of the directly recursive A rule and the indirectly recursive T rules generate arbitrarily long paths in the tree in Figure 18.1.

Standard tree derivations. In this case, since its nodes are strings, the search consists of applying the algorithm is

18.2 A Top-Down Approach

Paths in the tree. Our top-down parser finds a path for derivations of strings. The first node discovered and expanded is

To limit the search space, we consider only sentential forms that are prefixes of a string. A sentential form x is a prefix of a string y if x is a substring of y .

note that for
as infinitely

G , denoted

gs derivable
 \exists if $v \xrightarrow{L} w$,
he grammar
1 and in the
rc from v to
derivation of

the grammar
root. Since
e to use the
in particular,

ing whether
 w in $g(G)$.
1 using the

constructed
AE include
is chapter to

ential form
t or indirect
the directly
paths in the

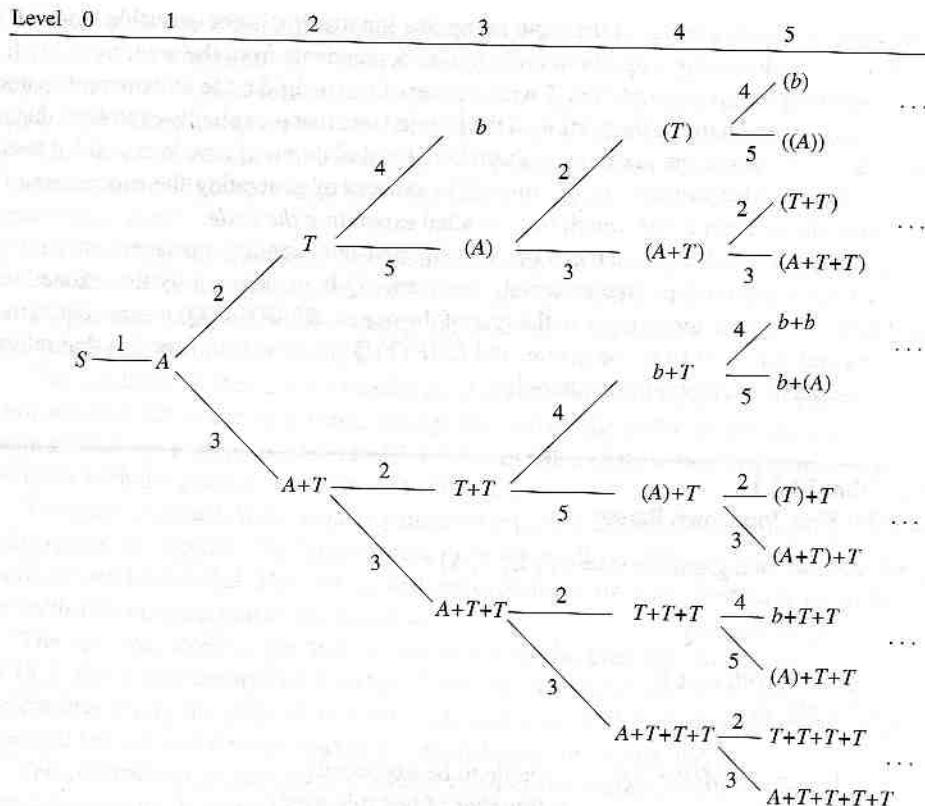


FIGURE 18.1 Tree of derivations of AE.

Standard tree searching techniques are used to examine the derivations in the tree of derivations. In tree searching terminology, the tree of derivations is called an *implicit tree* since its nodes have not been constructed prior to the invocation of the search algorithm. The search consists of building the tree as the paths are examined. An important feature of the algorithm is to explicitly construct as little of the implicit tree as possible.

18.2 A Top-Down Parser

Paths in the tree of derivations of a grammar represent leftmost derivations of the grammar. Our top-down parsing algorithm employs a breadth-first strategy to search the implicit tree for derivations of an input string. The algorithm accepts the input if a derivation of the string is discovered and rejects the input if the parser determines that no derivation is possible.

To limit the amount of searching required, the parser will use prefix matching to identify sentential forms that cannot appear in a derivation of the input string. The *terminal prefix* of a string is the substring occurring before the leftmost variable. That is, x is the terminal prefix of xBy if B is the first variable in the string. When a terminal prefix x of a string

xBy does not match a prefix of the input string, the input string is not derivable from xBy . We will call such a string a *dead end* and omit its descendants from the search.

The parser builds a search tree T with pointers from a child node to its parent (parent pointers). The search tree is the portion of the implicit tree that is explicitly examined during the parse. The rules of the grammar are numbered and children of a node are added to the tree according to the ordering of the rules. The process of generating the successors of a node and adding them to the search tree is called *expanding the node*.

A queue is used to implement the first-in, first-out memory management strategy required for a breadth-first tree traversal. The queue Q is maintained by three functions: $INSERT(x, Q)$ places the string x at the rear of the queue, $REMOVE(Q)$ returns the item at the front and deletes it from the queue, and $EMPTY(Q)$ is a Boolean function that returns true if the queue is empty, false otherwise.

Algorithm 18.2.1
Breadth-First Top-Down Parser

input: context-free grammar $G = (V, \Sigma, P, S)$

string $p \in \Sigma^*$

data structure: queue Q

1. initialize T with root S
 $\text{INSERT}(S, Q)$
 2. repeat
 - 2.1. $q := \text{REMOVE}(Q)$ (node to be expanded)
 - 2.2. $i := 0$ (number of last rule used)
 - 2.3. $\text{done} := \text{false}$ (Boolean indicator of expansion completion)

Let $q = uAv$ where A is the leftmost variable in q .

 - 2.4. repeat
 - 2.4.1. if there is no A rule numbered greater than i then $\text{done} := \text{true}$
 - 2.4.2. if not done then

Let $A \rightarrow w$ be the first A rule with number greater than i and let j be the number of this rule.

 - 2.4.2.1. if $uvw \notin \Sigma^*$ and the terminal prefix of uvw matches a prefix of p then
 - 2.4.2.1.1. $\text{INSERT}(uvw, Q)$
 - 2.4.2.1.2. Add node uvw to T . Set a pointer from uvw to q .

end if

end if

 - 2.4.3. $i := j$
 - until done or $p = uvw$
 - until $\text{EMPTY}(Q)$ or $p = uvw$
 3. if $p = uvw$ then accept else reject

from xBy .

ent (parent
ned during
ded to the
ssors of a

nt strategy
functions:
the item at
hat returns

The search tree is initialized with root S since a top-down algorithm attempts to find a derivation of an input string p from S . The algorithm consists of two nested repeat-until loops. The outer loop selects the first node q in the queue for expansion. The inner loop, step 2.4, generates the successors of q in the order specified by the numbering of the rules. There are three possibilities for each string uvw generated in the expansion of a string uAv : it may be a terminal string, it may be a dead end, or it may be a sentential form that requires further expansion.

If uvw is a terminal string, it represents the completion of a derivation and is not added to either the tree or the queue. The `until` statements check if it is the input string p . If so, the computation halts and accepts the string. Otherwise, the expansion of uAw continues with the generation of the next child.

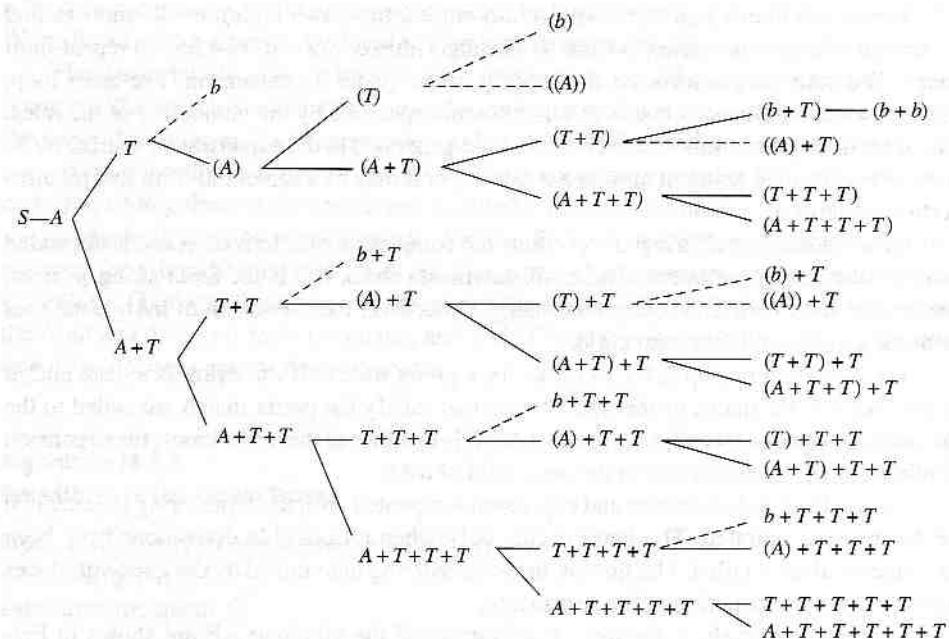
The condition in step 2.4.2.1 checks for a prefix match. If the string is a dead end, it is not added to the queue or the tree. Strings that satisfy the prefix match are added to the queue and the tree in steps 2.4.2.1.1 and 2.4.2.1.1. In either of these two cases, the expansion continues with the generation of the next child of “A”.

The cycle of node selection and expansion is repeated until the input string is generated or the queue is emptied. The latter occurs only when all possible derivations have been examined and have failed. The first-in, first-out ordering maintained by the queue produces a breadth-first construction of the search tree.

The first five levels of the tree of derivations of the grammar AE are shown in Figure 18.1. The parser evaluates the nodes of this tree in a level-by-level manner. The search tree constructed by the parse of $(b + b)$ is given in Figure 18.2. Sentential forms that are generated but not added to the search tree are indicated by dotted lines.

The comparison in step 2.4.2.1 checks whether the terminal prefix of the sentential form generated by the parser matches the input string. To obtain the information required for the match, the parser “reads” the input string as it builds derivations. The parser scans the input string in a left-to-right manner up to the leftmost variable in the derived sentential form. The growth of the terminal prefix causes the parser to read the entire input string. The derivation of $(b + b)$ exhibits the correspondence between the initial segment of the string scanned by the parser and the terminal prefix of the derived string:

Derivation	Input Read by Parser
$S \Rightarrow A$	λ
$\Rightarrow T$	λ
$\Rightarrow (A)$	(
$\Rightarrow (A + T)$	(
$\Rightarrow (T + T)$	(
$\Rightarrow (b + T)$	($b +$
$\Rightarrow (b + b)$	($b + b)$

FIGURE 18.2 A top-down parse of $(b + b)$.

A parser must not only be able to generate derivations for strings in the language, it must also determine when strings are not in the language. The bottom branch of the search tree in Figure 18.2 can potentially grow forever. The direct recursion of the rule $A \rightarrow A + T$ builds strings with any number of $+$'s as a suffix. In the search for a derivation of a string not in the language, the directly recursive A rule will never generate a prefix capable of terminating the search.

It may be argued that the string $A + T + T$ cannot lead to a derivation of $(b + b)$ and should be declared a dead end. It is true that the presence of two $+$'s guarantees that no sequence of rule applications can transform $A + T + T$ to $(b + b)$. However, such a determination requires a knowledge of the input string beyond the initial segment that has been scanned by the parser. The parsers in Chapter 19 will "look ahead" in the string, scanning beyond the terminal prefix generated by the parse, to aid in the selection of the subsequent action to be taken by the parser.

The possibility of entering an unending computation is caused by the presence of rules whose application does not increase the length of the terminal prefix. One approach to "fixing" Algorithm 18.2.1 is to use only grammars that do not allow this to happen. In Chapter 4 we showed that any context-free language is generated by a grammar in Greibach normal form. Every rule application in a Greibach normal form grammar either adds a terminal to the prefix of the derived string or completes a derivation. This is sufficient to

ensure that Algorithms 18.2.1 and 18.2.2 have a depth that is bounded by the size of the input string.

Although the search tree for a string in the language is finite, the search tree for a string not in the language can be very large. Lengthy derivations require a large search tree to incrementally build up the string. To keep the search tree from growing exponentially, parsing algorithms can be designed to terminate when they reach a certain depth. Such algorithms can be developed to ensure that the search tree does not grow with growth of the input string, provided that the required depth is bounded.

18.3 Reduction

In top-down parsing, the parser begins with the start symbol of the grammar until the input string is derived. The parser then produces the input string by applying the rules of the grammar. The parser generates the input string by applying the rules of the grammar. The parser consists of derivation trees.

Bottom-up parsers apply the rules "backwards." Bottom-up parsers are examined are those that are not in the language. The size of the search tree is bounded by the number of non-terminal symbols in the grammar. Only leftmost derivations are examined. Bottom-up parsers will examine only rightmost derivations. Bottom-up parsers search for an implicit rule.

The operation uses the expected, rule application.

A reduction replaces a non-terminal symbol on the left side. As implied in its name, a reduction is a reduction of the string. It is illustrated by the following example.

ensure that Algorithm 18.2.1 will halt for all input strings, since the explicit search tree will have a depth that is limited by the length of the input string.

Although the breadth-first algorithm succeeds in constructing a derivation for any string in the language, the practical application of this approach has several shortcomings. Lengthy derivations and grammars with a large number of rules cause the size of the search tree to increase rapidly. The exponential growth of the search tree is not limited to parsing algorithms but is a general property of breadth-first tree searches. If the grammar can be designed to utilize the prefix matching condition quickly or if other conditions can be developed to find dead ends in the search, the combinatorial problems associated with growth of the search tree may be delayed but not avoided. Better strategies are required.

18.3 Reductions and Bottom-Up Parsing

In top-down parsing, the search for a derivation examines paths in the tree of derivations of a grammar beginning with the start symbol. The search systematically constructs derivations until the input string is found or until it is determined that no derivation is capable of producing the input. The strategy is to perform an exhaustive search. With the exception of the pruning that results from the identification of dead ends, the same tree is generated for every input string. Searching in this manner examines many derivations that cannot possibly generate the input string. For example, the entire subtree with root $A + T$ in Figure 18.2 consists of derivations that cannot produce $(b + b)$.

Bottom-up parsing constructs a search tree whose root is the input string p and applies rules "backwards." By beginning the search with the input string, the only derivations that are examined are those that can generate p . This serves to focus the search and reduce the size of the search tree. To limit the size of the implicit graph, the top-down parser generated only leftmost derivations. Since the bottom-up parser constructs derivations backwards, it will examine only rightmost derivations. Bottom-up parsing may be considered to be a search of an implicit graph consisting of all strings that derive p by rightmost derivations.

The operation used to build a derivation in reverse is called a *reduction*. As may be expected, rule applications and reductions have an inverse relationship:

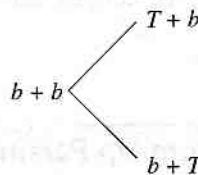
	Rule Application	Reduction
string	uAv	uvw
rule	$A \rightarrow w$	$\underline{A \rightarrow w}$
result	uvw	uAv

A reduction replaces the right-hand side of a rule with the single variable on the left-hand side. As implied in its name, a reduction is intended to reduce the length of a string as illustrated by the following examples.

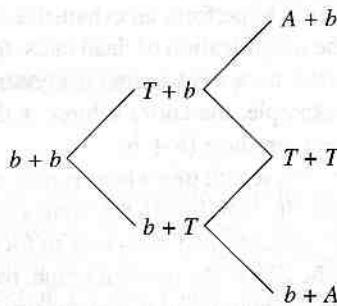
string	abb	$aAbAbbab$	BAA
rule	$A \rightarrow ab$	$A \rightarrow bAb$	$A \rightarrow AA$
reduction	Ab	$aAAbab$	BA

Whenever the length of the right-hand side of the rule is greater than one, a reduction produces a string of shorter length.

The grammar AE is used to illustrate the condition that is required to ensure that the search examines only rightmost derivations. Consider the two reductions of the string $b + b$ using the rule $T \rightarrow b$:



This tree represents the derivations $T + b \Rightarrow b + b$ and $b + T \Rightarrow b + b$. Building another level by adding all reductions of $b + T$ and $T + b$ produces



Notice that the string $T + T$ occurs twice, once in the derivation $T + T \Rightarrow T + b \Rightarrow b + b$ and once in $T + T \Rightarrow b + T \Rightarrow b + b$. The latter derivation is not rightmost and the corresponding reduction should not be considered in the search.

A reduction to uvw by a rule $A \rightarrow w$ produces a rightmost derivation only if the string v has no variables. If there is a variable in v , the corresponding derivation $uAv \Rightarrow uvw$ is not rightmost since the variable in v occurs to the right of A . This condition is incorporated into the bottom-up parser to ensure the generation of rightmost derivations. Example 18.3.1 illustrates the process of obtaining a rightmost derivation from a sequence of reductions.

Example 18.3.1

A reduction of the string $(b) + b$ to the start symbol S is given using the rules of the grammar AE.

Reductions with the rule $T \rightarrow b$ transforms the sentential form

Because the constructions are often said to co-

18.4 A Bottom-up Parser

The implicit graph $G = (V, \Sigma, P, S)$ is used to generate a rightmost derivation of p from v by one rightmost reduction at a time.

A breadth-first search manner. As with the operations *INSERT*,

Reduction	Rule
$(b) + b$	
$(T) + b$	$T \rightarrow b$
$(A) + b$	$A \rightarrow T$
$T + b$	$T \rightarrow (A)$
$A + b$	$A \rightarrow T$
$A + T$	$T \rightarrow b$
A	$A \rightarrow A + T$
S	$S \rightarrow A$

Reductions with the rules $T \rightarrow (A)$ and $A \rightarrow A + T$ reduce the length of the string and $T \rightarrow b$ transforms an occurrence of the terminal b into the variable T . Reversing the order of the sentential forms in the reduction of w to S produces the rightmost derivation

$$\begin{aligned} S &\Rightarrow A \\ &\Rightarrow A + T \\ &\Rightarrow A + b \\ &\Rightarrow T + b \\ &\Rightarrow (A) + b \\ &\Rightarrow (T) + b \\ &\Rightarrow (b) + b. \end{aligned}$$

Because the construction of a derivation terminates with the start symbol, bottom-up parsers are often said to construct rightmost derivations in reverse. \square

18.4 A Bottom-Up Parser

The implicit graph searched by a bottom-up parser is determined by both the grammar $G = (V, \Sigma, P, S)$ and the input string p . The nodes of the graph are strings that can derive p using rightmost rule applications. A node w is adjacent to a node v if w can be obtained from v by one rightmost rule application.

A breadth-first bottom-up parser builds a search tree with root p in a level-by-level manner. As with the top-down parser, the search tree T is constructed using the queue operations *INSERT*, *REMOVE*, and *EMPTY*.

Algorithm 18.4.1**Breadth-First Bottom-Up Parser**

input: context-free grammar $G = (V, \Sigma, P, S)$
 string $p \in \Sigma^*$

data structure: queue Q

1. initialize T with root p

$\text{INSERT}(p, Q)$

2. repeat

$q := \text{REMOVE}(Q)$

 2.1. for each rule $A \rightarrow w$ in P do

 2.1.1. for each decomposition uvw of w with $v \in \Sigma^*$ do

 2.1.1.1. $\text{INSERT}(uAv, Q)$

 2.1.1.2. Add node uAv to T . Set a pointer from uAv to q .

 end for

 end for

 until $q = S$ or $\text{EMPTY}(Q)$

3. if $q = S$ then accept else reject

($b + b$)

The search tree is initialized with root p . The remainder of the algorithm consists of selecting a node q for expansion, generating the reductions of q , and updating the queue and tree. Step 2.1.1 checks that no variable occurs to the right of the string w being reduced to ensure that only rightmost derivations are inserted into the queue and added to the search tree.

Figure 18.3 shows the search tree built when the string $(b + b)$ is analyzed by the bottom-up parser. Following the path from S to $(b + b)$ yields the rightmost derivation

$$\begin{aligned} S &\Rightarrow A \\ &\Rightarrow T \\ &\Rightarrow (A) \\ &\Rightarrow (A + T) \\ &\Rightarrow (A + b) \\ &\Rightarrow (T + b) \\ &\Rightarrow (b + b). \end{aligned}$$

Compare the search tree produced by the bottom-up parse of $(b + b)$ in Figure 18.3 with that produced by the top-down parse in Figure 18.2. Restricting the search to derivations that can produce $(b + b)$ significantly decreased the number of nodes generated.

The dramatic difference in the size of the search trees generated by the top-down and bottom-up parsers is shown for strings not in the language. Figure 18.4 shows the trees

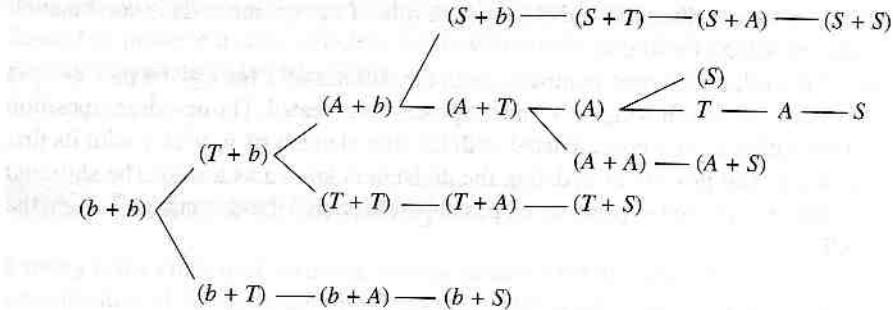
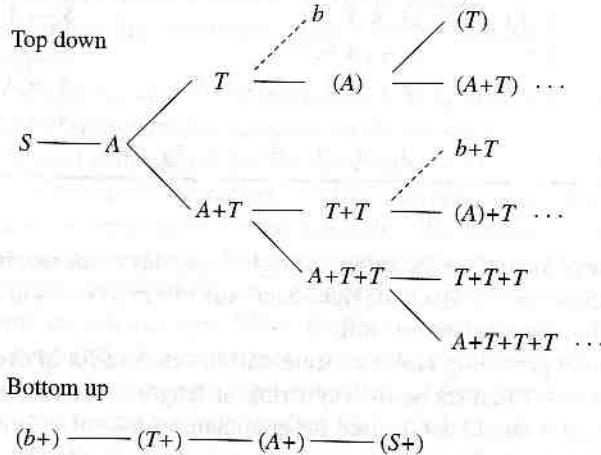
produced by the an
parse will never ter

One important
tions of a string q .

- i) q can be written
- ii) there is a rule

Determining the re
with substrings of t

A shift-and-con
string q is divided i
and y to q . The rig

FIGURE 18.3 Bottom-up parse of $(b + b)$.FIGURE 18.4 Top-down and bottom-up parse of $(b +)$.

produced by the analysis of the string $(b +)$. Due to the left recursion in AE, the top-down parse will never terminate. The bottom-up parse halts after examining four nodes.

One important step has been omitted in the preceding presentation—finding the reductions of a string q . We will now rectify that omission. A string q has a reduction if

- q can be written uvw , and
- there is a rule $A \rightarrow w$ in the grammar.

Determining the reductions of a string q requires matching the right-hand sides of the rules with substrings of the q .

A shift-and-compare strategy can be used to generate all reductions of a string q . The string q is divided into two substrings, $q = xy$. The initial division sets x to the null string and y to q . The right-hand side of each rule is compared with the suffixes of x . A match

occurs when x can be written uw and $A \rightarrow w$ is a rule of the grammar. This combination produces the reduction of q to uAy .

When all the rules have been compared with the suffixes of x for a given pair xy , q is divided into a new pair of substrings $x'y'$ and the process is repeated. The new decomposition is obtained by setting x' to x concatenated with the first element of y ; y' is y with its first element removed. The process of updating the division is known as a *shift*. The shift and compare operations are used to generate all possible reductions of the string $(A + T)$ in the grammar AE.

	x	y	Suffixes	Rule	Reduction
	λ	$(A + T)$	λ		
Shift	$($	$A + T)$	$(, \lambda$		
Shift	$(A$	$+ T)$	$(A, A, \lambda$	$S \rightarrow A$	$(S + T)$
Shift	$(A +$	$T)$	$(A +, A +, +, \lambda$		
Shift	$(A + T$	$)$	$(A + T, A + T, + T, T, \lambda$	$A \rightarrow A + T$	(A)
				$A \rightarrow T$	$(A + A)$
Shift	$(A + T)$	λ	$(A + T), A + T), + T), T),), \lambda$		

In generating the reductions of the string, the right-hand side of the rule must match a suffix of x . All other reductions in which the right-hand side of a rule occurs in x would have been discovered prior to the most recent shift.

As seen in the preceding table, a λ -rule will match a suffix in every decomposition xy and produce $n + 1$ reductions for any string of length n . Consequently, this bottom-up parsing algorithm should not be used for grammars with λ -rules. However, λ -rules will cause no problems for the bottom-up parsers considered in Chapter 20.

Does the breadth-first bottom-up parser halt for every possible input string, or is it possible for the algorithm to continue indefinitely in the repeat-until loop? If the string p is in the language of the grammar, a rightmost derivation will be found. If the length of the right-hand side of each rule is greater than 1, the reduction of a sentential form creates a new string of strictly smaller length. For grammars satisfying this condition, the depth of the search tree cannot exceed the length of the input string, assuring the termination of a parse with either a derivation or a failure. This condition, however, is not satisfied by grammars with rules of the form $A \rightarrow B$, $A \rightarrow a$, and $A \rightarrow \lambda$. In Exercise 11 you are asked to give a grammar and string for which Algorithm 18.4.1 will not terminate. Termination is assured for grammars without λ -rules and chain rules.

The efficiency of the bottom-up parser in Algorithm 18.4.1 is adversely affected by possible discovery multiple actions for a sentential form. For example, the string $A + T$ has two reductions and $b + b + b$ has three reductions using the rules of AE. The exhaustive search strategy will perform each reduction, add the resulting sentential forms to the search

tree, and generate needed to produce parsing are intro

18.5 Parsing

Parsing is the process specification of the code written in a language code is called parsing, and code compilation process briefly discuss the and parsing of a program.

Lexical analysis language. The tokens and special symbols space, comments, a source code that are errors when a sequence or special symbols letter, an underscore symbol that does not a regular language machine.

The parser checks syntactically correct a top-down or bottom-up the programming language (3.1) of the program.

The result of the application of the compiler must identify the grammar, and recover and special symbols a sequence of statements token string until it ends at the end of statement. At the end of the token string by

ation
, q is
sition
s first
t and
in the

action

T)

- A)

suffix
e been
osition
ottom-
es will

or is it
ring p
of the
eates a
n of the
a parse
mmars
give a
ssured

cted by
- T has
austive
search

tree, and generate their descendants. The ability to select a single action at each step is needed to produce a more efficient parse. Grammars that allow deterministic bottom-up parsing are introduced in Chapter 20.

18.5 Parsing and Compiling

Parsing is the process of verifying that the source code of a program satisfies the syntactic specification of the programming language. The entire process of transforming source code written in a high-level programming language into executable machine or assembly language code is *compiling* the program. Compiling a program consists of lexical analysis, parsing, and code generation. In addition to the analysis of syntax, the first two steps of the compilation process include semantic analysis and error identification and recovery. We will briefly discuss the additions beyond simple syntax analysis included in the lexical analysis and parsing of a program.

Lexical analysis scans the source code and creates a string of tokens of the programming language. The tokens of a programming language are the identifiers, reserved words, literals, and special symbols used in the language. The generation of a token string removes white space, comments, carriage return characters, linefeed characters, and other symbols in the source code that are not components of the language. The lexical analyzer also detects errors when a sequence of characters do not form syntactically correct identifiers, constants, or special symbols. The Java definition of an identifier requires the first symbol to be a letter, an underscore, or a dollar sign. When the lexical analyzer encounters a string of symbols that does not satisfy this requirement, or match any other Java reserved word or symbol, it generates an error message. Since the tokens of a programming language form a regular language, the lexical analysis is often performed with the aid of a finite-state machine.

The parser checks if the string of tokens produced by the lexical analyzer defines a syntactically correct program. This is accomplished by constructing a derivation, either in a top-down or bottom-up manner, of the string using the rules of the grammar that defines the programming language. A successful parse yields a derivation or parse tree (see Section 3.1) of the program.

The result of the parsers presented in the preceding two sections was simply an indication of the correctness of the input string—accept or reject. The parsing phase of a compiler must identify syntactic errors, generate informative error messages for the programmer, and recover from errors to continue the parse. Statement terminators, separators, and special symbols are invaluable for error recovery. If an error is discovered while parsing a sequence of statements, an error message is generated and the parser continues to read the token string until it encounters a symbol such as a semicolon or a bracket that designates the end of statement. At this point the parser will attempt to continue the parse of the remainder of the token string based on the token being read.

Semantic analysis uses information obtained during the parse to check for the semantic correctness of the statements generated by the parser. Semantic errors that may be identified in this phase of compilation include the declaration of a reserved word as an identifier, referencing a variable that has not been declared, multiple declarations of an identifier, and type incompatibility in assignments or operations.

After successful syntactic and semantic analysis, the parse tree is frequently used to create a representation of the program in an intermediate language. The intermediate representation is designed to facilitate the final step in the compilation: the translation into and optimization of the machine or assembly language code.

Exercises

1. Build the subgraph of the graph of the grammar of G consisting of the left sentential forms that are generated by derivations of length 3 or less.

$$\begin{aligned} G: S &\rightarrow aS \mid AB \mid B \\ A &\rightarrow abA \mid ab \\ B &\rightarrow BB \mid ba \end{aligned}$$

2. Build the subgraph of the graph of the grammar of G consisting of the left sentential forms that are generated by derivations of length 4 or less.

$$\begin{aligned} G: S &\rightarrow aSA \mid aB \\ A &\rightarrow bA \mid \lambda \\ B &\rightarrow cB \mid c \end{aligned}$$

Is G ambiguous?

In Exercises 3 through 7, trace the actions of the algorithm as it parses the input string using the grammar AE . If the input string is in the language, give the derivation constructed by the parser.

3. Algorithm 18.2.1 with input $(b) + b$.
4. Algorithm 18.2.1 with input $b + (b)$.
5. Algorithm 18.2.1 with input $((b))$.
6. Algorithm 18.4.1 with input $(b) + b$.
7. Algorithm 18.4.1 with input $(b))$.
8. Give the first five levels of the search tree generated by Algorithms 18.2.1 and 18.4.1 when parsing the string $b) + b$.

9. Let G be the

- a) Give a
- b) Give the
- c) Give the

10. Let G be the

- a) Give a
- b) Give the
- c) Give the

11. Construct a

loops indefi

12. Assume that

18.4.1 to no

modified alg

search tree i

Bibliographic

The parsers presented in this chapter are particular applications of the general LR(0) parser [1968] and in most cases are based on LR(0) grammars and compiling can be done by generating an LR(0) parser. For deterministic parsing, see the bibliography in Chapter 17.

he semantic
e identified
n identifier,
entifier, and

uently used
ntermediate
isolation into

ft sentential

ft sentential

string using
nstructed by

1 and 18.4.1

9. Let G be the grammar

1. $S \rightarrow aS$
2. $S \rightarrow AB$
3. $A \rightarrow bAa$
4. $A \rightarrow a$
5. $B \rightarrow bB$
6. $B \rightarrow b.$

- a) Give a set-theoretic definition for $L(G)$.
- b) Give the tree built by the top-down parse of $baab$.
- c) Give the tree built by the bottom-up parse of $baab$.

10. Let G be the grammar

1. $S \rightarrow A$
2. $S \rightarrow AB$
3. $A \rightarrow abA$
4. $A \rightarrow b$
5. $B \rightarrow baB$
6. $B \rightarrow a.$

- a) Give a regular expression for $L(G)$.
- b) Give the tree built by the top-down parse of $abbaaa$.
- c) Give the tree built by the bottom-up parse of $abbaaa$.

11. Construct a grammar G without λ -rules and a string $p \in \Sigma^*$ such that Algorithm 18.4.1 loops indefinitely in attempting to parse p .
12. Assume that the start symbol S of the grammar is nonrecursive. Modify Algorithm 18.4.1 to not continue the search whenever a string contains S . Trace the parse of your modified algorithm with grammar AE and input $(b + b)$. Compare your tree with the search tree in Figure 18.3.

Bibliographic Notes

The parsers presented in this chapter are graph searching algorithms modified for this particular application. A thorough exposition of graph and tree traversals is given in Knuth [1968] and in most texts on data structures. A comprehensive introduction to syntax analysis and compiling can be found in Aho, Sethi, and Ullman [1986]. Grammars amenable to deterministic parsing techniques are presented in Chapters 19 and 20. For references to parsing, see the bibliographic notes following those chapters.

CHAPTER 19

LL(k) Grammars

The fundamental cause of the inefficiency of the algorithms presented in Chapter 18 is the possibility of having several options when expanding a node in the search tree. The top-down parser extends the derivation by applying every A rule, where A is the leftmost variable in the sentential form. The bottom-up parser may have several reductions for a given string. In either case, the parsers perform all the possible actions, add the resulting sentential forms to the search tree, and generate their descendants.

A parsing algorithm is deterministic if, at each step, there is sufficient information to select a single action to be performed. For a top-down parser, this means being able to determine which of the possible rules to apply. The LL(k) grammars constitute the largest subclass of context-free grammars that permits deterministic top-down parsing using a k -symbol lookahead. The notation LL describes the parsing strategy for which these grammars are designed; the input string is scanned in a left-to-right manner and the parser generates a leftmost derivation. The lookahead, reading beyond the portion of input string generated by the parser, provides the additional information needed to select the appropriate action.

Throughout this chapter, all derivations and rule applications are leftmost. We also assume that the grammars are unambiguous and do not contain useless symbols. Techniques for detecting and removing useless symbols were presented in Section 4.4.

19.1 Lookahead in Context-Free Grammars

A top-down parser attempts to construct a leftmost derivation of an input string p . The parser extends derivations of the form $S \xrightarrow{*} uAv$, where u is a prefix of p , by applying an

A rule. “Looking ahead” in the input string can reduce the number of A rules that must be examined. If $p = uaw$, the terminal a is obtained by looking one symbol beyond the prefix of the input string that has been generated by the parser. Using the lookahead symbol permits an A rule whose right-hand side begins with a terminal other than a to be eliminated from consideration. The application of any such rule generates a terminal string that is not a prefix of p .

Consider a derivation of the string $acbb$ in the regular grammar

$$\begin{aligned} G: \quad S &\rightarrow aS \mid cA \\ A &\rightarrow bA \mid cB \mid \lambda \\ B &\rightarrow cB \mid a \mid \lambda. \end{aligned}$$

The derivation begins with the start symbol S and lookahead symbol a . The grammar contains two S rules, $S \rightarrow aS$ and $S \rightarrow cA$. Clearly, applying $S \rightarrow cA$ cannot lead to a derivation of $acbb$ since c does not match the lookahead symbol. It follows that a derivation of $acbb$ must begin with an application of the rule $S \rightarrow aS$.

After the application of the S rule, the lookahead symbol is advanced to c . Again, there is only one S rule that generates c . Comparing the lookahead symbol with the terminal in each of the appropriate rules permits the deterministic construction of derivations in G .

Prefix Generated	Lookahead Symbol	Rule	Derivation
λ	a	$S \rightarrow aS$	$S \Rightarrow aS$
a	c	$S \rightarrow cA$	$\Rightarrow acA$
ac	b	$A \rightarrow bA$	$\Rightarrow acbA$
acb	b	$A \rightarrow bA$	$\Rightarrow acbbA$
$acbb$	λ	$A \rightarrow \lambda$	$\Rightarrow acbb$

Looking ahead one symbol is sufficient to construct derivations deterministically in the grammar G . A more general approach allows the lookahead to consist of the portion of the input string that has not been generated. An intermediate step in a derivation of a terminal string p has the form $S \xrightarrow{*} uAv$, where $p = ux$. The string x is called a *lookahead string* for the variable A . The lookahead set of A consists of all lookahead strings for that variable.

Definition 19.1.1

Let $G = (V, \Sigma, P, S)$ be a context-free grammar and $A \in V$.

- i) The lookahead set of the variable A , $LA(A)$, is defined by

$$LA(A) = \{x \mid S \xrightarrow{*} uAv \xrightarrow{*} ux \in \Sigma^*\}.$$

- ii) For each rule $A \rightarrow w$ in P , the lookahead set of the rule $A \rightarrow w$ is defined by

$$LA(A \rightarrow w) = \{x \mid wv \xrightarrow{*} x \text{ where } x \in \Sigma^* \text{ and } S \xrightarrow{*} uAv\}.$$

$LA(A)$ consists of all left sentential forms whose subderivations Av end in Σ^* .

Let $A \rightarrow w_1, \dots, A \rightarrow w_n$ be used to select the rules for A . Then, that is, when the set $LA(A)$ is

- i) $LA(A) = \bigcup_{i=1}^n LA(A \rightarrow w_i)$
 ii) $LA(A \rightarrow w_i)$

The first condition is the definition of the lookahead set of A . The second condition is a partial derivation of A in G . The third condition is the definition of the lookahead set of A in $LA(A \rightarrow w_k)$. Condition i) is the union of the three conditions. Condition ii) is the successful completion of the derivation of A in $LA(A \rightarrow w_k)$.

Example 19.1.1

The lookahead sets of the variables in the grammar G are

$LA(S)$ consists of all strings x such that the rule $S \rightarrow Aabd$ or $S \rightarrow cAbcd$ generates ux .

Knowledge of the first character of the lookahead string is sufficient to determine the S rule.

To construct the lookahead set of A , we must consider all the left sentential forms av such that $A \rightarrow av$ is a rule of G . In this case, $A \rightarrow Aabd$ and $A \rightarrow cAbcd$. Thus, $LA(A) = abd \cup cabcd$.

The substring ab can be derived from A by the rule $A \rightarrow Aabd$. Thus $ab \in LA(A)$.

at must
ond the
symbol
minated
at is not

grammar
ead to a
erivation

in, there
minal in
in G.

lly in the
on of the
terminal
string for
variable.

$\text{LA}(A)$ consists of all terminal strings derivable from strings Av , where uAv is a left sentential form of the grammar. $\text{LA}(A \rightarrow w)$ is the subset of $\text{LA}(A)$ in which the subderivations $Av \xrightarrow{*} x$ are initiated with the rule $A \rightarrow w$.

Let $A \rightarrow w_1, \dots, A \rightarrow w_n$ be the A rules of a grammar G . The lookahead string can be used to select the appropriate A rule whenever the sets $\text{LA}(A \rightarrow w_i)$ partition $\text{LA}(A)$, that is, when the sets $\text{LA}(A \rightarrow w_i)$ satisfy

- i) $\text{LA}(A) = \bigcup_{i=1}^n \text{LA}(A \rightarrow w_i)$, and
- ii) $\text{LA}(A \rightarrow w_i) \cap \text{LA}(A \rightarrow w_j) = \emptyset$ for all $1 \leq i < j \leq n$.

The first condition is satisfied for every context-free grammar; it follows directly from the definition of the lookahead sets. If the lookahead sets satisfy (ii) and $S \xrightarrow{*} uAv$ is a partial derivation of a string $p = ux \in L(G)$, then x is an element of exactly one set $\text{LA}(A \rightarrow w_k)$. Consequently, $A \rightarrow w_k$ is the only A rule whose application can lead to a successful completion of the derivation.

Example 19.1.1

The lookahead sets are constructed for the variables and the rules of the grammar

$$\begin{aligned} G_1: S &\rightarrow Aabd \mid cAbcd \\ A &\rightarrow a \mid b \mid \lambda. \end{aligned}$$

$\text{LA}(S)$ consists of all terminal strings derivable from S . Every terminal string derivable from the rule $S \rightarrow Aabd$ begins with a or b . On the other hand, derivations initiated by the rule $S \rightarrow cAbcd$ generate strings beginning with c .

$$\begin{aligned} \text{LA}(S) &= \{aab, bab, abd, cab, cbb, cbd\} \\ \text{LA}(S \rightarrow Aabd) &= \{aab, bab, abd\} \\ \text{LA}(S \rightarrow cAbcd) &= \{cab, cbb, cbd\} \end{aligned}$$

Knowledge of the first symbol of the lookahead string is sufficient to select the appropriate S rule.

To construct the lookahead set for the variable A we must consider derivations from all the left sentential forms of G_1 that contain A . There are only two such sentential forms, $Aabd$ and $cAbcd$. The lookahead sets consist of terminal strings derivable from $Aabd$ and $Abcd$.

$$\begin{aligned} \text{LA}(A \rightarrow a) &= \{aab, abd\} \\ \text{LA}(A \rightarrow b) &= \{bab, bbd\} \\ \text{LA}(A \rightarrow \lambda) &= \{abd, bcd\} \end{aligned}$$

The substring ab can be obtained by applying $A \rightarrow a$ to $Abcd$ and by applying $A \rightarrow \lambda$ to $Aabd$. Thus a two-symbol lookahead is not sufficient for selecting the correct A .

rule. Looking ahead three symbols in the input string provides sufficient information to discriminate between these rules. A top-down parser with a three-symbol lookahead can deterministically construct derivations in the grammar G_1 . \square

A lookahead string of the variable A is the concatenation of the results of two derivations, one from the variable A and one from the portion of the sentential form following A . Example 19.1.2 emphasizes the dependence of the lookahead set on the sentential form.

Example 19.1.2

A lookahead string of G_2 receives at most one terminal from each of the variables A , B , and C .

$$G_2: S \rightarrow ABCabcd$$

$$A \rightarrow a \mid \lambda$$

$$B \rightarrow b \mid \lambda$$

$$C \rightarrow c \mid \lambda$$

The only left sentential form of G_2 that contains A is $ABCabcd$. The variable B appears in $aBCabcd$ and $BCabcd$, both of which can be obtained by the application of an A rule to $ABCabcd$. In either case, $BCabcd$ is used to construct the lookahead set. Similarly, the lookahead set $LA(C)$ consists of strings derivable from $Cabcd$.

$$LA(A \rightarrow a) = \{abcabcd, acabcd, ababcd, aabcd\}$$

$$LA(A \rightarrow \lambda) = \{bcabcd, cabcd, babcd, abcd\}$$

$$LA(B \rightarrow b) = \{bcabcd, babcd\}$$

$$LA(B \rightarrow \lambda) = \{cabcd, abcd\}$$

$$LA(C \rightarrow c) = \{cabcd\}$$

$$LA(C \rightarrow \lambda) = \{abcd\}$$

One-symbol lookahead is sufficient for selecting the B and C rules. A string with prefix abc can be derived from the sentential form $ABCabcd$ using the rule $A \rightarrow a$ or $A \rightarrow \lambda$. Four-symbol lookahead is required to parse the strings of G_2 deterministically. \square

The lookahead sets $LA(A)$ and $LA(A \rightarrow w)$ may contain strings of arbitrary length. The selection of rules in the previous examples needed only fixed-length prefixes of strings in the lookahead sets. The k -symbol lookahead sets are obtained by truncating the strings of the sets $LA(A)$ and $LA(A \rightarrow w)$. A function $trunc_k$ is introduced to simplify the definition of the fixed-length lookahead sets.

Definition 19.1.2

Let $G = (V, \Sigma, P, S)$ be a context-free grammar and let k be a natural number greater than zero.

i) $trunc_k$ is a fu

$trunc_k($

for all $X \in \Sigma$

ii) The length-

iii) The length-

Example 19.1.3

The length-three 1

Since there is no s
A rules, a three-sy

Example 19.1.4

The language $\{a^i a$
minimal-length lo
tions are given for

ation to
ead can

□

deriva-
wing A.
form.

s A, B,

- i) trunc_k is a function from $\mathcal{P}(\Sigma^*)$ to $\mathcal{P}(\Sigma^*)$ defined by

$$\text{trunc}_k(X) = \{u \mid u \in X \text{ with } \text{length}(u) \leq k \text{ or } uv \in X \text{ with } \text{length}(u) = k\}$$

for all $X \in \mathcal{P}(\Sigma^*)$.

- ii) The length- k lookahead set of the variable A is the set

$$\text{LA}_k(A) = \text{trunc}_k(\text{LA}(A)).$$

- iii) The length- k lookahead set of the rule $A \rightarrow w$ is the set

$$\text{LA}_k(A \rightarrow w) = \text{trunc}_k(\text{LA}(A \rightarrow w)).$$

Example 19.1.3

The length-three lookahead sets for the rules of the grammar G_1 from Example 19.1.1 are

$$\text{LA}_3(S \rightarrow Aabd) = \{aab, bab, abd\}$$

$$\text{LA}_3(S \rightarrow cAbcd) = \{cab, cbb, cbc\}$$

$$\text{LA}_3(A \rightarrow a) = \{aab, abc\}$$

$$\text{LA}_3(A \rightarrow b) = \{bab, bbc\}$$

$$\text{LA}_3(A \rightarrow \lambda) = \{abd, bcd\}.$$

Since there is no string in common in the length-three lookahead sets of the S rules or the A rules, a three-symbol lookahead is sufficient to determine the appropriate rule of G_1 . □

Example 19.1.4

The language $\{a^iabc^i \mid i > 0\}$ is generated by each of the grammars G_1 , G_2 , and G_3 . The minimal-length lookahead sets necessary for discriminating between alternative productions are given for these grammars.

	Rule	Lookahead Set
$G_1:$	$S \rightarrow aSc$	{aaa}
	$S \rightarrow aabc$	{aab}
$G_2:$	$S \rightarrow aA$	
	$A \rightarrow Sc$	{aa}
	$A \rightarrow abc$	{ab}
$G_3:$	$S \rightarrow aaAc$	
	$A \rightarrow aAc$	{a}
	$A \rightarrow b$	{b}

A one-symbol lookahead is insufficient for determining the S rule in G_1 since both of the alternatives begin with the symbol a . In fact, three-symbol lookahead is required to determine the appropriate rule. Grammar G_2 is constructed from G_1 by using the S rule to generate the leading a . The variable A is added to generate the remainder of the right-hand side of the S rules of G_1 . This technique is known as *left factoring* since the leading a is factored out of the rules $S \rightarrow aSc$ and $S \rightarrow aabc$. Left factoring the S rule reduces the length of the lookahead needed to select the rules.

A lookahead of length 1 is sufficient to parse strings with the rules of G_3 . The recursive A rule generates an a while the nonrecursive rule terminates the derivation by generating a b . \square

19.2 FIRST, FOLLOW, and Lookahead Sets

We have seen that lookahead sets can be used to select the appropriate rule to apply to derive a desired string. To incorporate this information into a parser, it is necessary to be able to generate the lookahead sets for each variable and rule. In this section we introduce the FIRST and FOLLOW sets, which will be used for constructing the lookahead sets directly from the rules of the grammar.

The lookahead set $LA_k(A)$ contains prefixes of at most length k of strings that can be derived from the variable A . If A derives strings of length less than k , the remainder of the lookahead comes from derivations that follow A in the sentential forms of the grammar. For each variable A , sets $FIRST_k(A)$ and $FOLLOW_k(A)$ are introduced to provide the information required for constructing the lookahead sets. $FIRST_k(A)$ contains prefixes of terminal strings derivable from A . $FOLLOW_k(A)$ contains prefixes of terminal strings that can follow the strings derivable from A . For convenience, a set $FIRST_k$ is defined for every string in $(V \cup \Sigma)^*$.

Definition 19.2.1

Let G be a context-free grammar. For every string $u \in (V \cup \Sigma)^*$ and $k > 0$, the set $FIRST_k(u)$ is defined by

$$FIRST_k(u) = trunc_k(\{x \mid u \xrightarrow{*} x, x \in \Sigma^*\}).$$

Example 19.2.1

FIRST sets are constructed for the strings S and ABC using the grammar G_2 from Example 19.1.2.

$$FIRST_1(ABC) = \{a, b, c, \lambda\}$$

$$FIRST_2(ABC) = \{ab, ac, bc, a, b, c, \lambda\}$$

$$FIRST_3(S) = \{abc, aca, aba, aab, bca, bab, cab\} \quad \square$$

Recall that
 $\{xy \mid x \in X$ and
for the $FIRST_k$

Lemma 19.2.2

For every $k > 0$

1. $FIRST_k(\lambda)$
2. $FIRST_k(a)$
3. $FIRST_k(au)$
4. $FIRST_k(uv)$
5. if $A \rightarrow w$ is

Definition 19.2.2

Let G be a context-free grammar defined by

The set $FOLLOW_k(A)$ contains prefixes of terminal strings that follow the variable A in derivations in the sentential form of the grammar.

Example 19.2.2

The FOLLOW sets for the grammar G_2 are

$FOLLOW_1(A) = \{c\}$
 $FOLLOW_1(B) = \{a\}$
 $FOLLOW_2(A) = \{b\}$
 $FOLLOW_2(B) = \{c\}$

The FOLLOW sets for the grammar G_3 are

$FOLLOW_1(A) = \{a\}$
 $FOLLOW_1(B) = \{b\}$
 $FOLLOW_2(A) = \{a, b\}$
 $FOLLOW_2(B) = \{b, c\}$

right-hand side. The strings that follow the variable A in derivations that follow A . If $A \rightarrow uBv$, then uBv follows A . The strings that follow B follow A .

Lemma 19.2.4

For every $k > 0$

1. $FOLLOW_k(\lambda)$
2. if $A \rightarrow uBv$ is
3. if $A \rightarrow uBv$ is

ince both
quired to
e S rule
he right-
leading a
uces the
ecursive
enerating
□

to derive
e able to
duce the
; directly

at can be
ler of the
grammar.
wide the
efixes of
ings that
for every

RST_k(u)

n Exam-

Recall that the concatenation of two sets X and Y is denoted by juxtaposition, XY = {xy | x ∈ X and y ∈ Y}. Using this notation, we can establish the following relationships for the FIRST_k sets.

Lemma 19.2.2

For every $k > 0$,

1. FIRST_k(λ) = { λ }
2. FIRST_k(a) = { a }
3. FIRST_k(au) = { av | $v \in \text{FIRST}_{k-1}(u)$ }
4. FIRST_k(uv) = trunc_k(FIRST_k(u)FIRST_k(v))
5. if $A \rightarrow w$ is a rule in G, then FIRST_k(w) ⊆ FIRST_k(A).

Definition 19.2.3

Let G be a context-free grammar. For every $A \in V$ and $k > 0$, the set FOLLOW_k(A) is defined by

$$\text{FOLLOW}_k(A) = \{x \mid S \xrightarrow{*} uAv \text{ and } x \in \text{FIRST}_k(v)\}.$$

The set FOLLOW_k(A) consists of prefixes of terminal strings that can follow the variable A in derivations in G. Since the null string follows every derivation from the sentential form consisting solely of the start symbol, $\lambda \in \text{FOLLOW}_k(S)$.

Example 19.2.2

The FOLLOW sets of length 1 and 2 are given for the variables of G₂.

FOLLOW ₁ (S) = { λ }	FOLLOW ₂ (S) = { λ }
FOLLOW ₁ (A) = { a, b, c }	FOLLOW ₂ (A) = { ab, bc, ba, ca }
FOLLOW ₁ (B) = { a, c }	FOLLOW ₂ (B) = { ca, ab }
FOLLOW ₁ (C) = { a }	FOLLOW ₂ (C) = { ab }

□

The FOLLOW sets of a variable B are obtained from the rules in which B occurs on the right-hand side. Consider the relationships generated by a rule of the form $A \rightarrow uBv$. The strings that follow B include those generated by v concatenated with all terminal strings that follow A . If the grammar contains a rule $A \rightarrow uB$, any string that follows A can also follow B . The preceding discussion is summarized in Lemma 19.2.4.

Lemma 19.2.4

For every $k > 0$,

1. FOLLOW_k(S) contains λ , where S is the start symbol of G
2. if $A \rightarrow uB$ is a rule of G, then FOLLOW_k(A) ⊆ FOLLOW_k(B)
3. if $A \rightarrow uBv$ is a rule of G, then trunc_k(FIRST_k(v)FOLLOW_k(A)) ⊆ FOLLOW_k(B).

□

The FIRST_k and FOLLOW_k sets are used to construct the lookahead sets for the rules of a grammar. Theorem 19.2.5 follows immediately from the definitions of the length- k lookahead sets and the function trunc_k .

Theorem 19.2.5

Let $G = (V, \Sigma, P, S)$ be a context-free grammar. For every $k > 0$, $A \in V$, and rule $A \rightarrow w = u_1 u_2 \dots u_n$ in P ,

- i) $\text{LA}_k(A) = \text{trunc}_k(\text{FIRST}_k(A)\text{FOLLOW}_k(A))$
- ii) $\text{LA}_k(A \rightarrow w) = \text{trunc}_k(\text{FIRST}_k(w)\text{FOLLOW}_k(A))$
 $= \text{trunc}_k(\text{FIRST}_k(u_1) \dots \text{FIRST}_k(u_n)\text{FOLLOW}_k(A)).$

Example 19.2.3

The FIRST_3 and FOLLOW_3 sets for the symbols in the grammar

$$\begin{aligned} G_1: \quad S &\rightarrow Aabd \mid cAbcd \\ A &\rightarrow a \mid b \mid \lambda \end{aligned}$$

from Example 19.1.1 are

$$\begin{aligned} \text{FIRST}_3(S) &= \{aab, bab, abd, cab, cbb, cbc\} \\ \text{FIRST}_3(A) &= \{a, b, \lambda\} \\ \text{FIRST}_3(a) &= \{a\} \\ \text{FIRST}_3(b) &= \{b\} \\ \text{FIRST}_3(c) &= \{c\} \\ \text{FIRST}_3(d) &= \{d\} \\ \text{FOLLOW}_3(S) &= \{\lambda\} \\ \text{FOLLOW}_3(A) &= \{abd, bcd\}. \end{aligned}$$

The set $\text{LA}_3(S \rightarrow Aabd)$ is explicitly constructed from the sets $\text{FIRST}_3(A)$, $\text{FIRST}_3(a)$, $\text{FIRST}_3(b)$, $\text{FIRST}_3(d)$, and $\text{FOLLOW}_3(S)$ using the strategy outlined in Theorem 19.2.5.

$$\begin{aligned} \text{LA}_3(S \rightarrow Aabd) &= \text{trunc}_3(\text{FIRST}_3(A)\text{FIRST}_3(a)\text{FIRST}_3(b)\text{FIRST}_3(d)\text{FOLLOW}_3(S)) \\ &= \text{trunc}_3(\{a, b, \lambda\}\{a\}\{b\}\{d\}\{\lambda\}) \\ &= \text{trunc}_3(\{aab, bab, abd\}) \\ &= \{aab, bab, abd\} \end{aligned}$$

The remainder of the length-three lookahead sets for the rules of G_1 can be found in Example 19.1.3. \square

19.3 Stronger Results

We have seen that when $\text{LA}(A)$ is context-free guarantees that

When empirical be examined. A guarantee that each symbol in the grammar is covered by each S rule. Otherwise $S' \rightarrow S^k$.

Definition 19.3.1

Let $G = (V, \Sigma, P, S)$ be a grammar. We say that G is *strongly LL(k)* if whenever there exist

where $u_j, w_i, z \in \Sigma^*$

We now establish the length- k lookahead sets for a grammar.

Theorem 19.3.2

A grammar G is strongly LL(k) if and only if each variable $A \in V$ is strongly LL(k).

Proof. Assume that G is strongly LL(k). Let $A \in V$ and $z \in \Sigma^*$ be a terminal string such that $z \in \text{LA}_k(A)$. Then z is in both $\text{LA}_k(A)$ and $\text{LA}_k(x)$.

Conversely, let G be a grammar that is strongly LL(k) and let $A \in V$. Then $\text{LA}_k(A)$ contains all strings z such that $z \in \text{LA}_k(x)$ for some $x \in \text{LA}_k(A)$.

he rules
length- k

nd rule

19.3 Strong LL(k) Grammars

We have seen that the lookahead sets can be used to select the A rule in a top-down parse when $\text{LA}(A)$ is partitioned by the sets $\text{LA}(A \rightarrow w_i)$. This section introduces a subclass of context-free grammars known as the strong LL(k) grammars. The strong LL(k) condition guarantees that the lookahead sets $\text{LA}_k(A)$ are partitioned by the sets $\text{LA}_k(A \rightarrow w_i)$.

When employing a k -symbol lookahead, it is often helpful if there are k symbols to be examined. An endmarker $\#^k$ is concatenated to the end of each string in the language to guarantee that every lookahead string contains exactly k symbols. If the start symbol S of the grammar is nonrecursive, the endmarker can be concatenated to the right-hand side of each S rule. Otherwise, the grammar can be augmented with a new start symbol S' and rule $S' \rightarrow S\#\#^k$.

Definition 19.3.1

Let $G = (V, \Sigma, P, S)$ be a context-free grammar with endmarker $\#^k$. G is **strong LL(k)** if whenever there are two leftmost derivations

$$\begin{aligned} S &\xrightarrow{*} u_1Av_1 \Rightarrow u_1xv_1 \xrightarrow{*} u_1zw_1 \\ S &\xrightarrow{*} u_2Av_2 \Rightarrow u_2yv_2 \xrightarrow{*} u_2zw_2, \end{aligned}$$

where $u_i, w_i, z \in \Sigma^*$ and $\text{length}(z) = k$, then $x = y$.

We now establish several properties of strong LL(k) grammars. First, we show that the length- k lookahead sets can be used to parse strings deterministically in a strong LL(k) grammar.

Theorem 19.3.2

A grammar G is strong LL(k) if, and only if, the sets $\text{LA}_k(A \rightarrow w_i)$ partition $\text{LA}_k(A)$ for each variable $A \in V$.

Proof. Assume that the sets $\text{LA}_k(A \rightarrow w_i)$ partition $\text{LA}_k(A)$ for each variable $A \in V$. Let z be a terminal string of length k that can be obtained by the derivations

$$\begin{aligned} S &\xrightarrow{*} u_1Av_1 \Rightarrow u_1xv_1 \xrightarrow{*} u_1zw_1 \\ S &\xrightarrow{*} u_2Av_2 \Rightarrow u_2yv_2 \xrightarrow{*} u_2zw_2, \end{aligned}$$

Then z is in both $\text{LA}_k(A \rightarrow x)$ and $\text{LA}_k(A \rightarrow y)$. Since the sets $\text{LA}_k(A \rightarrow w_i)$ partition $\text{LA}_k(A)$, $x = y$ and G is strong LL(k).

Conversely, let G be a strong LL(k) grammar and let z be an element of $\text{LA}_k(A)$. The strong LL(k) condition ensures that there is only one A rule that can be used to derive terminal strings of the form uzw from the sentential forms uAv of G . Consequently, z is in the lookahead set of exactly one A rule. This implies that the sets $\text{LA}_k(A \rightarrow w_i)$ partition $\text{LA}_k(A)$. ■

RST₃(a),
n 19.2.5.

W₃(S))

found in

□

Theorem 19.3.3

If G is strong LL(k) for some k , then G is unambiguous.

Intuitively, a grammar that can be deterministically parsed must be unambiguous; there is exactly one rule that can be applied at each step in the derivation of a terminal string. The formal proof of this proposition is left as an exercise.

Theorem 19.3.4

If G has a left-recursive variable, then G is not strong LL(k), for any $k > 0$.

Proof. Let A be a left-recursive variable. Since G does not contain useless variables, there is a derivation of a terminal string containing a left-recursive subderivation of the variable A . The proof is presented in two cases.

Case 1: A is directly left-recursive. A derivation containing direct left recursion uses A rules of the form $A \rightarrow Ay$ and $A \rightarrow x$, where the first symbol of x is not A .

$$S \xrightarrow{*} uAv \Rightarrow uAyv \Rightarrow uxv \xrightarrow{*} uw \in \Sigma^*$$

The prefix of w of length k is in both $LA_k(A \rightarrow Ay)$ and $LA_k(A \rightarrow x)$. By Theorem 19.3.2, G is not strong LL(k).

Case 2: A is indirectly left-recursive. A derivation with indirect recursion has the form

$$S \xrightarrow{*} uAv \Rightarrow uB_1yv \Rightarrow \dots \Rightarrow uB_nv_n \Rightarrow uAv_{n+1} \Rightarrow uxv_{n+1} \xrightarrow{*} uw \in \Sigma^*.$$

Again, G is not strong LL(k) since the sets $LA_k(A \rightarrow B_1y)$ and $LA_k(A \rightarrow x)$ are not disjoint. ■

19.4 Construction of FIRST $_k$ Sets

We now present algorithms to construct the length- k lookahead sets for a context-free grammar with endmarker $\#^k$. This is accomplished by generating the FIRST $_k$ and FOLLOW $_k$ sets for the variables of the grammar. The lookahead sets can then be constructed using the technique presented in Theorem 19.2.5.

The initial step in the construction of the lookahead sets begins with the generation of the FIRST $_k$ sets. Consider a rule of the form $A \rightarrow u_1u_2\dots u_n$. The subset of FIRST $_k(A)$ generated by this rule can be constructed from the sets FIRST $_k(u_1)$, FIRST $_k(u_2)$, ..., FIRST $_k(u_n)$, and FOLLOW $_k(A)$. The problem of constructing FIRST $_k$ sets for a string reduces to that of finding the sets for the variables in the string.

Algorithm 19.4.1
Construction of FIRST $_k$

input: context-free grammar G

1. for each $a \in \Sigma$ do
 2. for each $A \in V$ do
 3. repeat
 - 3.1 for each $B \in V$ do
 - 3.2 for each $u \in F(B)$ do
 4. $FIRST_k(A) = \dots$
-

The elements of the repeat-until loop are obtained from the sets $F(A)$ of G , which are then added to the sets $F(A)$ already obtained.

Example 19.4.1

Algorithm 19.4.1

The sets $F'(a)$ are the lookahead sets for a . They are prescribed by the assignment statement $F'(a) = \dots$

$F(A)$
 $F(A)$
 $F(A)$
 $F(A)$
 $F(A)$

Algorithm 19.4.1
Construction of FIRST_k Sets

input: context-free grammar $G = (V, \Sigma, P, S)$

1. for each $a \in \Sigma$ do $F'(a) := \{a\}$
2. for each $A \in V$ do $F(A) := \begin{cases} \{\lambda\} & \text{if } A \rightarrow \lambda \text{ is a rule in } P \\ \emptyset & \text{otherwise} \end{cases}$
3. repeat
 - 3.1 for each $A \in V$ do $F'(A) := F(A)$
 - 3.2 for each rule $A \rightarrow u_1u_2 \dots u_n$ with $n > 0$ do
 $F(A) := F(A) \cup \text{trunc}_k(F'(u_1)F'(u_2) \dots F'(u_n))$
 until $F(A) = F'(A)$ for all $A \in V$
4. $\text{FIRST}_k(A) = F(A)$

The elements of $\text{FIRST}_k(A)$ are generated in step 3.2. At the beginning of each iteration of the repeat-until loop, the auxiliary set $F'(A)$ is assigned the current value of $F(A)$. Strings obtained from the concatenation $F'(u_1)F'(u_2) \dots F'(u_n)$, where $A \rightarrow u_1u_2 \dots u_n$ is a rule of G , are then added to $F(A)$. The algorithm halts when an iteration occurs in which none of the sets $F(A)$ are altered.

Example 19.4.1

Algorithm 19.4.1 is used to construct the FIRST_2 sets for the variables of the grammar

$$\begin{aligned} G: S &\rightarrow A\#\# \\ A &\rightarrow aAd \mid BC \\ B &\rightarrow bBc \mid \lambda \\ C &\rightarrow acC \mid ad. \end{aligned}$$

The sets $F'(a)$ are initialized to $\{a\}$ for each $a \in \Sigma$. The action of the repeat-until loop is prescribed by the right-hand side of the rules of the grammar. Step 3.2 generates the assignment statements

$$\begin{aligned} F(S) &:= F(S) \cup \text{trunc}_2(F'(A)\{\#\}\{\#\}) \\ F(A) &:= F(A) \cup \text{trunc}_2(\{a\}F'(A)\{d\}) \cup \text{trunc}_2(F'(B)F'(C)) \\ F(B) &:= F(B) \cup \text{trunc}_2(\{b\}F'(B)\{c\}) \\ F(C) &:= F(C) \cup \text{trunc}_2(\{a\}\{c\}F'(C)) \cup \text{trunc}_2(\{a\}\{d\}) \end{aligned}$$

from the rules of G . The generation of the FIRST_2 sets is traced by giving the status of the sets $F(S)$, $F(A)$, $F(B)$, and $F(C)$ after each iteration of the loop. Recall that the concatenation of the empty set with any set yields the empty set.

	$F(S)$	$F(A)$	$F(B)$	$F(C)$
0	\emptyset	\emptyset	$\{\lambda\}$	\emptyset
1	\emptyset	\emptyset	$\{\lambda, bc\}$	$\{ad\}$
2	\emptyset	$\{ad, bc\}$	$\{\lambda, bc, bb\}$	$\{ad, ac\}$
3	$\{ab, bc\}$	$\{ad, bc, aa, ab, bb, ac\}$	$\{\lambda, bc, bb\}$	$\{ad, ac\}$
4	$\{ad, bc, aa, ab, bb, ac\}$	$\{ad, bc, aa, ab, bb, ac\}$	$\{\lambda, bc, bb\}$	$\{ad, ac\}$
5	$\{ad, bc, aa, ab, bb, ac\}$	$\{ad, bc, aa, ab, bb, ac\}$	$\{\lambda, bc, bb\}$	$\{ad, ac\}$

□

Theorem 19.4.2

Let $G = (V, \Sigma, P, S)$ be a context-free grammar. Algorithm 19.4.1 generates the sets $\text{FIRST}_k(A)$, for every variable $A \in V$.

Proof. The proof consists of showing that the repeat-until loop in step 3 terminates and, upon termination, $F(A) = \text{FIRST}_k(A)$.

- i) Algorithm 19.4.1 terminates. The number of iterations of the repeat-until loop is bounded since there are only a finite number of lookahead strings of length k or less.
- ii) $F(A) = \text{FIRST}_k(A)$. First we prove that $F(A) \subseteq \text{FIRST}_k(A)$ for all variables $A \in V$. To accomplish this we show that $F(A) \subseteq \text{FIRST}_k(A)$ at the beginning of each iteration of the repeat-until loop. By inspection, this inclusion holds prior to the first iteration. Assume $F(A) \subseteq \text{FIRST}_k(A)$ for all variables A after m iterations of the loop.

During the $m + 1$ st iteration, the only additions to $F(A)$ come from assignment statements of the form

$$F(A) := F(A) \cup \text{trunc}_k(F'(u_1)F'(u_2) \dots F'(u_n)),$$

where $A \rightarrow u_1u_2 \dots u_n$ is a rule of G . By the inductive hypothesis, each of the sets $F'(u_i)$ is the subset of $\text{FIRST}_k(u_i)$. If u is added to $F(A)$ on the iteration then

$$\begin{aligned} u &\in \text{trunc}_k(F'(u_1)F'(u_2) \dots F'(u_n)) \\ &\subseteq \text{trunc}_k(\text{FIRST}_k(u_1)\text{FIRST}_k(u_2) \dots \text{FIRST}_k(u_n)) \\ &= \text{FIRST}_k(u_1u_2 \dots u_n) \\ &\subseteq \text{FIRST}_k(A) \end{aligned}$$

and $u \in \text{FIRST}_k(A)$. The final two steps follow from parts 4 and 5 of Lemma 19.2.2.

We now show that $\text{FIRST}_k(A) \subseteq F(A)$ upon completion of the loop. Let $F_m(A)$ be the value of the set $F(A)$ after m iterations. Assume the repeat-until loop halts after j iterations. We begin with the observation that if a string can be shown to be in $F_m(A)$ for some $m > j$,

then it is in F_j after j iterations of the loop.

Let x be a string such that x is the prefix of a string in $F_m(A)$. Then the length of the derivation of x from A via rule application is at most m .

Assume that $x \in \text{trunc}_k(\{w \mid A \xrightarrow{*} w\})$ for some w . Then x is the prefix of a string in $F_{m+1}(A)$.

where $u_i \in V \cup \Sigma$ for $i = 1, 2, \dots, m + 1$. By the induction hypothesis, $x \in \text{FIRST}_k(u_i)$.

On the $m + 1$ st iteration, x is added to $F(A)$.

Thus,

and x is an element of $F(A)$, which is what was desired.

19.5 Construction of an LL(k) Grammar

The inclusions in the algorithm are guaranteed by the fact that $F(A) \subseteq \text{FIRST}_k(A)$. The sets $\text{FOLLOW}_k(A)$ are determined by the right-hand sides of the rules in P . The set $\text{FL}(A)$ is the union of the sets $\text{FOLLOW}_k(A)$ for all rules $A \rightarrow u$ in P . The set $\text{FL}(A)$ is the union of the sets $\text{FOLLOW}_k(A)$ for all rules $A \rightarrow u$ in P .

Algorithm 19.5.1 Construction of an LL(k) Grammar

input: context-free grammar $G = (V, \Sigma, P, S)$
output: $\text{FIRST}_k(A)$ for all $A \in V$

1. $\text{FL}(S) := \{\lambda\}$
2. for each $A \in V$ do

of the sets
atenation

)
, ac}
, ac}
, ac}
, ac}

□

s the sets

ates and,

bounded

$A \in V$. To
ion of the
. Assume

ent state-

ets $F'(u_i)$

.2.2.
 (A) be the
iterations.
ne $m > j$,

then it is in $F_j(A)$. This follows since the sets $F(A)$ and $F'(A)$ would be identical for all iterations of the loop past iteration j . We will show that $\text{FIRST}_k(A) \subseteq F_j(A)$.

Let x be a string in $\text{FIRST}_k(A)$. Then there is a derivation $A \xrightarrow{m} w$, where $w \in \Sigma^*$ and x is the prefix of w of length k . We show that $x \in F_m(A)$. The proof is by induction on the length of the derivation. The basis consists of terminal strings that can be derived with one rule application. If $A \rightarrow w \in P$, then x is added to $F_1(A)$.

Assume that $\text{trunc}_k(\{w \mid A \xrightarrow{m} w \in \Sigma^*\}) \subseteq F_m(A)$ for all variables A in V . Let $x \in \text{trunc}_k(\{w \mid A \xrightarrow{m+1} w \in \Sigma^*\})$; that is, x is a prefix of terminal string derivable from A by $m+1$ rule applications. We will show that $x \in F_{m+1}(A)$. The derivation of w can be written

$$A \Rightarrow u_1 u_2 \dots u_n \xrightarrow{m} x_1 x_2 \dots x_n = w,$$

where $u_i \in V \cup \Sigma$ and $u_i \xrightarrow{*} x_i$. Clearly, each subderivation $u_i \xrightarrow{*} x_i$ has length less than $m+1$. By the inductive hypothesis, the string obtained by truncating x_i at length k is in $F_m(u_i)$.

On the $m+1$ st iteration, $F_{m+1}(A)$ is augmented with the set

$$\text{trunc}_k(F'_{m+1}(u_1) \dots F'_{m+1}(u_n)) = \text{trunc}_k(F_m(u_1) \dots F_m(u_n)).$$

Thus,

$$\{x\} = \text{trunc}_k(x_1 x_2 \dots x_n) \subseteq \text{trunc}_k(F_m(u_1) \dots F_m(u_n))$$

and x is an element of $F_{m+1}(A)$. It follows that every string in $\text{FIRST}_k(A)$ is in $F_j(A)$, as desired. ■

19.5 Construction of FOLLOW_k Sets

The inclusions in Lemma 19.2.4 form the basis of an algorithm to generate the FOLLOW_k sets. FOLLOW_k(A) is constructed from the FIRST_k sets and the rules in which A occurs on the right-hand side. Algorithm 19.5.1 generates FOLLOW_k(A) using the auxiliary set FL(A). The set FL'(A), which triggers the halting condition, maintains the value assigned to FL(A) on the preceding iteration.

Algorithm 19.5.1 Construction of FOLLOW_k Sets

input: context-free grammar $G = (V, \Sigma, P, S)$

FIRST_k(A) for every $A \in V$

1. $\text{FL}(S) := \{\lambda\}$
2. for each $A \in V - \{S\}$ do $\text{FL}(A) := \emptyset$

3. repeat

- 3.1 **for** each $A \in V$ **do** $FL'(A) := FL(A)$
 - 3.2 **for** each rule $A \rightarrow w = u_1u_2 \dots u_n$ with $w \notin \Sigma^*$ **do**
 - 3.2.1. $L := FL'(A)$
 - 3.2.2. **if** $u_n \in V$ **then** $FL(u_n) := FL(u_n) \cup L$
 - 3.2.3. **for** $i := n - 1$ **to** 1 **do**
 - 3.2.3.1. $L := trunc_k(FIRST_k(u_{i+1})L)$
 - 3.2.3.2. **if** $u_i \in V$ **then** $FL(u_i) := FL(u_i) \cup L$
- end for**
- end for**
- until** $FL(A) = FL'(A)$ **for every** $A \in V$

4. FOLLOW _{k} (A) := $FL(A)$

The inclusion $FL(A) \subseteq FOLLOW_k(A)$ is established by showing that every element added to $FL(A)$ in statements 3.2.2 or 3.2.3.2 is in $FOLLOW_k(A)$. The opposite inclusion is obtained by demonstrating that every element of $FOLLOW_k(A)$ is added to $FL(A)$ prior to the termination of the repeat-until loop. The details are left as an exercise.

Example 19.5.1

Algorithm 19.5.1 is used to construct the set $FOLLOW_2$ for every variable of the grammar G from Example 19.4.1. The interior of the repeat-until loop processes each rule in a right-to-left fashion. The action of the loop is specified by the assignment statements obtained from the rules of the grammar.

Rule	Assignments
$S \rightarrow A\#\#$	$FL(A) := FL(A) \cup trunc_2(\{\#\#\}FL'(S))$
$A \rightarrow aAd$	$FL(A) := FL(A) \cup trunc_2(\{d\}FL'(A))$
$A \rightarrow BC$	$FL(C) := FL(C) \cup FL'(A)$
	$FL(B) := FL(B) \cup trunc_2(FIRST_2(C)FL'(A))$
	$= FL(B) \cup trunc_2(\{ad, ac\}FL'(A))$
$B \rightarrow bBc$	$FL(B) := FL(B) \cup trunc_2(\{c\}FL'(B))$

The rule $C \rightarrow acC$ has been omitted from the list since the assignment generated by this rule is $FL(C) := FL(C) \cup FL'(C)$. Tracing Algorithm 19.5.1 yields

	FL(S)	FL(A)	FL(B)	FL(C)
0	$\{\lambda\}$	\emptyset	\emptyset	\emptyset
1	$\{\lambda\}$	$\{\#\#\}$	\emptyset	\emptyset
2	$\{\lambda\}$	$\{\#\#, d\#}$	$\{ad, ac\}$	$\{\#\#}$
3	$\{\lambda\}$	$\{\#\#, d\#, dd\}$	$\{ad, ac, ca\}$	$\{\#\#, d\#\}$

Example 19.5.2

The length-two look-ahead sets $FIRST_2$ and $FOLLOW_2$

4

5

G is strong LL(2) s

The preceding
mar is strong LL(k)
using Algorithms 1
used to construct th
LL(k) if, and only
distinct A rules.

19.6 A Strong

The grammar AE
containing a single
recursive A rule. It
generates the addition
the length-one look-

The transforma
This ensures that a

	$\text{FL}(S)$	$\text{FL}(A)$	$\text{FL}(B)$	$\text{FL}(C)$
4	{ λ }	{ $\#\#, d\#, dd$ }	{ ad, ac, ca, cc }	{ $\#\#, d\#, dd$ }
5	{ λ }	{ $\#\#, d\#, dd$ }	{ ad, ac, ca, cc }	{ $\#\#, d\#, dd$ }

□

Example 19.5.2

The length-two lookahead sets for the rules of the grammar G are constructed from the FIRST_2 and FOLLOW_2 sets generated in Examples 19.4.1 and 19.5.1.

$$\text{LA}_2(S \rightarrow A\#\#) = \{ad, bc, aa, ab, bb, ac\}$$

$$\text{LA}_2(A \rightarrow aAd) = \{aa, ab\}$$

$$\text{LA}_2(A \rightarrow BC) = \{bc, bb, ad, ac\}$$

$$\text{LA}_2(B \rightarrow bBc) = \{bb, bc\}$$

$$\text{LA}_2(B \rightarrow \lambda) = \{ad, ac, ca, cc\}$$

$$\text{LA}_2(C \rightarrow acC) = \{ac\}$$

$$\text{LA}_2(C \rightarrow ad) = \{ad\}$$

G is strong LL(2) since the lookahead sets are disjoint for each pair of alternative rules. □

The preceding algorithms provide a decision procedure to determine whether a grammar is strong LL(k). The process begins by generating the FIRST_k and FOLLOW_k sets using Algorithms 19.4.1 and 19.5.1. The techniques presented in Theorem 19.2.5 are then used to construct the length- k lookahead sets. By Theorem 19.3.2, the grammar is strong LL(k) if, and only if, the sets $\text{LA}_k(A \rightarrow x)$ and $\text{LA}_k(A \rightarrow y)$ are disjoint for each pair of distinct A rules.

19.6 A Strong LL(1) Grammar

The grammar AE was introduced in Section 18.1 to generate infix additive expressions containing a single variable b . AE is not strong LL(k) since it contains a directly left-recursive A rule. In this section we modify AE to obtain a strong LL(1) grammar that generates the additive expressions. To guarantee that the resulting grammar is strong LL(1), the length-one lookahead sets are constructed for each rule.

The transformation begins by adding the endmarker # to the strings generated by AE. This ensures that a lookahead set does not contain the null string. The grammar

$$\begin{aligned} \text{AE: } S &\rightarrow A\# \\ A &\rightarrow T \\ A &\rightarrow A + T \\ T &\rightarrow b \\ T &\rightarrow (A) \end{aligned}$$

generates the strings in $L(AE)$ concatenated with the endmarker $\#$. The direct left recursion can be removed using the techniques presented in Section 4.5. The variable Z is used to convert the left recursion to right recursion, yielding the equivalent grammar AE_1 .

$$\begin{aligned} AE_1: \quad S &\rightarrow A\# \\ &A \rightarrow T \\ &A \rightarrow TZ \\ &Z \rightarrow +T \\ &Z \rightarrow +TZ \\ &T \rightarrow b \\ &T \rightarrow (A) \end{aligned}$$

AE_1 still cannot be strong LL(1) since both A rules have T as the first symbol occurring on the right-hand side. This difficulty is removed by left factoring the A rules using the new variable B . Similarly, the right-hand side of the Z rules begin with identical substrings. The variable Y is introduced by the factoring of the Z rules. AE_2 results from making these modifications to AE_1 .

$$\begin{aligned} AE_2: \quad S &\rightarrow A\# \\ &A \rightarrow TB \\ &B \rightarrow Z \\ &B \rightarrow \lambda \\ &Z \rightarrow +TY \\ &Y \rightarrow Z \\ &Y \rightarrow \lambda \\ &T \rightarrow b \\ &T \rightarrow (A) \end{aligned}$$

To show that AE_2 is strong LL(1), the length-one lookahead sets for the variables of the grammar must satisfy the partition condition of Theorem 19.3.2. We begin by tracing the sequence of sets generated by Algorithm 19.4.1 in the construction of the $FIRST_1$ sets.

	$F(S)$	$F(A)$	$F(B)$	$F(Z)$	$F(Y)$	$F(T)$
0	\emptyset	\emptyset	$\{\lambda\}$	\emptyset	$\{\lambda\}$	\emptyset
1	\emptyset	\emptyset	$\{\lambda\}$	$\{+\}$	$\{\lambda\}$	$\{b, ()\}$
2	\emptyset	$\{b, ()\}$	$\{\lambda, +\}$	$\{+\}$	$\{\lambda, +\}$	$\{b, ()\}$
3	$\{b, ()\}$	$\{b, ()\}$	$\{\lambda, +\}$	$\{+\}$	$\{\lambda, +\}$	$\{b, ()\}$
4	$\{b, ()\}$	$\{b, ()\}$	$\{\lambda, +\}$	$\{+\}$	$\{\lambda, +\}$	$\{b, ()\}$

Similarly, the F

The length-

Since the lookah

19.7 A Strong LL(1) Grammar

Parsing with a strong LL(1) grammar is similar to the LR(0) parser. For each of the nonterminals, we construct a lookahead set for the parsing of the string. The lookahead sets are constructed using the techniques presented in Algorithm 19.4.1. The lookahead sets for the grammar AE_2 are shown in the following table.

Unlike the LR(0) parser, the lookahead sets for the grammar AE_2 are not partitioned. Instead, they are combined into a single set for each nonterminal. This is because the grammar AE_2 is strong LL(1). The lookahead sets for the grammar AE_2 are shown in the following table.

recursion
is used to

Similarly, the FOLLOW₂ sets are generated using Algorithm 19.5.1.

1.

	FL(S)	FL(A)	FL(B)	FL(Z)	FL(Y)	FL(T)
0	{λ}	∅	∅	∅	∅	∅
1	{λ}	{#, ()}	∅	∅	∅	∅
2	{λ}	{#, ()}	{#, ()}	∅	∅	∅
3	{λ}	{#, ()}	{#, ()}	{#, ()}	∅	∅
4	{λ}	{#, ()}	{#, ()}	{#, ()}	{#, ()}	∅
5	{λ}	{#, ()}	{#, ()}	{#, ()}	{#, ()}	{#, ()}
6	{λ}	{#, ()}	{#, ()}	{#, ()}	{#, ()}	{#, ()}

occurring
ing the new
rings. The
king these

The length-one lookahead sets are obtained from the FIRST₁ and FOLLOW₁ sets.

$$\text{LA}_1(S \rightarrow A\#) = \{b, ()\}$$

$$\text{LA}_1(A \rightarrow TB) = \{b, ()\}$$

$$\text{LA}_1(B \rightarrow Z) = \{+\}$$

$$\text{LA}_1(B \rightarrow \lambda) = \{\#\}$$

$$\text{LA}_1(Z \rightarrow + TY) = \{+\}$$

$$\text{LA}_1(Y \rightarrow Z) = \{+\}$$

$$\text{LA}_1(Y \rightarrow \lambda) = \{\#\}$$

$$\text{LA}_1(T \rightarrow b) = \{b\}$$

$$\text{LA}_1(T \rightarrow (A)) = \{\}\}$$

Since the lookahead sets for alternative rules are disjoint, the grammar AE₂ is strong LL(1).

bles of the
tracing the
T₁ sets.

19.7 A Strong LL(k) Parser

Parsing with a strong LL(k) grammar begins with the construction of the lookahead sets for each of the rules of the grammar. Once these sets have been built, they are available for the parsing of any number of strings. The strategy for parsing strong LL(k) grammars presented in Algorithm 19.7.1 consists of a loop that compares the lookahead string with the lookahead sets and applies the appropriate rule.

Unlike the examination of multiple rules in the top-down parser given in Algorithm 18.2.1, node expansion using a strong LL(k) grammar is limited to the application of at most one rule. The lookahead string and lookahead sets provide sufficient information to eliminate other rules from consideration.

Algorithm 19.7.1**Deterministic Parser for a Strong LL(k) Grammar**

input: strong LL(k) grammar $G = (V, \Sigma, P, S)$
 string $p \in \Sigma^*$
 lookahead sets $LA_k(A \rightarrow w)$ for each rule in P

1. $q := S$ (q is the sentential form to be expanded)
2. **repeat**
 - Let $q = uAv$ where A is the leftmost variable in q and
 let $p = uyz$ where $\text{length}(y) = k$.
 - 2.1. if $y \in LA_k(A \rightarrow w)$ for some A rule then $q := uwv$
 - until $q = p$ or $y \notin LA_k(A \rightarrow w)$ for all A rules
3. if $q = p$ then accept else reject

The presence of the endmarker in the grammar ensures that the lookahead string y contains k symbols. The input string is rejected whenever the lookahead string is not an element of one of the lookahead sets. When the lookahead string is in $LA_k(A \rightarrow w)$, a new sentential form is constructed by applying $A \rightarrow w$ to the current string uAv . The input is accepted if this rule application generates the input string. Otherwise, the loop is repeated for the sentential form uwv .

Example 19.7.1

Algorithm 19.7.1 and the lookahead sets of the strong LL(1) grammar AE_2 from Section 19.6 are used to parse the string $(b + b)\#$. Each row in the table that follows represents one iteration of step 2 of Algorithm 19.7.1.

u	A	v	Lookahead	Rule	Derivation
λ	S	λ	($S \rightarrow A\#$	$S \Rightarrow A\#$
λ	A	#	($A \rightarrow TB$	$\Rightarrow TB\#$
λ	T	$B\#$	($T \rightarrow (A)$	$\Rightarrow (A)B\#$
(A	$)B\#$	b	$A \rightarrow TB$	$\Rightarrow (TB)B\#$
(T	$B)B\#$	b	$T \rightarrow b$	$\Rightarrow (bB)B\#$
$(b$	B	$)B\#$	+	$B \rightarrow Z$	$\Rightarrow (bZ)B\#$
$(b$	Z	$)B\#$	+	$Z \rightarrow + TY$	$\Rightarrow (b+TY)B\#$
$(b+$	T	$Y)B\#$	b	$T \rightarrow b$	$\Rightarrow (b+bY)B\#$
$(b+b$	Y	$)B\#$)	$Y \rightarrow \lambda$	$\Rightarrow (b+b)B\#$
$(b+b)$	B	#	#	$B \rightarrow \lambda$	$\Rightarrow (b+b)\#$

19.8 LL(k)

The lookahead set of a rule. When A is the leftmost variable in q , the lookahead string provides information to select the rule containing A that the rule depends upon.

Definition 19.8.1

Let $G = (V, \Sigma, P, S)$, whenever there are

where $u, w_i, z \in \Sigma^*$

Notice the difference between a strong LL(k) condition and a string z from any set of strings. It must be unique for a fixed u and w and be defined for each u and w .

Definition 19.8.2

Let $G = (V, \Sigma, P, S)$, the lookahead set of G .

- The lookahead set of a rule A .
- The lookahead set of a sentential form uAv .

A result similar to Theorem 19.6 can be obtained by showing that the lookahead sets of a strong LL(k), then

Example 19.8.1

An LL(k) grammar

19.8 LL(k) Grammars

The lookahead sets in a strong LL(k) grammar provide a global criterion for selecting a rule. When A is the leftmost variable in the sentential form being extended by the parser, the lookahead string generated by the parser and the lookahead sets provide sufficient information to select the appropriate A rule. This choice does not depend upon the sentential form containing A . The LL(k) grammars provide a local selection criterion; the choice of the rule depends upon both the lookahead and the sentential form.

Definition 19.8.1

Let $G = (V, \Sigma, P, S)$ be a context-free grammar with endmarker $\#^k$. G is LL(k) if whenever there are two leftmost derivations

$$S \xrightarrow{*} uAv \Rightarrow uxv \xrightarrow{*} uzw_1$$

$$S \xrightarrow{*} uAv \Rightarrow uyw \xrightarrow{*} uzw_2,$$

where $u, w_i, z \in \Sigma^*$ and $\text{length}(z) = k$, then $x = y$.

d string y
is not an
w), a new
ne input is
s repeated

from Sec-
represents

Notice the difference between the derivations in Definitions 19.3.1 and 19.8.1. The strong LL(k) condition requires that there be a unique A rule that can derive the lookahead string z from any sentential form containing A . An LL(k) grammar only requires the rule to be unique for a fixed sentential form uAv . The lookahead sets for an LL(k) grammar must be defined for each sentential form.

Definition 19.8.2

Let $G = (V, \Sigma, P, S)$ be a context-free grammar with endmarker $\#^k$ and uAv a sentential form of G .

- i) The lookahead set of the sentential form uAv is defined by $\text{LA}_k(uAv) = \text{FIRST}_k(Av)$.
- ii) The lookahead set for the sentential form uAv and rule $A \rightarrow w$ is defined by $\text{LA}_k(uAv, A \rightarrow w) = \text{FIRST}_k(wv)$.

A result similar to Theorem 19.3.2 can be established for LL(k) grammars. The unique selection of a rule for the sentential form uAv requires the set $\text{LA}_k(uAv)$ to be partitioned by the lookahead sets $\text{LA}_k(uAv, A \rightarrow w_i)$ generated by the A rules. If the grammar is strong LL(k), then the partition is guaranteed and the grammar is also LL(k).

Example 19.8.1

An LL(k) grammar need not be strong LL(k). Consider the grammar

$$G_1: S \rightarrow Aabd \mid cAbcd$$

$$A \rightarrow a \mid b \mid \lambda$$

B#
B#

□

whose lookahead sets were given in Example 19.1.1. G_1 is strong LL(3) but not strong LL(2) since the string ab is in both $\text{LA}_2(A \rightarrow a)$ and $\text{LA}_2(A \rightarrow \lambda)$. The length-two lookahead sets for the sentential forms containing the variables S and A are

$$\begin{array}{ll} \text{LA}_2(S, S \rightarrow Aabd) = \{aa, ba, ab\} & \\ \text{LA}_2(S, S \rightarrow cAbcd) = \{ca, cb\} & \\ \text{LA}_2(Aabd, A \rightarrow a) = \{aa\} & \text{LA}_2(cAbcd, A \rightarrow a) = \{ab\} \\ \text{LA}_2(Aabd, A \rightarrow b) = \{ba\} & \text{LA}_2(cAbcd, A \rightarrow b) = \{bb\} \\ \text{LA}_2(Aabd, A \rightarrow \lambda) = \{ab\} & \text{LA}_2(cAbcd, A \rightarrow \lambda) = \{bc\}. \end{array}$$

Since the alternatives for a given sentential form are disjoint, the grammar is LL(2). \square

Example 19.8.2

A three-symbol lookahead is sufficient for a local selection of rules in the grammar

$$\begin{aligned} G: S &\rightarrow aBAd \mid bBbAd \\ A &\rightarrow abA \mid c \\ B &\rightarrow ab \mid a. \end{aligned}$$

The S and A rules can be selected with a one-symbol lookahead; so we turn our attention to selecting the B rule. The lookahead sets for the B rules are

$$\begin{array}{l} \text{LA}_3(aBAd, B \rightarrow ab) = \{aba, abc\} \\ \text{LA}_3(aBAd, B \rightarrow a) = \{aab, acd\} \\ \text{LA}_3(bBbAd, B \rightarrow ab) = \{abb\} \\ \text{LA}_3(bBbAd, B \rightarrow a) = \{aba, abc\}. \end{array}$$

The length-three lookahead sets for the two sentential forms that contain B are partitioned by the B rules. Consequently, G is LL(3). The strong LL(k) conditions can be checked by examining the lookahead sets for the B rules.

$$\begin{aligned} \text{LA}(B \rightarrow ab) &= ab(ab)^*cd \cup abb(ab)^*cd \\ \text{LA}(B \rightarrow a) &= a(ab)^*cd \cup ab(ab)^*cd \end{aligned}$$

For any integer k , there is a string of length greater than k in both $\text{LA}(B \rightarrow ab)$ and $\text{LA}(B \rightarrow a)$. Consequently, G is not strong LL(k) for any k . \square

Parsing deterministically with LL(k) grammars requires the construction of the local lookahead sets for the sentential forms generated during the parse. The lookahead set for a sentential form can be constructed directly from the FIRST_k sets of the variables and

terminals of the grammar and $v = v_1 \dots v_n$. \square

A parsing algorithm for the construction of the lookahead sets

Algorithm 19.8.3

Deterministic Parsing

input: LL(k) grammar G , string $p \in L(G)$, $k \geq 1$
 $\text{FIRST}_k(A)$ for all terminals A

1. $q := S$
2. repeat
 - Let $q = uA$
 - let $p = uy_2y_3\dots y_ky_{k+1}\dots y_n$
 - 2.1. for each terminal A in $\text{FIRST}_k(A)$
 - 2.2. if $y_i \in \text{LA}(A)$ then $q := qy_i$
3. if $q = p$ then

The family of lookahead sets provides a local selection of the appropriate rule. The grammar is strong LL(k) if there is no prefix u already generated by the grammar. The LL(k) parser is a deterministic parser.

Exercises

1. Let G be a context-free grammar.
2. Give the lookahead sets for the following grammar:
 - a) $S \rightarrow Aa \mid Bb$
 $A \rightarrow a \mid ab$
 $B \rightarrow b \mid bb$
 - c) $S \rightarrow AA \mid BB$
 $A \rightarrow a \mid ab$
 $B \rightarrow b \mid bb$

strong LL(2)
cahead sets

terminals of the grammar. The lookahead set $\text{LA}_k(uAv, A \rightarrow w)$, where $w = w_1 \dots w_n$ and $v = v_1 \dots v_m$, is given by

$\text{trunc}_k(\text{FIRST}_k(w_1) \dots \text{FIRST}_k(w_n) \text{FIRST}_k(v_1) \dots \text{FIRST}_k(v_m))$.

A parsing algorithm for LL(k) grammars can be obtained from Algorithm 19.7.1 by adding the construction of the local lookahead sets.

Algorithm 19.8.3

Deterministic Parser for an LL(k) Grammar

input: LL(k) grammar $G = (V, \Sigma, P, S)$

string $n \in \Sigma^*$

FIRST: (A) for every $A \in V$

- $q := S$
 - repeat**
 - Let $q = uAv$ where A is the leftmost variable in q and let $p = uyz$ where $\text{length}(y) = k$.
 - 2.1. for each rule $A \rightarrow w$ construct the set $\text{LA}_k(uAv, A \rightarrow w)$
 - 2.2. if $y \in \text{LA}_k(uAv, A \rightarrow w)$ for some A rule then $q := uwv$ until $q = p$ or $y \notin \text{LA}_k(uAv, A \rightarrow w)$ for all A rules
 - 3. if $q = p$ then accept else reject

The family of strong LL(k) grammars is a proper subset of the LL(k). The local lookahead sets permit more contextual information to be used in the selection of the appropriate rule. In the determination of the rule to apply to a sentential form uAv , a strong LL(k) grammar considers the variable A and the lookahead string. The terminal prefix u already generated by the parser may also be used in rule selection in an LL(k) grammar. The LL(k) grammars do not generate every context-free language that can be parsed deterministically. Exercise 14 gives an example of a language that can be parsed by a deterministic pushdown automaton, but is not generated by any LL(k) grammar.

Exercises

1. Let G be a context-free grammar with start symbol S . Prove that $\text{LA}(S) = L(G)$.
 2. Give the lookahead sets for each variable and rule of the following grammars.

- a) $S \rightarrow ABab \mid bAcc$
 $A \rightarrow a \mid c$
 $B \rightarrow b \mid c \mid \lambda$

b) $S \rightarrow aS \mid A$
 $A \rightarrow ab \mid b$

c) $S \rightarrow AB \mid ab$
 $A \rightarrow aa \mid \lambda$
 $B \rightarrow bB \mid \lambda$

d) $S \rightarrow aAbBc$
 $A \rightarrow aA \mid cA \mid \lambda$
 $B \rightarrow bBc \mid bc$

3. Give the FIRST₁ and FOLLOW₁ sets for each of the variables of the following grammars. Which of these grammars are strong LL(1)?

a) $S \rightarrow aAB\#$ $A \rightarrow a \mid \lambda$ $B \rightarrow b \mid \lambda$	b) $S \rightarrow AB\#$ $A \rightarrow aAb \mid B$ $B \rightarrow aBc \mid \lambda$
c) $S \rightarrow ABC\#$ $A \rightarrow aA \mid \lambda$ $B \rightarrow bBc \mid \lambda$ $C \rightarrow cA \mid dB \mid \lambda$	d) $S \rightarrow aAd\#$ $A \rightarrow BCD$ $B \rightarrow bB \mid \lambda$ $C \rightarrow cC \mid \lambda$ $D \rightarrow bD \mid \lambda$

4. Give strong LL(1) grammars that generate each of the following languages.

- a) $\{a^i b^j c^i \mid i \geq 0, j \geq 0\}$
b) $\{a^i b^j c \mid i \geq 1, j \geq 0\}$

5. Show that the grammar

$$S \rightarrow aSa \mid bSb \mid \lambda$$

is strong LL(2) but not strong LL(1).

6. Use Algorithms 19.4.1 and 19.5.1 to construct the FIRST₂ and FOLLOW₂ sets for variables of the following grammars. Construct the length-two lookahead sets for the rules of the grammars. Are these grammars strong LL(2)?

a) $S \rightarrow ABC\#\#$ $A \rightarrow aA \mid a$ $B \rightarrow bB \mid \lambda$ $C \rightarrow cC \mid a \mid b \mid c$	b) $S \rightarrow A\#\#$ $A \rightarrow bBA \mid BcAa \mid \lambda$ $B \rightarrow acB \mid b$
---	--

7. Prove parts 3, 4, and 5 of Lemma 19.2.2.

- *8. Prove Theorem 19.3.3.

9. Show that each of the grammars defined below is not strong LL(k) for any k . Construct a deterministic PDA that accepts the language generated by the grammar.

a) $S \rightarrow aSb \mid A$ $A \rightarrow aAc \mid \lambda$	b) $S \rightarrow A \mid B$ $A \rightarrow aAb \mid ab$ $B \rightarrow aBc \mid ac$
c) $S \rightarrow A$ $A \rightarrow aAb \mid B$ $B \rightarrow aB \mid a$	

10. Prove that Algorithm 19.5.1 generates the sets FOLLOW _{k} (A).

11. Modify the grammars given below to obtain an equivalent strong LL(1) grammar. Build the lookahead sets to ensure that the modified grammar is strong LL(1).

a) $S \rightarrow A\#$ $A \rightarrow aB \mid Ab \mid Ac$ $B \rightarrow bBc \mid \lambda$	b) $S \rightarrow aA\# \mid abB\# \mid abcC\#$ $A \rightarrow aA \mid \lambda$ $B \rightarrow bB \mid \lambda$ $C \rightarrow cC \mid \lambda$
--	---

12. Parse the following grammar. The actions of the nonterminals are given in parentheses.
a) $b + (b)\#$
b) $((b))\#$
c) $b + b + b$
d) $b + +b\#$

13. Construct the grammar that generates the language of each sentential form.

a) $S \rightarrow aA \mid aB$ $A \rightarrow a$
b) $S \rightarrow aA \mid aB$ $A \rightarrow a$
c) $S \rightarrow aA \mid aB$ $A \rightarrow a$

- *14. Prove that the

Design a determinist

- *15. Prove that a grammar

16. Prove that a grammar $L_{k+1}(uAv)$ is equivalent to $L_k(uAv)$.

Bibliographic Notes

Parsing with LL(k) grammars is a well-known topic. The theory of LL(k) grammars was developed by Robert Stearns [1970]. Robert Stearns [1971] gives a survey of languages that can be parsed with LL(k) grammars. The LL(k) hierarchy is discussed by Robert Stearns [1971], and the construction of modifying grammars is discussed by Robert Stearns [1971].

The construction of LR(k) grammars employs the method of LR(k) item sets. This code to accompany the book is available at www.cs.vassar.edu/~alvarez/compilers.html. The book is available at www.cs.vassar.edu/~alvarez/compilers.html.

12. Parse the following strings with the LL(1) parser and the grammar AE_2 . Trace the actions of the parser using the format of Example 19.7.1. The lookahead sets for AE_2 are given in Section 19.6.

- a) $b + (b)\#$
- b) $((b))\#$
- c) $b + b + b\#$
- d) $b + +b\#$

13. Construct the lookahead sets for the rules of the grammar. What is the minimal k such that the grammar is strong LL(k)? Construct the lookahead sets for the combination of each sentential form and rule. What is the minimal k such that the grammar is LL(k)?

- a) $S \rightarrow aAcaa \mid bAbcc$
 $A \rightarrow a \mid ab \mid \lambda$
- b) $S \rightarrow aAbc \mid bABbd$
 $A \rightarrow a \mid \lambda$
 $B \rightarrow a \mid b$
- c) $S \rightarrow aAbB \mid bAbA$
 $A \rightarrow ab \mid a$
 $B \rightarrow aB \mid b$

- * 14. Prove that there is no LL(k) grammar that generates the language

$$L = \{a^i \mid i \geq 0\} \cup \{a^i b^i \mid i \geq 0\}.$$

Design a deterministic pushdown automaton that accepts L .

- * 15. Prove that a grammar is strong LL(1) if, and only if, it is LL(1).
- 16. Prove that a context-free grammar G is LL(k) if, and only if, the lookahead set $LA_k(uAv)$ is partitioned by the sets $LA_k(uAv, A \rightarrow w_i)$ for each left sentential form uAv .

Construct

Bibliographic Notes

Parsing with LL(k) grammars was introduced by Lewis and Stearns [1968]. The theory of LL(k) grammars and deterministic parsing was further developed in Rosenkrantz and Stearns [1970]. Relationships between the class of LL(k) languages and other classes of languages that can be parsed deterministically are examined in Aho and Ullman [1973]. The LL(k) hierarchy was presented in Kurki-Suonio [1969]. Foster [1968], Wood [1969], Stearns [1971], and Soisalon-Soininen and Ukkonen [1979] introduced techniques for modifying grammars to satisfy the LL(k) or strong LL(k) conditions.

The construction of compilers for languages defined by LL(1) grammars frequently employs the method of recursive descent. This approach allows the generation of machine code to accompany the syntax analysis. A comprehensive introduction to syntax analysis and compiling can be found in Aho, Sethi, and Ullman [1986].

CHAPTER 20

LR(k) Grammars

A bottom-up parser generates a sequence of shifts and reductions to reduce the input string to the start symbol of the grammar. A deterministic parser must incorporate additional information into the process to select the correct alternative when more than one operation is possible. A grammar is LR(k) if a k -symbol lookahead provides sufficient information to make this selection. LR signifies that these strings are parsed in a left-to-right manner to construct a rightmost derivation. The LR(k) grammars are theoretically significant because every context-free language that can be parsed deterministically reading the input string in a left-to-right manner is generated by an LR(k) grammar. The practical significance is that the LR approach provides the foundation for bottom-up parser generators, programs used to automatically generate a parser directly from the rules of the grammar.

All derivations in this chapter are rightmost. We also assume that grammars have a nonrecursive start symbol and that all the symbols in a grammar are useful.

20.1 LR(0) Contexts

A deterministic bottom-up parser attempts to reduce the input string to the start symbol of the grammar. Nondeterminism in bottom-up parsing is illustrated by examining reductions of the string $aabb$ using the grammar

$$\begin{aligned} G: S &\rightarrow aAb \mid BaAa \\ A &\rightarrow ab \mid b \\ B &\rightarrow Bb \mid b. \end{aligned}$$

The parser scans the prefix aab before finding a reducible substring. The suffixes b and ab of aab both constitute the right-hand side of a rule of G . Three reductions of $aabb$ can be obtained by replacing these substrings.

Rule	Reduction
$A \rightarrow b$	$aaAb$
$A \rightarrow ab$	aAb
$B \rightarrow b$	$aaBb$

The objective of a bottom-up parser is to repeatedly reduce the input string until the start symbol is obtained. Can a reduction of $aabb$ initiated with the rule $A \rightarrow b$ eventually produce the start symbol? Equivalently, is $aaAb$ a right sentential form of G ? Rightmost derivations of the grammar G have the form

$$\begin{aligned} S &\Rightarrow aAb \Rightarrow aabb \\ S &\Rightarrow aAb \Rightarrow abb \\ S &\Rightarrow BaAa \Rightarrow Baaba \xrightarrow{i} Bb^i aaba \Rightarrow bb^i aaba \quad i \geq 0 \\ S &\Rightarrow BaAa \Rightarrow Baba \xrightarrow{i} Bb^i aba \Rightarrow bb^i aba \quad i \geq 0. \end{aligned}$$

Successful reductions of strings in $L(G)$ can be obtained by “reversing the arrows” in the preceding derivations. Since the strings $aaAb$ and $aaBb$ do not occur in any of these derivations, a reduction of $aabb$ initiated by the rule $A \rightarrow b$ or $B \rightarrow b$ cannot produce S . With this additional information, the parser need only reduce aab using the rule $A \rightarrow ab$.

Successful reductions were obtained by examining rightmost derivations of G . A parser that does not use lookahead must decide whether to perform a reduction with a rule $A \rightarrow w$ as soon as a string uw is scanned by the parser. We now introduce the set of LR(0) contexts of a rule $A \rightarrow w$, which defines the contexts in which a reduction should be performed when w is read by the scanner.

Definition 20.1.1

Let $G = (V, \Sigma, P, S)$ be a context-free grammar. The string uw is an LR(0) context of a rule $A \rightarrow w$ if there is a derivation

$$S \xrightarrow[R]{*} uAv \xrightarrow[R]{} uwv,$$

where $u \in (V \cup \Sigma)^*$ and $v \in \Sigma^*$. The set of LR(0) contexts of the rule $A \rightarrow w$ is denoted $\text{LR}(0)\text{-CONTEXT}(A \rightarrow w)$.

The LR(0) contexts of a rule $A \rightarrow w$ are obtained from the rightmost derivations that terminate with the application of the rule. In terms of reductions, uw is an LR(0) context of $A \rightarrow w$ if there is a reduction of a string uwv to S that begins by replacing w with A . If $uw \notin \text{LR}(0)\text{-CONTEXT}(A \rightarrow w)$ then there is no sequence of reductions beginning with

$A \rightarrow w$ that produces uw . If known, can be reduced to w .

The LR(0) contexts of $A \rightarrow w$ are the strings uw such that uwv is a rightmost derivation of S . To determine the LR(0) contexts of $A \rightarrow w$, we must find all strings uw such that uwv is a rightmost derivation of S .

The only rightmost derivations of S begin with A . Thus $\text{LR}(0)\text{-CONTEXT}(A \rightarrow w)$ contains all strings uw such that uwv is a rightmost derivation of S .

The LR(0) contexts of $A \rightarrow w$ are the strings uw such that uwv is a rightmost derivation of S . To determine the LR(0) contexts of $A \rightarrow w$, we must find all strings uw such that uwv is a rightmost derivation of S .

Consequently, the LR(0) contexts of $A \rightarrow w$ are the strings uw such that uwv is a rightmost derivation of S .

Example 20.1.1

The LR(0) contexts of $A \rightarrow ab$ are the strings uw such that uwv is a rightmost derivation of S .

tes b and ab
 $aabb$ can be

ing until the
 b eventually
? Rightmost

“arrows” in
any of these
t produce S .
le $A \rightarrow ab$.
G. A parser
rule $A \rightarrow w$
(0) contexts
formed when

context of a

v is denoted
ivations that
R(0) context
 w with A . If
ginning with

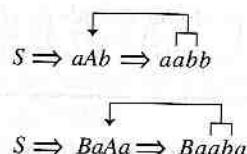
$A \rightarrow w$ that produces S from a string of the form uvw with $v \in \Sigma^*$. The LR(0) contexts, if known, can be used to eliminate reductions from consideration by the parser. The parser need only reduce a string uw with the rule $A \rightarrow w$ when uw is an LR(0) context of $A \rightarrow w$.

The LR(0) contexts of the rules of G are constructed from the rightmost derivations of G . To determine the LR(0) contexts of $S \rightarrow aAb$, we consider all rightmost derivations that contain an application of the rule $S \rightarrow aAb$. The only two such derivations are

$$\begin{aligned} S &\Rightarrow aAb \Rightarrow aabb \\ S &\Rightarrow aAb \Rightarrow abb. \end{aligned}$$

The only rightmost derivation terminating with the application of $S \rightarrow aAb$ is $S \Rightarrow aAb$. Thus $\text{LR}(0)\text{-CONTEXT}(S \rightarrow aAb) = \{aAb\}$.

The LR(0) contexts of $A \rightarrow ab$ are obtained from the rightmost derivations that terminate with an application of $A \rightarrow ab$. There are only two such derivations. The reduction is indicated by the arrow from ab to A . The context is the prefix of the sentential form up to and including the occurrence of ab that is reduced.



Consequently, the LR(0) contexts of $A \rightarrow ab$ are aab and $Baab$. In a similar manner we can obtain the LR(0) contexts for all the rules of G .

Rule	LR(0) Contexts
$S \Rightarrow aAb$	$\{aAb\}$
$S \Rightarrow BaAa$	$\{BaAa\}$
$A \rightarrow ab$	$\{aab, Bab\}$
$A \rightarrow b$	$\{ab, Bab\}$
$B \rightarrow Bb$	$\{Bb\}$
$B \rightarrow b$	$\{b\}$

Example 20.1.1

The LR(0) contexts are constructed for the rules of the grammar

$$\begin{aligned} S &\rightarrow aA \mid bB \\ A &\rightarrow abA \mid bB \\ B &\rightarrow bBc \mid bc. \end{aligned}$$

The rightmost derivations initiated by the rule $S \rightarrow aA$ have the form

$$S \Rightarrow aA \xrightarrow{i} a(ab)^i A \Rightarrow a(ab)^i bB \xrightarrow{j} a(ab)^i bb^j Bc^j \Rightarrow a(ab)^i bb^j bcc^j,$$

where $i, j \geq 0$. Derivations beginning with $S \rightarrow bB$ can be written

$$S \Rightarrow bB \xrightarrow{i} bb^i Bc^i \Rightarrow bb^i bcc^i.$$

The LR(0) contexts can be obtained from the sentential forms generated in the preceding derivations.

Rule	LR(0) Contexts
$S \rightarrow aA$	$\{aA\}$
$S \rightarrow bB$	$\{bB\}$
$A \rightarrow abA$	$\{a(ab)^i A \mid i > 0\}$
$A \rightarrow bB$	$\{a(ab)^i bB \mid i \geq 0\}$
$B \rightarrow bBc$	$\{a(ab)^i bb^j Bc, bb^j Bc \mid i \geq 0, j > 0\}$
$B \rightarrow bc$	$\{a(ab)^i bb^j c, bb^j c \mid i \geq 0, j > 0\}$

□

The contexts can be used to eliminate reductions from consideration by the parser. When the LR(0) contexts provide sufficient information to eliminate all but one action, the grammar is called an LR(0) grammar.

Definition 20.1.2

A context-free grammar $G = (V, \Sigma, P, S)$ with nonrecursive start symbol S is LR(0) if, for every $u \in (V \cup \Sigma)^*$ and $v \in \Sigma^*$,

$$u \in \text{LR}(0)\text{-CONTEXT}(A \rightarrow w_1) \quad \text{and} \quad uv \in \text{LR}(0)\text{-CONTEXT}(B \rightarrow w_2)$$

implies $v = \lambda$, $A = B$, and $w_1 = w_2$.

The grammar from Example 20.1.1 is LR(0). Examining the table of LR(0) contexts, we see that there is no LR(0) context of a rule that is a prefix of an LR(0) context of another rule.

The contexts of an LR(0) grammar provide the information needed to select the appropriate action. Upon scanning the string u , the parser takes one of three mutually exclusive actions:

1. If $u \in \text{LR}(0)\text{-CONTEXT}(A \rightarrow w)$, then u is reduced with the rule $A \rightarrow w$.
2. If u is not an LR(0) context but is a prefix of some LR(0) context, then the parser effects a shift.
3. If u is not the prefix of any LR(0) context, then the input string is rejected.

Since a string u is a unique action. A that uv is an LR(0) shift operations pr

Example 20.1.2

The grammar

is not LR(0). The r

for $i \geq 0$. The LR(0) sentential forms in

The grammar G is

20.2 An LR(0)

Incorporating the grammar into a box string p is scanned is determined by co prefix of the senten The operation shift end of u .

Since a string u is an LR(0) context for at most one rule $A \rightarrow w$, the first condition specifies a unique action. A string u is called a *viable prefix* if there is a string $v \in (V \cup \Sigma)^*$ such that uv is an LR(0) context. If u is a viable prefix and not an LR(0) context, a sequence of shift operations produces the LR(0) context uv .

Example 20.1.2

The grammar

$$G: S \rightarrow aA \mid aB$$

$$A \rightarrow aAb \mid b$$

$$B \rightarrow bBa \mid b$$

is not LR(0). The rightmost derivations of G have the form

$$S \Rightarrow aA \xrightarrow{i} aa^i Ab^i \Rightarrow aa^i bb^i$$

$$S \Rightarrow aB \xrightarrow{i} ab^i Ba^i \Rightarrow ab^i ba^i$$

for $i \geq 0$. The LR(0) contexts for the rules of the grammar can be obtained from the right sentential forms in the preceding derivations.

Rule	LR(0) Contexts
$S \rightarrow aA$	$\{aA\}$
$S \rightarrow aB$	$\{aB\}$
$A \rightarrow aAb$	$\{aa^i Ab \mid i > 0\}$
$A \rightarrow b$	$\{aa^i b \mid i \geq 0\}$
$B \rightarrow bBa$	$\{ab^i Ba \mid i > 0\}$
$B \rightarrow b$	$\{ab^i \mid i > 0\}$

The grammar G is not LR(0) since ab is an LR(0) context of both $B \rightarrow b$ and $A \rightarrow b$. \square

20.2 An LR(0) Parser

Incorporating the information provided by the LR(0) contexts of the rules of an LR(0) grammar into a bottom-up parser produces a deterministic parsing algorithm. The input string p is scanned in a left-to-right manner. The action of the parser in Algorithm 20.2.1 is determined by comparing the LR(0) contexts with the string scanned. The string u is the prefix of the sentential form scanned by the parser, and v is the remainder of the input string. The operation $shift(u, v)$ removes the first symbol from v and concatenates it to the right end of u .

Algorithm 20.2.1**Parser for an LR(0) Grammar**

input: LR(0) grammar $G = (V, \Sigma, P, S)$
 string $p \in \Sigma^*$

1. $u := \lambda, v := p$
2. dead-end := false
3. repeat
 - 3.1. if $u \in \text{LR}(0)\text{-CONTEXT}(A \rightarrow w)$ for the rule $A \rightarrow w$ in P
 where $u = xw$ then $u := xA$
 else if u is a viable prefix and $v \neq \lambda$ then shift(u, v)
 else dead-end := true
- until $u = S$ or dead-end
4. if $u = S$ then accept else reject

The decision to reduce with the rule $A \rightarrow w$ is made as soon as a substring $u = xw$ is encountered. The decision does not use any information contained in v , the unscanned portion of the string. The parser does not look beyond the string xw , hence the zero in LR(0) indicating no lookahead is required.

One detail has been overlooked in Algorithm 20.2.1. No technique has been provided for deciding whether a string is a viable prefix or an LR(0) context of a rule of the grammar. In the next section we will design a finite automaton whose computations identify LR(0) contexts and viable prefixes.

Example 20.2.1

The string $aabbcc$ is parsed using the rules and LR(0) contexts of the grammar presented in Example 20.1.1 and the parsing algorithm for LR(0) grammars.

u	v	Rule	Action
λ	$aabbcc$		shift
a	$abbcc$		shift
aa	$bbcc$		shift
aab	bc		shift
$aabb$	c		shift
$aabb$	bcc		shift
$aabb$	c		shift
$aabb$	c	$B \rightarrow bc$	reduce
$aabbB$	c		shift
$aabbB$	λ	$B \rightarrow bBc$	reduce

20.3 The LR(k) Grammars

To select the appropriate reduction rule, the parser must determine which rule may contain infinite loops. This is done by generating sets of LR(0) items. These sets are avoided in LL(k) grammars because they force the parser to make the decision to reduce before it has enough lookahead. The LR(0) grammar is not LL(k).

The LR(0) context of a rule is the set of strings that satisfy the prefix condition. It is the union of all LR(0) items containing an element of both the left-hand side and right-hand side of the rule. To discriminate between rules, the parser must determine which LR(0) item is active at any given time.

One may be tempted to alter the suffix of the string to be parsed to make it a viable prefix. This alters the suffix of the string to be parsed.

u	v	Rule	Action
$aabbB$	λ	$A \rightarrow bB$	reduce
$aabA$	λ	$A \rightarrow abA$	reduce
aA	λ	$S \rightarrow aA$	reduce
S			

□

20.3 The LR(0) Machine

To select the appropriate action, the LR(0) parser compares the string u being processed with the LR(0)-contexts of the rules of the grammar. Since the set of LR(0) contexts of a rule may contain infinitely many strings and strings in set may be arbitrarily long, we cannot generate these sets for a direct comparison. The problem of dealing with infinite sets was avoided in LL(k) grammars by restricting the length of the lookahead strings. Unfortunately, the decision to reduce a string requires knowledge of the entire scanned string (the context). The LR(0) grammars G_1 and G_2 demonstrate this dependence.

The LR(0) contexts of the rules $A \rightarrow aAb$ and $A \rightarrow ab$ of G_1 form disjoint sets that satisfy the prefix conditions. If these sets are truncated at any length k , the string a^k will be an element of both of the truncated sets. The final two symbols of the context are required to discriminate between these reductions.

Rule	LR(0) Contexts
$G_1: S \rightarrow A$	$\{A\}$
$A \rightarrow aAa$	$\{a^i Aa \mid i > 0\}$
$A \rightarrow aAb$	$\{a^i Ab \mid i > 0\}$
$A \rightarrow ab$	$\{a^i b \mid i > 0\}$

One may be tempted to consider only fixed-length suffixes of contexts, since a reduction alters the suffix of the scanned string. The grammar G_2 exhibits the futility of this approach.

Rule	LR(0) Contexts
$G_2: S \rightarrow A$	$\{A\}$
$S \rightarrow bB$	$\{bB\}$
$A \rightarrow aA$	$\{a^i A \mid i > 0\}$
$A \rightarrow ab$	$\{a^i b \mid i > 0\}$
$B \rightarrow aB$	$\{ba^i B \mid i > 0\}$
$B \rightarrow ab$	$\{ba^i b \mid i > 0\}$

The sole difference between the LR(0) contexts of $A \rightarrow ab$ and $B \rightarrow ab$ is the first element of the string. A parser will be unable to discriminate between these rules if the selection process uses only fixed-length suffixes of the LR(0) contexts.

The grammars G_1 and G_2 demonstrate that the entire scanned string is required by the LR(0) parser to select the appropriate action. Fortunately, this does not imply that the complete set of LR(0) contexts is required. For a given grammar, a finite automaton can be constructed whose computations determine whether a string is a viable prefix of the grammar. The states of the machine, called LR(0) items, are constructed directly from the rules of the grammar.

Definition 20.3.1

Let $G = (V, \Sigma, P, S)$ be a context-free grammar. The **LR(0) items** of G are defined as follows:

- If $A \rightarrow uv \in P$, then $A \rightarrow u.v$ is an LR(0) item.
- If $A \rightarrow \lambda \in P$, then $A \rightarrow .$ is an LR(0) item.

The LR(0) items are obtained from the rules of the grammar by placing the marker “.” in the right-hand side of a rule. An item “ $A \rightarrow u.$ ” is called a *complete item*. A rule whose right-hand side has length n generates $n + 1$ items, one for each possible position of the marker.

Definition 20.3.2

Let $G = (V, \Sigma, P, S)$ be a context-free grammar. The **nondeterministic LR(0) machine** of G is an NFA- λ $M = (Q, V \cup \Sigma, \delta, q_0, Q)$, where Q is the set of LR(0) items augmented with the state q_0 . The transition function is defined by

- $\delta(q_0, \lambda) = \{S \rightarrow .w \mid S \rightarrow w \in P\}$
- $\delta(A \rightarrow u.av, a) = \{A \rightarrow ua.v\}$
- $\delta(A \rightarrow u.Bv, B) = \{A \rightarrow u.B.v\}$
- $\delta(A \rightarrow u.Bv, \lambda) = \{B \rightarrow .w \mid B \rightarrow w \in P\}.$

The computations of the nondeterministic LR(0) machine M of a grammar G completely process strings that are viable prefixes of the grammar. All other computations halt prior to reading the entire input. Since all the states of M are accepting, M accepts precisely the viable prefixes of the original grammar. A computation of M records the progress made toward matching the right-hand side of a rule of G . The item $A \rightarrow u.v$ indicates that the string u has been scanned and the automaton is looking for the string v to complete the match.

The symbol following the marker in an item defines the arcs leaving a node. If the marker precedes a terminal, the only arc leaving the node is labeled by that terminal. Arcs labeled B or λ may leave a node containing an item of the form $A \rightarrow u.Bv$. To extend the match of the right-hand side of the rule, the machine is looking for a B . The node $A \rightarrow u.B.v$

is entered if the path
B may be obtained
for the right-hand side.

Definition 20.3.2
G given in the fol-
the associated NF

The NFA- λ is
prefix of a context
the LR(0) machine
 $A.B, B \rightarrow .bBa,$
beginning with $A.$
prefix of the rule $B \rightarrow .bBa$.

The technique
from the nondeter-
machine of G , is
 λ -closure of q_0 , t-
failure, the empty
string u successfu

incorporates the L

is the first
rules if the
required by
only that the
maton can
efix of the
y from the

defined as

marker “.”
rule whose
tion of the

) machine
augmented

ar G com-
tations halt
ts precisely
gress made
tes that the
mplete the

ode. If the
ninal. Arcs
extend the
 $A \rightarrow uB.v$

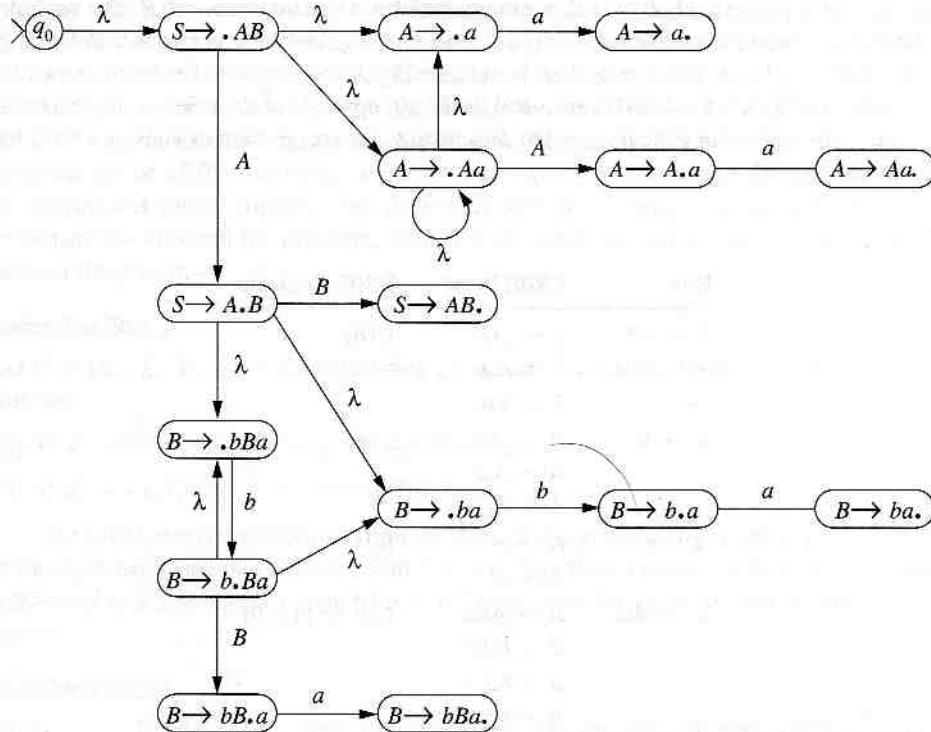
is entered if the parser reads B . It is also looking for strings that may produce B . The variable B may be obtained by a reduction using a B rule. Consequently, the parser is also looking for the right-hand side of a B rule. This is indicated by λ -transitions to the items $B \rightarrow .w$.

Definition 20.3.2, the LR(0) items, and the LR(0) contexts of the rules of the grammar G given in the following table are used to demonstrate the recognition of viable prefixes by the associated NFA- λ .

Rule	LR(0) Items	LR(0) Contexts
$S \rightarrow AB$	$S \rightarrow .AB$	$\{AB\}$
	$S \rightarrow A.B$	
	$S \rightarrow AB.$	
$A \rightarrow Aa$	$A \rightarrow .Aa$	$\{Aa\}$
	$A \rightarrow A.a$	
	$A \rightarrow Aa.$	
$A \rightarrow a$	$A \rightarrow .a$	$\{a\}$
	$A \rightarrow a.$	
$B \rightarrow bBa$	$B \rightarrow .bBa$	$\{Ab^i Ba \mid i > 0\}$
	$B \rightarrow b.Ba$	
	$B \rightarrow bB.a$	
	$B \rightarrow bBa.$	
$B \rightarrow ba$	$B \rightarrow .ba$	$\{Ab^i ba \mid i \geq 0\}$
	$B \rightarrow b.a$	
	$B \rightarrow ba.$	

The NFA- λ in Figure 20.1 is the LR(0) machine of the grammar G . A string w is a prefix of a context of the rule $A \rightarrow uv$ if $A \rightarrow u.v \in \hat{\delta}(q_0, w)$. The computation $\hat{\delta}(q_0, A)$ of the LR(0) machine in Figure 20.1 halts in the states containing the items $A \rightarrow A.a$, $S \rightarrow A.B$, $B \rightarrow .bBa$, and $B \rightarrow .ba$. These are precisely the rules that have LR(0) contexts beginning with A . Similarly, the computation with input AbB indicates that AbB is a viable prefix of the rule $B \rightarrow bBa$ and no other.

The techniques presented in Chapter 5 can be used to construct an equivalent DFA from the nondeterministic LR(0) machine of G . This machine, the **deterministic LR(0) machine** of G , is given in Figure 20.2. The start state q_s of the deterministic machine is the λ -closure of q_0 , the start state of the nondeterministic machine. The state that represents failure, the empty set, has been omitted. When the computation obtained by processing the string u successfully terminates, u is an LR(0) context or a viable prefix. Algorithm 20.3.3 incorporates the LR(0) machine into the LR(0) parsing strategy.

FIGURE 20.1 Nondeterministic LR(0) machine of G .**Algorithm 20.3.3****Parser Utilizing the Deterministic LR(0) Machine**input: LR(0) grammar $G = (V, \Sigma, P, S)$ string $p \in \Sigma^*$ deterministic LR(0) machine of G

1. $u := \lambda, v := p$
2. dead-end := false
3. repeat
 - 3.1. if $\hat{\delta}(q_s, u)$ contains $A \rightarrow w$, where $u = xw$ then $u := xA$
 else if $\hat{\delta}(q_s, u)$ contains an item $A \rightarrow y.z$ and $v \neq \lambda$ then shift(u, v)
 else dead-end := true
 - until $u = S$ or dead-end
4. if $u = S$ then accept else reject

The decision computation $\hat{\delta}(q_s, u)$ for $A \rightarrow w$, then $u := xA$ with the resulting u to extend the resulting u until it is empty.

Example 20.3.1

The string $aabb$ is an LR(0) grammar. Figure 20.2. Upon a reduction using shifts and constituents Aa is an LR(0) grammar.

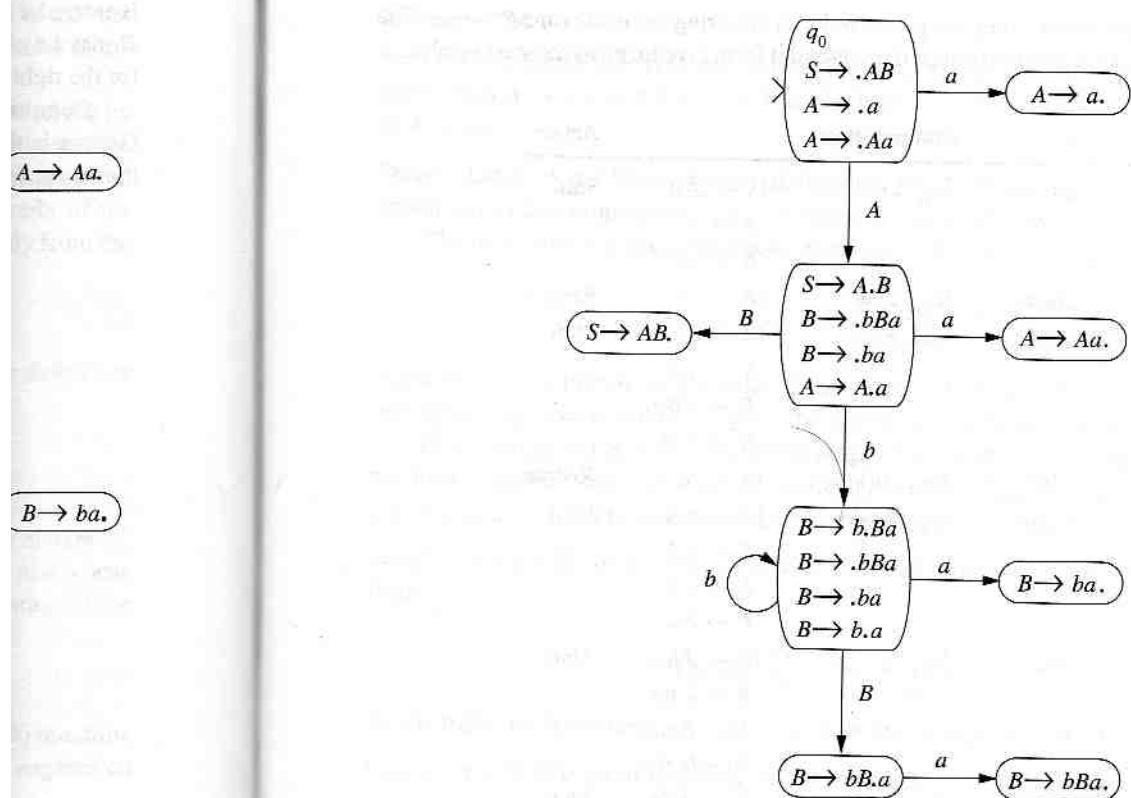


FIGURE 20.2 Deterministic LR(0) machine of G.

The decision of which action to take is made in step 3.1 based on the result of the computation $\hat{\delta}(q_s, u)$ by the LR(0) machine. If $\hat{\delta}(q_s, u)$ contains a complete LR(0) item $A \rightarrow w.$, then a reduction with the rule $A \rightarrow w$ is performed and the loop is repeated with the resulting string. If $\hat{\delta}(q_s, u)$ contains an LR(0) item $A \rightarrow y.z$, a shift is performed to extend the match of the viable prefix. Finally, the computation halts if $\hat{\delta}(q_s, u)$ is empty.

Example 20.3.1

The string $aabbbaa$ is parsed using Algorithm 20.3.3 and the deterministic LR(0) machine in Figure 20.2. Upon processing the leading a , the machine enters the state $A \rightarrow a.$, specifying a reduction using the rule $A \rightarrow a$. Since $\hat{\delta}(q_s, A)$ does not contain a complete item, the parser shifts and constructs the string Aa . The computation $\hat{\delta}(q_s, Aa) = \{A \rightarrow Aa.\}$ indicates that Aa is an LR(0) context of $A \rightarrow Aa$ and that it is not a prefix of a context of any other rule.

Having generated a complete item, the parser reduces the string using the rule $A \rightarrow Aa$. The shift and reduction cycle continues until the sentential form is reduced to the start symbol S .

u	v	Computation	Action
λ	$aabbba$	$\hat{\delta}(q_s, \lambda) = \{S \rightarrow .AB, A \rightarrow .a, A \rightarrow .Aa\}$	Shift
a	$abbaa$	$\hat{\delta}(q_s, a) = \{A \rightarrow a.\}$	Reduce
A	$abbaa$	$\hat{\delta}(q_s, A) = \{A \rightarrow A.a, S \rightarrow A.B, B \rightarrow .bBa, B \rightarrow .ba\}$	Shift
Aa	baa	$\hat{\delta}(q_s, Aa) = \{A \rightarrow Aa.\}$	Reduce
A	baa	$\hat{\delta}(q_s, A) = \{A \rightarrow A.a, S \rightarrow A.B, B \rightarrow .bBa, B \rightarrow .ba\}$	Shift
Ab	baa	$\hat{\delta}(q_s, Ab) = \{B \rightarrow .bBa, B \rightarrow b.Ba, B \rightarrow .ba, B \rightarrow b.a\}$	Shift
Abb	aa	$\hat{\delta}(q_s, Abb) = \{B \rightarrow .bBa, B \rightarrow b.Ba, B \rightarrow .ba, B \rightarrow b.a\}$	Shift
$Abba$	a	$\hat{\delta}(q_s, Abba) = \{B \rightarrow ba.\}$	Reduce
AbB	a	$\hat{\delta}(q_s, AbB) = \{B \rightarrow b.B.a\}$	Shift
$AbBa$	λ	$\hat{\delta}(q_s, AbBa) = \{B \rightarrow bBa.\}$	Reduce
AB	λ	$\hat{\delta}(q_s, AB) = \{S \rightarrow AB.\}$	Reduce
S			

□

Theorem 20.4.1

Let G be a context-free grammar with an LR(0) item $A \rightarrow u.v$. Then $A \rightarrow uv$ if and only if $A \rightarrow u.v$ is a final item.

Proof. Let $A \rightarrow u.v$ be a final item. We show that $A \rightarrow uv$ has transitions in the LR(0) item $A \rightarrow u.v$.

The basis case is

where $S \rightarrow q$ is a reduction transition.

Setting $p = \lambda$, $u = v$.

Now let $\hat{\delta}(q_0, p) = \{x \in V \cup \Sigma \cup \{\lambda\} : T(x) \text{ is a final item}\}$.

Case 1: $x = a \in \Sigma$. Then $T(a)$ is a final item.

By the inductive hypothesis.

Case 2: $x \in V$. Then $T(x)$ is a final item.

Case 3: $x = \lambda$. If $T(\lambda)$ is a final item, then $A \rightarrow uv$ is a final item.

The inductive hypothesis holds for the context of $B \rightarrow rA$.

The application of the theorem.

The final step of the proof.

To establish the theorem, we show that $A \rightarrow u.v$ whenever $\hat{\delta}(q_0, p) = \{x \in V \cup \Sigma \cup \{\lambda\} : T(x) \text{ is a final item}\}$ and $\hat{\delta}(q_0, p)$ contains $A \rightarrow u.v$ from conditions (ii).

20.4 Acceptance by the LR(0) Machine

The LR(0) machine has been constructed to decide whether a string is a viable prefix of the grammar. Theorem 20.4.1 establishes that computations of the LR(0) machine provide the desired information.

Aa. The symbol S .

Theorem 20.4.1

Let G be a context-free grammar and M the nondeterministic LR(0) machine of G . The LR(0) item $A \rightarrow u.v$ is in $\hat{\delta}(q_0, w)$ if, and only if, $w = pu$, where puv is an LR(0) context of $A \rightarrow uv$.

Proof. Let $A \rightarrow u.v$ be an element of $\hat{\delta}(q_0, w)$. We prove, by induction on the number of transitions in the computation $\hat{\delta}(q_0, w)$, that wv is an LR(0) context of $A \rightarrow uv$.

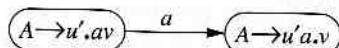
The basis consists of computations of length 1. All such computations have the form



where $S \rightarrow q$ is a rule of the grammar. These computations process the input string $w = \lambda$. Setting $p = \lambda$, $u = \lambda$, and $v = q$ gives the desired decomposition of w .

Now let $\hat{\delta}(q_0, w)$ be a computation of length $k > 1$ with $A \rightarrow u.v$ in $\hat{\delta}(q_0, w)$. Isolating the final transition, we can write this computation as $\delta(\hat{\delta}(q_0, y), x)$, where $w = yx$ and $x \in V \cup \Sigma \cup \{\lambda\}$. The remainder of the proof is divided into three cases.

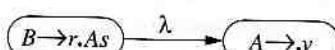
Case 1: $x = a \in \Sigma$. In this case, $u = u'a$. The final transition of the computation has the form



By the inductive hypothesis, $pu'av = wv$ is an LR(0) context of $A \rightarrow uv$.

Case 2: $x \in V$. The proof is similar to that of case 1.

Case 3: $x = \lambda$. If $x = \lambda$, then $y = w$ and the computation terminates at an item $A \rightarrow .v$. The final transition has the form



The inductive hypothesis implies that w can be written $w = pr$, where $prAs$ is an LR(0) context of $B \rightarrow rAs$. Thus there is a rightmost derivation

$$S \xrightarrow[R]{*} pBq \xrightarrow[R]{*} prAsq.$$

The application of $A \rightarrow v$ yields

$$S \xrightarrow[R]{*} pBq \xrightarrow[R]{*} prAsq \xrightarrow[R]{*} prvsq.$$

The final step of this derivation shows that $prv = wv$ is an LR(0) context of $A \rightarrow v$.

To establish the opposite implication, we must show that $\hat{\delta}(q_0, pu)$ contains the item $A \rightarrow u.v$ whenever puv is an LR(0) context of a rule $A \rightarrow uv$. First, we note that if $\hat{\delta}(q_0, p)$ contains $A \rightarrow .uv$, then $\hat{\delta}(q_0, pu)$ contains $A \rightarrow u.v$. This follows immediately from conditions (ii) and (iii) of Definition 20.3.2.

Since puv is an LR(0) context of $A \rightarrow uv$, there is a derivation

$$S \xrightarrow[R]{*} pAq \xrightarrow[R]{*} puvq.$$

We prove, by induction on the length of the derivation $S \xrightarrow[R]{*} pAq$, that $\hat{\delta}(q_0, p)$ contains $A \rightarrow .uv$. The basis consists of derivations $S \Rightarrow pAq$ of length 1. The desired computation consists of traversing the λ -arc to $S \rightarrow .pAq$ followed by the arcs that process the string p . The computation is completed by following the λ -arc from $S \rightarrow p.Aq$ to $A \rightarrow .uv$.

Now consider a derivation in which the variable A is introduced on the k th rule application. A derivation of this form can be written

$$S \xrightarrow[R]{k-1} xBy \xrightarrow[R]{*} xwAzy.$$

The inductive hypothesis asserts that $\hat{\delta}(q_0, x)$ contains the item $B \rightarrow .wAz$. Hence $B \rightarrow w.Az \in \hat{\delta}(q_0, xw)$. The λ -transition to $A \rightarrow .uv$ completes the computation. ■

The relationships in Lemma 20.4.2 between derivations in a context-free grammar and the items in the nodes of the deterministic LR(0) machine of the grammar follow from Theorem 20.4.1. The proof of Lemma 20.4.2 is left as an exercise. Recall that q_s is the start symbol of the deterministic machine.

Lemma 20.4.2

Let M be the deterministic LR(0) machine of a context-free grammar G . Assume $\hat{\delta}(q_s, w)$ contains an item $A \rightarrow u.Bv$.

- i) If $B \xrightarrow{*} \lambda$, then $\hat{\delta}(q_s, w)$ contains an item of the form $C \rightarrow .$ for some variable $C \in V$.
- ii) If $B \xrightarrow{*} x \in \Sigma^+$, then there is an arc labeled by a terminal symbol leaving the node $\hat{\delta}(q_s, w)$ or $\hat{\delta}(q_s, w)$ contains an item of the form $C \rightarrow .$ for some variable $C \in V$.

Lemma 20.4.3

Let M be the deterministic LR(0) machine of an LR(0) grammar G . Assume $\hat{\delta}(q_s, u)$ contains the complete item $A \rightarrow w..$. Then $\hat{\delta}(q_s, ua)$ is undefined for all terminal symbols $a \in \Sigma$.

Proof. By Theorem 20.4.1, u is an LR(0) context of $A \rightarrow w$. Assume that $\hat{\delta}(q_s, ua)$ is defined for some terminal a . Then ua is a prefix of an LR(0) context of some rule $B \rightarrow y$. This implies that there is a derivation

$$S \xrightarrow[R]{*} pBv \Rightarrow pyv = uazv$$

with $z \in (V \cup \Sigma)^*$ and $v \in \Sigma^*$. Consider the possibilities for the string z . If $z \in \Sigma^*$, then uaz is an LR(0) context of the rule $B \rightarrow y$. If z is not a terminal string, then there is a terminal string derivable from z

$$z \xrightarrow[R]{*} rCs \Rightarrow rts \quad r, s, t \in \Sigma^*,$$

where $C \rightarrow t$. Combining this with either case, u is an LR(0) context of $B \rightarrow y$.

The previous section shows that G is LR(0).

Theorem 20.4.4

Let G be a context-free grammar. If, the extended LR(0) machine M following condition

- i) If $\hat{\delta}(q_s, u)$ contains an item $A \rightarrow .$ for some variable $A \in V$, then $\hat{\delta}(q_s, u)$ contains other items of the form $C \rightarrow .$ for some variable $C \in V$.
- ii) If $\hat{\delta}(q_s, u)$ contains an item of the form $C \rightarrow .$ for some variable $C \in V$, then $\hat{\delta}(q_s, u)$ contains all other items of the form $C \rightarrow .$ for some variable $C \in V$.

Proof. First we show that M is an extended LR(0) machine. Consider the rule $A \rightarrow w..$ in M . It is clear that $w..$ is an LR(0) context of A . Since $w..$ is an LR(0) context of A , it is an LR(0) context of $A \rightarrow w..$ in M .

Conversely, let $A \rightarrow w..$ be an LR(0) context of M . Then $w..$ is an LR(0) context of A . By Theorem 20.4.1, $w..$ is an LR(0) context of A . Therefore, $w..$ is an LR(0) context of $A \rightarrow w..$ in M . This shows that M is an extended LR(0) machine.

Now assume that M is an extended LR(0) machine. Let $A \rightarrow w..$ be an LR(0) context of M . By Theorem 20.4.2, if there is a derivation $S \xrightarrow[R]{*} pBv$ such that p is a complete item of $\hat{\delta}(q_s, u)$, then u is an LR(0) context of $B \rightarrow y$. This shows that G is LR(0). The proof is complete.

Intuitively, if $\hat{\delta}(q_s, u)$ contains an item $A \rightarrow .$ for some variable $A \in V$, then $\hat{\delta}(q_s, u)$ contains other items of the form $C \rightarrow .$ for some variable $C \in V$. This is because the items generated by A contain items generated by C . The rules $S \xrightarrow[R]{*} pBv$ and $B \xrightarrow{*} y$ are related by the fact that p is a complete item of $\hat{\delta}(q_s, u)$.

The string a is an LR(0) context of $B \rightarrow b$. The proof of Theorem 20.4.4 is complete.

Example 20.4.1

where $C \rightarrow t$ is the final rule application in the derivation of the terminal string from z . Combining the derivations from S and z shows that $uart$ is an LR(0) context of $C \rightarrow t$. In either case, u is an LR(0) context and ua is a viable prefix. This contradicts the assumption that G is LR(0). ■

The previous results can be combined with Definition 20.1.2 to obtain a characterization of LR(0) grammars in terms of the structure of the deterministic LR(0) machine.

Theorem 20.4.4

Let G be a context-free grammar with a nonrecursive start symbol. G is LR(0) if, and only if, the extended transition function $\hat{\delta}$ of the deterministic LR(0) machine of G satisfies the following conditions:

- i) If $\hat{\delta}(q_s, u)$ contains a complete item $A \rightarrow w$, with $w \neq \lambda$, then $\hat{\delta}(q_s, u)$ contains no other items.
- ii) If $\hat{\delta}(q_s, u)$ contains a complete item $A \rightarrow .$, then the marker is followed by a variable in all other items in $\hat{\delta}(q_s, u)$.

Proof. First we show that a grammar G with nonrecursive start symbol is LR(0) when the extended transition function satisfies conditions (i) and (ii). Let u be an LR(0) context of the rule $A \rightarrow w$. Then $\hat{\delta}(q_s, uv)$ is defined only when v begins with a variable. Thus, for all strings $v \in \Sigma^*$, $uv \in \text{LR}(0)\text{-CONTEXT}(B \rightarrow x)$ implies $v = \lambda$, $B = A$, and $w = x$.

Conversely, let G be an LR(0) grammar and u an LR(0) context of the rule $A \rightarrow w$. By Theorem 20.4.1, $\hat{\delta}(q_s, u)$ contains the complete item $A \rightarrow w$. The state $\hat{\delta}(q_s, u)$ does not contain any other complete items $B \rightarrow v$, since this would imply that u is also an LR(0) context of $B \rightarrow v$. By Lemma 20.4.3, all arcs leaving $\hat{\delta}(q_s, u)$ must be labeled by variables.

Now assume that $\hat{\delta}(q_s, u)$ contains a complete item $A \rightarrow w$, where $w \neq \lambda$. By Lemma 20.4.2, if there is an arc labeled by a variable with tail $\hat{\delta}(q_s, u)$, then $\hat{\delta}(q_s, u)$ contains a complete item $C \rightarrow .$ or $\hat{\delta}(q_s, u)$ has an arc labeled by a terminal leaving it. In the former case, u is an LR(0) context of both $A \rightarrow w$ and $C \rightarrow \lambda$, contradicting the assumption that G is LR(0). The latter possibility contradicts Lemma 20.4.3. Thus $A \rightarrow w$ is the only item in $\hat{\delta}(q_s, u)$. ■

Intuitively, we would like to say that a grammar is LR(0) if a state containing a complete item contains no other items. This condition is satisfied by all states containing complete items generated by nonnull rules. The previous theorem permits a state containing $A \rightarrow .$ to contain items in which the marker is followed by a variable. Consider the derivation using the rules $S \rightarrow aABC$, $A \rightarrow \lambda$, and $B \rightarrow b$.

$$S \xrightarrow{R} aABC \xrightarrow{R} aAbc \xrightarrow{R} abc$$

The string a is an LR(0) context of $A \rightarrow \lambda$ and a prefix of aAb , which is an LR(0) context of $B \rightarrow b$. The effect of reductions by λ -rules in an LR(0) parser is demonstrated in Example 20.4.1.

Example 20.4.1

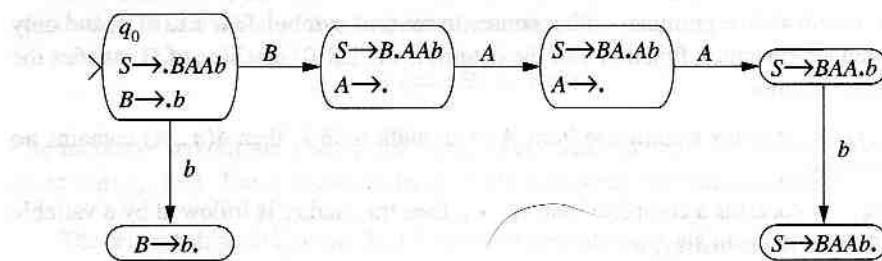
The deterministic LR(0) machine for the grammar

$$G: S \rightarrow BAAb$$

$$A \rightarrow \lambda$$

$$B \rightarrow b$$

is



The analysis of the string bb is traced using the computations of the machine to specify the actions of the parser.

u	v	Computation	Action
λ	bb	$\hat{\delta}(q_s, \lambda) = \{S \rightarrow .BAAb, B \rightarrow .b\}$	Shift
b	b	$\hat{\delta}(q_s, b) = \{B \rightarrow b.\}$	Reduce
B	b	$\hat{\delta}(q_s, B) = \{S \rightarrow B.AAb, A \rightarrow .\}$	Reduce
BA	b	$\hat{\delta}(q_s, BA) = \{S \rightarrow BA.Ab, A \rightarrow .\}$	Reduce
BAA	b	$\hat{\delta}(q_s, BAA) = \{S \rightarrow BAA.b\}$	Shift
$BAAb$	λ	$\hat{\delta}(q_s, BAAb) = \{S \rightarrow BAAb.\}$	Reduce
S			

The parser reduces the sentential form with the rule $A \rightarrow \lambda$ whenever the LR(0) machine halts in a state containing the complete item $A \rightarrow ..$. This reduction adds an A to the end of the currently scanned string. In the next iteration, the LR(0) machine follows the arc labeled A to the subsequent state. An A is generated by a λ reduction only when its presence adds to the prefix of an item being recognized. \square

Theorem 20.4.4 establishes a procedure for deciding whether a grammar is LR(0). The process begins by constructing the deterministic LR(0) machine of the grammar. A grammar

with a nonrecursing (ii) of Theorem 2

Example 20.4.2

The grammar A is

is LR(0). The determinstic states containing

Example 20.4.3

The grammar

is not LR(0). This multiplicative substates of the deter

The computation of the item $T \rightarrow T ..$ action: Reduce using

with a nonrecursive start symbol is LR(0) if the restrictions imposed by conditions (i) and (ii) of Theorem 20.4.4 are satisfied by the LR(0) machine.

Example 20.4.2

The grammar AE augmented with the endmarker #,

$$\begin{aligned} \text{AE: } S &\rightarrow A\# \\ A &\rightarrow A + T \mid T \\ T &\rightarrow b \mid (A), \end{aligned}$$

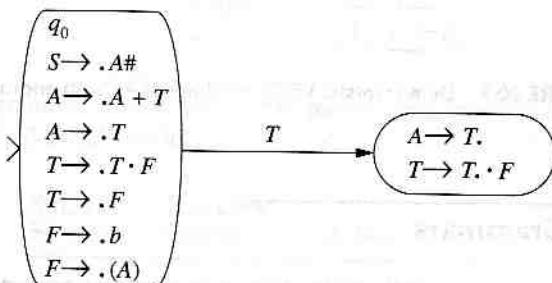
is LR(0). The deterministic LR(0) machine of AE is given in Figure 20.3. Since each of the states containing a complete item is a singleton set, the grammar is LR(0). \square

Example 20.4.3

The grammar

$$\begin{aligned} S &\rightarrow A\# \\ A &\rightarrow A + T \mid T \\ T &\rightarrow T \cdot F \mid F \\ F &\rightarrow b \mid (A) \end{aligned}$$

is not LR(0). This grammar is obtained by adding the variable F (factor) to AE to generate multiplicative subexpressions. We show that this grammar is not LR(0) by constructing two states of the deterministic LR(0) machine.



The computation generated by processing T contains the complete item $A \rightarrow T.$ and the item $T \rightarrow T. \cdot F.$ When the parser scans the string $T,$ there are two possible courses of action: Reduce using $A \rightarrow T$ or shift in an attempt to construct the string $T \cdot F.$ \square

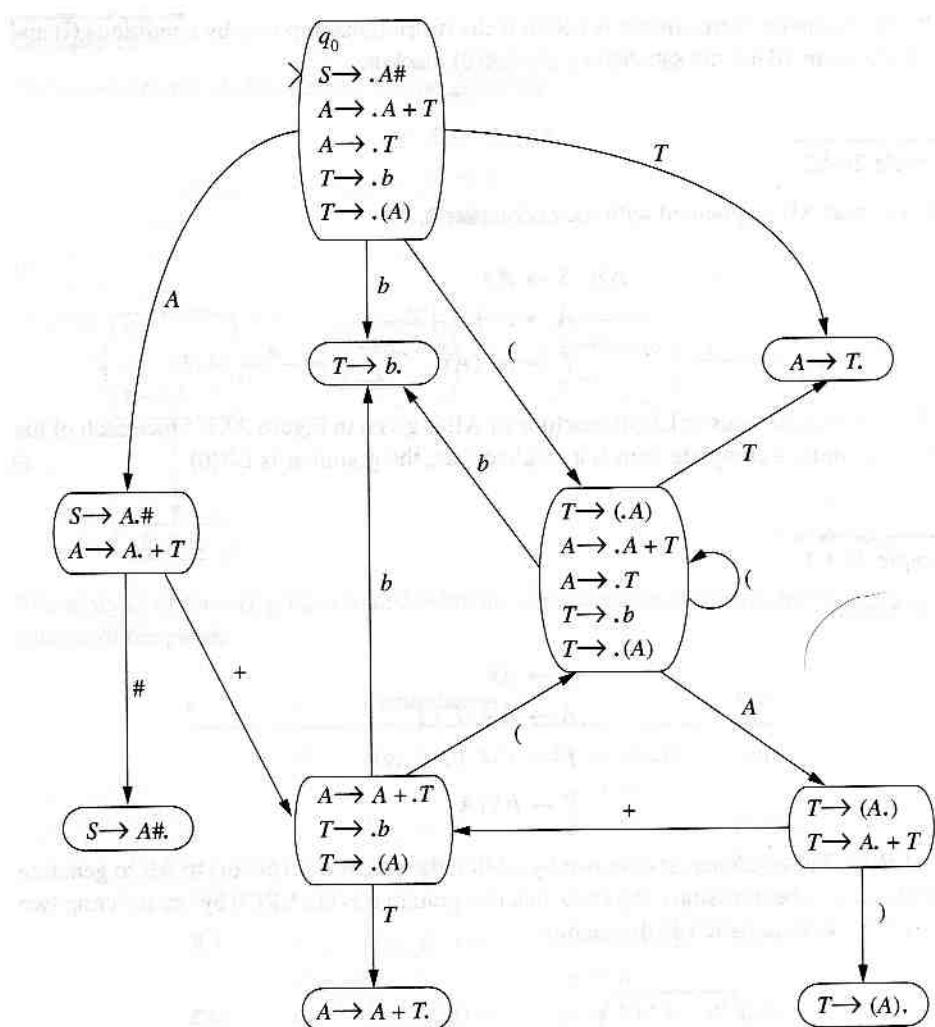


FIGURE 20.3 Deterministic LR(0) machine of AE with endmarker.

20.5 LR(1) Grammars

The LR(0) conditions are generally too restrictive to construct grammars that define programming languages. In this section the LR parser is modified to utilize information obtained by looking beyond the substring that matches the right-hand side of the rule. The lookahead is limited to a single symbol. The definitions and algorithms, with obvious modifications, can be extended to utilize a lookahead of arbitrary length.

A grammar ahead is called LR(0). A substring to be reduced upon scanning a string in grammars, a string

where z is the first by a bottom-up procedure the null string.

The role of t
examined by the

When an LR(0) p

- i) Reduce with
 - ii) Reduce with
 - iii) Shift to obtain

One-symbol look
underlined in each
is scanned by the

In the preceding
determined by the

A grammar in which strings can be deterministically parsed using a one-symbol lookahead is called LR(1). The lookahead symbol is the symbol to the immediate right of the substring to be reduced by the parser. The decision to reduce with the rule $A \rightarrow w$ is made upon scanning a string of the form uwz , where $z \in \Sigma \cup \{\lambda\}$. Following the example of LR(0) grammars, a string uwz is called an **LR(1) context** if there is a derivation

$$S \xrightarrow[R]{*} uAv \Rightarrow uwv,$$

where z is the first symbol of v or the null string if $v = \lambda$. Since the derivation constructed by a bottom-up parser is rightmost, the lookahead symbol z is either a terminal symbol or the null string.

The role of the lookahead symbol in reducing the number of possibilities that must be examined by the parser is demonstrated by considering reductions in the grammar

$$\begin{aligned} G: S &\rightarrow A \mid Bc \\ A &\rightarrow aA \mid a \\ B &\rightarrow a \mid ab. \end{aligned}$$

When an LR(0) parser reads the symbol a , there are three possible actions:

- i) Reduce with $A \rightarrow a$.
- ii) Reduce with $B \rightarrow a$.
- iii) Shift to obtain either aA or ab .

One-symbol lookahead is sufficient to determine the appropriate operation. The symbol underlined in each of the following derivations is the lookahead symbol when the initial a is scanned by the parser.

$$\begin{array}{llll} S \Rightarrow A & S \Rightarrow A & S \Rightarrow Bc & S \Rightarrow Bc \\ \Rightarrow a\underline{-} & \Rightarrow a\underline{A} & \Rightarrow a\underline{c} & \Rightarrow a\underline{bc} \\ & \Rightarrow aaA & & \\ & \Rightarrow aaa & & \end{array}$$

In the preceding grammar, the action of the parser when reading an a is completely determined by the lookahead symbol.

String Scanned	Lookahead Symbol	Action
a	λ	Reduce with $A \rightarrow a$
a	a	Shift
a	b	Shift
a	c	Reduce with $B \rightarrow a$

The action of an LR(0) parser is determined by the result of a computation of the LR(0) machine of the grammar. An LR(1) parser incorporates the lookahead symbol into the decision procedure. An LR(1) item is an ordered pair consisting of an LR(0) item and a set containing the possible lookahead symbols.

Definition 20.5.1

Let $G = (V, \Sigma, P, S)$ be a context-free grammar. The LR(1) items of G have the form

$$[A \rightarrow u.v, \{z_1, z_2, \dots, z_n\}],$$

where $A \rightarrow uv \in P$ and $z_i \in \Sigma \cup \{\lambda\}$. The set $\{z_1, z_2, \dots, z_n\}$ is the lookahead set of the LR(1) item.

The lookahead set of an item $[A \rightarrow u.v, \{z_1, \dots, z_n\}]$ consists of the first symbol in the terminal strings y that follow uv in rightmost derivations.

$$S \xrightarrow[R]{*} xAy \xrightarrow[R]{} xuvy$$

Since the S rules are nonrecursive, the only derivation terminated by a rule $S \rightarrow w$ is the derivation $S \Rightarrow w$. The null string follows w in this derivation. Consequently, the lookahead set of an S rule is always the singleton set $\{\lambda\}$.

As before, a complete item is an item in which the marker follows the entire right-hand side of the rule. The LR(1) machine, which specifies the actions of an LR(1) parser, is constructed from the LR(1) items of the grammar.

Definition 20.5.2

Let $G = (V, \Sigma, P, S)$ be a context-free grammar. The nondeterministic LR(1) machine of G is an NFA- λ $M = (Q, V \cup \Sigma, \delta, q_0, Q)$, where Q is a set of LR(1) items augmented with the state q_0 . The transition function is defined by

- i) $\delta(q_0, \lambda) = \{[S \rightarrow .w, \{\lambda\}] \mid S \rightarrow w \in P\}$
- ii) $\delta([A \rightarrow u.Bv, \{z_1, \dots, z_n\}], B) = \{[A \rightarrow u.B.v, \{z_1, \dots, z_n\}]\}$
- iii) $\delta([A \rightarrow u.av, \{z_1, \dots, z_n\}], a) = \{[A \rightarrow ua.v, \{z_1, \dots, z_n\}]\}$
- iv) $\delta([A \rightarrow u.Bv, \{z_1, \dots, z_n\}], \lambda) = \{[B \rightarrow .w, \{y_1, \dots, y_k\}] \mid B \rightarrow w \in P \text{ where } y_i \in \text{FIRST}_1(vz_j) \text{ for some } j\}$

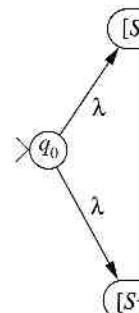
If we disregard the lookahead sets, the transitions of the LR(1) machine defined in (i), (ii), and (iii) have the same form as those of the LR(0) machine. The LR(1) item $[A \rightarrow u.v, \{z_1, \dots, z_n\}]$ indicates that the parser has scanned the string u and is attempting to find v to complete the match of the right-hand side of the rule. The transitions generated by conditions (ii) and (iii) represent intermediate steps in matching the right-hand side of a rule and do not alter the lookahead set. Condition (iv) introduces transitions of the form



Following this arc $B \rightarrow w$. If the state B is reached, the lookahead set contains the strings derived from w .

A bottom-up grammar. An LR(0) item is reduced only if

The state diagram for a grammar G are given below:



[B]

tion of the symbol into item and

the form

set of the

symbol in

$\rightarrow w$ is the lookahead

entire right-
LR(1) parser,

1) machine augmented

where

defined in LR(1) item attempting to generate end side of a the form

Following this arc, the LR(1) machine attempts to match the right-hand side of the rule $B \rightarrow w$. If the string w is found, a reduction of uwv produces $uB.v$, as desired. The lookahead set consists of the symbols that follow w , that is, the first terminal symbol in strings derived from v and the lookahead set $\{z_1, \dots, z_n\}$ if $v \xrightarrow{*} \lambda$.

A bottom-up parser may reduce the string uw to uA whenever $A \rightarrow w$ is a rule of the grammar. An LR(1) parser uses the lookahead set to decide whether to reduce or to shift when this occurs. If $\delta(q_0, uw)$ contains a complete item $[A \rightarrow w., \{z_1, \dots, z_n\}]$, the string is reduced only if the lookahead symbol is in the set $\{z_1, \dots, z_n\}$.

The state diagrams of the nondeterministic and deterministic LR(1) machines of the grammar G are given in Figures 20.4 and 20.5, respectively.

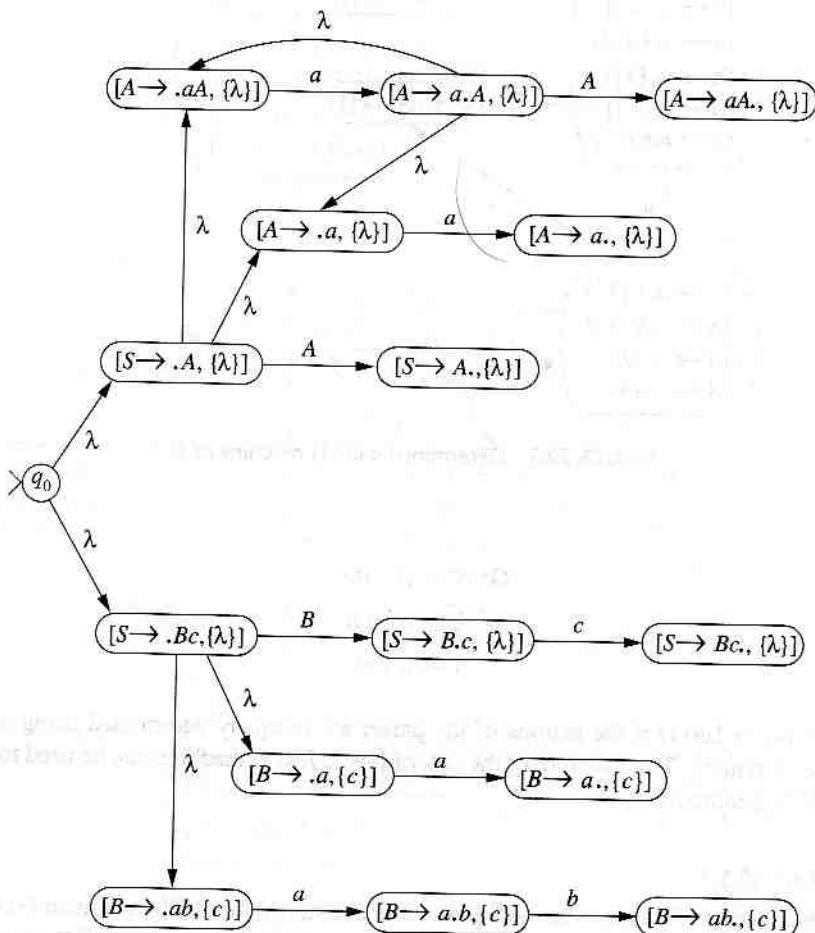
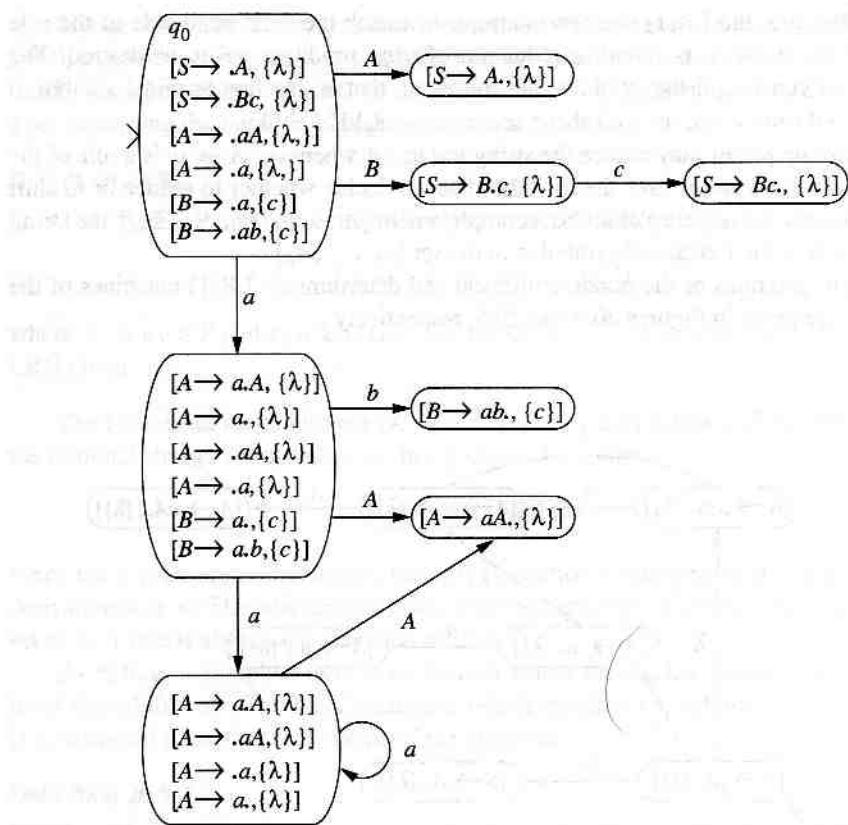


FIGURE 20.4 Nondeterministic LR(1) machine of G .

FIGURE 20.5 Deterministic LR(1) machine of G .

$$G: S \rightarrow A \mid Bc$$

$$A \rightarrow aA \mid a$$

$$B \rightarrow a \mid ab$$

A grammar is LR(1) if the actions of the parser are uniquely determined using a single lookahead symbol. The structure of the deterministic LR(1) machine can be used to define the LR(1) grammars.

Definition 20.5.3

Let G be a context-free grammar with a nonrecursive start symbol. The grammar G is LR(1) if the extended transition function $\hat{\delta}$ of the deterministic LR(1) machine of G satisfies the following conditions:

- i) If $\hat{\delta}(q_s, u)$ contains item $[B \rightarrow r]$
- ii) If $\hat{\delta}(q_s, u)$ contains item $[B \rightarrow v, \{y\}]$

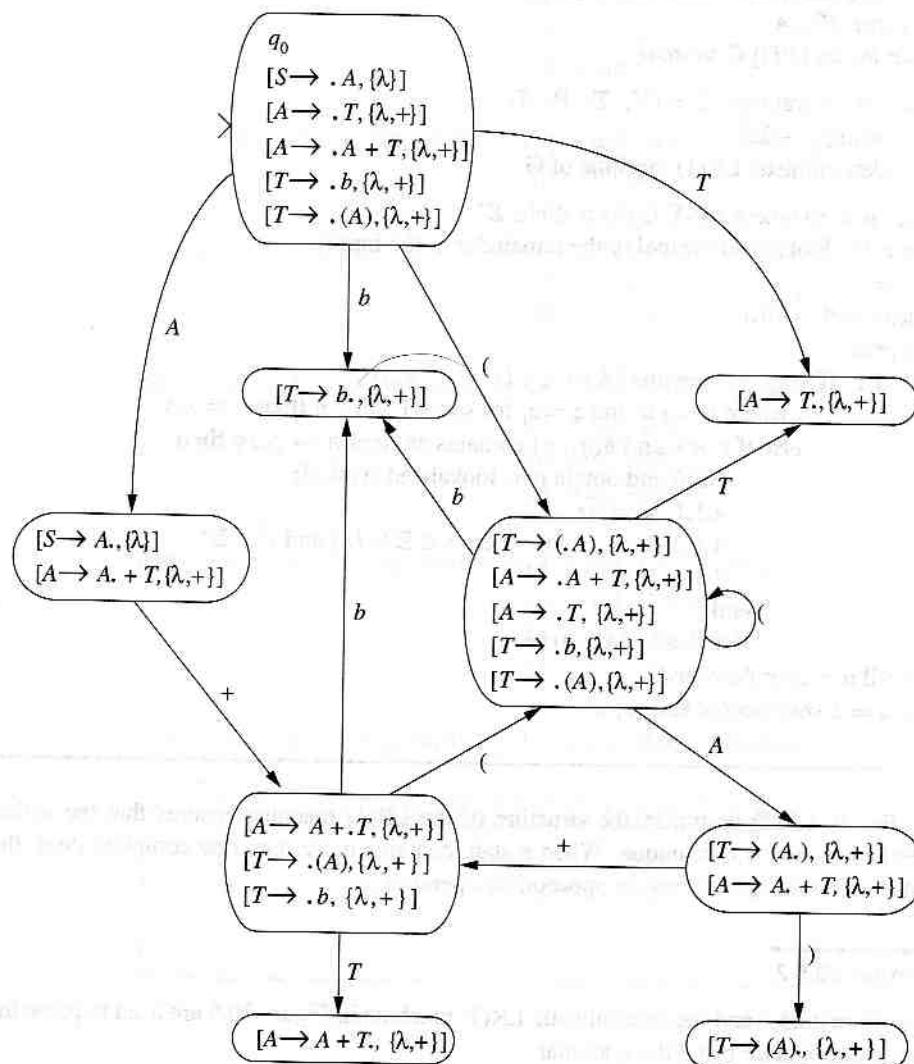
Example 20.5.1

The deterministic LR(1) machine for the grammar

- If $\hat{\delta}(q_s, u)$ contains a complete item $[A \rightarrow w., \{z_1, \dots, z_n\}]$ and $\hat{\delta}(q_s, u)$ contains an item $[B \rightarrow r.as, \{y_1, \dots, y_k\}]$, then $a \neq z_i$ for all $1 \leq i \leq n$.
- If $\hat{\delta}(q_s, u)$ contains two complete items $[A \rightarrow w., \{z_1, \dots, z_n\}]$ and $[B \rightarrow v., \{y_1, \dots, y_k\}]$, then $y_i \neq z_j$ for all $1 \leq i \leq k, 1 \leq j \leq n$.

Example 20.5.1

The deterministic LR(1) machine is constructed for the grammar AE.



single
define

LR(1)
es the

The state containing the complete item $S \rightarrow A.$ also contains $A \rightarrow A. + T.$ It follows that AE is not LR(0). Upon entering this state, the LR(1) parser halts unsuccessfully unless the lookahead symbol is $+$ or the null string. In the latter case, the entire input string has been read and a reduction with the rule $S \rightarrow A$ is specified. When the lookahead symbol is $+$, the parser shifts in an attempt to construct the string $A + T.$ \square

The action of a parser for an LR(1) grammar upon scanning the string u is selected by the result of the computation $\hat{\delta}(q_s, u).$ Algorithm 20.5.4 gives a deterministic algorithm for parsing an LR(1) grammar.

Algorithm 20.5.4
Parser for an LR(1) Grammar

input: LR(1) grammar $G = (V, \Sigma, P, S)$

string $p \in \Sigma^*$

deterministic LR(1) machine of G

1. Let $p = zv$ where $z \in \Sigma \cup \{\lambda\}$ and $v \in \Sigma^*$
(z is the lookahead symbol, v the remainder of the input)
 2. $u := \lambda$
 3. dead-end := false
 4. repeat
 - 4.1. if $\hat{\delta}(q_s, u)$ contains $[A \rightarrow w., \{z_1, \dots, z_n\}]$
where $u = xw$ and $z = z_i$ for some $1 \leq i \leq n$ then $u := xA$
 - else if $z \neq \lambda$ and $\hat{\delta}(q_s, u)$ contains an item $A \rightarrow p.zq$ then
(shift and obtain new lookahead symbol)
 - 4.1.1. $u := uz$
 - 4.1.2. Let $v = zv'$ where $z \in \Sigma \cup \{\lambda\}$ and $v' \in \Sigma^*$
 - 4.1.3. $v := v'$
 - end if
 - else dead-end := true
 - until $u = S$ or dead-end
 5. if $u = S$ then accept else reject
-

For an LR(1) grammar, the structure of the LR(1) machine ensures that the action specified in step 4.1 is unique. When a state contains more than one complete item, the lookahead symbol specifies the appropriate operation.

Example 20.5.2

Algorithm 20.5.4 and the deterministic LR(1) machine in Figure 20.5 are used to parse the strings aaa and ac using the grammar

It follows
illy unless
string has
symbol is

□

selected by
orithm for

parse tree

$$G: S \rightarrow A \mid Bc$$

$$A \rightarrow aA \mid a$$

$$B \rightarrow a \mid ab.$$

u	z	v	Computation	Action
λ	a	aa	$\hat{\delta}(q_s, \lambda) = \{[S \rightarrow .A, \{\lambda\}], [S \rightarrow .Bc, \{\lambda\}], [A \rightarrow .aA, \{\lambda\}], [A \rightarrow .a, \{\lambda\}], [B \rightarrow .a, \{c\}], [B \rightarrow .ab, \{c\}]\}$	Shift
a	a	a	$\hat{\delta}(q_s, a) = \{[A \rightarrow a.A, \{\lambda\}], [A \rightarrow a., \{\lambda\}], [A \rightarrow .aA, \{\lambda\}], [A \rightarrow .a, \{\lambda\}], [B \rightarrow a., \{c\}], [B \rightarrow a.b, \{c\}]\}$	Shift
aa	a	λ	$\hat{\delta}(q_s, aa) = \{[A \rightarrow a.A, \{\lambda\}], [A \rightarrow .aA, \{\lambda\}], [A \rightarrow .a, \{\lambda\}], [A \rightarrow a., \{\lambda\}]\}$	Shift
aaa	λ	λ	$\hat{\delta}(q_s, aaa) = \{[A \rightarrow a.A, \{\lambda\}], [A \rightarrow .aA, \{\lambda\}], [A \rightarrow .a, \{\lambda\}], [A \rightarrow a., \{\lambda\}]\}$	Reduce
aaA	λ	λ	$\hat{\delta}(q_s, aaA) = \{[A \rightarrow aA., \{\lambda\}]\}$	Reduce
aA	λ	λ	$\hat{\delta}(q_s, aA) = \{[A \rightarrow aA., \{\lambda\}]\}$	Reduce
A	λ	λ	$\hat{\delta}(q_s, A) = \{[S \rightarrow A., \{\lambda\}]\}$	Reduce
<hr/>				
S				

the action
item, the

to parse the

u	z	v	Computation	Action
λ	a	c	$\hat{\delta}(q_s, \lambda) = \{[S \rightarrow .A, \{\lambda\}], [S \rightarrow .Bc, \{\lambda\}], [A \rightarrow .aA, \{\lambda\}], [A \rightarrow .a, \{\lambda\}], [B \rightarrow .a\{c\}], [B \rightarrow .ab\{c\}]\}$	Shift
a	c	λ	$\hat{\delta}(q_s, a) = \{[A \rightarrow a.A, \{\lambda\}], [A \rightarrow a., \{\lambda\}], [A \rightarrow .aA, \{\lambda\}], [A \rightarrow .a, \{\lambda\}], [B \rightarrow a., \{c\}], [B \rightarrow a.b, \{c\}]\}$	Reduce
B	c	λ	$\hat{\delta}(q_s, B) = \{[S \rightarrow B.c, \{\lambda\}]\}$	Shift
Bc	λ	λ	$\hat{\delta}(q_s, Bc) = \{[S \rightarrow Bc., \{\lambda\}]\}$	Reduce
<hr/>				
S				

□

Exercises

1. Give the LR(0) contexts for the rules of the following grammars. Build the nondeterministic LR(0) machine. Use this to construct the deterministic LR(0) machine. Is the grammar LR(0)?

- | | |
|--|---|
| a) $S \rightarrow AB$
$A \rightarrow aA \mid b$
$B \rightarrow bB \mid a$ | b) $S \rightarrow Ac$
$A \rightarrow BA \mid \lambda$
$B \rightarrow aB \mid b$ |
| c) $S \rightarrow A$
$A \rightarrow aAb \mid bAa \mid \lambda$ | d) $S \rightarrow aA \mid AB$
$A \rightarrow aAb \mid b$
$B \rightarrow ab \mid b$ |
| e) $S \rightarrow BA \mid BAB$
$A \rightarrow aA \mid \lambda$
$B \rightarrow Bb \mid b$ | f) $S \rightarrow A \mid aB$
$A \rightarrow BC \mid \lambda$
$B \rightarrow Bb \mid C$
$C \rightarrow Cc \mid c$ |

2. Build the deterministic LR(0) machine for the grammar

$$\begin{aligned} S &\rightarrow aAb \mid aB \\ A &\rightarrow Aa \mid \lambda \\ B &\rightarrow Ac. \end{aligned}$$

Use the tech
and ac .

3. Show that the

4. Prove Lemma

5. Prove that an

6. Define the L

7. For each of the

LR(1) machines

a) $S \rightarrow A$

$A \rightarrow B$

$B \rightarrow a$

c) $S \rightarrow A$

$A \rightarrow a$

$B \rightarrow B$

e) $S \rightarrow A$

$A \rightarrow A$

8. Construct the

grammar LR

9. Parse the fol-

actions of the

machine of A

a) $b + b$

b) (b)

c) $b + + b$

Bibliographic

LR grammars were
LR machine made
Korenjak [1969]
these difficulties.
LR) grammars. The
grammars that can
in Aho and Ullma

Use the technique presented in Example 20.3.1 to trace the parse of the strings *aaab* and *ac*.

3. Show that the grammar AE without an endmarker is not LR(0).
4. Prove Lemma 20.4.2.
5. Prove that an LR(0) grammar is unambiguous.
6. Define the LR(*k*) contexts of a rule $A \rightarrow w$.
7. For each of the following grammars, construct the nondeterministic and deterministic LR(1) machines. Is the grammar LR(1)?
 - a) $S \rightarrow Ac$
 $A \rightarrow BA \mid \lambda$
 $B \rightarrow aB \mid b$
 - b) $S \rightarrow A$
 $A \rightarrow AaAb \mid \lambda$
 - c) $S \rightarrow A$
 $A \rightarrow aAb \mid B$
 $B \rightarrow Bb \mid b$
 - d) $S \rightarrow A$
 $A \rightarrow BB$
 $B \rightarrow aB \mid b$
 - e) $S \rightarrow A$
 $A \rightarrow AAa \mid AAb \mid c$
8. Construct the LR(1) machine for the grammar introduced in Example 20.4.3. Is this grammar LR(1)?
9. Parse the following strings using the LR(1) parser and the grammar AE. Trace the actions of the parser using the format of Example 20.5.2. The deterministic LR(1) machine of AE is given in Example 20.5.1.
 - a) *b + b*
 - b) *(b)*
 - c) *b + +b*

deter-
Is the

Bibliographic Notes

LR grammars were introduced by Knuth [1965]. The number of states and transitions in the LR machine made the use of LR techniques impractical for parsers of computer languages. Korenjak [1969] and De Remer [1969, 1971] developed simplifications that eliminated these difficulties. The latter works introduced the SLR (simple LR) and LALR (lookahead LR) grammars. The relationships between the class of LR(*k*) grammars and other classes of grammars that can be deterministically parsed, including the LL(*k*) grammars, are presented in Aho and Ullman [1972, 1973].

APPENDIX I

Index of Notation

Symbol	Page	Interpretation
\in	8	is an element of
\notin	8	is not an element of
$\{x \mid \dots\}$	8	the set of x such that . . .
\mathbb{N}	8	the set of natural numbers
\emptyset	8	empty set
\subseteq	8	is a subset of
$\mathcal{P}(X)$	9	power set of X
\cup	9	union
\cap	9	intersection
$-$	9	$X - Y$: set difference
\bar{X}	9	complement
\times	11	$X \times Y$: Cartesian product
$[x, y]$	11	ordered pair
$f: X \rightarrow Y$	12	f is a function from X to Y
$f(x)$	12	value assigned to x by the function f
$f(x) \uparrow$	13	$f(x)$ is undefined
$f(x) \downarrow$	13	$f(x)$ is defined

Symbol	Page	Interpretation	Symbol
div	13	integer division	\Rightarrow_R
\equiv	15	equivalence relation	A_{opt}
$[]_e$	15	equivalence class	δ
card	16	cardinality	$L(M)$
s	24, 300	successor function	\vdash
$\sum_{i=n}^m$	31, 398	bounded summation	\models
\dashv	39, 395	proper subtraction	$\hat{\delta}$
λ	42	null string	$\lambda\text{-closure}$
Σ^*	42	set of strings over Σ	Γ
length	43	length of a string	B
uv	44	concatenation of u and v	lo
u^n	44	concatenation of u n times	x_L
u^R	45	reversal of u	\hat{x}_L
XY	47	concatenation of sets X and Y	\bar{i}
X^i	47	concatenation of X with itself i times	z
X^*	48	strings over X	e
X^+	48	nonnull strings over X	$p_i^{(k)}$
∞	48	infinity	id
\emptyset	50	regular expression for the empty set	$pred$
λ	50	regular expression for the null string	\circ
a	50	regular expression for the set $\{a\}$	$c_i^{(k)}$
\cup	50	regular expression union operation	$[x]$
\rightarrow	65, 69, 326	rule of a grammar	P
\Rightarrow	67, 69, 326	is derivable by one rule application	U
$\stackrel{*}{\Rightarrow}$	69, 326	is derivable from	L_H
$\stackrel{+}{\Rightarrow}$	69	is derivable by one or more rule applications	P
$\stackrel{n}{\Rightarrow}$	69	is derivable by n rule applications	!
$L(G)$	70, 326	language of the grammar G	$\prod_{i=0}^n$
$n_x(u)$	84	number of occurrences of x in u	$\mu z[p]$
$\stackrel{L}{\Rightarrow}$	91	leftmost rule application	$\stackrel{y}{\mu z}[p]$

Symbol	Page	Interpretation
\Rightarrow_R	91	rightmost rule application
A_{opt}	94, 631	occurrence of A is optional
δ	147, 163, 222, 256	transition function
$L(M)$	148, 163, 234, 260	language of the machine M
\vdash	149, 224, 258	yields by one transition
\vdash^*	149, 224, 258	yields by zero or more transitions
$\hat{\delta}$	151, 185	extended transition function
λ -closure	170	lambda closure function
Γ	222, 256	stack or tape alphabet
B	256	blank tape symbol
lo	286	lexicographical ordering
χ_L	298	characteristic function of language L
$\hat{\chi}_L$	298	partial characteristic function of language L
\bar{i}	299, 471	representation of i
z	300, 390	zero function
e	300	empty function
$p_i^{(k)}$	300, 390	k -variable projection function
id	301	identity function
$pred$	301	predecessor function
\circ	308	composition
$c_i^{(k)}$	311, 391	k -variable constant function
$\lfloor x \rfloor$	320	greatest integer less than or equal to x
P	343	decision problem
U	356	universal Turing machine
L_H	357, 365	language of the Halting Problem
\mathbb{P}	372	property of recursively enumerable languages
!	393	factorial
$\prod_{i=0}^n$	398	bounded product
$\mu z[p]$	400, 413	unbounded minimalization
$\mu^y z[p]$	401	bounded minimalization

Symbol	Page	Interpretation
<i>quo</i>	404	quotient function
<i>pn(i)</i>	405	<i>i</i> th prime function
<i>gn_k</i>	406	(<i>k</i> + 1)-variable Gödel numbering function
<i>dec(i, x)</i>	407	decoding function
<i>gn_f</i>	408	bounded Gödel numbering function
<i>tr_M</i>	417, 420	Turing machine trace function
\mathcal{P}	431, 468	class of polynomial languages
NP	431, 469	class of nondeterministically polynomial languages
$O(g)$	436	big oh of <i>g</i> , the order of the function <i>g</i>
$\Theta(g)$	438	big theta of <i>g</i>
$ i $	438	absolute value of <i>i</i>
tc_M	443	time complexity function
$[x]$	451	least integer greater than or equal to <i>x</i>
<i>rep(p)</i>	471	representation of problem instance <i>p</i>
\wedge	481	conjunction
\vee	481	disjunction
\neg	481	negation
L_{SAT}	483	language of the Satisfiability Problem
NPC	492	class of NP-complete languages
Co-NP	531	complement of NP
sc_M	532	space complexity function
<i>inf</i>	538	infimum, greatest lower bound
\mathcal{P} -Space	540	class of polynomial space languages
NP -Space	540	class of nondeterministic polynomial space languages
<i>g(G)</i>	556	graph of the grammar <i>G</i>
$\text{LA}(A)$	572	lookahead set of variable <i>A</i>
$\text{LA}(A \rightarrow w)$	572	lookahead set of the rule $A \rightarrow w$
<i>trunc_k</i>	575	length- <i>k</i> truncation function
$\text{FIRST}_k(u)$	576	FIRST_k set of the string <i>u</i>
$\text{FOLLOW}_k(A)$	577	FOLLOW_k set of the variable <i>A</i>
<i>shift</i>	599	shift function

APPENDIX II

The Greek Alphabet

Uppercase	Lowercase	Name
A	α	alpha
B	β	beta
Γ	γ	gamma
Δ	δ	delta
Ε	ϵ	epsilon
Z	ζ	zeta
H	η	eta
Θ	θ	theta
I	ι	iota
K	κ	kappa
Λ	λ	lambda
M	μ	mu
N	ν	nu
Ξ	ξ	xi
O	$\ο$	omicron
Π	π	pi
R	ρ	rho
Σ	σ	sigma
T	τ	tau
Υ	υ	upsilon
Φ	ϕ	phi
X	χ	chi
Ψ	ψ	psi
Ω	ω	omega

APPENDIX III

The ASCII Character Set

The American Standard Code for Information Interchange, more commonly referred to as the ASCII code, is a code that represents printable symbols and special functions using the binary representation of the numbers 0 to 127. Numbers 0 through 31 are control characters and the column labeled Name gives an abbreviation for the action associated with the character. For example, numbers 14 and 15 indicate that the printer should begin a new line (LF, line feed) or a new page (FF, form feed) when this character is encountered. Numbers 32 (a blank space) to 126 have become widely accepted as the standard encoding for text documents.

Code	Char	Name	Code	Char	Code	Char	Code	Char
0	^@	NUL	32		64	@	96	'
1	^A	SOH	33	!	65	A	97	a
2	^B	STX	34	"	66	B	98	b
3	^C	ETX	35	#	67	C	99	c
4	^D	EOT	36	\$	68	D	100	d
5	^E	ENQ	37	%	69	E	101	e
6	^F	ACK	38	&	70	F	102	f
7	^G	BEL	39	,	71	G	103	g
8	^H	BS	40	(72	H	104	h
9	^I	TAB	41)	73	I	105	i
10	^J	LF	42	*	74	J	106	j
11	^K	VT	43	+	75	K	107	k

630 Appendix III The ASCII Character Set

Code	Char	Name	Code	Char	Code	Char	Code	Char
12	^L	FF	44	,	76	L	108	I
13	^M	CR	45	-	77	M	109	m
14	^N	SO	46	.	78	N	110	n
15	^O	SI	47	/	79	O	111	o
16	^P	DLE	48	0	80	P	112	p
17	^Q	DC1	49	1	81	Q	113	q
18	^R	DC2	50	2	82	R	114	r
19	^S	DC3	51	3	83	S	115	s
20	^T	DC4	52	4	84	T	116	t
21	^U	NAK	53	5	85	U	117	u
22	^V	SYN	54	6	86	V	118	v
23	^W	ETB	55	7	87	W	119	w
24	^X	CAN	56	8	88	X	120	x
25	^Y	EM	57	9	89	Y	121	y
26	^Z	SUB	58	:	90	Z	122	z
27	^_	ESC	59	;	91	[123	{
28	^`	FS	60	<	92	\	124	
29	^]	GS	61	=	93]	125	}
30	^~	RS	62	>	94	^	126	~
31	^_	US	63	?	95	_	127	DEL

The programm
Sun Microsyst
programming
its introduc
applications.

The gramm
[2000]. The ru
with the exce
placing the sub
are needed, bu
context-free ru
rules; in one, t
example, $A \rightarrow$
subscripted wi
variable $\langle Comp$

1. $\langle Compilat$

Declaratio

2. $\langle ImportDe$

3. $\langle TypeDecl$

Char

l
m
n
o
p
q
r
s
t
u
v
w
x
y
z
{
|
}
~
DEL

APPENDIX IV

Backus-Naur Form Definition of Java

The programming language Java was developed under the direction of James Gosling at Sun Microsystems. Java was introduced in 1995 as a platform independent, object-oriented programming language particularly suitable for Internet and network applications. Since its introduction, Java has become one of the most commonly used languages for Internet applications.

The grammar for the language Java is derived from the BNF definition in Gosling et al. [2000]. The rules have been transformed into the standard context-free grammar notation, with the exception of retaining the designation of a terminal or a variable as optional by placing the subscript *opt* on the symbol. The use of *opt* reduces the number of rules that are needed, but rules with optional components can easily be transformed into equivalent context-free rules. A rule with a variable B_{opt} on the right-hand side can be replaced by two rules; in one, the occurrence of B_{opt} is replaced with B , and it is deleted in the other. For example, $A \rightarrow B_{opt}C$ is replaced by $A \rightarrow BC \mid C$. A rule with n occurrences of symbols subscripted with *opt* creates 2^n context-free rules. The start symbol of the grammar is the variable $\langle CompilationUnit \rangle$.

1. $\langle CompilationUnit \rangle \rightarrow \langle PackageDeclaration \rangle_{opt} \langle ImportDeclarations \rangle_{opt} \langle TypeDeclarations \rangle_{opt}$
Declarations
2. $\langle ImportDeclarations \rangle \rightarrow \langle ImportDeclarations \rangle \mid \langle ImportDeclarations \rangle \langle ImportDeclaration \rangle$
3. $\langle TypeDeclarations \rangle \rightarrow \langle TypeDeclaration \rangle \mid \langle TypeDeclarations \rangle \langle TypeDeclaration \rangle$

4. $\langle \text{PackageDeclaration} \rangle \rightarrow \text{package } \langle \text{PackageName} \rangle ;$
5. $\langle \text{ImportDeclaration} \rangle \rightarrow \langle \text{SingleTypeImportDeclaration} \rangle \mid \langle \text{TypeImportOnDemand} \rangle$
6. $\langle \text{SingleTypeImportDeclaration} \rangle \rightarrow \text{import } \langle \text{TypeName} \rangle ;$
7. $\langle \text{TypeImportOnDemandDeclaration} \rangle \rightarrow \text{import } \langle \text{PackageName} \rangle . * ;$
8. $\langle \text{TypeDeclaration} \rangle \rightarrow \langle \text{ClassDeclaration} \rangle \mid \langle \text{Declaration} \rangle ;$
9. $\langle \text{Type} \rangle \rightarrow \langle \text{PrimitiveType} \rangle \langle \text{ReferenceType} \rangle$
10. $\langle \text{PrimitiveType} \rangle \rightarrow \langle \text{NumericType} \rangle \text{ boolean}$
11. $\langle \text{NumericType} \rangle \rightarrow \langle \text{IntegralType} \rangle \mid \langle \text{FloatingPointType} \rangle$
12. $\langle \text{IntegralType} \rangle \rightarrow \text{byte} \mid \text{short} \mid \text{int} \mid \text{long} \mid \text{char}$
13. $\langle \text{FloatingPointType} \rangle \rightarrow \text{float} \mid \text{double}$
31. $\langle \text{VariableList} \rangle$
32. $\langle \text{VariableList} \rangle$
33. $\langle \text{VariableList} \rangle$
34. $\langle \text{FieldModifier} \rangle$
35. $\langle \text{FieldModifier} \rangle$
- Method Declaration
36. $\langle \text{MethodDeclaration} \rangle$
37. $\langle \text{MethodHeader} \rangle$
38. $\langle \text{ResultType} \rangle$
39. $\langle \text{MethodDeclaration} \rangle$
40. $\langle \text{FormalParameter} \rangle$
41. $\langle \text{FormalParameter} \rangle$
42. $\langle \text{MethodModifier} \rangle$
43. $\langle \text{MethodModifier} \rangle$
44. $\langle \text{Throws} \rangle$
45. $\langle \text{ClassType} \rangle$
46. $\langle \text{MethodBody} \rangle$
- Constructor Declaration
47. $\langle \text{ConstructorDeclaration} \rangle$
48. $\langle \text{ConstructorDeclaration} \rangle$
49. $\langle \text{ConstructorDeclaration} \rangle$
50. $\langle \text{ConstructorDeclaration} \rangle$
51. $\langle \text{ConstructorDeclaration} \rangle$
52. $\langle \text{ExplicitConstructor} \rangle$

Reference Types and Values

14. $\langle \text{ReferenceType} \rangle \rightarrow \langle \text{ClassOrInterfaceType} \rangle \mid \langle \text{ArrayType} \rangle$
15. $\langle \text{ClassOrInterfaceType} \rangle \rightarrow \langle \text{ClassType} \rangle \mid \langle \text{InterfaceType} \rangle$
16. $\langle \text{ClassType} \rangle \rightarrow \langle \text{TypeName} \rangle$
17. $\langle \text{InterfaceType} \rangle \rightarrow \langle \text{TypeName} \rangle$
18. $\langle \text{ArrayType} \rangle \rightarrow \langle \text{Type} \rangle []$

Class Declarations

19. $\langle \text{ClassDeclaration} \rangle \rightarrow \langle \text{ClassModifier} \rangle_{\text{opt}} \text{ class } \langle \text{Identifier} \rangle \langle \text{Super} \rangle_{\text{opt}} \langle \text{Interfaces} \rangle_{\text{opt}} \langle \text{Classbody} \rangle$
20. $\langle \text{ClassModifiers} \rangle \rightarrow \langle \text{ClassModifier} \rangle \mid \langle \text{ClassModifiers} \rangle \langle \text{ClassModifier} \rangle$
21. $\langle \text{ClassModifier} \rangle \rightarrow \text{public} \mid \text{abstract} \mid \text{final}$
22. $\langle \text{Super} \rangle \rightarrow \text{extends } \langle \text{ClassType} \rangle$
23. $\langle \text{Interfaces} \rangle \rightarrow \text{implements } \langle \text{InterfaceTypeList} \rangle$
24. $\langle \text{InterfaceTypeList} \rangle \rightarrow \langle \text{InterfaceType} \rangle \mid \langle \text{InterfaceTypeList} \rangle \langle \text{InterfaceType} \rangle$
25. $\langle \text{ClassBody} \rangle \rightarrow \{ \langle \text{ClassBodyDeclarations} \rangle_{\text{opt}} \}$
26. $\langle \text{ClassBodyDeclarations} \rangle \rightarrow \langle \text{ClassBodyDeclaration} \rangle \mid \langle \text{ClassBodyDeclaration} \rangle \langle \text{ClassBodyDeclarations} \rangle$
27. $\langle \text{ClassBodyDeclaration} \rangle \rightarrow \langle \text{ClassMemberDeclaration} \rangle \mid \langle \text{StaticInitializer} \rangle \mid \langle \text{ConstructorDeclarations} \rangle$
28. $\langle \text{ClassMemberDeclaration} \rangle \rightarrow \langle \text{FieldDeclaration} \rangle \mid \langle \text{MethodDeclaration} \rangle$

Field Declarations

29. $\langle \text{FieldDeclaration} \rangle \rightarrow \langle \text{FieldModifiers} \rangle_{\text{opt}} \langle \text{Type} \rangle \langle \text{VariableDeclarators} \rangle ;$
30. $\langle \text{VariableDeclarators} \rangle \rightarrow \langle \text{VariableDeclarator} \rangle \mid \langle \text{VariableDeclarators} \rangle , \langle \text{VariableDeclarator} \rangle$

Interface Declaration

31. $\langle \text{VariableList} \rangle$
32. $\langle \text{VariableList} \rangle$
33. $\langle \text{VariableList} \rangle$
34. $\langle \text{FieldModifier} \rangle$
35. $\langle \text{FieldModifier} \rangle$
- Method Declaration
36. $\langle \text{MethodDeclaration} \rangle$
37. $\langle \text{MethodHeader} \rangle$
38. $\langle \text{ResultType} \rangle$
39. $\langle \text{MethodDeclaration} \rangle$
40. $\langle \text{FormalParameter} \rangle$
41. $\langle \text{FormalParameter} \rangle$
42. $\langle \text{MethodModifier} \rangle$
43. $\langle \text{MethodModifier} \rangle$
44. $\langle \text{Throws} \rangle$
45. $\langle \text{ClassType} \rangle$
46. $\langle \text{MethodBody} \rangle$
- Constructor Declaration
47. $\langle \text{ConstructorDeclaration} \rangle$
48. $\langle \text{ConstructorDeclaration} \rangle$
49. $\langle \text{ConstructorDeclaration} \rangle$
50. $\langle \text{ConstructorDeclaration} \rangle$
51. $\langle \text{ConstructorDeclaration} \rangle$
52. $\langle \text{ExplicitConstructor} \rangle$

- Demand)*
31. $\langle \text{VariableDeclarator} \rangle \rightarrow \langle \text{VariableDeclaratorID} \rangle | \langle \text{VariableDeclaratorsID} \rangle = \langle \text{VariableInitializer} \rangle$
 32. $\langle \text{VariableDeclaratorID} \rangle \rightarrow \langle \text{Identifier} \rangle | \langle \text{VariableDeclaratorsID} \rangle []$
 33. $\langle \text{VariableInitializer} \rangle \rightarrow \langle \text{Expression} \rangle | \langle \text{ArrayInitializer} \rangle$
 34. $\langle \text{FieldModifiers} \rangle \rightarrow \langle \text{FieldModifier} \rangle | \langle \text{FieldModifiers} \rangle \langle \text{FieldModifier} \rangle$
 35. $\langle \text{FieldModifier} \rangle \rightarrow \text{public} | \text{protected} | \text{private} | \text{final} | \text{static} | \text{transient} | \text{volatile}$

Method Declarations

36. $\langle \text{MethodDeclaration} \rangle \rightarrow \langle \text{MethodHeader} \rangle \langle \text{MethodBody} \rangle$
37. $\langle \text{MethodHeader} \rangle \rightarrow \langle \text{MethodModifiers} \rangle_{\text{opt}} \langle \text{ResultType} \rangle \langle \text{MethodDeclarator} \rangle \langle \text{Throws} \rangle_{\text{opt}}$
38. $\langle \text{ResultType} \rangle \rightarrow \langle \text{Type} \rangle | \text{void}$
39. $\langle \text{MethodDeclarator} \rangle \rightarrow \langle \text{Identifier} \rangle (\langle \text{FormalParameterList} \rangle_{\text{opt}}) \langle \text{MethodDeclarator} \rangle []$
40. $\langle \text{FormalParameterList} \rangle \rightarrow \langle \text{FormalParameter} \rangle | \langle \text{FormalParameterList} \rangle \langle \text{FormalParameter} \rangle$
41. $\langle \text{FormalParameter} \rangle \rightarrow \langle \text{Type} \rangle \langle \text{VariableDeclaratorId} \rangle$
42. $\langle \text{MethodModifiers} \rangle \rightarrow \langle \text{MethodModifier} \rangle | \langle \text{MethodModifiers} \rangle \langle \text{MethodModifiers} \rangle$
43. $\langle \text{MethodModifier} \rangle \rightarrow \text{public} | \text{protected} | \text{private} | \text{abstract} | \text{final} | \text{static} | \text{synchronized} | \text{native}$
44. $\langle \text{Throws} \rangle \rightarrow \text{throws} \langle \text{ClassTypeList} \rangle$
45. $\langle \text{ClassTypeList} \rangle \rightarrow \langle \text{ClassType} \rangle | \langle \text{ClassTypeList} \rangle , \langle \text{ClassType} \rangle$
46. $\langle \text{MethodBody} \rangle \rightarrow \langle \text{Block} \rangle | ;$

Constructor Declarations

47. $\langle \text{ConstructorDeclaration} \rangle \rightarrow \langle \text{ConstructorModifiers} \rangle_{\text{opt}} \langle \text{ConstructorDeclarator} \rangle \langle \text{Throws} \rangle_{\text{opt}} \langle \text{ConstructorBody} \rangle$
48. $\langle \text{ConstructorDeclarator} \rangle \rightarrow \langle \text{SimpleTypeName} \rangle (\langle \text{FormalParameterList} \rangle_{\text{opt}})$
49. $\langle \text{ConstructorModifiers} \rangle \rightarrow \langle \text{ConstructorModifier} \rangle | \langle \text{ConstructorModifiers} \rangle \langle \text{ConstructorModifier} \rangle$
50. $\langle \text{ConstructorModifier} \rangle \rightarrow \text{public} | \text{private} | \text{protected}$
51. $\langle \text{ConstructorBody} \rangle \rightarrow \{ \langle \text{ExplicitConstructorInvocation} \rangle_{\text{opt}} \langle \text{BlockStatements} \rangle_{\text{opt}} \}$
52. $\langle \text{ExplicitConstructorInvocation} \rangle \rightarrow \text{this} (\langle \text{ArgumentList} \rangle_{\text{opt}}) ; \text{mid} \text{super} (\langle \text{ArgumentList} \rangle_{\text{opt}}) ;$

Interface Declarations

53. $\langle \text{InterfaceDeclaration} \rangle \rightarrow \langle \text{InterfaceModifiers} \rangle_{\text{opt}} \text{interface} \langle \text{Identifier} \rangle \langle \text{ExtendsInterface} \rangle_{\text{opt}} \langle \text{InterfaceBody} \rangle$

54. $\langle \text{InterfaceModifiers} \rangle \rightarrow \langle \text{InterfaceModifier} \rangle \mid \langle \text{InterfaceModifiers} \rangle \langle \text{InterfaceModifier} \rangle$
 55. $\langle \text{InterfaceModifier} \rangle \rightarrow \text{public} \mid \text{abstract}$
 56. $\langle \text{ExtendsInterfaces} \rangle \rightarrow \text{extends } \langle \text{InterfaceType} \rangle \mid \langle \text{ExtendsInterfaces} \rangle , \langle \text{InterfaceType} \rangle$
 57. $\langle \text{InterfaceBody} \rangle \rightarrow \{ \langle \text{InterfaceMemberDeclaration} \rangle_{\text{opt}} \}$
 58. $\langle \text{InterfaceMemberDeclarations} \rangle \rightarrow \langle \text{InterfaceMemberDeclaration} \rangle \mid \langle \text{InterfaceMemberDeclarations} \rangle \langle \text{InterfaceMemberDeclaration} \rangle$
 59. $\langle \text{InterfaceMemberDeclaration} \rangle \rightarrow \langle \text{ConstantDeclaration} \rangle \mid \langle \text{AbstractMethodDeclaration} \rangle$

74. $\langle \text{StatementN} \rangle$ **Constant Declarations**

60. $\langle \text{ConstantDeclaration} \rangle \rightarrow \langle \text{ConstantModifiers} \rangle_{\text{opt}} \langle \text{Type} \rangle \langle \text{VariableDeclarator} \rangle$
 61. $\langle \text{ConstantModifiers} \rangle \rightarrow \text{public} \mid \text{static} \mid \text{final}$

Empty, Label

76. $\langle \text{EmptyStatement} \rangle$ 77. $\langle \text{LabeledStatement} \rangle$ 78. $\langle \text{LabeledStatement} \rangle$ 79. $\langle \text{ExpressionStatement} \rangle$ 80. $\langle \text{StatementExecution} \rangle$ **Abstract Method Declarations**

62. $\langle \text{AbstractMethodDeclaration} \rangle \rightarrow \langle \text{AbstractMethodModifiers} \rangle_{\text{opt}} \langle \text{ResultType} \rangle \langle \text{MethodDeclarator} \rangle \langle \text{Throws} \rangle_{\text{opt}}$
 63. $\langle \text{AbstractMethodModifiers} \rangle \rightarrow \langle \text{AbstractMethodModifier} \rangle \mid \langle \text{AbstractMethodModifiers} \rangle \langle \text{AbstractMethodModifier} \rangle$
 64. $\langle \text{AbstractMethodModifier} \rangle \rightarrow \text{public} \mid \text{abstract}$

Array Initializers

65. $\langle \text{ArrayInitializer} \rangle \rightarrow \{ \langle \text{VariableInitializers} \rangle_{\text{opt}} ,_{\text{opt}} \}$
 66. $\langle \text{VariableInitializers} \rangle \rightarrow \langle \text{VariableInitializer} \rangle \mid \langle \text{VariableInitializers} \rangle \langle \text{VariableInitializers} \rangle$

If Statement

81. $\langle \text{IfThenStatement} \rangle$ 82. $\langle \text{IfThenElseStatement} \rangle$ 83. $\langle \text{IfThenElseStatement} \rangle$ **Blocks and Local Variable Declaration**

67. $\langle \text{Block} \rangle \rightarrow \{ \langle \text{BlockStatements} \rangle_{\text{opt}} \}$
 68. $\langle \text{BlockStatements} \rangle \rightarrow \langle \text{BlockStatement} \rangle \mid \langle \text{BlockStatements} \rangle \langle \text{BlockStatement} \rangle$
 69. $\langle \text{BlockStatement} \rangle \rightarrow \langle \text{LocalVariableDeclarationStatement} \rangle \mid \langle \text{Statement} \rangle$
 70. $\langle \text{StaticInitializer} \rangle \rightarrow \text{static } \langle \text{Block} \rangle$
 71. $\langle \text{LocalVariableDeclarationStatement} \rangle \rightarrow \langle \text{LocalVariableDeclaration} \rangle$
 72. $\langle \text{LocalVariableDeclaration} \rangle \rightarrow \langle \text{Type} \rangle \langle \text{VariableDeclarators} \rangle$

Switch Statement

84. $\langle \text{SwitchStatement} \rangle$ 85. $\langle \text{SwitchBlock} \rangle$ 86. $\langle \text{SwitchBlock} \rangle$ 87. $\langle \text{SwitchBlock} \rangle$ 88. $\langle \text{SwitchLabels} \rangle$ 89. $\langle \text{SwitchLabel} \rangle$ **Statements**

73. $\langle \text{Statement} \rangle \rightarrow \langle \text{StatementWithoutTrailingSubstatement} \rangle \mid \langle \text{LabeledStatement} \rangle \mid \langle \text{IfThenStatement} \rangle \mid \langle \text{IfThenElseStatement} \rangle \mid \langle \text{WhileStatement} \rangle \mid \langle \text{ForStatement} \rangle$

While, Do, an

90. $\langle \text{WhileStatement} \rangle$

74. $\langle \text{StatementNoShortIf} \rangle \rightarrow \langle \text{StatementWithoutTrailingSubstatement} \rangle \mid \langle \text{LabeledStatementNoShortIf} \rangle \mid \langle \text{IfThenStatementNoShortIf} \rangle \mid \langle \text{IfThenElseStatementNoShortIf} \rangle \mid \langle \text{ForStatementNoShortIf} \rangle$
75. $\langle \text{StatementWithoutTrailingSubstatement} \rangle \rightarrow \langle \text{Block} \rangle \langle \text{EmptyStatement} \rangle \mid \langle \text{ExpressionStatement} \rangle \mid \langle \text{SwitchStatement} \rangle \mid \langle \text{DoStatement} \rangle \mid \langle \text{BreakStatement} \rangle \mid \langle \text{ContinueStatement} \rangle \mid \langle \text{ReturnStatement} \rangle \mid \langle \text{SynchronizedStatement} \rangle \mid \langle \text{ThrowStatement} \rangle \mid \langle \text{TryStatement} \rangle$

Empty, Labeled, and Expression Statements

76. $\langle \text{EmptyStatement} \rangle \rightarrow ;$
77. $\langle \text{LabeledStatement} \rangle \rightarrow \langle \text{Identifier} \rangle : \langle \text{Statement} \rangle$
78. $\langle \text{LabeledStatementNoShortIf} \rangle \rightarrow \langle \text{Identifier} \rangle : \langle \text{StatementNoShortIf} \rangle$
79. $\langle \text{ExpressionStatement} \rangle \rightarrow \langle \text{StatementExpression} \rangle ;$
80. $\langle \text{StatementExpression} \rangle \rightarrow \langle \text{Assignment} \rangle \mid \langle \text{PreincrementExpression} \rangle \mid \langle \text{PredecrementExpression} \rangle \mid \langle \text{PostincrementExpression} \rangle \mid \langle \text{PostdecrementExpression} \rangle \mid \langle \text{MethodInvocation} \rangle \mid \langle \text{ClassInstanceCreationExpression} \rangle$

If Statements

81. $\langle \text{IfThenStatement} \rangle \rightarrow \text{if} (\langle \text{Expression} \rangle) \langle \text{Statement} \rangle$
82. $\langle \text{IfThenElseStatement} \rangle \rightarrow \text{if} (\langle \text{Expression} \rangle) \langle \text{StatementNoShortIf} \rangle \text{else} \langle \text{Statement} \rangle$
83. $\langle \text{IfThenElseStatementNoShortIf} \rangle \rightarrow \text{if} (\langle \text{Expression} \rangle) \langle \text{StatementNoShortIf} \rangle \text{else} \langle \text{StatementNoShortIf} \rangle$

Switch Statement

84. $\langle \text{SwitchStatement} \rangle \rightarrow \text{switch} (\langle \text{Expression} \rangle) \langle \text{SwitchBlock} \rangle$
85. $\langle \text{SwitchBlock} \rangle \rightarrow \{ \langle \text{SwitchBlockStatementGroups} \rangle_{opt} \langle \text{SwitchLabel} \rangle_{opt} \}$
86. $\langle \text{SwitchBlockStatementGroups} \rangle \rightarrow \langle \text{SwitchBlockStatementGroup} \rangle \mid \langle \text{SwitchBlockStatementGroups} \rangle \langle \text{SwitchBlockStatementGroups} \rangle$
87. $\langle \text{SwitchBlockStatementGroup} \rangle \rightarrow \langle \text{SwitchLabels} \rangle \langle \text{BlockStatements} \rangle$
88. $\langle \text{SwitchLabels} \rangle \rightarrow \langle \text{SwitchLabel} \rangle \mid \langle \text{SwitchLabels} \rangle \langle \text{SwitchLabel} \rangle$
89. $\langle \text{SwitchLabel} \rangle \rightarrow \text{case} \langle \text{ConstantExpression} \rangle : \mid \text{default} :$

While, Do, and For Statements

90. $\langle \text{WhileStatement} \rangle \rightarrow \text{while} (\langle \text{Expression} \rangle) \langle \text{Statement} \rangle$

91. $\langle \text{WhileStatementNoShortIf} \rangle \rightarrow \text{while} (\langle \text{Expression} \rangle) \langle \text{StatementNoShortIf} \rangle$ 116. $\langle \text{MethodDecl} \rangle$
92. $\langle \text{DoStatement} \rangle \rightarrow \text{do} \langle \text{Statement} \rangle \text{while} (\langle \text{Expression} \rangle);$ 117. $\langle \text{ArrayAccess} \rangle$
93. $\langle \text{ForStatement} \rangle \rightarrow \text{for} (\langle \text{ForInit} \rangle_{\text{opt}}; \langle \text{Expression} \rangle_{\text{opt}}; \langle \text{ForUpdate} \rangle_{\text{opt}}) \langle \text{Statement} \rangle$
94. $\langle \text{ForStatementNoShortIf} \rangle \rightarrow \text{for} (\langle \text{ForInit} \rangle_{\text{opt}}; \langle \text{Expression} \rangle_{\text{opt}}; \langle \text{ForUpdate} \rangle_{\text{opt}}) \langle \text{StatementNoShortIf} \rangle$ 118. $\langle \text{Expression} \rangle$
95. $\langle \text{ForInit} \rangle \rightarrow \langle \text{StatementExpressionList} \rangle \mid \langle \text{LocalVariableDeclaration} \rangle$ 119. $\langle \text{Constant} \rangle$
96. $\langle \text{ForUpdate} \rangle \rightarrow \langle \text{StatementExpressionList} \rangle$
97. $\langle \text{StatementExpressionList} \rangle \rightarrow \langle \text{StatementExpression} \rangle \mid \langle \text{StatementExpressionList} \rangle, \langle \text{StatementExpression} \rangle$
- Break, Continue, Return, Throw, Synchronized, and Try Statements**
98. $\langle \text{BreakStatement} \rangle \rightarrow \text{break} \langle \text{Identifier} \rangle_{\text{opt}};$ 120. $\langle \text{Assignment} \rangle$
99. $\langle \text{ContinueStatement} \rangle \rightarrow \text{continue} \langle \text{Identifier} \rangle_{\text{opt}};$ 121. $\langle \text{Assignment} \rangle$
100. $\langle \text{ReturnStatement} \rangle \rightarrow \text{return} \langle \text{Expression} \rangle_{\text{opt}};$ 122. $\langle \text{LeftHandSide} \rangle$
101. $\langle \text{ThrowStatement} \rangle \rightarrow \text{throw} \langle \text{Expression} \rangle;$ 123. $\langle \text{Assignment} \rangle$
102. $\langle \text{SynchronizedStatement} \rangle \rightarrow \text{synchronized} (\langle \text{Expression} \rangle) \langle \text{Block} \rangle$
103. $\langle \text{TryStatement} \rangle \rightarrow \text{try} \langle \text{Block} \rangle \langle \text{Catches} \rangle \mid \text{try} \langle \text{Block} \rangle \langle \text{Catches} \rangle_{\text{opt}} \langle \text{Finally} \rangle$ **Postfix Expressions**
104. $\langle \text{Catches} \rangle \rightarrow \langle \text{CatchClause} \rangle \mid \langle \text{Catches} \rangle \langle \text{CatchClause} \rangle$ 124. $\langle \text{PostfixExpression} \rangle$
105. $\langle \text{CatchClause} \rangle \rightarrow \text{catch} (\langle \text{FormalParamenter} \rangle) \langle \text{Block} \rangle$ 125. $\langle \text{PostIncremente} \rangle$
106. $\langle \text{Finally} \rangle \rightarrow \text{finally} \langle \text{Block} \rangle$ 126. $\langle \text{PostDecremente} \rangle$
- Creation and Access Expressions**
107. $\langle \text{Primary} \rangle \rightarrow \langle \text{PrimaryNoNewArray} \rangle \mid \langle \text{ArrayCreationExpression} \rangle$ 127. $\langle \text{UnaryExpression} \rangle$
108. $\langle \text{PrimaryNoNewArray} \rangle \rightarrow \langle \text{Literal} \rangle \mid \text{this} \mid (\langle \text{Expression} \rangle) \mid \langle \text{ClassInstanceCreationExpression} \rangle \mid \langle \text{FieldAccess} \rangle \mid \langle \text{MethodInvocation} \rangle \mid \langle \text{ArrayAccess} \rangle$ 128. $\langle \text{PreIncrement} \rangle$
109. $\langle \text{ClassInstanceCreationExpression} \rangle \rightarrow \text{new} \langle \text{ClassType} \rangle (\langle \text{ArgumentList} \rangle_{\text{opt}})$ 129. $\langle \text{PreDecrement} \rangle$
110. $\langle \text{ArgumentList} \rangle \rightarrow \langle \text{Expression} \rangle \mid \langle \text{ArgumentList} \rangle, \langle \text{Expression} \rangle$ 130. $\langle \text{UnaryExpression} \rangle$
111. $\langle \text{ArrayCreationExpression} \rangle \rightarrow \text{new} \langle \text{PrimitiveType} \rangle \langle \text{DimExprs} \rangle \langle \text{Dims} \rangle_{\text{opt}} \mid \text{new} \langle \text{TypeName} \rangle \langle \text{DimExprs} \rangle \langle \text{Dims} \rangle_{\text{opt}}$ 131. $\langle \text{CastExpression} \rangle$
112. $\langle \text{DimExprs} \rangle \rightarrow \langle \text{DimExpr} \rangle \mid \langle \text{DimExprs} \rangle \langle \text{DimExpr} \rangle$
113. $\langle \text{DimExpr} \rangle \rightarrow [\langle \text{Expression} \rangle]$
114. $\langle \text{Dims} \rangle \rightarrow [] \mid \langle \text{Dims} \rangle []$
115. $\langle \text{FieldAccess} \rangle \rightarrow \langle \text{Primary} \rangle . \langle \text{Identifier} \rangle \mid \text{super} . \langle \text{Identifier} \rangle$ **Operators**
132. $\langle \text{Multiplicand} \rangle$

-)
statement
*ate*_{opt})
ssion)
116. $\langle \text{MethodInvocation} \rangle \rightarrow \langle \text{MethodName} \rangle (\langle \text{ArgumentList} \rangle_{\text{opt}}) |$
 $\langle \text{Primary} \rangle . \langle \text{Identifier} \rangle (\langle \text{ArgumentList} \rangle_{\text{opt}}) |$
 $\text{super} . \langle \text{Identifier} \rangle (\langle \text{ArgumentList} \rangle_{\text{opt}})$
 117. $\langle \text{ArrayAccess} \rangle \rightarrow \langle \text{ExpressionName} \rangle [\langle \text{Expression} \rangle] |$
 $\langle \text{PrimaryNoNewArray} \rangle [\langle \text{Expression} \rangle]$

Expressions

118. $\langle \text{Expression} \rangle \rightarrow \langle \text{AssignmentExpression} \rangle$
119. $\langle \text{ConstantExpression} \rangle \rightarrow \langle \text{Expression} \rangle$

Assignment Operators

120. $\langle \text{AssignmentExpression} \rangle \rightarrow \langle \text{ConditionalExpression} \rangle | \langle \text{Assignment} \rangle$
121. $\langle \text{Assignment} \rangle \rightarrow \langle \text{LeftHandSide} \rangle \langle \text{AssignmentOperator} \rangle \langle \text{AssignmentExpression} \rangle$
122. $\langle \text{LeftHandSide} \rangle \rightarrow \langle \text{ExpressionName} \rangle | \langle \text{FieldAccess} \rangle | \langle \text{ArrayAccess} \rangle$
123. $\langle \text{AssignmentOperator} \rangle \rightarrow = | *= | / = | \% = | += | -= | <=> |$
 $>= | >>= | \&= | = | |=$

Postfix Expressions

124. $\langle \text{PostfixExpression} \rangle \rightarrow \langle \text{Primary} \rangle | \langle \text{ExpressionName} \rangle |$
 $\langle \text{PostIncrementExpression} \rangle | \langle \text{PostDecrementExpression} \rangle$
125. $\langle \text{PostIncrementExpression} \rangle \rightarrow \langle \text{PostfixExpression} \rangle ++$
126. $\langle \text{PostDecrementExpression} \rangle \rightarrow \langle \text{PostfixExpression} \rangle --$

Unary Operators

127. $\langle \text{UnaryExpression} \rangle \rightarrow \langle \text{PreIncrementExpression} \rangle | \langle \text{PreDecrementExpression} \rangle |$
 $+ \langle \text{UnaryExpression} \rangle | - \langle \text{UnaryExpression} \rangle |$
 $\langle \text{UnaryExpressionNotPlusMinus} \rangle$
128. $\langle \text{PreIncrementExpression} \rangle \rightarrow ++ \langle \text{UnaryExpression} \rangle$
129. $\langle \text{PreDecrementExpression} \rangle \rightarrow -- \langle \text{UnaryExpression} \rangle$
130. $\langle \text{UnaryExpressionNotPlusMinus} \rangle \rightarrow \langle \text{PostfixExpression} \rangle | \langle \text{UnaryExpression} \rangle |$
 $! \langle \text{UnaryExpression} \rangle | \langle \text{CastExpression} \rangle$
131. $\langle \text{CastExpression} \rangle \rightarrow (\langle \text{PrimitiveType} \rangle \langle \text{Dims} \rangle_{\text{opt}}) \langle \text{UnaryExpression} \rangle |$
 $(\langle \text{PrimitiveType} \rangle) \langle \text{UnaryExpressionNotPlusMinus} \rangle$

Operators

132. $\langle \text{MultiplicativeExpression} \rangle \rightarrow \langle \text{UnaryExpression} \rangle |$
 $\langle \text{MultiplicativeExpression} \rangle * \langle \text{UnaryExpression} \rangle |$
 $\langle \text{MultiplicativeExpression} \rangle / \langle \text{UnaryExpression} \rangle |$
 $\langle \text{MultiplicativeExpression} \rangle \% \langle \text{UnaryExpression} \rangle$

133. $\langle \text{AdditiveExpression} \rangle \rightarrow \langle \text{MultiplicativeExpression} \rangle \mid \langle \text{AdditiveExpression} \rangle + \langle \text{MultiplicativeExpression} \rangle \mid \langle \text{AdditiveExpression} \rangle - \langle \text{MultiplicativeExpression} \rangle$
134. $\langle \text{ShiftExpression} \rangle \rightarrow \langle \text{AdditiveExpression} \rangle \mid \langle \text{ShiftExpression} \rangle << \langle \text{AdditiveExpression} \rangle \mid \langle \text{ShiftExpression} \rangle >> \langle \text{AdditiveExpression} \rangle \mid \langle \text{ShiftExpression} \rangle >>> \langle \text{AdditiveExpression} \rangle$
135. $\langle \text{RelationalExpression} \rangle \rightarrow \langle \text{ShiftExpression} \rangle \mid \langle \text{RelationalExpression} \rangle < \langle \text{ShiftExpression} \rangle \mid \langle \text{RelationalExpression} \rangle > \langle \text{ShiftExpression} \rangle \mid \langle \text{RelationalExpression} \rangle <= \langle \text{ShiftExpression} \rangle \mid \langle \text{RelationalExpression} \rangle >= \langle \text{ShiftExpression} \rangle \mid \langle \text{RelationalExpression} \rangle \text{ instanceof } \langle \text{ReferenceType} \rangle$
136. $\langle \text{EqualityExpression} \rangle \rightarrow \langle \text{RelationalExpression} \rangle \mid \langle \text{RelationalExpression} \rangle == \langle \text{RelationalExpression} \rangle \mid \langle \text{RelationalExpression} \rangle != \langle \text{RelationalExpression} \rangle$
137. $\langle \text{AndExpression} \rangle \rightarrow \langle \text{EqualityExpression} \rangle \mid \langle \text{AndExpression} \rangle \& \langle \text{EqualityExpression} \rangle$
138. $\langle \text{ExclusiveOrExpression} \rangle \rightarrow \langle \text{EqualityExpression} \rangle \mid \langle \text{ExclusiveOrExpression} \rangle \langle \text{AndExpression} \rangle$
139. $\langle \text{InclusiveOrExpression} \rangle \rightarrow \langle \text{ExclusiveOrExpression} \rangle \mid \langle \text{InclusiveOrExpression} \rangle \mid \langle \text{AndExpression} \rangle$
140. $\langle \text{ConditionalAndExpression} \rangle \rightarrow \langle \text{InclusiveOrExpression} \rangle \mid \langle \text{ConditionalAndExpression} \rangle \&\& \langle \text{InclusiveOrExpression} \rangle$
141. $\langle \text{ConditionalOrExpression} \rangle \rightarrow \langle \text{ConditionalAndExpression} \rangle \mid \langle \text{ConditionalOrExpression} \rangle \parallel \langle \text{ConditionalAndExpression} \rangle$
142. $\langle \text{ConditionalExpression} \rangle \rightarrow \langle \text{ConditionalOrExpression} \rangle \mid \langle \text{ConditionalOrExpression} \rangle ? \langle \text{Expression} \rangle : \langle \text{ConditionalExpression} \rangle$

Literals

143. $\langle \text{Literal} \rangle \rightarrow \langle \text{IntegerLiteral} \rangle \mid \langle \text{FloatingPointLiteral} \rangle \mid \langle \text{BooleanLiteral} \rangle \mid \langle \text{CharacterLiteral} \rangle \mid \langle \text{StringLiteral} \rangle \mid \langle \text{NullLiteral} \rangle$
144. $\langle \text{IntegerLiteral} \rangle \rightarrow \langle \text{DecimalIntegerLiteral} \rangle \mid \langle \text{HexIntegerLiteral} \rangle \mid \langle \text{OctalIntegerLiteral} \rangle$
145. $\langle \text{DecimalIntegerLiteral} \rangle \rightarrow \langle \text{DecimalNumeral} \rangle \langle \text{IntegerTypeSuffix} \rangle_{\text{opt}}$
146. $\langle \text{HexIntegerLiteral} \rangle \rightarrow \langle \text{HexNumeral} \rangle \langle \text{IntegerTypeSuffix} \rangle_{\text{opt}}$
147. $\langle \text{OctalIntegerLiteral} \rangle \rightarrow \langle \text{OctalNumeral} \rangle \langle \text{IntegerTypeSuffix} \rangle_{\text{opt}}$

148. $\langle \text{OctalIntegerLiteral} \rangle \rightarrow \langle \text{OctalNumeral} \rangle \langle \text{IntegerTypeSuffix} \rangle_{\text{opt}}$
149. $\langle \text{IntegerType} \rangle \rightarrow \langle \text{SignedIntegerType} \rangle \mid \langle \text{UnsignedIntegerType} \rangle$
150. $\langle \text{DecimalIntegerType} \rangle \rightarrow \langle \text{SignedIntegerType} \rangle$
151. $\langle \text{Digits} \rangle \rightarrow \langle \text{NonZeroDigit} \rangle \langle \text{Digits} \rangle \mid \langle \text{NonZeroDigit} \rangle$
152. $\langle \text{Digit} \rangle \rightarrow \langle \text{Zero} \rangle \mid \langle \text{NonZeroDigit} \rangle$
153. $\langle \text{NonZeroDigit} \rangle \rightarrow \langle \text{Digit} \rangle$
154. $\langle \text{HexNumber} \rangle \rightarrow \langle \text{HexDigit} \rangle \langle \text{HexNumber} \rangle \mid \langle \text{HexDigit} \rangle$
155. $\langle \text{HexDigit} \rangle \rightarrow \langle \text{HexDigitA} \rangle \mid \langle \text{HexDigitB} \rangle \mid \langle \text{HexDigitC} \rangle \mid \langle \text{HexDigitD} \rangle \mid \langle \text{HexDigitE} \rangle \mid \langle \text{HexDigitF} \rangle$
156. $\langle \text{OctalNumber} \rangle \rightarrow \langle \text{OctalDigit} \rangle \langle \text{OctalNumber} \rangle \mid \langle \text{OctalDigit} \rangle$
157. $\langle \text{OctalDigit} \rangle \rightarrow \langle \text{OctalDigitA} \rangle \mid \langle \text{OctalDigitB} \rangle \mid \langle \text{OctalDigitC} \rangle \mid \langle \text{OctalDigitD} \rangle \mid \langle \text{OctalDigitE} \rangle \mid \langle \text{OctalDigitF} \rangle$
158. $\langle \text{FloatingPointNumber} \rangle \rightarrow \langle \text{Sign} \rangle \langle \text{Exponent} \rangle \langle \text{Fraction} \rangle \langle \text{Sign} \rangle \langle \text{Fraction} \rangle$
159. $\langle \text{Exponent} \rangle \rightarrow \langle \text{Sign} \rangle \langle \text{Digits} \rangle$
160. $\langle \text{Exponent} \rangle \rightarrow \langle \text{Sign} \rangle \langle \text{Digit} \rangle \langle \text{Digits} \rangle$
161. $\langle \text{SignedIntegerType} \rangle \rightarrow \langle \text{Signed} \rangle \langle \text{IntegerType} \rangle$
162. $\langle \text{Sign} \rangle \rightarrow \langle \text{Plus} \rangle \mid \langle \text{Minus} \rangle$
163. $\langle \text{FloatType} \rangle \rightarrow \langle \text{Sign} \rangle \langle \text{Exponent} \rangle$
164. $\langle \text{BooleanLiteral} \rangle \rightarrow \langle \text{True} \rangle \mid \langle \text{False} \rangle$
165. $\langle \text{CharacterLiteral} \rangle \rightarrow \langle \text{Character} \rangle$
166. $\langle \text{StringLiteral} \rangle \rightarrow \langle \text{String} \rangle$
167. $\langle \text{NullLiteral} \rangle \rightarrow \langle \text{Null} \rangle$
168. $\langle \text{Identifier} \rangle \rightarrow \langle \text{JavaLetter} \rangle \langle \text{Identifier} \rangle^*$
169. $\langle \text{Identifier} \rangle \rightarrow \langle \text{JavaLetter} \rangle \langle \text{Identifier} \rangle^*$

The variable $\langle \text{JavaLetter} \rangle$ denotes input, literals, and identifiers from the alphabet so that the character of an identifier by any number of characters for which the reserved and case-sensitive.

148. $\langle \text{OctalIntegerLiteral} \rangle \rightarrow \langle \text{OctalNumeral} \rangle \langle \text{IntegerTypeSuffix} \rangle_{\text{opt}}$
149. $\langle \text{IntegerTypeSuffix} \rangle \rightarrow \text{l} \mid \text{L}$
150. $\langle \text{DecimalNumeral} \rangle \rightarrow 0 \mid \langle \text{NonZeroDigit} \rangle \langle \text{Digits} \rangle_{\text{opt}}$
151. $\langle \text{Digits} \rangle \rightarrow \langle \text{Digit} \rangle \mid \langle \text{Digits} \rangle \langle \text{Digit} \rangle$
152. $\langle \text{Digit} \rangle \rightarrow 0 \mid \langle \text{NonZeroDigit} \rangle$
153. $\langle \text{NonZeroDigit} \rangle \rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
154. $\langle \text{HexNumeral} \rangle \rightarrow 0x \langle \text{HexDigit} \rangle \mid 0X \langle \text{HexDigit} \rangle \mid \langle \text{HexNumeral} \rangle \langle \text{HexDigit} \rangle$
155. $\langle \text{HexDigit} \rangle \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid a \mid b \mid c \mid d \mid e \mid A \mid B \mid C \mid D \mid E$
156. $\langle \text{OctalNumeral} \rangle \rightarrow 0 \langle \text{OctalDigit} \rangle \mid 0 \langle \text{OctalNumeral} \rangle \langle \text{OctalDigit} \rangle$
157. $\langle \text{OctalDigit} \rangle \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7$
158. $\langle \text{FloatingPointLiteral} \rangle \rightarrow \langle \text{Digits} \rangle . \langle \text{Digits} \rangle_{\text{opt}} \langle \text{ExponentPart} \rangle_{\text{opt}}$
 $\quad \quad \quad \langle \text{FloatTypeSuffix} \rangle_{\text{opt}} \mid$
 $\quad \quad \quad . \langle \text{Digits} \rangle \langle \text{ExponentPart} \rangle_{\text{opt}} \langle \text{FloatTypeSuffix} \rangle_{\text{opt}} \mid$
 $\quad \quad \quad \langle \text{Digits} \rangle \langle \text{ExponentPart} \rangle \langle \text{FloatTypeSuffix} \rangle_{\text{opt}} \mid$
 $\quad \quad \quad \langle \text{Digits} \rangle \langle \text{ExponentPart} \rangle_{\text{opt}} \langle \text{FloatTypeSuffix} \rangle$
159. $\langle \text{ExponentPart} \rangle \rightarrow \langle \text{ExponentIndicator} \rangle \langle \text{SignedInteger} \rangle$
160. $\langle \text{ExponentIndicator} \rangle \rightarrow e \mid E$
161. $\langle \text{SignedInteger} \rangle \rightarrow \langle \text{Sign} \rangle_{\text{opt}} \langle \text{Digits} \rangle$
162. $\langle \text{Sign} \rangle \rightarrow + \mid -$
163. $\langle \text{FloatTypeSuffix} \rangle \rightarrow f \mid F \mid d \mid D$
164. $\langle \text{BooleanLiteral} \rangle \rightarrow \text{true} \mid \text{false}$
165. $\langle \text{CharacterLiteral} \rangle \rightarrow ' \langle \text{InputCharacter} \rangle ' \mid ' \langle \text{EscapeCharacter} \rangle '$
166. $\langle \text{StringLiteral} \rangle \rightarrow " \langle \text{StringCharacters} \rangle_{\text{opt}} "$
167. $\langle \text{NullLiteral} \rangle \rightarrow \text{null}$

Identifier

168. $\langle \text{Identifier} \rangle \rightarrow \langle \text{IdentifierChars} \rangle$
169. $\langle \text{IdentifierChars} \rangle \rightarrow \langle \text{JavaLetter} \rangle \mid \langle \text{IdentifierChars} \rangle \langle \text{JavaLetterOrDigit} \rangle$

The variables $\langle \text{SingleCharacter} \rangle$, $\langle \text{InputCharacter} \rangle$, $\langle \text{EscapeSequence} \rangle$, and $\langle \text{JavaLetter} \rangle$ define the subsets of the 16-bit Unicode character set that can be used in input, literals, and identifiers.

Identifiers are defined by the variable $\langle \text{Identifier} \rangle$ and use characters from the Unicode alphabet so that programmers can write the source code in their own language. The first character of an identifier must be a letter, an underscore (_), or a dollar sign (\$) followed by any number of Java letters or digits. Java letters and digits consist of Unicode characters for which the method Character.isJavaIdentifierPart returns true. The Java keywords are reserved and cannot be used as identifiers.

Input characters are Unicode characters, not including the representation of linefeed or carriage return. A *<SingleCharacter>* is an input character but not ' or \. An escape sequence consists of a \ followed by an ASCII symbol to signify a nongraphic character. For example, \n is the escape sequence representing linefeed. Details on both the syntax and semantics of the Java programming language can be found in Gosling et al. [2000].

Bibliog

- Ackermann, W. *Annalen*, 99
- Aho, A. V., and J. D. Ullman. Vol. I: *Parsing Techniques*. New York: American Elsevier, 1972.
- Aho, A. V., and J. D. Ullman. Vol. II: *Compilers*. New York: Addison-Wesley, 1978.
- Aho, A. V., J. E. Hopcroft, and J. D. Ullman. *Algorithms for Minimum-Time Scheduling*. New York: Addison-Wesley, 1974.
- Backus, J. W. [1959]. The formal description of programming languages. In *Information and Control*, 2, 161–200.
- Bar-Hillel, Y., M. Perles, and E. Shamir. Structure of Chinese. *Language Research Institute*, 1960.
- Bavel, Z. [1983]. On the complexity of the word problem in semigroups. *Journal of Computer and System Sciences*, 27, 339–356.
- Blum, M. [1967]. Complexity of functions. *J. ACM*, 14, 257–267.
- Blum, M., and R. M. Kalai. *Information and Computation*, 1992, 102, 287–309.
- Bobrow, L. S., and M. Green. *Computer and Information Science*. Cambridge, MA: MIT Press, 1973.
- Bondy, J. A., and U. S. R. Murty. *Graph Theory with Applications*. London: American Elsevier, 1976.
- Brainerd, W. S., and R. E. Brainerd. *Formal Languages and Computation*. San Francisco: Holden-Day, 1973.

Bibliography

- Ackermann, W. [1928], "Zum Hilbertschen Aufbau der reellen Zahlen," *Mathematische Annalen*, 99, pp. 118–133.
- Aho, A. V., and J. D. Ullman [1972], *The Theory of Parsing, Translation and Compilation*, Vol. I: *Parsing*, Prentice Hall, Englewood Cliffs, NJ.
- Aho, A. V., and J. D. Ullman [1973], *The Theory of Parsing, Translation and Compilation*, Vol. II: *Compiling*, Prentice Hall, Englewood Cliffs, NJ.
- Aho, A. V., J. E. Hopcroft, and J. D. Ullman [1974], *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA.
- Aho, A. V., R. Sethi, and J. D. Ullman [1986], *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Reading, MA.
- Backus, J. W. [1959], "The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference," *Proc. of the International Conference on Information Processing*, pp. 125–132.
- Bar-Hillel, Y., M. Perles, and E. Shamir [1961], "On formal properties of simple phrase-structure grammars," *Zeitschrift für Phonetik, Sprachwissenschaft, und Kommunikationsforschung*, 14, pp. 143–177.
- Bavil, Z. [1983], *Introduction to the Theory of Automata*, Reston Publishing, Reston, VA.
- Blum, M. [1967], "A machine independent theory of the complexity of recursive functions," *J. ACM*, 14, pp. 322–336.
- Blum, M., and R. Koch [1999], "Greibach normal form transformation, revisited," *Information and Computation*, 150, pp. 112–118.
- Bobrow, L. S., and M. A. Arbib [1974], *Discrete Mathematics: Applied Algebra for Computer and Information Science*, Saunders, Philadelphia, PA.
- Bondy, J. A., and U. S. R. Murty [1977], *Graph Theory with Applications*, Elsevier, New York.
- Brainerd, W. S., and L. H. Landweber [1974], *Theory of Computation*, Wiley, New York.

- Brassard, G., and P. Bratley [1996], *Fundamentals of Algorithms*, Prentice Hall, Englewood Cliffs, NJ.
- Busacker, R. G., and T. L. Saaty [1965], *Finite Graphs and Networks: An Introduction with Applications*, McGraw-Hill, New York.
- Cantor, D. C. [1962], "On the ambiguity problems of Backus systems," *J. ACM*, 9, pp. 477–479.
- Cantor, G. [1947], *Contributions to the Foundations of the Theory of Transfinite Numbers* (reprint), Dover, New York.
- Chomsky, N. [1956], "Three models for the description of languages," *IRE Trans. on Information Theory*, 2, pp. 113–124.
- Chomsky, N. [1959], "On certain formal properties of grammars," *Information and Control*, 2, pp. 137–167.
- Chomsky, N. [1962], "Context-free grammar and pushdown storage," *Quarterly Progress Report* 65, M.I.T. Research Laboratory in Electronics, pp. 187–194.
- Chomsky, N., and G. A. Miller [1958], "Finite state languages," *Information and Control*, 1, pp. 91–112.
- Chomsky, N., and M. P. Schutzenberger [1963], "The algebraic theory of context free languages," in *Computer Programming and Formal Systems*, North-Holland, Amsterdam, pp. 118–161.
- Christofides, N. [1975], "Worst case analysis of a new heuristic for the traveling salesman problem," *Research Report* 338, Management Sciences, Carnegie Mellon University, Pittsburgh, PA.
- Church, A. [1936], "An unsolvable problem of elementary number theory," *American Journal of Mathematics*, 58, pp. 345–363.
- Church, A. [1941], "The calculi of lambda-conversion," *Annals of Mathematics Studies*, 6, Princeton University Press, Princeton, NJ.
- Cobham, A. [1964], "The intrinsic computational difficulty of functions," *Proceedings of the 1964 Congress for Logic, Mathematics and Philosophy of Science*, North-Holland, New York, pp. 24–30.
- Cook, S. A. [1971], "The complexity of theorem proving procedures," *Proc. of the Third Annual ACM Symposium on the Theory of Computing*, Association for Computing Machinery, New York, pp. 151–158.
- Cook, S. A., and R. A. Reckhow [1973], "Time bounded random access machines," *Journal of Computer and System Science*, 7, pp. 354–375.
- Cormen T. H., C. E. Leiserson, R. L. Rivest, and C. Stein. [2001], *Introduction to Algorithms*, McGraw-Hill, New York, NY.
- Davis, M. D. [1965], *The Undecidable*, Raven Press, Hewlett, NY.
- Davis, M. D., and E. J. Weyuker [1983], *Computability, Complexity and Languages: Fundamentals of Theoretical Computer Science*, Academic Press, New York.
- Denning, P. J. [1978], *Operating Systems: Theory and Practice*, Prentice Hall, Englewood Cliffs, NJ.
- De Remer, F. [1962], "Joint Coding of Programs and Data," *Proc. of the 1962 ACM Symposium on the Theory of Computing*, Association for Computing Machinery, New York, pp. 14–20.
- De Remer, F. [1963], "A Comparison of Two Coding Techniques," *Proc. of the 1963 ACM Symposium on the Theory of Computing*, Association for Computing Machinery, New York, pp. 449–454.
- Engelfriet, J. [1973], "On the Complexity of Pushdown Automata," *Proc. of the 1973 ACM Symposium on the Theory of Computing*, Association for Computing Machinery, New York, pp. 526–532.
- Fischer, P. C. [1965], "A Complexity Trade-off Between Time and Space," *Proc. of the 1965 ACM Symposium on the Theory of Computing*, Association for Computing Machinery, New York, pp. 13–18.
- Floyd, R. W. [1964], "Space-Time Trade-offs," *Proc. of the 1964 Congress for Logic, Mathematics and Philosophy of Science*, North-Holland, New York, pp. 1–10.
- Foster, J. M. [1965], "A Note on the Complexity of Computations," *Proc. of the 1965 ACM Symposium on the Theory of Computing*, Association for Computing Machinery, New York, pp. 11–15.
- Fraenkel, A. A. [1964], "Complexity of Computations," *Proc. of the 1964 Congress for Logic, Mathematics and Philosophy of Science*, North-Holland, New York, pp. 11–15.
- Garey, M. R., and D. S. Johnson [1979], *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, San Francisco.
- Ginsburg, S. [1966], "The Mathematical Structure of Language," *Proc. of the 1966 Congress for Logic, Mathematics and Philosophy of Science*, North-Holland, New York, pp. 1–10.
- Ginsburg, S., and G. Markov [1963], "On Some Properties of Context-Free Languages," *Proc. of the 1963 ACM Symposium on the Theory of Computing*, Association for Computing Machinery, New York, pp. 9–14.
- Ginsburg, S., and E. N. Ziv [1965], "On the Complexity of Computations," *Proc. of the 1965 ACM Symposium on the Theory of Computing*, Association for Computing Machinery, New York, pp. 11–15.
- Ginsburg, S., and E. N. Ziv [1966], "On the Complexity of Computations," *Proc. of the 1966 Congress for Logic, Mathematics and Philosophy of Science*, North-Holland, New York, pp. 1–10.
- Ginsburg, S., and E. N. Ziv [1967], "On the Complexity of Computations," *Proc. of the 1967 Congress for Logic, Mathematics and Philosophy of Science*, North-Holland, New York, pp. 1–10.
- Gödel, K. [1931], "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I," *Monatshefte für Mathematik*, 37, pp. 343–354.

- Jewood
on with
p. 477–
umbers
ans. on
Control,
rogress
Control,
ree lan-
terdam,
lesman
versity,
erican
dies, 6,
ings of
olland,
? Third
puting
ournal
Algo-
uages:
- Denning, P. J., J. B. Dennis, and J. E. Qualitz [1978], *Machines, Languages, and Computation*, Prentice Hall, Englewood Cliffs, NJ.
- De Remer, F. L. [1969], "Generating parsers for BNF grammars," *Proc. of the 1969 Fall Joint Computer Conference*, AFIPS Press, Montvale, NJ, pp. 793–799.
- De Remer, F. L. [1971], "Simple LR(k) grammars," *Comm. ACM*, 14, pp. 453–460.
- Edmonds, J. [1965], "Paths, trees and flowers," *Canadian Journal of Mathematics*, 3, pp. 449–467.
- Engelfriet, J. [1992], "An elementary proof of double Greibach normal form," *Information Processing Letters*, 44, pp. 291–293.
- Evey, J. [1963], "Application of pushdown store machines," *Proc. of the 1963 Fall Joint Computer Science Conference*, AFIPS Press, pp. 215–217.
- Fischer, P. C. [1963], "On computability by certain classes of restricted Turing machines," *Proc. of the Fourth Annual Symposium on Switching Circuit Theory and Logical Design*, pp. 23–32.
- Floyd, R. W. [1962], "On ambiguity in phrase structure languages," *Comm. ACM*, 5, pp. 526–534.
- Floyd, R. W. [1964], *New Proofs and Old Theorems in Logic and Formal Linguistics*, Computer Associates, Wakefield, MA.
- Foster, J. M. [1968], "A syntax improving program," *Computer J.*, 11, pp. 31–34.
- Fraenkel, A. A., Y. Bar-Hillel, and A. Levy [1984], *Foundations of Set Theory*, 2d ed., North-Holland, New York.
- Garey, M. R., and D. S. Johnson [1979], *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, New York.
- Ginsburg, S. [1966], *The Mathematical Theory of Context-Free Languages*, McGraw-Hill, New York.
- Ginsburg, S., and H. G. Rice [1962], "Two families of languages related to ALGOL," *J. ACM*, 9, pp. 350–371.
- Ginsburg, S., and G. F. Rose [1963a], "Some recursively unsolvable problems in ALGOL-like languages," *J. ACM*, 10, pp. 29–47.
- Ginsburg, S., and G. F. Rose [1963b], "Operations which preserve definability in languages," *J. ACM*, 10, pp. 175–195.
- Ginsburg, S., and J. S. Ullian [1966a], "Ambiguity in context-free languages," *J. ACM*, 13, pp. 62–89.
- Ginsburg, S., and J. S. Ullian [1966b], "Preservation of unambiguity and inherent ambiguity in context-free languages," *J. ACM*, 13, pp. 364–368.
- Gödel, K. [1931], "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I," *Monatshefte für Mathematik und Physik*, 38, pp. 173–198. (English translation in Davis [1965].)

- Gosling, J., B. Joy, G. Steele, and G. Bracha [2000], *The Java Language Specification*, 2d ed., Addison-Wesley, Boston, MA.
- Greibach, S. [1965], "A new normal form theorem for context-free phrase structure grammars," *J. ACM*, 12, pp. 42–52.
- Halmos, P. R. [1974], *Naive Set Theory*, Springer-Verlag, New York.
- Harrison, M. A. [1978], *Introduction to Formal Language Theory*, Addison-Wesley, Reading, MA.
- Hartmanis, J., and J. E. Hopcroft [1971], "An overview of the theory of computational complexity," *J. ACM*, 18, pp. 444–475.
- Hennie, F. C. [1977], *Introduction to Computability*, Addison-Wesley, Reading, MA.
- Hermes, H. [1965], *Enumerability, Decidability, Computability*, Academic Press, New York.
- Hochbaum, D. S., ed. [1997], *Approximation Algorithms for NP-Complete Problems*, PWS Publishing, Boston, MA.
- Hopcroft, J. E. [1971], "An $n \log n$ algorithm for minimizing the states in a finite automaton," in *The Theory of Machines and Computation*, ed. by Z. Kohavi, Academic Press, New York, pp. 189–196.
- Hopcroft, J. E., and J. D. Ullman [1979], *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, Reading, MA.
- Ibarra, O.H., and C. E. Kim [1975], "Fast approximation problems for the knapsack and sum of subsets problems," *J. ACM*, 22, no. 4, pp. 463–468.
- Jensen, K., and N. Wirth [1974], *Pascal: User Manual and Report*, 2d ed., Springer-Verlag, New York.
- Karp, R. M. [1972], "Reducibility among combinatorial problems," in *Complexity of Computer Computations*, Plenum Press, New York, pp. 85–104.
- Karp, R. M. [1986], "Combinatorics, complexity and randomness," *Comm. ACM*, 29, no. 2, pp. 98–109.
- Kfoury, A. J., R. N. Moll, and M. A. Arbib [1982], *A Programming Approach to Computability*, Springer-Verlag, New York.
- Kleene, S. C. [1936], "General recursive functions of natural numbers," *Mathematische Annalen*, 112, pp. 727–742.
- Kleene, S. C. [1952], *Introduction to Metamathematics*, Van Nostrand, Princeton, NJ.
- Kleene, S. C. [1956], "Representation of events in nerve nets and finite automata," in *Automata Studies*, ed. by C. E. Shannon and J. McCarthy, Princeton University Press, Princeton, NJ, pp. 3–42.
- Knuth, D. E. [1965], "On the translation of languages from left to right," *Information and Control*, 8, pp. 607–639.

- Knuth, D. E. [1965], *On the Translation of Languages from Left to Right*, Addison-Wesley, Reading, MA.
- Koch, R., and J. W. Thatcher [1968], "State transition systems," in *Formal Languages STAF*, ed. by J. W. Thatcher, Academic Press, New York, pp. 47–54.
- Korenjak, A. [1971], "The state transition system of a pushdown automaton," *J. ACM*, 12, pp. 444–475.
- Kurki-Suonio, T. [1972], "On the equivalence of finite automata," *Information and Control*, 19, pp. 22–36.
- Kuroda, S. Y. [1961], "On the equivalence problem for push-down automata," *Information and Control*, 4, pp. 305–320.
- Ladner, R. E. [1975], "On the computational power of pushdown automata," *Information and Control*, 22, pp. 257–265.
- Landweber, H. [1966], "On the equivalence of pushdown automata," *Information and Control*, 9, pp. 411–422.
- Levitin, A. [1970], *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, San Francisco.
- Lewis, H. R. [1969], *Computability of Functions of Finite Types*, Prentice-Hall, Englewood Cliffs, NJ.
- Lewis, P. M. [1973], "Computability of functions of finite types," *Information and Control*, 20, pp. 465–473.
- Markov, A. A. [1947], "Theory of algorithms," in *Selected Works of A. A. Markov*, ed. by A. A. Markov and V. A. Ufnarovsky, Israel Program for Scientific Translations, Jerusalem, 1961.
- McNaughton, L. [1960], "Irreducibility among finite automata," *Information and Control*, 3, pp. 280–292.
- Mealy, G. H. [1955], "A method for synthesis of sequential circuit," *Journal of the Franklin Institute*, 254, pp. 141–155.
- Meyer, A. R. [1968], "A note on the complexity of computation," in *Complexity of Computer Computations*, ed. by J. B. Rosenthal, Plenum Press, New York, pp. 151–160.
- Minsky, M. L. [1967], *Computation: Finite and Infinite Machines*, Prentice-Hall, Englewood Cliffs, NJ.
- Moore, E. F. [1950], "Recursiveness of partial recursive functions," in *Recursive Function Theory*, ed. by J. B. Rosenthal, Plenum Press, New York, pp. 129–140.
- Myhill, J. [1957], "Equivalence relations on partially ordered sets," *Information and Control*, 1, pp. 57–67.
- Myhill, J. [1959], "Equivalence relations on partially ordered sets," *Information and Control*, 2, pp. 247–265.
- Naur, P., ed. [1973], *Programming Languages*, Academic Press, London.
- ACM, 6, pp. 607–639.

- ification, 2d
ture gram-
ley, Read-
putational
MA.
ress, New
ems, PWS
tomaton,"
ress, New
uages and
psack and
er-Verlag,
y of Com-
29, no. 2,
to Com-
ematische
, NJ.
mata," in
ity Press,
ation and
- Knuth, D. E. [1968], *The Art of Computer Programming*, Vol. 1: *Fundamental Algorithms*, Addison-Wesley, Reading, MA.
- Koch, R., and N. Blum [1997], "Greibach normal form transformation revisited," *Proceedings STACS 97*, Lecture Notes in Computer Science 1200, Springer-Verlag, New York, pp. 47-54.
- Korenjak, A. J. [1969], "A practical method for constructing LR(k) processors," *Comm. ACM*, 12, pp. 613-623.
- Kurki-Suonio, R. [1969], "Notes on top-down languages," *BIT*, 9, pp. 225-238.
- Kuroda, S. Y. [1964], "Classes of languages and linear-bounded automata," *Information and Control*, 7, pp. 207-223.
- Ladner, R. E. [1975], "On the structure of polynomial time reducibility," *Journal of the ACM*, 22, pp. 155-171.
- Landweber, P. S. [1963], "Three theorems of phrase structure grammars of type 1," *Information and Control*, 6, pp. 131-136.
- Levitin, A. [2003], *The Design and Analysis of Algorithms*, Addison-Wesley, Boston, MA.
- Lewis, H. R., and C. H. Papadimitriou [1981], *Elements of the Theory of Computation*, Prentice Hall, Englewood Cliffs, NJ.
- Lewis, P. M., II, and R. E. Stearns [1968], "Syntax directed transduction," *J. ACM*, 15, pp. 465-488.
- Markov, A. A. [1961], *Theory of Algorithms*, Israel Program for Scientific Translations, Jerusalem.
- McNaughton, R., and H. Yamada [1960], "Regular expressions and state graphs for automata," *IEEE Trans. on Electronic Computers*, 9, pp. 39-47.
- Mealy, G. H. [1955], "A method for synthesizing sequential circuits," *Bell System Technical Journal*, 34, pp. 1045-1079.
- Meyer, A. R., and L. J. Stockmeyer [1973], "The equivalence problem for regular expressions with squaring requires exponential space," *Proc. of the Thirteenth Annual IEEE Symposium on Switching and Automata Theory* pp. 125-129.
- Minsky, M. L. [1967], *Computation: Finite and Infinite Machines*, Prentice Hall, Englewood Cliffs, NJ.
- Moore, E. F. [1956], "Gendanken-experiments on sequential machines," in *Automata Studies*, ed. by C. E. Shannon and J. McCarthy, Princeton University Press, Princeton, NJ, pp. 129-153.
- Myhill, J. [1957], "Finite automata and the representation of events," *WADD Technical Report 57-624*, Wright Patterson Air Force Base, OH, pp. 129-153.
- Myhill, J. [1960], "Linear bounded automata," *WADD Technical Note 60-165*, Wright Patterson Air Force Base, OH.
- Naur, P., ed. [1963], "Revised report on the algorithmic language ALGOL 60," *Comm. ACM*, 6, pp. 1-17.

- Nerode, A. [1958], "Linear automaton transformations," *Proc. AMS*, 9, pp. 541–544.
- Oettinger, A. G. [1961], "Automatic syntax analysis and the pushdown store," *Proc. on Symposia on Applied Mathematics*, 12, American Mathematical Society, Providence, RI, pp. 104–129.
- Ogden, W. G. [1968], "A helpful result for proving inherent ambiguity," *Mathematical Systems Theory*, 2, pp. 191–194.
- Ore, O. [1963], *Graphs and Their Uses*, Random House, New York.
- Papadimitriou, C. H. [1994], *Computational Complexity*, Addison-Wesley, Reading, MA.
- Papadimitriou, C. H., and K. Steiglitz [1982], *Combinatorial Optimization: Algorithms and Complexity*, Prentice Hall, Englewood Cliffs, NJ.
- Parikh, R. J. [1966], "On context-free languages," *J. ACM*, 13, pp. 570–581.
- Péter, R. [1967], *Recursive Functions*, Academic Press, New York.
- Post, E. L. [1936], "Finite combinatory processes—formulation I," *Journal of Symbolic Logic*, 1, pp. 103–105.
- Post, E. L. [1946], "A variant of a recursively unsolvable problem," *Bulletin of the American Mathematical Society*, 52, pp. 264–268.
- Post, E. L. [1947], "Recursive unsolvability of a problem of Thue," *Journal of Symbolic Logic*, 12, pp. 1–11.
- Pratt, V. [1975], "Every prime has a succinct certificate," *SIAM Journal of Computation*, 4, pp. 214–220.
- Rabin, M. O., and D. Scott [1959], "Finite automata and their decision problems," *IBM J. Res.*, 3, pp. 115–125.
- Rice, H. G. [1953], "Classes of recursively enumerable sets and their decision problems," *Trans. of the American Mathematical Society*, 89, pp. 25–29.
- Rice, H. G. [1956], "On completely recursively enumerable classes and their key arrays," *Journal of Symbolic Logic*, 21, pp. 304–341.
- Rogers, H., Jr. [1967], *Theory of Recursive Functions and Effective Computation*, McGraw-Hill, New York.
- Rosenkrantz, D. J., and R. E. Stearns [1970], "Properties of deterministic top-down grammars," *Information and Control*, 17, pp. 226–256.
- Sahni, A. [1975], "Approximate algorithms for the 0/1 knapsack problem," *J. ACM*, 22, no. 1, pp. 115–124.
- Salomaa, A. [1973], *Formal Languages*, Academic Press, New York.
- Salomaa, S. [1966], "Two complete axiom systems for the algebra of regular events," *J. ACM*, 13, pp. 156–199.
- Savitch, W. J. [1970], "Relationships between nondeterministic and deterministic tape complexities," *J. Computer and Systems Sciences*, 4, no. 2, pp. 177–192.

- Scheinberg, S. [1966], "A note on the complexity of computation and communication," *Information and Control*, 10, pp. 217–226.
- Schutzenberger, M. P. [1960], "On the equivalence of two formal languages," *Information and Control*, 3, pp. 247–265.
- Sheperdson, J. C. [1965], "The complexity of the word problem for groups," *J. Res.*, 3, pp. 10, pp. 217–226.
- Sheperdson, J. C. [1966], "The complexity of the word problem for groups," *J. Res.*, 3, pp. 10, pp. 217–226.
- Soisalon-Soininen, E. [1978], "On the equivalence of LR(k) and LL(k) form," *Information and Control*, 35, pp. 137–147.
- Stearns, R. E. [1956], "The equivalence of two formal languages," *Conference Record*, 1956, pp. 1–10.
- Stoll, R. [1963], *Set Theory and Logic*, Addison-Wesley, Reading, MA.
- Thue, A. [1914], "Über die gegenseitige Lage der Elemente in den unendlichen Zeichenketten und über allgemeine Entscheidungsprobleme," *Natur-Vidensk. Medd.*, 7, pp. 1–67.
- Turing, A. M. [1936], "On computable numbers, with an application to the Entscheidungsproblem," *Proceedings of the Royal Society of London A*, 1936, no. 43, pp. 42–48.
- von Neumann, J. [1955], "Theory of self-reproducing automata," in A. H. Taub, ed., *Engineering Cybernetics*, 1955, pp. 1–100.
- Wand, M. [1980], "A note on the complexity of the word problem for groups," *Information and Control*, 45, pp. 1–10.
- Wilson, R. J. [1969], "The complexity of the word problem for groups," *Information and Control*, 16, pp. 356–366.
- Wood, D. [1969], "The complexity of the word problem for groups," *Information and Control*, 16, pp. 356–366.
- Younger, D. [1967], "A linear-time solution for the membership problem for LR(k) languages," *Information and Control*, 12, pp. 317–326.

- 1–544.
- ,” Proc. on Providence,
- athematical
- ading, MA.
- orithms and
- of Symbolic
- e American
- of Symbolic
- putation, 4,
- ns,” IBM J.
- problems,”
- key arrays,”
- t, McGraw-
- own gram-
- CM, 22, no.
- ar events,”
- nistic tape
- Scheinberg, S. [1960], “Note on the Boolean properties of context-free languages,” *Information and Control*, 3, pp. 372–375.
- Schutzenberger, M. P. [1963], “On context-free languages and pushdown automata,” *Information and Control*, 6, pp. 246–264.
- Sheperdson, J. C. [1959], “The reduction of two-way automata to one-way automata,” *IBM J. Res.*, 3, pp. 198–200.
- Sheperdson, J. C., and H. E. Sturgis [1963], “Computability of recursive functions,” *J. ACM*, 10, pp. 217–255.
- Soisalon-Soininen, E., and E. Ukkonen [1979], “A method for transforming grammars into LL(k) form,” *Acta Informatica*, 12, pp. 339–369.
- Stearns, R. E. [1971], “Deterministic top-down parsing,” *Proc. of the Fifth Annual Princeton Conference of Information Sciences and Systems*, pp. 182–188.
- Stoll, R. [1963], *Set Theory and Logic*, W. H. Freeman, San Francisco, CA.
- Thue, A. [1914], “Probleme über Veränderungen von Zeichenreihen nach gegebenen Regeln,” *Skrifter utgit av Videnskappsselskapet i Kristiana*, I., Matematisk-natur-videnskabelig klasse, 10.
- Turing, A. M. [1936], “On computable numbers with an application to the Entscheidungsproblem,” *Proc. of the London Mathematical Society*, 2, no. 42, pp. 230–265; no. 43, pp. 544–546.
- von Neumann, J. [1945], *First Draft of a Report on EDVAC*, Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia, PA. Reprinted in: Stern, N. [1981], *From Eniac to Univac*, Digital Press, Bedford, MA.
- Wand, M. [1980], *Induction, Recursion and Programming*, North-Holland, New York.
- Wilson, R. J. [1985], *Introduction to Graph Theory*, 3d ed., American Elsevier, New York.
- Wood, D. [1969], “The theory of left factored languages,” *Computer Journal*, 12, pp. 349–356.
- Younger, D. [1967], “Recognition and parsing of context-free languages in time n^3 ,” *Information and Control*, 10, no. 2, pp. 189–208.

Subject Index

- Abnormal termination, 257
Abstract machine, 147
Acceptance
 by deterministic finite automaton, 147–148
 by empty stack, 230
 by entering, 263, 289
 by final state, 229, 260
 by halting, 262–263
 by nondeterministic finite automaton, 161
 by nondeterministic Turing machine, 274
 by pushdown automaton, 224, 229–230
 by Turing machine, 260
Accepted string, 148, 224
Accepting state, 146–147
Ackermann's function, 411–413
Acyclic graph, 33
Adjacency relation, 32
AE, 205, 556, 585
 nonregularity of, 205
ALGOL, 1, 94, 553
Algorithm, 343–344
Alphabet, 42–43, 147, 163
 input, 222, 256
 stack, 222
 tape, 256
Ambiguity, 91–93
 inherent, 92
Ancestor, 34–35
Approximation algorithm, 519–521
Approximation schema, 523–526
 fully polynomial, 526
Arithmetization, 416
ASCII character set, 21–22
Associativity, 44–45
Atomic pushdown automaton, 227
Atomic Turing machine, 290
Backus-Naur Form (BNF), 94, 553, 631
Barber's paradox, 21–22
Big oh, 436–438
Big theta, 438
Bin Packing Problem, 516
Binary relation, 11–12
Binary tree, 35
Blank Tape Problem, 366–368
BNF (Backus-Naur Form), 94, 553, 631
Boolean variable, 481
Bottom-up parser, 555, 561–567
 depth-first, 563–567
 LR(0), 599–601, 604
 LR(1), 618
Bounded operators, 398–404
Bounded sum, 398
Breadth-first bottom-parser, 563–567
Breadth-first top-down parser, 557–561
Cardinality, 16–21
Cartesian product, 11–12
Chain, 114
Chain rule, 113
 elimination of, 113–116
Characteristic function, 298–299
Child, 33
Chomsky hierarchy, 64, 338–339
Chomsky normal form, 121–124, 239–240
Church-Turing Thesis, 2, 253, 344, 352–354, 421–423
Clause, 482
Closure properties
 of context-free languages, 243–246
 of countable sets, 18–19
 of regular languages, 200–203
Co-NP, 531
Compatible transitions, 225
Compilation, 567
Complement, 9
 acceptance of, 158
Complete binary tree, 40
Complete item, 602
Complexity, 433–436
 nondeterministic, 466–468
 space complexity, 532–535
 time complexity, 442–446
Composition of functions, 308–309
Computable function, 7, 296, 353
 Church-Turing Thesis for, 353–354, 421–423
Concatenation
 of languages, 47
 of strings, 43–44
Conjunctive normal form, 482
Context, 69
Context-free grammar, 68–69
 ambiguous, 91, 384
 equivalent, 79
 for Java, 94–97, 631–639
 language of, 70
 left-linear, 220
 left-regular, 219–220
 right-linear, 102, 219

- Context-free grammar (*continued*)
undecidable problems of, 382–386
- Context-free language, 70
acceptance by pushdown automaton, 232–239
closure properties of, 243–246
examples of, 76–81
inherently ambiguous, 92
pumping lemma for, 239–242
- Context-sensitive grammar, 332–334
- Context-sensitive language, 333
- Context-sensitive Turing machine, 290
- Cook's Theorem, 485
- Countable set, 7, 17–19
- Countably infinite set, 17–19
- Course-of-values recursion, 409
- Cycle, 33
- Cyclic graph, 33
- CYK algorithm, 124–128
- Dead end, 558
- Decidable language, 260
in polynomial space, 540
in polynomial time, 468
problem, 343–344
- Decision problem, 343–346
Bin Packing Problem, 516
Blank Tape Problem, 366–368
Church-Turing Thesis for, 353
Halting Problem, 357, 362–365
- Hamiltonian Circuit Problem, 473–477, 503–509
- Hitting Set Problem, 515–516
- intractable, 431, 465, 548–550
- Knapsack Problem, 518
- NP-complete, 480
- Partition Problem, 513–515
- Post Correspondence Problem, 377–382
reduction of, 348–352
representation of, 344–346, 469–471
- Satisfiability Problem, 472, 481–483
- Subset-Sum Problem, 473, 509–513
- 3-Satisfiability Problem, 498–500
- Traveling Salesman Problem, 517–518
undecidable, 361
- Vertex Cover Problem, 500–503, 527
- Word Problem, 373–376
- DeMorgan's Laws, 9–10
- Denumerable set, 17–19
- Depth of a node, 34
- Derivable string, 69, 326
- Derivation, 66–67, 69
directly left recursive, 129
leftmost, 71, 89–91
length of, 69
recursive, 71
rightmost, 71
- Derivation tree, 71–74
- Descendant, 34–35
- Deterministic finite automaton (DFA), 2–3, 147
examples, 150–159
extended transition function, 151
incompletely specified, 158
language of, 148
minimization, 178–183
state diagram of, 146–147, 151–153
transition table, 150
- Deterministic LR(0) machine, 603
- Deterministic pushdown automaton, 225
- DFA. *See* Deterministic finite automaton
- Diagonalization, 7, 19–23
- Difference of sets, 9
- Direct left recursion, removal of, 129–131
- Directed graph, 32–33, 347
- Disjoint sets, 9
- Distinguishable state, 178
- Distinguished element, 32
- Domain, 12
- Dynamic programming, 125
- Effective procedure, 253, 344
- Empty set, 8
- Empty stack (acceptance by), 230
- Enumeration, by Turing machine, 282–288
- Equivalence class, 15
- Equivalence relation, 14–16
right-invariant, 212
- Equivalent grammars, 79
machines, 158
regular expressions, 53
states, 178
- Essentially noncontracting grammar, 110
- Expanding a node, 558
- Exponential growth, 441
- Expression graph, 193–196
- Extended pushdown automaton, 225
- Extended transition function, 151
- Factorial, 392–393
- Fibonacci numbers, 407
- Final state, 147, 259
acceptance by, 229, 260
- Finite automaton. *See*
Deterministic finite automaton, Finite-state machine, Nondeterministic finite automaton
- Finite-state machine, 145–147
- FIRST_k set, 576
construction of, 580–583
- Fixed point, 20
- FOLLOW_k set, 577
construction of, 583–585
- Frontier (of a tree), 35
- Fully space constructible, 538
- Function, 14
Ackermann's, 411–413
characteristic, 298–299
composition of, 308–309
computable, 7
computation of, 295–298
identity, 301
input transition, 170
macro-computable, 430
 μ -recursive, 414
 n -variable, 12
number-theoretic, 299
one-to-one, 13–14
onto, 13–14
partial, 13
- polynomially bounded, 440–441
- primitive recursive projection, 300, 39
- rates of growth, 43
- total, 13
- transition, 147, 162
- Turing computable, 312
- Gödel numbering, 401
- Grammar. *See also* Context-sensitive grammar; Regular grammar; Context-free grammar; Essentially noncontracting grammar; Graph; LL(k); LR(0); LR(1); Non-deterministic finite automaton; Finite-state machine; Deterministic finite automaton; Non-deterministic finite automaton; Expression graph; Frontier (of a tree); Fully space constructible; Function; Input transition; Macro-computable; μ -recursive; Number-theoretic function; One-to-one function; Onto function; Partial function; Polynomially bounded function; Primitive recursive function; Projection function; Rates of growth function; Total function; Transition function
- graph of, 556
language of, 70
linear, 250
- LL(k), 571, 589–591
- LR(0), 598, 609
- LR(1), 612–618
- noncontracting, 10
phrase-structure, 33
- regular, 81–83, 196
- right-linear, 102, 210
- strong LL(k), 579–580
- unrestricted, 254, 312
- Graph
acyclic, 33
cyclic, 33, 358
- directed, 32–34, 347
- expression, 193–196
of a grammar, 556
- Graph of a grammar, 556
- Greibach normal form, 232–233
- Grep, 55–58
- Halting, 257
acceptance by, 262–263
- Halting Problem, 357
- Hamiltonian Circuit Problem, 473–477, 503–509
- Hard for a class, 479
- Hitting Set Problem, 515–516
- Home position, 314
- Homomorphic image, 219
- Homomorphism, 219, 314
- Identity function, 301
- Implicit tree, 557

- ring
88
5
, 14–16
2
- s, 53
- acting
58
441
3–196
- automaton,
unction,
07
- , 260
e
nite
e-state
terministic
145–147
- 0–583
- 3–585
5
ible, 538
- 413
299
3–309
- 5–298
- 0
430
- 99
- polynomially bounded,
440–441
primitive recursive, 389–391
projection, 300, 390
rates of growth, 436–441
total, 13
transition, 147, 163, 166, 222
Turing computable, 296
uncomputable, 312–313
- Gödel numbering, 406
- Grammar. *See also* Context-free grammar; Regular grammar
context-sensitive, 332–334
essentially noncontracting,
110
graph of, 556
language of, 70
linear, 250
 $LL(k)$, 571, 589–591
 $LR(0)$, 598, 609
 $LR(1)$, 612–618
noncontracting, 107
phrase-structure, 325–326
regular, 81–83, 196
right-linear, 102, 219
strong $LL(k)$, 579–580
unrestricted, 254, 325–332
- Graph
acyclic, 33
cyclic, 33, 358
directed, 32–34, 347
expression, 193–196
of a grammar, 556
Graph of a grammar, 556
Greibach normal form, 131–138,
232–233
Grep, 55–58
- Halting, 257
acceptance by, 262–263
Halting Problem, 357, 362–365
Hamiltonian Circuit Problem,
473–477, 503–509
Hard for a class, 479
Hitting Set Problem, 515–516
Home position, 314
Homomorphic image, 219
Homomorphism, 219, 257
- Identity function, 301
Implicit tree, 557
- In-degree of a node, 32
Indistinguishable
state, 178
string, 211–212
- Induction. *See* Mathematical induction
- Infinite set, 17
- Infix notation, 205
- Inherent ambiguity, 92
- Input alphabet, 222
- Input transition function, 170
- Intersection of sets, 9
- Intractable decision problem,
431, 465, 548–550
- Invariance, right, 212
- Inverse homomorphic image,
251
- Item
complete, 602
 $LL(0)$, 602
 $LR(1)$, 614
- Java, 94–97, 631–639
- Kleene star operation, 47–48
- Kleene's Theorem, 196
- Knapsack Problem, 518
approximation algorithm for,
524–526
- L_{REG} , 545
- L_{SAT} , 483
- λ -closure, 170
- λ -rule, 69
elimination of, 106–113
 λ -transition, 165–166
- Language, 41–43, 326
context-free, 70
context-sensitive, 332
finite specification of, 45–49
of finite-state machine, 63,
148
inherently ambiguous, 92
of nondeterministic finite
automaton, 163
nonregular, 203–204
of phrase-structure grammar,
326
polynomial, 468
of pushdown automaton, 224
recognition, 260
recursive, 260
- recursive definition of, 46–47
recursively enumerable, 260
regular, 49, 82, 200
of Turing machine, 260
- Language acceptor, Turing
machine as, 259–262
- Language enumerator, Turing
machine as, 282–288
- LBA. *See* Linear-bounded
automaton
- Leaf, 34
- Left factoring, 576
- Left-linear context-free
grammar, 220
- Left-recursive rule, 129
- Left-regular context-free
grammar, 219–220
- Leftmost derivation, 71, 89–91
ambiguity and, 91–93
- Lexical analysis, 553, 567
- Lexicographical ordering, 286
- L'Hospital's Rule, 439–440
- Linear-bounded automaton
(LBA), 334–338
- Linear grammar, 250
- Linear speedup, 448–451
- Literal, 482
- $LL(k)$ grammar, 571, 589–591
strong, 579–580
- Lookahead
set, 572, 575, 589
string, 572
- Lower-order terms, 436
- $LR(0)$
context, 595–596
grammar, 598, 609
item, 602
machine, 602–603, 606–610
parser, 599–601, 604
- $LR(1)$
context, 613
grammar, 612–618
item, 614
machine, 614–617
parser, 618
- Machine configuration
of deterministic finite
automaton, 149
- of pushdown automaton, 224
of Turing machine, 256–258
- Macro, 302–305

- Macro-computable function, 430
 Mathematical induction, 27–32
 simple, 30
 strong, 30
 Microsoft Word, 58
 Minimal common ancestor, 34–35
 Minimization, 400
 bounded, 401
 unbounded, 413
 Monotonic rule, 333
 Moore machine, 156
 μ -recursive function, 414
 Turing computability of, 414–415
 Multitape Turing machine, 268–274
 time complexity of, 447–448
 Multitrack Turing machine, 263–265
 time complexity of, 446
 Myhill-Nerode Theorem, 211–217

n-ary relation, 12
n-variable function, 12
 Natural language, 1, 5
 Natural numbers, 8
 recursive definition of, 24
 NFA. *See* Nondeterministic finite automaton
 $NFA-\lambda$, 165–166
 Node, 32
 Noncontracting derivation, 333
 Noncontracting grammar, 107
 Nondeterminism, removing, 170–178
 Nondeterministic complexity, 442–446
 Nondeterministic finite automaton (NFA), 159–163
 acceptance by, 161
 examples, 164–165
 input transition function, 169
 λ -transition, 165–166
 language of, 163
 Nondeterministic LR(0) machine, 602–603
 Nondeterministic LR(1) machine, 616

 Nondeterministic polynomial time, 469
 Nondeterministic Turing machine, 274–282
 Nonregular language, 203–205
 Nonterminal symbol, 65, 69
 Normal form, 103
 Chomsky, 121–124, 239–240
 conjunctive, 482
 Greibach, 131–138, 232–235
 3-conjunctive, 498
 NP, 431, 469
 NPC, 492
 NP-complete problem, 480
 Bin Packing Problem, 516
 Hamiltonian Circuit Problem, 473–477, 503–509
 Hitting Set Problem, 515–516
 Knapsack Problem, 518
 Partition Problem, 513–515
 Satisfiability Problem, 473, 481–483
 Subset-Sum Problem, 473, 509–513
 3-Satisfiability Problem, 498–500
 Traveling Salesman Problem, 517–518
 Vertex Cover Problem, 500–503, 527
 NP-hard problem, 480
 NPJ, 530
 NP-Space, 540
 Null
 path, 33
 rule, 69
 string, 42
 Nullable variable, 107
 Number-theoretic function, 299
 Numeric computation, 299–301

 One-to-one function, 13–14
 Onto function, 13–14
 Operator, bounded, 398–404
 Optimization problem, 517–518
 Order of a function, 436
 Ordered *n*-tuple, 12
 Ordered tree, 33–34
 Out-degree of a node, 32
 Output tape, 282

 P , 431, 468

 P-Space, 540
 completeness, 544–545
 Palindrome, 60, 77–78, 204, 226
 Parsing, 553, 567–568
 bottom-up parser, 555, 561–567
 breadth-first bottom-parser, 563–567
 breadth-first top-down parser, 557–561
 CYK algorithm, 124–128
 deterministic, 554, 571
 LL(k), 591
 LR(0), 599–601, 604
 LR(1), 618
 strong LL(k), 587–588
 top-down, 555
 Partial function, 13
 Partition, 9
 Partition Problem, 513–515
 Path, 32–33
 null, 33
 PDA. *See* Pushdown automaton
 Phrase-structure grammar, 325–326
 Pigeonhole principle, 206
 Polynomial with integral coefficients, 438
 Polynomial language, 468
 Polynomially bounded function, 440–441
 Post Correspondence Problem, 377–382
 Post correspondence system, 377
 Power set, 9, 20
 Prefix, 45
 terminal prefix, 129, 557
 viable, 599
 Primitive recursion, 390–391
 Primitive recursive function, 389–391, 410–413
 basic, 389–390
 examples of, 391–398
 Turing computability of, 393–394
 Primitive recursive predicate, 395
 Problem reduction, 348–352, 365–367
 and NP-completeness, 497–498, 513–514

 polynomial-time, 4
 and undecidability, 3
 Projection function, 3
 Proof by contradiction
 Proper subset, 9
 Proper subtraction, 39
 Pseudo-polynomial problem, 471
 Pumping lemma
 for context-free languages, 239–242
 for regular languages, 205–209
 Pushdown automaton, 2–3, 221–222
 acceptance, 224
 acceptance by emptiness, 230
 acceptance by final atomic, 227
 context-free languages, 232–239
 deterministic, 225
 extended, 228
 language of, 230
 stack alphabet, 222
 state diagram, 222–223
 variations, 227–232

 Random access machine, 2
 Range, 12
 Rates of growth, 436–437
 Reachable variable, 11
 Recognition of languages, 2
 Recursion
 course-of-values, 400
 primitive, 390–391
 removal of direct left recursion, 129–132
 simultaneous, 427
 Recursive definition, 2–3, 45–46
 Recursive language, 2
 Recursive variable, 71
 Recursively enumerable language, 260
 Reduction, 555–556, 561
 also Problem reduction
 Regular expression, 42
 defining pattern with, 53
 equivalent, 53
 examples, 51–53

- 545
8, 204, 226
8
555,
i-parser,
wn parser,
4–128
571
4
588
1–515
utomaton
mar,
206
ral
468
function,
Problem,
ystem,
557
0–391
ction,
3
8
y of,
dicate,
3–352,
s,
+
- polynomial-time, 477
and undecidability, 365–368
Projection function, 300, 390
Proof by contradiction, 19–23
Proper subset, 9
Proper subtraction, 39, 395
Pseudo-polynomial problem, 471
Pumping lemma
for context-free languages, 239–242
for regular languages, 205–209
Pushdown automaton (PDA), 2–3, 221–222
acceptance, 224
acceptance by empty stack, 230
acceptance by final state, 229
atomic, 227
context-free language and, 232–239
deterministic, 225
extended, 228
language of, 230
stack alphabet, 222
state diagram, 222–223
variations, 227–232
- Random access machine, 323
Range, 12
Rates of growth, 436–441
Reachable variable, 119
Recognition of language, 260
Recursion
course-of-values, 409
primitive, 390–391, 413–414
removal of direct left recursion, 129–131
simultaneous, 427
Recursive definition, 23–27, 45–46
Recursive language, 260, 346
Recursive variable, 71, 390
Recursively enumerable language, 260
Reduction, 555–556, 561. *See also* Problem reduction
Regular expression, 42, 50
defining pattern with, 54–58
equivalent, 53
examples, 51–53
- with squaring, 548–550
Regular grammar, 81–83, 196
finite automaton and, 196–200
Regular language, 49, 82, 200
acceptance by finite automaton, 191–193
closure properties of, 200–203
decision procedures for, 209–210
generation by regular grammar, 198–199
pumping lemma for, 205–209
Regular set, 49–50
finite automaton and, 191–193
Relation
adjacency, 32
binary, 11–12
characteristic function of, 299
equivalence, 14–16
 n -ary, 12
Turing computable, 299
Reversal of a string, 45
Rice's Theorem, 371–373
Right invariant equivalence relation, 212
Right-linear grammar, 102, 219
Rightmost derivation, 71
Root, 33
Rule, 65, 326
chain rule, 113
context-free, 65–66, 69
 λ -rule, 69
left-recursive, 70, 129
monotonic, 333
null, 69
of phrase-structure grammar, 326
recursive, 67–68, 70
regular, 81
right recursive, 70, 130
of semi-Thue system, 373
of unrestricted grammar, 326
Russell's paradox, 21–23
- Satisfiability Problem, 472, 481–483
NP-completeness of, 483–492
Savitch's Theorem, 542
Schröder-Bernstein Theorem, 16–17
Search tree, 558–561
Self-reference, 21–23, 363–364
- Semi-Thue system, 373–376
Sentence, 65–68, 70
Sentential form, 70
terminal prefix of, 129
Set, 8–11
cardinality of, 16–21
complement, 9
countable, 7, 17–19
countably infinite, 17–19
denumerable, 17–19
difference, 9
disjoint, 9
empty, 8
equality, 8, 11
infinite, 17
intersection, 9
lookahead, 572, 575, 589
partition, 9
power, 9, 20
proper subset of, 9
regular, 49–50
subset of, 8
uncountable, 7, 17
union, 9
Shift, 556
Simple cycle, 33
Simple induction, 30
Space bounded Turing machine, 534
Space complexity, 532–535
Speedup Theorem, 448–451
Stack
acceptance by empty stack, 230
alphabet, 222
Standard Turing machine, 255–257
State, 145–147
accepting, 147
equivalent, 178
start, 147, 256
State diagram, 146–147
of deterministic finite automaton, 151–153
of multtape machine, 268
of nondeterministic finite automaton, 163
of pushdown automaton, 222–223
of Turing machine, 254
Strictly binary tree, 35–36

- String**, 41–45
 accepted string, 148, 224
 concatenation, 43–44
 derivable, 69
 homomorphism, 219, 257
 languages and, 41, 43
 length, 43
 lookahead, 572
 null, 42
 prefix of, 45
 reversal, 45
 substring, 44–45
 suffix of, 45
Strong induction, 30
Strong LL(k)
 grammar, 579–580
 parser, 587–588
Subset, 8
Subset-Sum Problem, 473,
 509–513
Substring, 44–45
Successful computation, 224
Suffix, 45
Symbol
 nonterminal, 65, 69
 terminal, 65
 useful, 116
 useless, 116
Tape, 147–148, 255–256
 multitrack, 263–265
 output, 282–283
 two-way infinite, 265–268
Tape alphabet, 256
Tape number, 416
Terminal prefix, 129, 557
Terminal symbol, 65
Termination, abnormal, 257
3-conjunctive normal form, 498
3-Satisfiability Problem,
 498–500
 reductions from, 500–513
Time complexity, 442–446
 nondeterministic, 466
 properties of, 451–458
 and representation, 469–471
Token, 553, 567
Top-down parser, 555
 breadth-first, 557–561
 LL(k), 591
 strong LL(k), 587–588
Total function, 13
Tour, 359, 474
Transition function
 of deterministic finite
 automaton, 147
 extended, 151
 input, 170
 multitape, 268
 multitrack, 264
 of NFA- λ , 166
 of nondeterministic finite
 automaton, 163
 of nondeterministic Turing
 machine, 274–275
 of pushdown automaton,
 222–223
 of Turing machine, 256
Transition table, 150
Traveling Salesman Problem,
 517–518
 approximation algorithm for,
 521–523
Tree, 33
 binary, 35
 complete binary, 40
 derivation, 71–74
 frontier of, 35
 ordered, 33–34
 search, 558–561
 strictly binary, 35–36
Truth assignment, 481–482
Turing computable
 function, 296
 relation, 299
Turing machine, 2, 255–257
 abnormal termination of, 296
 acceptance by entering, 263,
 289
 acceptance by final state, 260
 acceptance by halting,
 262–263
 arithmetization of, 416–417
 atomic, 290
 context-sensitive, 290
 halting, 257
 Halting Problem for, 357,
 362–365
 as language acceptor, 259–262
 as language enumerator,
 282–288
 linear speedup, 448–451
multitape machine, 268–274
multitrack machine, 263–265
nondeterministic Turing
 machine, 274–282
 sequential operation of,
 301–302
space complexity, 532–535
standard, 255–257
state diagram, 257
time complexity of, 442–443,
 466
two-way, 265–268
universal, 354–358
Two-way Turing machine,
 265–268
Unary representation, 299
Uncomputable function,
 312–313
Uncountable set, 7, 17
 examples of, 19–20
Undecidable problem, 361
 Blank Tape Problem, 366–368
 for context-free grammars,
 382–386
 Halting Problem, 362–365
 Post Correspondence
 Problem, 377–382
 Word Problem, 373–376
Union of sets, 9
Universal Turing machine,
 354–358
Unrestricted grammar, 254,
 325–332
Useful symbol, 116
Useless symbol, 116
 removal of, 116–121
Variable
 Boolean, 481
 of a grammar, 65, 68–69
 nullable, 107
 reachable, 119
 recursive, 71, 390
Vertex, 32
Vertex Cover Problem, 500–503,
 527
Viable prefix, 599
Well-formed formula, 481
Word Problem, 373–376

x 2150 $\frac{2}{5}$
Word Problem, 373–376

$(\$55.80 + \$3.49)$
 $+ \$35.260$
Feb. '06