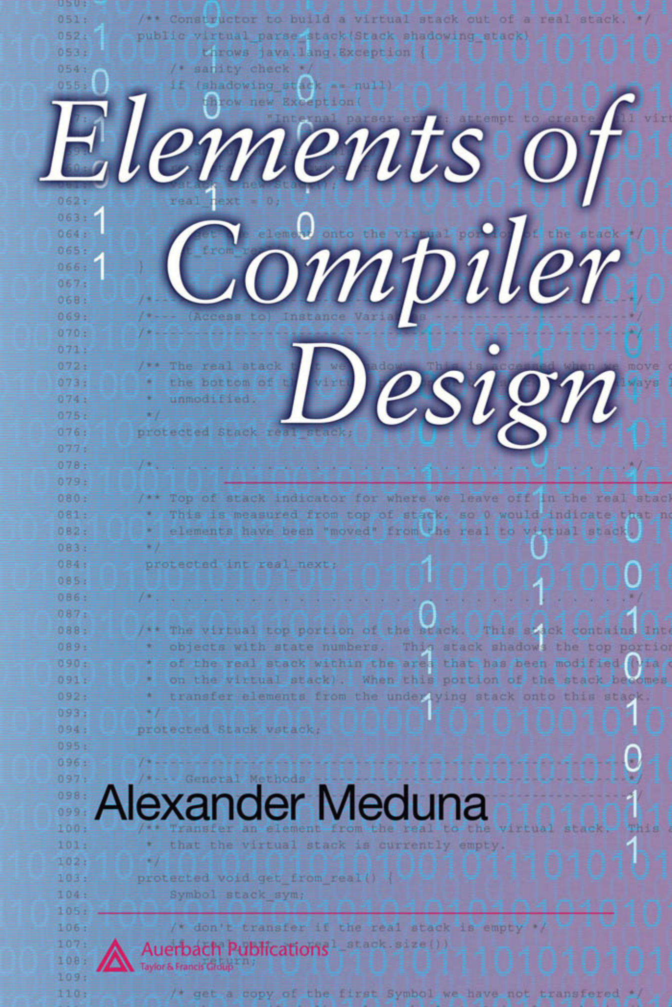
Elements of Compiler Design

Alexander Meduna

Auerbach Publications
Taylor & Francis Group



This page intentionally left blank

Elements of Compiler Design

Elements of Compiler Design

Alexander Meduna

Auerbach Publications
Taylor & Francis Group
Boca Raton New York

Auerbach Publications is an imprint of the
Taylor & Francis Group, an **informa** business

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Version Date: 20110614

International Standard Book Number-13: 978-1-4200-6325-7 (eBook - PDF)

in memory of St. John of the Cross

This page intentionally left blank

Contents

Preface

This book is intended as a text for a one-term introductory course in compiler writing at a senior undergraduate level. It maintains a balance between a theoretical and practical approach to this subject. From a theoretical viewpoint, it introduces rudimental models underlying compilation and its essential phases. Based on these models, it demonstrates the concepts, methods, and techniques employed in compilers. It also sketches the mathematical foundations of compilation and related topics, including the theory of formal languages, automata, and transducers. Simultaneously, from a practical point of view, this book describes how the compiler techniques are implemented. Running throughout the book, a case study designs a new Pascal-like programming language and constructs its compiler; while discussing various methods concerning compilers, the case study illustrates their implementation. Additionally, many detailed examples and computer programs are presented to emphasize the actual applications of the compilation algorithms. Essential software tools are also covered. After studying this book, the student should be able to grasp the compilation process, write a simple real compiler, and follow advanced books on the subject.

From a logical standpoint, the book divides compilation into six cohesive phases. At the same time, it points out that a real compiler does not execute these phases in a strictly consecutive manner; instead, their execution somewhat overlaps to speed up and enhance the entire compilation process as much as possible. Accordingly, the book covers the compilation process phase by phase while simultaneously explaining how each phase is connected during compilation. It describes how this mutual connection is reflected in the compiler construction to achieve the most effective compilation as a whole.

On the part of the student, no previous knowledge concerning compilation is assumed. Although this book is self-contained, in the sense that no other sources are needed for understanding the material, a familiarity with an assembly language and a high-level language, such as Pascal or C, is helpful for quick comprehension. Every new concept or algorithm is preceded by an explanation of its purpose and followed by some examples, computer program passages, and comments to reinforce its understanding. Each complicated material is preceded by its intuitive explanation. All applications are given in a quite realistic way to clearly demonstrate a strong relation between the theoretical concepts and their uses.

In computer science, strictly speaking, every algorithm requires a verification that it terminates and works correctly. However, the termination of the algorithms given in this book is always so obvious that its verification is omitted throughout. The correctness of complicated algorithms is verified in detail. On the other hand, we most often give only the gist of the straightforward algorithms and leave their rigorous verification as an exercise. The text describes the algorithms in Pascal-like notation, which is so simple and intuitive that even the student unfamiliar with Pascal can immediately pick it up. In this description, a Pascal **repeat** loop is sometimes ended with **until no change**, meaning that the loop is repeated until no change can result from its further repetition. As the clear comprehensibility is a paramount importance in the book, the description of algorithms is often enriched by an explanation in words.

Algorithms, conventions, definitions, lemmas, and theorems are sequentially numbered within chapters and are ended with ■. Examples and figures are analogously organized. At the end of each chapter, a set of exercises is given to reinforce and augment the material covered. Selected exercises, denoted by ***Solved*** in the text, have their solutions at the chapter's conclusion. The appendix contains a C++ implementation of a substantial portion of a real compiler. Further backup materials, including lecture notes, teaching tips, homework assignments, errata, exams, solutions, programs, and implementation of compilers, are available at

http://www.fit.vutbr.cz/~meduna/books/eocd

This page intentionally left blank

Acknowledgements

This book is based on lecture notes I have used for my classes about compilers and related computer science topics, such as the automata theory, at various American, European, and Japanese universities over the past three decades. Notes made at the Kyoto Sangyo University in Japan, the National Taiwan University, and the University of Buenos Aires in Argentina were particularly helpful. Nine years I taught compiler writing at the University of Missouri—Columbia in the United States back in the 1990's, and since 2000, I have taught this subject at the Brno University of Technology in the Czech Republic. The lecture notes I wrote at these two universities underlie this book, and I have greatly benefited from conversations with many colleagues and students there. Writing this book was supported by the GACR 201/07/0005 and MSM 0021630528 grants.

My special thanks go to Erzsebet Csuhaj-Varju, Jan Hrouzek, Masami Ito, Miroslav Novotný, Dušan Kolář, Jan Lipowski, and Hsu-Chun Yen for fruitful discussions about compilers and related topics, such as formal languages and their automata. Roman Lukáš carefully read and verified the mathematical passages of this book, and his comments were invaluable. This book would be hardly possible without Zbyněk Křivka, whose help was enormous during its preparation. I am also grateful to Andrea Demby, Katy E. Smith, Jessica Vakili, and John Wyzalek at Taylor and Francis for an excellent editorial and production work. Most importantly, I thank my wife Ivana for her support and love.

A. M.

This page intentionally left blank

About the Author

Alexander Meduna, **PhD**, is a full professor of computer science at the Brno University of Technology in the Czech Republic, where he earned his doctorate in 1988. From 1988 until 1997, he taught computer science at the University of Missouri—Columbia in the United States. Even more intensively, since 2000, he has taught computer science and mathematics at the Brno University of Technology. In addition to these two universities, he has taught computer science at several other American, European, and Japanese universities for shorter periods of time. His classes have been primarily focused on compiler writing. His teaching has also covered various topics including automata, discrete mathematics, formal languages, operating systems, principles of programming languages, and the theory of computation.

Alexander Meduna is the author of *Automata and Languages* (Springer, 2000) and a co-author of the book *Grammars with Context Conditions and Their Applications* (Wiley, 2005). He has published over seventy studies in prominent international journals, such as *Acta Informatica* (Springer), *International Journal of Computer Mathematics* (Taylor and Francis), and *Theoretical Computer Science* (Elsevier). All his scientific work discusses compilers, the subject of this book, or closely related topics, such as formal languages and their models.

Alexander Meduna's Web site is http://www.fit.vutbr.cz/~meduna. His scientific work is described in detail at http://www.fit.vutbr.cz/~meduna/work.

CHAPTER 1

Introduction

In this chapter, we introduce the subject of this book by describing the process of compiling and the components of a compiler. We also define some mathematical notions and concepts in order to discuss this subject clearly and precisely.

Synopsis. We first review the mathematical notions used throughout this text (Section 1.1). Then, we describe the process of compiling and the construction of a compiler (Section 1.2). Finally, we introduce rewriting systems as the fundamental models that formalize the components of a compiler (Section 1.3).

1.1 Mathematical Preliminaries

This section reviews well-known mathematical notions, concepts, and techniques used in this book. Specifically, it reviews sets, languages, relations, translations, graphs, and proof techniques.

Sets and Sequences

A *set*, Σ, is a collection of elements, which are taken from some pre-specified *universe*. If Σ contains an element a, then we symbolically write $a \in \Sigma$ and refer to a as a *member* of Σ. On the other hand, if a is not in Σ, we write $a \notin \Sigma$. The *cardinality* of Σ, *card*(Σ), is the number of Σ's members. The set that has no member is the *empty set*, denoted by $\emptyset$; note that *card*($\emptyset$) = 0. If Σ has a finite number of members, then Σ is a *finite set*; otherwise, Σ is an *infinite set*.

A finite set, Σ, is customarily specified by listing its members; that is, $\Sigma = \{a_1, a_2, \dots, a_n\}$ where a_1 through a_n are all members of Σ. An infinite set, Ω, is usually specified by a property, π, so that Ω contains all elements satisfying π; in symbols, this specification has the following general format $\Omega = \{a| \ \pi(a)\}$. Sets whose members are other sets are usually called *families* of sets rather than sets of sets.

Let Σ and Ω be two sets. Σ is a *subset* of Ω, symbolically written as $\Sigma \subseteq \Omega$, if each member of Σ also belongs to Ω. Σ is a *proper subset* of Ω, written as $\Sigma \subset \Omega$, if $\Sigma \subseteq \Omega$ and Ω contains an element that is not in Σ. If $\Sigma \subseteq \Omega$ and $\Omega \subseteq \Sigma$, Σ *equals* Ω, denoted by $\Sigma = \Omega$. The *power set* of Σ, denoted by *Power*(Σ), is the set of all subsets of Σ.

For two sets, Σ and Ω, their *union*, *intersection*, and *difference* are denoted by $\Sigma \cup \Omega$, $\Sigma \cap \Omega$, and $\Sigma - \Omega$, respectively, and defined as $\Sigma \cup \Omega = \{a| \ a \in \Sigma \text{ or } a \in \Omega\}$, $\Sigma \cap \Omega = \{a| \ a \in \Sigma \text{ and } a \in \Omega\}$, and $\Sigma - \Omega = \{a| \ a \in \Sigma \text{ and } a \notin \Omega\}$. If Σ is a set over a universe U, the *complement* of Σ is denoted by *complement*(Σ) and defined as *complement*(Σ) = $U - \Sigma$. The operations of union, intersection, and complement are related by *DeMorgan's rules* stating that *complement*(*complement*(Σ) $\cup$ *complement*(Ω)) = $\Sigma \cap \Omega$ and *complement*(*complement*(Σ) $\cap$ *complement*(Ω)) = $\Sigma \cup \Omega$, for any two sets Σ and Ω. If $\Sigma \cap \Omega = \emptyset$, Σ and Ω are *disjoint*. More generally, n sets $\Delta_1, \Delta_2, \dots, \Delta_n$, where $n \geq 2$, are *pairwise disjoint* if $\Delta_i \cap \Delta_j = \emptyset$ for all $1 \leq i, j \leq n$ such that $i \neq j$.

A *sequence* is a list of elements from some universe. A sequence is *finite* if it represents a finite list of elements; otherwise, it is *infinite*. The *length of* a finite sequence x, denoted by $|x|$, is the number of elements in x. The *empty sequence*, denoted by ε, is the sequence consisting of no element; that is, $|\varepsilon| = 0$. A finite sequence is usually specified by listing its elements. For instance, consider a finite sequence x specified as $x = (0, 1, 0, 0)$, and observe that $|x| = 4$.

Languages

An *alphabet* Σ is a finite non-empty set, whose members are called *symbols*. Any non-empty subset of Σ is a *subalphabet* of Σ. A finite sequence of symbols from Σ is a *string* over Σ; specifically, ε is referred to as the *empty string*. By Σ^*, we denote the set of all strings over Σ; $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. Let $x \in \Sigma^*$. Like for any sequence, $|x|$ denotes the length of x. For any $a \in \Sigma$, *occur*(x, a) denotes the number of occurrences of as in x, so *occur*(x, a) always satisfies $0 \leq$ *occur*(x, a) $\leq |x|$. Furthermore, if $x \neq \varepsilon$, *symbol*(x, i) denotes the ith symbol in x, where $i = 1, \dots, |x|$. Any subset $L \subseteq \Sigma^*$ is a *language* over Σ. Set *symbol*(L, i) = $\{a|\ a =$ *symbol*(x, i), $x \in L - \{\varepsilon\}, 1 \leq i \leq |x|\}$. Any subset of L is a *sublanguage* of L. If L represents a finite set of strings, L is a *finite language*; otherwise, L is an *infinite language*. For instance, Σ^*, which is called the *universal language* over Σ, is an infinite language while $\emptyset$ and $\{\varepsilon\}$ are finite; noteworthy, $\emptyset \neq \{\varepsilon\}$ because *card*($\emptyset$) = 0 $\neq$ *card*($\{\varepsilon\}$) = 1. Sets whose members are languages are called *families of languages*.

Convention 1.1. In strings, for brevity, we simply juxtapose the symbols and omit the parentheses and all separating commas. That is, we write $a_1a_2\dots a_n$ instead of $(a_1, a_2, \dots, a_n)$.

■

Operations. Let $x, y \in \Sigma^*$ be two strings over an alphabet Σ, and let $L, K \subseteq \Sigma^*$ be two languages over Σ. As languages are defined as sets, all set operations apply to them. Specifically, $L \cup K$, $L \cap K$, and $L - K$ denote the union, intersection, and difference of languages L and K, respectively. Perhaps most importantly, the *concatenation of x with y*, denoted by xy, is the string obtained by appending y to x. Notice that from an algebraic point of view, Σ^* and Σ^+ are the *free monoid* and the *free semigroup*, respectively, generated by Σ under the operation of concatenation. Notice that for every $w \in \Sigma^*$, $w\varepsilon = \varepsilon w = w$. The *concatenation* of L and K, denoted by LK, is defined as $LK = \{xy|\ x \in L, y \in K\}$.

Apart from binary operations, we also make some unary operations with strings and languages. Let $x \in \Sigma^*$ and $L \subseteq \Sigma^*$. The *complement* of L is denoted by *complement*(L) and defined as *complement*(L) = $\Sigma^* - L$. The *reversal of x*, denoted by *reversal*(x), is x written in the reverse order, and the *reversal* of L, *reversal*(L), is defined as *reversal*(L) = $\{$*reversal*(x)$|\ x \in L\}$. For all $i \geq 0$, the *ith power of x*, denoted by x^i, is recursively defined as (1) $x^0 = \varepsilon$, and (2) $x^i = xx^{i-1}$, for $i \geq 1$. Observe that this definition is based on the *recursive definitional method*. To demonstrate the recursive aspect, consider, for instance, the ith power of x^i with $i = 3$. By the second part of the definition, $x^3 = xx^2$. By applying the second part to x^2 again, $x^2 = xx^1$. By another application of this part to x^1, $x^1 = xx^0$. By the first part of this definition, $x^0 = \varepsilon$. Thus, $x^1 = xx^0 = x\varepsilon = x$. Hence, $x^2 = xx^1 = xx$. Finally, $x^3 = xx^2 = xxx$. By using this recursive method, we frequently introduce new notions, including the *ith power of L*, L^i, which is defined as (1) $L^0 = \{\varepsilon\}$ and (2) $L^i = LL^{i-1}$, for $i \geq 1$. The *closure of L*, L^*, is defined as $L^* = L^0 \cup L^1 \cup L^2 \cup \dots$, and the *positive closure of L*, L^+, is defined as $L^+ = L^1 \cup L^2 \cup \dots$. Notice that $L^+ = LL^* = L^*L$, and $L^* = L^+ \cup \{\varepsilon\}$. Let $w, x, y, z \in \Sigma^*$. If $xz = y$, then x is a *prefix* of y; if in addition, $x \notin \{\varepsilon, y\}$, x is a *proper prefix* of y. By *prefixes*(y), we denote the set of all prefixes of y. Set *prefixes*(L) = $\{x|\ x \in$ *prefixes*(y) for some $y \in L\}$. For $i = 0, \dots, |y|$, *prefix*(y, i) denotes y's prefix of length i; notice that *prefix*(y, 0) = ε and *prefix*(y, $|y|$) = y. If $zx = y$, x is a *suffix* of y; if in addition, $x \notin \{\varepsilon, y\}$, x is a *proper suffix* of y. By *suffixes*(y), we denote the set of all suffixes of y. Set *suffixes*(L) = $\{x|\ x \in$ *suffixes*(y) for some $y \in L\}$. For $i = 0, \dots, |y|$, *suffix*(y, i) denotes y's suffix of length i. If $wxz = y$, x is a *substring* of y; if in addition, $x \notin \{\varepsilon, y\}$, x is a *proper substring* of y. By *substrings*(y), we denote the set of all substrings of y. Observe that for all $v \in \Sigma^*$, *prefixes*(v) $\subseteq$ *substrings*(v), *suffixes*(v) $\subseteq$ *substrings*(v), and $\{\varepsilon,$

$v\} \in$ *prefixes*$(v) \cap$ *suffixes*$(v) \cap$ *substrings*(v). Set *substrings*$(L) = \{x|\ x \in$ *substrings*(y) for some $y \in L\}$.

Example 1.1 *Operations.* Consider a *binary alphabet*, {0, 1}. For instance, ε, 1, and 010 are strings over {0, 1}. Notice that $|\varepsilon| = 0$, $|1| = 1$, $|010| = 3$. The concatenation of 1 and 010 is 1010. The third power of 1010 equals 101010101010. Observe that *reversal*(1010) = 0101. String 10 and 1010 are prefixes of 1010. The former is a proper prefix of 1010 whereas the latter is not. We have *prefixes*(1010) = {ε, 1, 10, 101, 1010}. Strings 010 and ε are suffixes of 1010. String 010 is a proper suffix of 1010 while ε is not. We have *suffixes*(1010) = {ε, 0, 10, 010, 1010} and *substrings*(1010) = {ε, 0, 1, 01, 10, 010,101, 1010}.

Set $K = \{0, 01\}$ and $L = \{1, 01\}$. Observe that $L \cup K$, $L \cap K$, and $L - K$ equal to {0, 1, 01}, {01}, and {0}, respectively. The concatenation of K and L is KL = {01, 001, 011, 0101}. For L, *complement*$(L) = \Sigma^* - L$, so every binary string is in *complement*(L) except 1 and 01. Furthermore, *reversal*(L) = {1, 10} and L^2 = {11, 101, 011, 0101}. $L^* = L^0 \cup L^1 \cup L^2 \cup ....$; the strings in L^* that consists of four or fewer symbols are ε, 1, 01, 11, 101, 011, and 0101. $L^+ = L^* - \{\varepsilon\}$. Notice that *prefixes*$(L)$ = {ε, 1, 0, 01}, *suffixes*(L) = {ε, 1, 01}, and *substrings*(L) = {ε, 0, 1, 01}.

■

Relations and Translations

For two object, a and b, (a, b) denotes the *ordered pair* consisting of a and b in this order. Let A and B be two sets. The *Cartesian product* of A and B, $A \times B$, is defined as $A \times B = \{(a, b)|\ a \in A$ and $b \in B\}$. A *binary relation* or, briefly, a *relation*, ρ, from A to B is any subset of $A \times B$; that is, $\rho \subseteq A \times B$. If ρ represents a finite set, then it is a *finite relation*; otherwise, it is an *infinite relation*. The *domain of* ρ, denoted by *domain*(ρ), and the *range of* ρ, denoted by *range*(ρ), are defined as *domain*$(\rho) = \{a|\ (a, b) \in \rho$ for some $b \in B\}$ and *range*$(\rho) = \{b|\ (a, b) \in \rho$ for some $a \in A\}$. If $A = B$, then ρ is a *relation on* Σ. A relation σ is a *subrelation* of ρ if $\sigma \subseteq \rho$. The *inverse of* ρ, denoted by *inverse*(ρ), is defined as *inverse*$(\rho) = \{(b, a)|\ (a, b) \in \rho\}$. A *function* from A to B is a relation ϕ from A to B such that for every $a \in A$, *card*$(\{b|\ b \in B$ and $(a, b) \in \phi)\} \leq 1$. If *domain*$(\phi) = A$, ϕ is *total*; otherwise, ϕ is *partial*. If for every $b \in B$, *card*$(\{a|\ a \in A$ and $(a, b) \in \phi)\} \leq 1$, ϕ is an *injection*. If for every $b \in B$, *card*$(\{a|\ a \in A$ and $(a, b) \in \phi)\} = 1$, ϕ is a *surjection*. If ϕ is both a surjection and an injection, ϕ represents a *bijection*.

Convention 1.2. Instead of $(a, b) \in \rho$, we often write $b \in \rho(a)$ or $a\rho b$; in other words, $(a, b) \in \rho$, $a\rho b$, and $a \in \rho(b)$ are used interchangeably. If ρ is a function, we usually write $\rho(a) = b$.

■

Let A be a set, ρ be a relation on A, and $a, b \in A$. For $k \geq 1$, the *k-fold product* of ρ, ρ^k, is recursively defined as (1) $a\rho^1 b$ if and only if $a\rho b$, and (2) $a\rho^k b$ if and only if there exists $c \in A$ such that $a\rho c$ and $c\rho^{k-1}$b, for $k \geq 2$. Furthermore, $a\rho^0$b if and only if $a = b$. The *transitive closure* of ρ, ρ^+, is defined as $a\rho^+ b$ if and only if $a\rho^k$b, for some $k \geq 1$, and the *reflexive and transitive closure* of ρ, ρ^*, is defined as $a\rho^* b$ if and only if $a\rho^k$b, for some $k \geq 0$.

Let K and L be languages over alphabets T and U, respectively. A *translation* from K to L is a relation σ from T^* to U^* with *domain*$(\sigma) = K$ and *range*$(\sigma) = L$. A total function τ from T^* to *Power*(U^*) such that $\tau(uv) = \tau(u)\tau(v)$ for every $u, v \in T^*$ is a *substitution* from T^* to U^*. By this definition, $\tau(\varepsilon) = \{\varepsilon\}$ and $\tau(a_1a_2\ldots a_n) = \tau(a_1)\tau(a_2)\ldots\tau(a_n)$, where $a_i \in T$, $1 \leq i \leq n$, for some $n \geq 1$, so τ is completely specified by defining $\tau(a)$ for every $a \in T$.

A total function υ from T^* to U^* such that $\upsilon(uv) = \upsilon(u)\upsilon(v)$ for every $u, v \in T^*$ is a *homomorphism* from T^* to U^*. As any homomorphism is obviously a special case of a substitution, we simply specify υ by defining $\upsilon(a)$ for every $a \in T$. If for every $a, b \in T$, $\upsilon(a) = \upsilon(b)$ implies $a = b$, υ is an *injective homomorphism*.

Example 1.2 *Polish Notation*. There exists a useful way of representing ordinary *infix arithmetic expressions* without using parentheses. This notation is referred to as *Polish notation*, which has two fundamental forms—*postfix* and *prefix notation*. The former is defined recursively as follows.

Let Ω be a set of binary operators, and let Σ be a set of operands.

1. Every $a \in \Sigma$ is a postfix representation of a.
2. Let AoB be an infix expression, where $o \in \Omega$, and A, B are infix expressions. Then, CDo is the postfix representation of AoB, where C and D are the postfix representations of A and B, respectively.
3. Let C be the postfix representation of an infix expression A. Then, C is the postfix representation of (A).

Consider the infix expression $(a + b) * c$. The postfix expression for c is c. The postfix expression for $a + b$ is $ab+$, so the postfix expression for $(a + b)$ is $ab+$, too. Thus the postfix expression for $(a + b) * c$ is $ab+c*$.

The prefix notation is defined analogically except that in the second part of the definition, o is placed in front of AB; the details are left as an exercise.

To illustrate homomorphisms and substitutions, set $\Xi = \{0, 1, \dots, 9\}$ and $\Psi = (\{A, B, \dots, Z\} \cup \{|\})$ and consider the homomorphism h from Ξ^* to Ψ^* defined as $h(0) = ZERO|$, $h(1) = ONE|$, …, $h(9) = NINE|$. For instance, h maps 91 to $NINE|ONE|$. Notice that h is an injective homomorphism. Making use of h, define the infinite substitution s from Ξ^* to Ψ^* as $s(x) = \{h(x)\}\{|\}^*$. As a result, $s(91) = \{NINE|\}\{|\}^*\{ONE|\}\{|\}^*$, including, for instance, $NINE|ONE|$ and $NINE||||ONE||$.

■

Graphs

Let A be a set. A *directed graph* or, briefly, a *graph* is a pair $G = (A, \rho)$, where ρ is a relation on A. Members of A are called *nodes*, and ordered pairs in ρ are called *edges*. If $(a, b) \in \rho$, then edge (a, b) *leaves* a and *enters* b. Let $a \in A$; then, the *in-degree* of a and the *out-degree* of a are $card(\{b| (b, a) \in \rho\})$ and $card(\{c| (a, c) \in \rho\})$. A sequence of nodes, $(a_0, a_1, \dots, a_n)$, where $n \geq 1$, is a *path of length n* from a_0 to a_n if $(a_{i-1}, a_i) \in \rho$ for all $1 \leq i \leq n$; if, in addition, $a_0 = a_n$, then $(a_0, a_1, \dots, a_n)$ is a *cycle of length n*. In this book, we frequently *label* G's edges with some attached information. Pictorially, we represent $G = (A, \rho)$ so we draw each edge $(a, b) \in \rho$ as an arrow from a to b possibly with its label as illustrated in the next example.

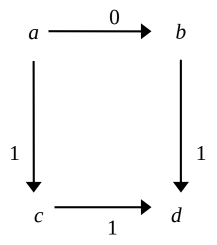
$$\begin{tikzcd}
a \arrow[r] \arrow[d] & b \arrow[d] \\
c \arrow[r] & d
\end{tikzcd}$$

Figure 1.1 *Graph*.

Example 1.3 *Graphs*. Consider a program p and its *call graph* $G = (P, \rho)$, where P represents the set of subprograms in p, and $(x, y) \in \rho$ if and only if subprogram x calls subprogram y. Specifically, let $P = \{a, b, c, d\}$, and $\rho = \{(a, b), (a, c), (b, d), (c, d)\}$, which says that a calls b and c, b calls d, and c calls d as well (see Figure 1.1). The in-degree of a is 0, and its out-degree is 2.

Notice that (a, b, d) is a path of length 2 in G. G contains no cycle because none of its paths starts and ends in the same node.

Suppose we use G to study the value of a global variable during the four calls. Specifically, we want to express that this value is zero when call (a, b) occurs; otherwise, it is one. Pictorially, we express this by labeling G's edges in the way given in Figure 1.2.



$$\begin{tikzcd} a \arrow[r, "0"] \arrow[d, "1"'] & b \arrow[d, "1"] \\ c \arrow[r, "1"'] & d \end{tikzcd}$$

Figure 1.2 *Labeled Graph.*

■

Let $G = (A, \rho)$ be a graph. G is an *acyclic graph* if it contains no cycle. If $(a_0, a_1, \ldots, a_n)$ is a path in G, then a_0 is an *ancestor* of a_n and a_n is a *descendent* of a_0; if in addition, $n = 1$, then a_0 is a *direct ancestor* of a_n and a_n a *direct descendent* of a_0. A *tree* is an acyclic graph $T = (A, \rho)$ such that A contains a specified node, called the *root* of T and denoted by *root*(T), and every $a \in A -$ *root*(T) is a descendent of *root*(A) and its in-degree is one. If $a \in A$ is a node whose out-degree is 0, a is a *leaf*; otherwise, it is an *interior node*. In this book, a tree T is always considered as an *ordered tree* in which each interior node a $\in A$ has all its direct descendents, b_1 through b_n, where $n \geq 1$, ordered from the left to the right so that b_1 is the leftmost direct descendent of a and b_n is the rightmost direct descendent of a. At this point, a is the *parent* of its *children* b_1 through b_n, and all these nodes together with the edges connecting them, (a, b_1) through (a, b_n), are called a *parent-children portion of T*. The *frontier* of T, denoted by *frontier*(T), is the sequence of T's leaves ordered from the left to the right. The *depth* of T, *depth*(T), is the length of the longest path in T. A tree $S = (B, \upsilon)$ is a *subtree of T* if $\emptyset \subset B \subseteq A$, $\upsilon \subseteq \rho \cap (B \times B)$, and in T, no node in $A - B$ is a descendent of a node in B. Like any graph, a tree T can be described as a two-dimensional structure. Apart from this two-dimensional representation, however, it is frequently convenient to specify T by a one-dimensional representation, $\Re(T)$, in which each subtree of T is represented by the expression appearing inside a balanced pair of $\langle$ and $\rangle$ with the node which is the root of that subtree appearing immediately to the left of $\langle$. More precisely, $\Re(T)$ is defined by the following recursive rules to T:

1. If T consists of a single node a, then $\Re(T) = a$.
2. Let (a, b_1) through (a, b_n), where $n \geq 1$, be the parent-children portion of T, *root*(T) $= a$, and T_k be the subtree rooted at b_k, $1 \leq k \leq n$, then $\Re(T) = a\langle\Re(T_1)\ \Re(T_2) \ldots \Re(T_n)\rangle$.

Apart from a one-dimensional representation of T, we sometimes make use of a *postorder of T*'s nodes, denoted by *postorder*(T), obtained by recursively applying the next procedure **POSTORDER**, starting from *root*(T).

POSTORDER: Let **POSTORDER** be currently applied to node a.
- If a is an interior node with children a_1 through a_n,
  recursively apply **POSTORDER** to a_1 through a_n, then list a;
- if a is a leaf, list a and halt.

Convention 1.3. Graphically, we draw a tree T with its root on the top and with all edges directed down. Each parent has its children drawn from the left to the right according to its ordering. Drawing T in this way, we always omit all arrowheads. When T is actually implemented, ☞T denotes the pointer to T's root. Regarding T's one-dimensional representation, if *depth*(T) = 0 and, therefore, $\Re(T)$ consists of a single leaf a, we frequently point this out by writing **leaf** a rather than a plain a. Throughout this book, we always use $\Re$ as a one-dimensional representation of trees.

■

Example 1.4 ***Trees***. Graph G discussed in Figure 1.2 is acyclic. However, it is no tree because the in-degree of node d is two. By removing edge (b, d), we obtain a tree $T = (P, \tau)$, where $P = \{a, b, c, d\}$ and $\tau = \{(a, b), (a, c), (c, d)\}$. Nodes a and c are interior nodes while b and d are leaves. The root of T is a. We define b and c as a's first child and a's second child, respectively. A parent-children portion of T is, for instance, (a, b) and (a, c). Notice that *frontier*(T) = bd, and *depth*(T) = 2. T's one-dimensional representation $\Re(T) = a\langle bc\langle d\rangle\rangle$, and *postorder*($T$) = $bdca$. Its subtrees are $a\langle bc\langle d\rangle\rangle$, $c\langle d\rangle$, b, and d. For clarity, we usually write the one-leaf subtrees b and d as **leaf** b and **leaf** d, respectively. In Figure 1.3, we pictorially give $a\langle bc\langle d\rangle\rangle$ and $c\langle d\rangle$.

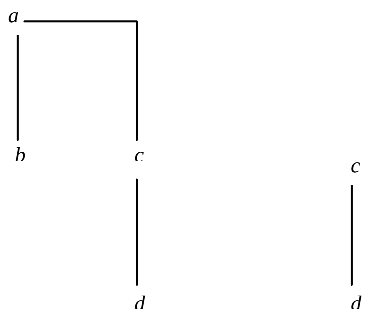


Figure 1.3 ***Tree and Subtree.***

■

Proofs

Next, we review the basics of elementary logic. We pay a special attention to the fundamental proof techniques used in this book.

In general, a *formal mathematical system S* consists of basic *symbols*, *formation rules*, *axioms*, and *inference rules*. Basic symbols, such as constants and operators, form components of *statements*, which are composed according to formation rules. Axioms are primitive statements, whose validity is accepted without justification. By inference rules, some statements infer other statements. A *proof* of a statement s in S consists of a sequence of statements $s_1, \ldots, s_i, \ldots, s_n$ such that $s = s_n$ and each s_i is either an axiom of S or a statement inferred by some of the statements $s_1, \ldots, s_{i-1}$ according to the inference rules; s proved in this way represents a *theorem* of S.

Logical connectives join statements to create more complicated statements. The most common logical connectives are *not*, *and*, *or*, *implies*, and *if and only if*. In this list, *not* is unary while the other connectives are binary. That is, if s is a statement, then *not* s is a statement as well. Similarly, if s_1 and s_2 are statements, then s_1 *and* s_2, s_1 *or* s_2, s_1 *implies* s_2, and s_1 *if and only if* s_2 are statements, too. We often write $\wedge$ and $\vee$ instead of *and* and *or*, respectively. The following *truth table* presents the rules governing the *truth*, denoted by 1, or *falsehood*, denoted by 0, concerning statements connected by the binary connectives. Regarding the unary connective *not*, if s is true, then *not* s is false, and if s is false, then *not* s is true.

s_1	s_2	*and*	*or*	*implies*	*if and only if*
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	0
1	1	1	1	1	1

Figure 1.4 ***Truth Table.***

By this table, s_1 *and* s_2 is true if both statements are true; otherwise, s_1 *and* s_2 is false. Analogically, we can interpret the other rules governing the truth or falsehood of a statement containing the other connectives from this table. A statement of *equivalence*, which has the form s_1 *if and only if* s_2, plays a crucial role in this book. A proof that it is true usually consists of two parts. The *only-if part* demonstrates that s_1 *implies* s_2 is true while the *if part* proves that s_2 *implies* s_1 is true. There exist many logic laws useful to demonstrate that an implication is true. Specifically, the *contrapositive law* says (s_1 *implies* s_2) *if and only if* ((*not* s_2) *implies* (*not* s_1)), so we can prove s_1 *implies* s_2 by demonstrating that (*not* s_2) *implies* (*not* s_1) holds true. We also often use a *proof by contradiction* based upon the law saying ((*not* s_2) *and* s_1) *implies* 0 is true. Less formally, if from the assumption that s_2 is false and s_1 is true, we obtain a false statement, s_1 *implies* s_2 is true.

Example 1.5 ***Proof by Contradiction.*** Let P be the set of all primes (a natural number n is prime if its only positive divisors are 1 and n). By contradiction, we next prove that P is infinite. That is, assume that P is finite. Set $k = card(P)$. Thus, P contains k numbers, $p_1, p_2, \ldots, p_k$. Set $n = p_1p_2{\ldots}p_k + 1$. Observe that n is not divisible by any p_i, $1 \leq i \leq k$. As a result, either n is a new prime or n equals a product of new primes. In either case, there exists a prime out of P, which contradicts that P contains all primes. Thus, P is infinite.

■

A *proof by induction* demonstrates that a statement s_i is true for all integers $i \geq b$, where b is a non-negative integer. In general, a proof of this kind is made in this way:

Basis. Prove that s_b is true.
Inductive Hypothesis. Suppose that there exists an integer n such that $n \geq b$ and s_m is true for all $b \leq m \leq n$.
Inductive Step. Prove that s_{n+1} is true under the assumption that the inductive hypothesis holds.

Example 1.6 ***Proof by Induction.*** Consider statement s_i as

$$1 + 3 + 5 + ... + 2i - 1 = i^2$$

for all $i \geq 1$. In other words, s_i states that the sum of odd integers is a perfect square. An inductive proof of this statement follows next.

Basis. As $1=1^2$, s_1 is true.
Inductive Hypothesis. Assume that s_m is true for all $1 \leq m \leq n$, where n is a natural number.
Inductive Step. Consider

$$s_{n+1} = 1 + 3 + 5 + ... + (2n - 1) + (2(n + 1) - 1) = (n + 1)^2.$$

By the inductive hypothesis, $s_n = 1 + 3 + 5 + ... + (2n - 1) = n^2$. Hence,

$$1 + 3 + 5 + ... + (2n - 1) + (2(n + 1) - 1) = n^2 + 2n + 1 = (n + 1)^2.$$

Consequently, s_{n+1} holds, and the inductive proof is completed.

■

1.2 Compilation

A *compiler* reads a *source program* written in a *source language* and translates this program into a *target program* written in a *target language* so that both programs are functionally equivalent—that is, they specify the same computational task to perform. As a rule, the source language is a high-level language, such as Pascal or C, while the target language is the machine language of a particular computer or an assembly language, which is easy to transform to the machine language. During the translation, the compiler first *analyzes* the source program to verify that the source program is correctly written in the source language. If so, the compiler generates the target program; otherwise, the compiler reports the errors and unsuccessfully ends the translation.

Compilation Phases

In greater detail, the compiler first makes the lexical, syntax, and semantic analysis of the source program. Then, from the information gathered during this threefold analysis, it generates the intermediate code of the source program, makes its optimization, and creates the resulting target code. As a whole, the *compilation* thus consists of these six *compilation phases*, each of which transforms the source program from one inner representation to another:

- lexical analysis
- syntax analysis
- semantic analysis
- intermediate code generation
- optimized intermediate code generation
- target code generation

Lexical analysis breaks up the source program into *lexemes*—that is, logically cohesive lexical entities, such as identifiers or integers. It verifies that these entities are well-formed, produces *tokens* that uniformly represent lexemes in a fixed-sized way, and sends these tokens to the syntax analysis. If necessary, the tokens are associated with attributes to specify them in more detail. The lexical analysis recognizes every single lexeme by its *scanner*, which reads the sequence of characters that make up the source program to recognize the next portion of this sequence that forms the lexeme. Having recognized the lexeme in this way, the lexical analysis creates its tokenized representation and sends it to the syntax analysis.

Syntax analysis determines the syntax structure of the tokenized source program, provided by the lexical analysis. This compilation phase makes use of the concepts and techniques developed by modern mathematical linguistics. Indeed, the source-language syntax is specified by *grammatical rules*, from which the syntax analysis constructs a *parse*—that is, a sequence of rules that generates the program. The way by which a *parser*, which is the syntax-analysis component responsible for this construction, works is usually explained graphically. That is, a parse is displayed as a *parse tree* whose leaves are labeled with the tokens and each of its parent-children portion forms a *rule tree* that graphically represents a rule. The parser constructs this tree by smartly selecting and composing appropriate rule trees. Depending on the way it makes this construction, we distinguish two fundamental types of parsers. A *top-down parser* builds the parse tree from the root and proceeds down toward the frontier while a *bottom-up parser* starts from the frontier and works up toward the root. If the parser eventually obtains a complete parse tree for the source program, it not only verifies that the program is syntactically correct but also obtains its syntax structure. On the other hand, if this tree does not exist, the source program is syntactically erroneous.

Semantic analysis checks that the source program satisfies the semantic conventions of the source language. Perhaps most importantly, it performs type checking to verify that each operator has operands permitted by the source-language specification. If the operands are not permitted, this compilation phase takes an appropriate action to handle this incompatibility. That is, it either indicates an error or makes type coercion, during which the operands are converted so they are compatible.

Intermediate code generation turns the tokenized source program to a functionally equivalent program in a uniform intermediate language. As its name indicates, this language is at a level intermediate between the source language and the target language because it is completely independent of any particular machine code, but its conversion to the target code represents a relatively simple task. The intermediate code fulfills a particularly important role in a *retargetable* compiler, which is adapt or retarget for several different computers. Indeed, an installation of a compiler like this on a specific computer only requires the translation of the intermediate code to the computer's machine code while all the compiler part preceding this simple translation remains unchanged.

As a matter of fact, this generation usually makes several conversions of the source program from one internal representation to another. Typically, this compilation phase first creates the *abstract syntax tree*, which is easy to generate by using the information obtained during syntax analysis. Indeed, this tree compresses the essential syntactical structure of the parse tree. Then, the abstract syntax tree is transformed to the *three-address code*, which represents every single source-program statement by a short sequence of simple instructions. This kind of representation is particularly convenient for the optimization.

Optimized intermediate code generation or, briefly, *optimization* reshapes the intermediate code so it works in a more efficient way. This phase usually involves numerous subphases, many of which are applied repeatedly. It thus comes as no surprise that this phase slows down the translation significantly, so a compiler usually allows optimization to be turned off.

In greater detail, we distinguish two kinds of optimizations—*machine-independent optimization* and *machine-dependent optimization*. The former operates on the intermediate code while the latter is applied to the target code, whose generation is sketched next.

Target code generation maps the optimized intermediate representation to the target language, such as a specific assembly language. That is, it translates this intermediate representation into a sequence of the assembly instructions that perform the same task. As obvious, this generation requires detailed information about the target machine, such as memory locations available for each variable used in the program. As already noted, the optimized target code generation attempts to make this translation as economically as possible so the resulting instructions do not waste space or time. Specifically, considering only a tiny target-code fragment at a time, this optimization shortens a sequence of target-code instructions without any functional change by some simple improvements. Specifically, it eliminates useless operations, such as a load of a value into a register when this value already exists in another register.

All the six compilation phases make use of error handler and symbol table management, sketched next.

Error Handler. The three analysis phases can encounter various errors. For instance, the lexical analysis can find out that the upcoming sequence of numeric characters represents no number in the source language. The syntax analysis can find out that the tokenized version of the source program cannot be parsed by the grammatical rules. Finally, the semantic analysis may detect an incompatibility regarding the operands attached to an operator. The error handler must be able to detect any error of this kind. After issuing an error diagnostic, however, it must somehow recover from the error so the compiler can complete the analysis of the entire source program. On the other hand, the error handler is no mind reader, so it can hardly figure out what the author actually meant by an erroneous passage in the source program code. As a result, no mater how sophisticatedly the compiler handles the errors, the author cannot expect that a compiler turns an erroneous program to a properly coded source program. Therefore, no generation of the target program follows the analysis of an erroneous program.

Symbol table management is a mechanism that associates each identifier with relevant information, such as its name, type, and scope. Most of this information is collected during the analysis; for instance, the identifier type is obtained when its declaration is processed. This mechanism assists almost every compilation phase, which can obtain the information about an identifier whenever needed. Perhaps most importantly, it provides the semantic analyzer with information to check the source-program semantic correctness, such as the proper declaration of identifiers. Furthermore, it aids the proper code generation. Therefore, the symbol-table management must allow the compiler to add new entries and find existing entries in a speedy and effective way. In addition, it has to reflect the source-program structure, such as identifier scope in nested program blocks. Therefore, a compiler writer should carefully organize the symbol-table so it meets all these criteria. Linked lists, binary search trees, and hash tables belong to commonly used symbol-table data structures.

Convention 1.4. For a variable *x*, ☞*x* denotes a pointer to the symbol-table entry recording the information about *x* throughout this book.

■

Case Study 1/35 FUN *Programming Language.* While discussing various methods concerning compilers in this book, we simultaneously illustrate how they are used in practice by designing a new Pascal-like programming language and its compiler. This language is called FUN because it is particularly suitable for the computation of mathematical *fun*ctions.

In this introductory part of the case study, we consider the following trivial FUN program that multiplies an integer by two. With this source program, we trace the six fundamental compilation phases described above. Although we have introduced all the notions used in these phases quite informally so far, they should be intuitively understood in terms of this simple program.

```
program DOUBLE;
{This FUN program reads an integer value and multiplies it by two.}

integer u;

begin
    read(u);
    u = u * 2;
    write(u);
end.
```

Lexical analyzer divides the source program into lexemes and translates them into tokens, some of which have attributes. In general, an attributed token has the form $t\{a\}$, where t is a token and a

represents *t*'s attribute that provides further information about *t*. Specifically, the FUN lexical analyzer represents each identifier *x* by an attributed token of the form *i*{☞*x*}, where *i* is the token specifying an identifier as a generic type of lexemes and the attribute ☞*x* is a pointer to the symbol-table entry that records all needed information about this particular identifier *x*, such as its type. Furthermore, #{*n*} is an attributed token, where # represents an integer in general and its attribute *n* is the integer value of the integer in question. Next, we give the tokenized version of program DOUBLE, where | separates the tokens of this program. Figure 1.5 gives the symbol table created for DOUBLE's identifiers.

program | *i*{☞DOUBLE} | ; | ***integer*** | *i*{☞*u*} | ; | ***begin*** | ***read*** | (| *i*{☞*u*} |) | ; | *i*{☞*u*} | = | *i*{☞*u*} | * | #{2} | ; | ***write***| (| *i*{☞*u*} |) | ***end*** | .

Name	***Type***	…
DOUBLE		
u	***integer***	
⋮		

Figure 1.5 ***Symbol Table.***

Syntax analyzer reads the tokenized source program from left to right and verifies its syntactical correctness by grammatical rules. Graphically, this grammatical verification is expressed by constructing a parse tree, in which each parent-children portion represents a rule. This analyzer works with tokens without any attributes, which play no role during the syntax analysis. In DOUBLE, we restrict our attention just to the expression *i*{☞*u*} * #{2}, which becomes *i* * # without the attributes. Figure 1.6 gives the parse tree for this expression.

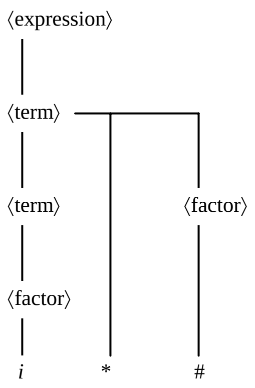


Figure 1.6 ***Parse Tree.***

Semantic analyzer checks semantic aspects of the source program, such as type checking. In DOUBLE, it consults the symbol table to find out that *u* is declared as **integer**.

Intermediate code generator produces the intermediate code of DOUBLE. First, it implements its syntax tree (see Figure 1.7).

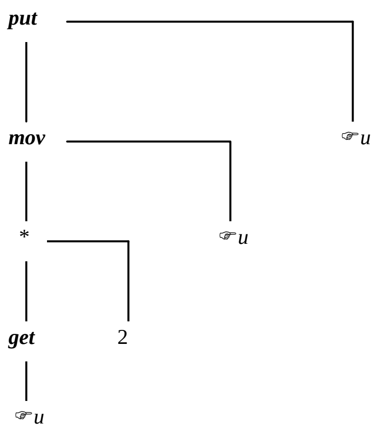


Figure 1.7 ***Syntax Tree.***

Then, it transforms this tree to the following three-address code, which makes use of a temporary variable *t* produced by the compiler. The ***get*** instruction moves the input integer value into *u*. The ***mul*** instruction multiplies the value of *u* by 2 and sets *t* to the result of this multiplication. The ***mov*** instruction moves the value of *t* to *u*. Finally, the ***put*** instruction prints the value of *u* out.

[***get***, , , ☞*u*]
[***mul***, ☞*u*, 2, ☞*t*]
[***mov***, ☞*t*, , ☞*u*]
[***put***, , , ☞*u*]

Optimizer reshapes the intermediate code to perform the computational task more efficiently. Specifically, in the above three-address program, it replaces ☞*t* with ☞*u*, and removes the third instruction to obtain this shorter one-variable three-address program

[***get***, , , ☞*u*]
[***mul***, ☞*u*, 2, ☞ *u*]
[***put***, , , ☞*u*]

Target code generator turns the optimized three-address code into a target program, which performs the computational task that is functionally equivalent to the source program. Of course, like the previous optimizer, the target code generator produces the target program code as succinctly as possible. Specifically, the following hypothetical assembly-language program, which is functionally equivalent to DOUBLE, consists of three instructions and works with a single register, *R*. First, instruction *GET R* reads the input integer value into *R*. Then, instruction *MUL R*, 2 multiplies the contents of *R* by 2 and places the result back into *R*, which the last instruction *PUT R* prints out.

```
GET    R
MUL    R, 2
PUT    R
```

■

Compiler Construction

The six fundamental compilation phases—lexical analysis, syntax analysis, semantic analysis, intermediate code generation, optimization, and target code generation—are abstracted from the translation process made by a real compiler, which does not execute these phases strictly consecutively. Rather, their execution somewhat overlaps in order to complete the whole compilation process as fast as possible (see Figure 1.8). Since the source-program syntax structure represents probably the most important information to the analysis as a whole, the syntax analyzer guides the performance of all the analysis phases as well as the intermediate code generator. Indeed, the lexical analyzer goes into operation only when the syntax analyzer requests the next token. The syntax analyzer also calls the semantic analysis routines to make their semantic-related checks. Perhaps most importantly, the syntax analyzer directs the intermediate code generation actions, each of which translates a bit of the tokenized source program to a functionally equivalent portion of the intermediate code. This *syntax-directed translation* is based on grammatical rules with associated actions over *attributes* attached to symbols occurring in these rules to provide the intermediate code generator with specific information needed to produce the intermediate code. For instance, these actions generate some intermediate code operations with operands addressed by the attributes. When this generation is completed, the resulting intermediate code usually contains some useless or redundant instructions, which are removed by a machine-independent optimizer. Finally, in a close cooperation with a machine-dependent optimizer, the target code generator translates the optimized intermediate program into the target program and, thereby, completes the compilation process.

Passes. Several compilation phases may be grouped into a single *pass* consisting of reading an internal version of the program from a file and writing an output file. As passes obviously slow down the translation, *one-pass compilers* are usually faster than *multi-pass compilers*. Nevertheless, some aspects concerning the source language, the target machine, or the compiler design often necessitate an introduction of several passes. Regarding the source language, some questions raised early in the source program may remain unanswered until the compiler has read the rest of the program. For example, there may exist references to procedures that appear later in the source code. Concerning the target machine, unless there is enough memory available to hold all the intermediate results obtained during compilation, these results are stored into a file, which the compiler reads during a subsequent pass. Finally, regarding the compiler design, the compilation process is often divided into two passes corresponding to the two ends of a compiler as explained next.

Ends. The *front end* of a compiler contains the compilation portion that heavily depends on the source language and has no concern with the target machine. On the other hand, the *back end* is primarily dependent on the target machine and largely independent of the source language. As a result, the former contains all the three analysis phases, the intermediate code generation, and the machine-independent optimization while the latter includes the machine-dependent optimization and the target code generator. In this two-end way, we almost always organize a retargetable compiler. Indeed, to adapt it for various target machines, we use the same front end and only redo its back end as needed. On the other hand, to obtain several compilers that translate different programming languages to the same target language, we use the same back end with different front ends.

Compilation in Computer Context. To sketch where the compiler fits into the overall context of writing and executing programs, we sketch the computational tasks that usually precede or follow a compilation process.

Before compilation, a source program may be stored in several separate files, so a preprocessor collects them together to create a single source program, which is subsequently translated as a whole.

After compilation, several post-compilation tasks are often needed to run the generated program on computer. If a compiler generates assembly code as its target language, the resulting target program is translated into the machine code by an assembler. Then, the resulting machine code is usually linked together with some library routines, such as numeric functions, character string operations, or file handling routines. That is, the required library services are identified, *loaded* into memory, and *linked* together with the machine code program to create an executable code (the discussion of linkers and loaders is beyond the scope of this book). Finally, the resulting executable code is placed in memory and executed, or by a specific request, this code is stored on a disk and executed later on.

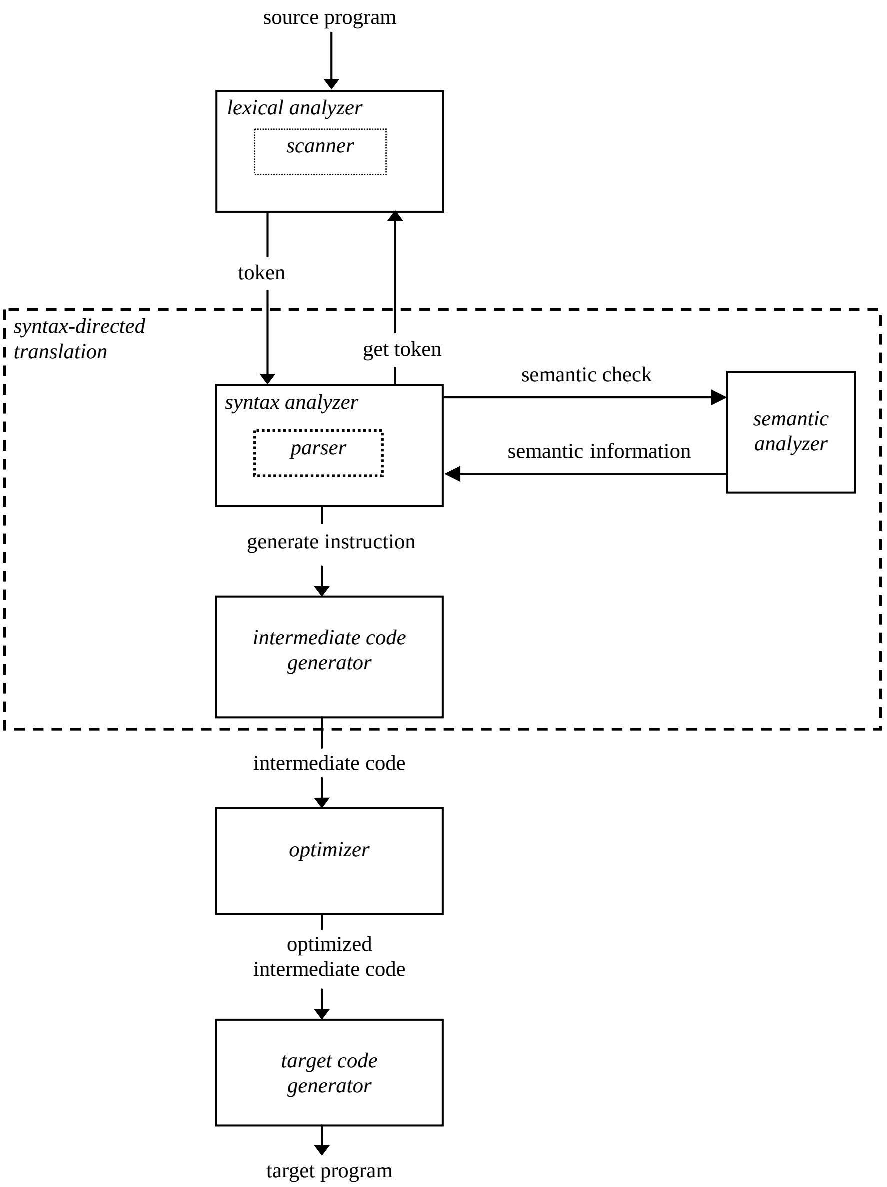


Figure 1.8 *Compiler Construction.*

1.3 Rewriting Systems

As explained in the previous section, each compilation phase actually transforms the source program from one compiler inner representation to another. In other words, it *rewrites* a string that represents an inner form of the source program to a string representing another inner form that is closer to the target program, and this rewriting is obviously ruled by an algorithm. It is thus only natural to formalize these phases by rewriting systems, which are based on finite many rules that abstractly represent the algorithms according to which compilation phases are performed.

Definition 1.5 ***Rewriting System.*** A *rewriting system* is a pair, $M = (\Sigma, R)$, where Σ is an alphabet, and R is a finite relation on Σ^*. Σ is called the *total alphabet of M* or, simply, *M*'s *alphabet*. A member of R is called a *rule of M*, so R is referred to as *M*'s *set of rules*.

■

Convention 1.6. Each rule $(x, y) \in R$ is written as $x \rightarrow y$ throughout this book. For brevity, we often denote $x \rightarrow y$ with a label r as r: $x \rightarrow y$, and instead of r: $x \rightarrow y \in R$, we sometimes write $r \in R$. For r: $x \rightarrow y \in R$, x and y represent r's *left-hand side*, denoted by ***lhs***(r), and r's *right-hand side*, denoted by ***rhs***(r), respectively. R^* denotes the set of all *sequences of rules* from R; as a result, by $\rho \in R^*$, we briefly express that ρ is a sequence consisting of $|\rho|$ rules from R. By analogy with strings (see Convention 1.1), in sequences of rules, we simply juxtapose the rules and omit the parentheses as well as all separating commas in them. That is, if $\rho = (r_1, r_2, \dots, r_n)$, we simply write ρ as $r_1r_2\dots r_n$. To explicitly express that Σ and R represent the components of M, we write ${}_M\Sigma$ and ${}_MR$ instead of Σ and R, respectively.

■

Definition 1.7 ***Rewriting Relation.*** Let $M = (\Sigma, R)$ be a rewriting system. The *rewriting relation* over Σ^* is denoted by $\Rightarrow$ and defined so that for every $u, v \in \Sigma^*$, $u \Rightarrow v$ in M if and only if there exist $x \rightarrow y \in R$ and $w, z \in \Sigma^*$ such that $u = wxz$ and $v = wyz$.

■

Let $u, v \in \Sigma^*$. If $u \Rightarrow v$ in M, we say that *M directly rewrites u to v*. As usual, for every $n \geq 0$, the n-fold product of $\Rightarrow$ is denoted by $\Rightarrow^n$. If $u \Rightarrow^n v$, *M rewrites u to v in n steps*. Furthermore, the transitive-reflexive closure and the transitive closure of $\Rightarrow$ are $\Rightarrow^*$ and $\Rightarrow^+$, respectively. If $u \Rightarrow^* v$, we simply say that *M rewrites u to v*, and if $u \Rightarrow^+ v$, *M rewrites u to v in a nontrivial way*. In this book, we sometimes need to explicitly specify the rules used during rewriting. Suppose M makes $u \Rightarrow v$ so that $u = wxz$, $v = wyz$ and M replaces x with y by applying r: $x \rightarrow y \in R$. To express this application, we write $u \Rightarrow v$ $[r]$ or, in greater detail, $wxz \Rightarrow wyz$ $[r]$ in M and say that *M directly rewrites uxv to uyv by r*. More generally, let n be a non-negative integer, $w_0, w_1, \dots, w_n$ be a sequence with $w_i \in \Sigma^*$, $0 \leq i \leq n$, and $r_j \in R$ for $1 \leq j \leq n$. If $w_{j-1} \Rightarrow w_j$ $[r_j]$ in M for $1 \leq j \leq n$, *M rewrites w_0 to w_n in n steps by $r_1r_2\dots r_n$*, symbolically written as $w_0 \Rightarrow^n w_n$ $[r_1r_2\dots r_n]$ in M ($n = 0$ means $w_0 \Rightarrow^0 w_0$ $[\varepsilon]$). By $u \Rightarrow^* v$ $[\rho]$, where $\rho \in R^*$, we express that M makes $u \Rightarrow^* v$ by using ρ; $u \Rightarrow^+ v$ $[\rho]$ has an analogical meaning. Of course, whenever the specification of applied rules is superfluous, we omit it and write $u \Rightarrow v$, $u \Rightarrow^n v$, and $u \Rightarrow^* v$ for brevity.

Language Models

The language constructs used during some compilation phases, such as the lexical and syntax analysis, are usually represented by formal languages defined by a special case of rewriting systems, customarily referred to as *language-defining models* underlying the phase. Accordingly, the compiler parts that perform these phases are usually based upon algorithms that implement the corresponding language models.

Convention 1.8. Throughout this book, the language defined by a model M is denoted by $L(M)$. In an algorithm that implements M working with a string, w, we write **ACCEPT** to announce that $w \in L(M)$ while **REJECT** means $w \notin L(M)$.

■

By the investigation of the language models, we obtain a systematized body of knowledge about the compiler component that performs the compilation phase, and making use of this valuable knowledge, we try to design the component as sophisticatedly as possible. In particular, the language models underlying the phases that are completely independent of the target machine, such as the analysis phases, allow us to approach these phases in a completely general and rigorous way. This approach to the study of compilation phases has become so common and fruitful that it has given rise to several types of models, some of which define the same languages. We refer to the models that define the same language as *equivalent models*, and from a broader perspective, we say that some types of models are *equally powerful* if the family of languages defined by models of each of these types is the same.

Synopsis of the Book

Specifically, regarding the compilation process discussed in this book, the lexical analysis is explained by equally powerful language-defining models called *finite automata* and *regular expressions* (see Chapter 2). The syntax analysis is expressed in terms of equally powerful *pushdown automata* and *grammars* (see Chapters 3 through 5), and the syntax-directed translation is explained by *attribute grammars* (see Chapter 6), which represent an extension of the grammars underlying the syntax analysis. The optimization and the target code generation are described without any formal models in this introductory book (see Chapter 7). In its conclusion, we suggest the contents of an advanced course about compilers following a class based upon the present book (see Chapter 8). In the appendix, we demonstrate how to implement a real compiler.

Exercises

1.1. Let $L = \{a^n | \ n \geq 2\}$ be a language over an alphabet, Σ. Determine *complement*(L) for $\Sigma = \{a\}$, $\Sigma = \{a, b\}$, and $\Sigma = \{a, b, \ldots, \mathrm{z}\}$.

1.2. By using the notions introduced in Section 1.1, such as various language operations, define integer and real numbers in your favorite programming language.

1.3. Let Σ be a subset of the set of all non-negative integers, and let ϕ be the total function from the set of all non-negative integers to $\{0, 1\}$ defined by this equivalence $\phi(i) = 1$ if and only if $i \in \Sigma$, for all non-negative integers, i. Then, ϕ is the *characteristic function* of Σ. Illustrate this notion in terms of formal languages.

1.4. Let $X = \{i | \ i \text{ is a positive integer}\} \cup \{and, are, even, Integers, odd, respectively\}$. Describe the language of all well-constructed English sentences consisting of X's members, commas and a period. For instance, this language contains *Integers* 2 *and* 5 *are even and odd*, *respectively*. Is this language infinite? Can you define this language by using X and operations concatenation, union, and iteration?

1.5. Let Σ and Ω be two sets, and let ρ and ρ' be two relations from Σ to Ω. If ρ and ρ' represent two identical subsets of $\Sigma \times \Omega$, then ρ *equals* ρ', symbolically written as $\rho = \rho'$. Illustrate this definition in terms of language translations.

1.6. Let Σ be a set, and let ρ be a relation on Σ. Then,

- if for all $a \in \Sigma$, $a\rho a$, then ρ is *reflexive*;
- if for all $a, b \in \Sigma$, $a\rho b$ implies $b\rho a$, then ρ is *symmetric*;
- if for all $a, b \in \Sigma$, ($a\rho b$ and $b\rho a$) implies $a = b$, then ρ is *antisymmetric*;
- if for all $a, b, c \in \Sigma$, ($a\rho b$ and $b\rho c$) implies $a\rho c$, then ρ is *transitive*.

Illustrate these notions in terms of language translations.

1.7. Let $\Sigma = \{1, 2, \ldots, 8\}$. Consider the following binary relations over Σ:

- $\emptyset$;
- $\{(1, 3), (3, 1), (8, 8)\}$;
- $\{(1, 1), (2, 2), (8, 8)\}$;
- $\{(a, a) | a \in \Sigma\}$;
- $\{(a, b) | a, b \in \Sigma, a < b\}$;
- $\{(a, b) | a, b \in \Sigma, a \leq b\}$;
- $\{(a, b) | a, b \in \Sigma, a + b = 9\}$;
- $\{(a, b) | a, b \in \Sigma$, b is divisible by $a\}$;
- $\{(a, b) | a, b \in \Sigma$, $a - b$ is divisible by $3\}$.

Note that a is divisible by b if there exists a positive integer c such that $a = bc$. For each of these relations, determine whether it is reflexive, symmetric, antisymmetric, or transitive.

1.8. Let Σ be a set, and let ρ be a relation on Σ. If ρ is reflexive, symmetric, and transitive, then ρ is an *equivalence relation*. Let ρ be an equivalence relation on Σ. Then, ρ partitions Σ into disjoint subsets, called *equivalence classes*, so that for each $a \in \Sigma$, the equivalence class of a is denoted by $[a]$ and defined as $[a] = \{b | a\rho b\}$. Prove that for all a and b in Σ, either $[a] = [b]$ or $[a] \cap [b] = \emptyset$.

1.9*Solved*. Let Σ be a set, and let ρ be a relation on Σ. If ρ is reflexive, antisymmetric, and transitive, then ρ is a *partial order*. If ρ is a partial order satisfying for all $a, b \in \Sigma$, $a\rho b$, $b\rho a$, or $a = b$, then ρ is a *linear order*. Formalize the usual dictionary order as a *lexicographic order* based upon a linear order.

1.10. Write a program that implements the lexicographic order constructed in Exercise 1.9. Test this program on a large file of English words.

1.11. Let Σ be a set. Define the relation ρ on *Power*(Σ) as $\rho = \{(A, B) | A, B \in Power(\Sigma), A \subseteq B\}$ (see Section 1.1 for *Power*(Σ)). Prove that ρ represents a partial order.

1.12. By induction, prove that for any set Σ, $card(Power(\Sigma)) = 2^{card(\Sigma)}$.

1.13*Solved*. Prove Theorem 1.9, given next.

Theorem 1.9. Let Σ be a set, ρ be a relation on Σ, and ρ^+ be the transitive closure of ρ. Then, (1) ρ^+ is a transitive relation, and (2) if ρ' is a transitive relation such that $\rho \subseteq \rho'$, then $\rho^+ \subseteq \rho'$.

1.14. Prove Theorem 1.10, given next.

Theorem 1.10. Let Σ be a set, ρ be a relation on Σ, and ρ^* be the transitive and reflexive closure of ρ. Then, (1) ρ^* is a transitive and reflexive relation, and (2) if ρ' be a transitive and reflexive relation such that $\rho \subseteq \rho'$, then $\rho^* \subseteq \rho'$.

1.15. Generalize the notion of a binary relation to the notion of an *n*-ary relation, where *n* is a natural number.

1.16*Solved*. Define the prefix Polish notation.

1.17*Solved*. In Section 1.1, we translate the infix expression $(a + b) * c$ into the postfix notation $ab{+}c{*}$. Translate $(a + b) * c$ into the prefix notation. Describe this translation in a step-by-step way.

1.18. Consider the one-dimensional representation of trees, $\mathfrak{R}$ (see Section 1.1). Prove that the prefix Polish notation is the same as $\mathfrak{R}$ with parentheses deleted.

1.19. Design a one-dimensional representation for trees, different from $\mathfrak{R}$.

1.20. Introduce the notion of an *n*-ary function, where *n* is a natural number. Illustrate this notion in terms of compilation.

1.21. Consider the directed graphs defined and discussed in Section 1.1. Intuitively, *undirected graphs* are similar to these graphs except that their edges are undirected. Define them rigorously.

1.22. Recall that Section 1.1 has defined a tree as an acyclic graph, $G = (\Sigma, \rho)$, satisfying these three properties:

(1) *G* has a specified node whose in-degree is 0; this node represents the *root* of *G*, denoted by *root*(*G*).
(2) If $a \in \Sigma$ and $a \neq root(G)$, then *a* is a descendent of *root*(*G*) and the in-degree of *a* is 1.
(3) Each node, $a \in \Sigma$, has its direct descendents, b_1 through b_n, ordered from the left to the right so that b_1 is the leftmost direct descendent of *a* and b_n is the rightmost direct descendent of *a*.

Reformulate (3) by using the notion of a partial order, defined in Exercise 1.9.

1.23. A *tautology* is a statement that is true for all possible truth values of the statement variables. Explain why every theorem of a formal mathematical system represents a tautology.

1.24. Prove that the contrapositive law represents a tautology. State a theorem and prove this theorem by using the contrapositive law.

1.25. A *Boolean algebra* is a formal mathematical system, which consists of a set, Σ, and operations *and*, *or*, and *not*. The axioms of Boolean algebra follow next.

Associativity. (1) *a or* (*b or c*) = (*a or b*) *or c*, and
(2) *a and* (*b and c*) = (*a and b*) *and c*, for all $a, b, c \in \Sigma$.
Commutativity. (1) *a or b* = *b or a*, and
(2) *a and b* = *b and a*, for all $a, b \in \Sigma$.
Distributivity. (1) *a and* (*b or c*) = (*a and b*) *or* (*a and c*), and
(2) *a or* (*b and c*) = (*a or b*) *and* (*a or c*), for all $a, b \in \Sigma$.

In addition, Σ contains two distinguished members, 0 and 1, which satisfy these laws for all $a \in \Sigma$,

(1) a *or* $0 = a$, (2) a *and* $1 = a$, (3) a *or* (*not* a) = 1, and (4) a *and* (*not* a) = 0.

The rule of inference is substitution of equals for equals. Discuss the Boolean algebra in which Σ's only members are 0 and 1, representing falsehood and truth, respectively. Why is this algebra important to the mathematical foundations of computer science?

1.26. Consider your favorite programming language, such as Pascal or C. Define its lexical units by the language operations introduced in Section 1.1. Can the syntax be defined in the same way? Justify your answer.

1.27. Learn the *syntax diagram* from a good high-level programming language manual. Design a simple programming language and describe its syntax by these diagrams.

1.28*Solved*. Recall that a rewriting system is a pair, $M = (\Sigma, R)$, where Σ is an alphabet, and R is a finite relation on Σ^* (see Definition 1.5). Furthermore, the rewriting relation over Σ^* is denoted by $\Rightarrow$ and defined so that for every $u, v \in \Sigma^*$, $u \Rightarrow v$ in M if and only if there exist $x \rightarrow y \in R$ and $w, z \in \Sigma^*$ such that $u = wxz$ and $v = wyz$ (see Definition 1.7). For every $n \geq 0$, the n-fold product of $\Rightarrow$ is denoted by $\Rightarrow^n$. Determine a non-negative integer, $m \geq 0$, satisfying for all $u, v \in \Sigma^*$, if $u \Rightarrow^n v$ in M, then $|v| \leq nm|u|$.

Solutions to Selected Exercises

1.9. Let Σ be a set, and let β be a linear order on Σ. We extend β to Σ^* so that for any $x, y \in \Sigma^*$, $x \beta y$ if $x \in prefixes(y) - \{y\}$, or for some $k \geq 1$ such that $|x| > k$ and $|y| > k$, $prefix(x, k - 1) = prefix(y, k - 1)$ and $symbol(x, k) \beta symbol(y, k)$. This extended definition of β is referred to as the *lexicographic order* β on Σ^*. Take, for instance, Σ as the English alphabet and β as its alphabetical order. Then, the lexical order β extended in the above way represents the usual dictionary order on Σ^*.

1.13. To demonstrate that Theorem 1.9 holds, we next prove that (1) and (2) hold true.

(1) To prove that ρ^+ is a transitive relation, we demonstrate that if $a\rho^+b$ and $b\rho^+c$, then $a\rho^+c$. As $a\rho^+b$, there exist $x_1, \ldots, x_n$ in Σ so $x_1\rho x_2, \ldots, x_{n-1}\rho x_n$, where $x_1 = a$ and $x_n = b$. As $b\rho^+c$, there also exist $y_1, \ldots, y_m$ in Σ so $y_1\rho y_2, \ldots, y_{m-1}\rho y_m$, where $y_1 = b$ and $y_m = c$. Consequently, $x_1\rho x_2, \ldots, x_{n-1}\rho x_n, y_1\rho y_2, \ldots, y_{m-1}\rho y_m$, where $x_1 = a$, $x_n = b = y_1$, and $y_m = c$. As a result, $a\rho^+c$.

(2) We demonstrate that if ρ' is a transitive relation such that $\rho \subseteq \rho'$, then $\rho^+ \subseteq \rho'$. Less formally, this implication means that ρ^+ is the smallest transitive relation that includes ρ. Let ρ' be a transitive relation such that $\rho \subseteq \rho'$, and let $a\rho^+b$. Then, there exist $x_1, \ldots, x_n$ in Σ so $x_1\rho x_2, \ldots, x_{n-1}\rho x_n$, where $x_1 = a$ and $x_n = b$. As $\rho \subseteq \rho'$, $x_1\rho' x_2, \ldots, x_{n-1}\rho' x_n$ where $x_1 = a$ and $x_n = b$. Because ρ' is transitive, $a\rho' b$. Consequently, $a\rho^+b$ implies $a\rho' b$.

1.16. The prefix notation is defined recursively as follows. Let Ω be a set of binary operators, and let Σ be a set of operands.

- Every $a \in \Sigma$ is a prefix representation of a.
- Let AoB be an infix expression, where $o \in \Omega$, and A, B are infix expressions. Then, oCD is the prefix representation of AoB, where C and D are the prefix representations of A and B, respectively.
- Let C be the prefix representation of an infix expression A. Then, C is the prefix representation of (A).

1.17. The expression $(a + b) * c$ is of the form $A * B$ with $A = (a + b)$ and $B = c$. The prefix notation for B is c. The prefix notation for A is $+ab$. Thus the prefix expression for $(a + b) * c$ is $*+abc$.

1.28. Take m as any non-negative integer satisfying $m \geq |y| - |x|$, for all $x \rightarrow y \in R$.

CHAPTER 2

Lexical Analysis

Upon receiving a request for the next token from the syntax analyzer, the lexical analyzer's scanner reads the stream of characters forming a source program from left to right in order to recognize a lexeme—that is, the next instance of meaningful lexical elements, such as identifiers or integers. After recognizing this next lexeme and verifying its lexical correctness, the lexical analyzer produces a token that represents the recognized lexeme in a simple and uniform way. Having fulfilled its task, it sends the newly produced token to the syntax analysis and, thereby, satisfies its request.

Besides this fundamental task, the lexical analyzer usually fulfils several minor tasks. Specifically, the lexical analyzer usually closely and frequently cooperates with the symbol table handler to store or find an identifier in the symbol table whenever needed. In addition, it performs some trivial tasks simplifying the source-program text, such as case conversion or removal of the superfluous passages, including comments and white spaces.

Synopsis. Section 2.1 discusses the fundamental language models underlying the lexical analysis. Making use of these models, Section 2.2 describes methods and techniques used in this analysis. Finally, Section 2.3 presents the theoretical background of lexical analysis.

2.1 Models

This chapter describes these fundamental models behind lexical analysis

- regular expressions
- finite automata and finite transducers

Regular expressions represent simple language-denoting formulas, based upon the operations of concatenation, union, and iteration. These expressions are used to specify programming language lexemes. Finite automata are language-accepting devices used to recognize lexemes. Based on these automata, finite transducer represents language-translating models that not only recognize lexemes but also translate them to the corresponding tokens.

Regular Expressions

To allow computer programmers to denote their lexical units as flexibly as possible, every high-level programming language offers them a wide variety of lexemes. These lexemes are usually specified by regular expressions, defined next.

Definition 2.1 ***Regular Expressions*.** Let Δ be an alphabet. The *regular expressions* over Δ and the *languages that these expressions denote* are defined recursively as follows:

- $\varnothing$ is a regular expression denoting the empty set;
- ε is a regular expression denoting $\{\varepsilon\}$;
- a, where $a \in \Delta$, is a regular expression denoting $\{a\}$;
- if r and s are regular expressions denoting the languages R and S, respectively, then
  $(r|s)$ is a regular expression denoting $R \cup S$;

(rs) is a regular expression denoting RS;
(r^*) is a regular expression denoting R^*.

The languages denoted by regular expressions are customarily called the *regular languages*.

■

In practice, we frequently simplify the *fully parenthesized regular expressions* created strictly by this definition by reducing the number of parentheses in them. To do so, we assume that * has higher precedence than operation concatenation, which has higher precedence than |. In addition, the expression rr^* is usually written as r^+. These simplifications make regular expressions more succinct; for instance, $((a)^*(b.(b)^*))$ is written as a^*b^+ under these conventions.

As the next two examples illustrate, most programming language lexemes are specified by regular expressions, some of which may contain several identical subexpressions. Therefore, we often give names to some simple regular expressions so that we can define more complex regular expressions by using these names, which then actually refer to the subexpressions they denote.

Case Study 2/35 ***Lexemes.*** The FUN lexemes are easily and elegantly specified by regular expressions. In Figure 2.1, its identifiers, defined as arbitrarily long alphanumerical words beginning with a letter, and real numbers are specified.

Lexemes	*Regular Expression*
letter	A|…|Z|a|…|z
digit	0|…|9
digits	$digit^+$
identifier	$letter(digit\|letter)^*$
real number	*digits*(ε|.*digits*)(ε|E(ε | + | –)*digits*)

Figure 2.1 ***Lexemes Specified by Regular Expressions.***

Observe that *digits*(ε|.*digits*)(ε|E(ε | + | –)*digits*) is a convenient equivalent to $(0|\ldots|9)^+(\varepsilon|.(0|\ldots|9)^+)(\varepsilon|E(\varepsilon\,|\,+\,|\,-)(0|\ldots|9)^+)$. Notice that the language denoted by this expression includes 1.2E+45, so 1.2E+45 is a correct FUN real number; however, 1.2E+ is not correct because it is not in this language (no digit follows +). A complete set of FUN lexemes is defined by regular expressions in Case Study 3/35.

■

There exist a number of important language properties and algebraic laws obeyed by regular expressions. These properties and laws significantly simplify manipulation with these expressions and, thereby, specification of lexemes. We discuss them in Section 2.3.

Finite Automata

Next, we discuss several variants of finite automata as the fundamental models of lexical analyzers. We proceeded from quite general variants towards more restricted variants of these automata. The general variants represent mathematically convenient models, which are difficult to apply in practice though. On the other hand, the restricted variants are easy to use in practice, but their restrictions make them inconvenient from a theoretical point of view. More specifically, first we study finite automata that can change states without reading input symbols. Then, we rule out changes of this kind and discuss finite automata that read a symbol during every computational step. In general, these automata work non-deterministically because with the same symbol, they can make several different steps from the same state. As this non-determinism obviously complicates the implementation of lexical analyzers, we pay a special attention to deterministic finite automata, which disallow different steps from the same state with the same symbol. All these variants have the same power, so we can always use any of them without any loss of generality. In fact, later in Section 2.3.2, we present algorithms that convert the general versions of finite automata to their equivalent restricted versions.

Definition 2.2 ***Finite Automaton and its Language.*** A *finite automaton* is a rewriting system, $M = (\Sigma, R)$, where

- Σ is divided into two pairwise disjoint subalphabets Q and Δ;
- R is a finite *set of rules* of the form $qa \to p$, where $q, p \in Q$ and $a \in \Delta \cup \{\varepsilon\}$.

Q and Δ are referred to as the *set of states* and the *alphabet of input symbols*, respectively. Q contains a state called the *start state*, denoted by s, and a *set of final states*, denoted by F. Like in any rewriting system, we define $u \Rightarrow v$, $u \Rightarrow^n v$ with $n \geq 0$, and $u \Rightarrow^* v$, where $u, v \in \Sigma^*$ (see Definition 1.5). If $sw \Rightarrow^* f$ in M, where $w \in \Delta^*$, *M accepts w*. The set of all strings that M accepts is the *language accepted by M*, denoted by $L(M)$.

■

Convention 2.3. For any finite automaton, M, we automatically assume that Σ, Δ, Q, s, F, and R denote M's total alphabet, the alphabet of input symbols, the set of states, the start state, the set of final states, and the set of rules, respectively. If there exists any danger of confusion, we mark Σ, Δ, Q, s, F, and R with M as ${}_M\Sigma$, ${}_M\Delta$, ${}_MQ$, ${}_Ms$, ${}_MF$, and ${}_MR$, respectively, in order to explicitly relate these components to M (in particular, we make these marks when several automata are simultaneously discussed).

Symbols in the input alphabet are usually represented by early lowercase letters a, b, c, and d while states are usually denoted by s, f, o, p, and q.

■

Taking advantage of Convention 2.3, we now explain Definition 2.2 informally. Consider a *configuration*, qv, where $q \in Q$ and $v \in \Delta^*$. This configuration actually represents an instantaneous description of M. Indeed, q is the current state and u represents the remaining suffix of the *in*put *s*tring, symbolically denoted by *ins*. Let r: $qa \to p \in R$, where $q, p \in Q$, $a \in \Delta \cup \{\varepsilon\}$, and $y \in \Delta^*$. By using this rule, M directly rewrites qay to qy, which is usually referred to as a *move* from qay to qy. If in this way M makes a sequence of moves from $sw \Rightarrow^* f$, where $w \in \Delta^*$, then M accepts w. The set of all strings accepted by M is the language of M, denoted by $L(M)$. To describe the acceptance of w symbol by symbol, suppose that $w = a_1 \ldots a_n$ with each $a_j \in \Delta$, $1 \leq j \leq n$, for some $n \geq 1$. M reads w from left to right by performing moves according to its rules. During a move, by using a rule of the form $qa \to p$, M changes its current state q to p while reading a as its current input symbol. Suppose that M has already made a sequence of moves during which it has read the first $i - 1$ symbols of w, so M's current input symbol is $a_i \in \Delta$. Furthermore, suppose that its current state is q. At this point, M can make a move according to a rule of the form $q \to p$ or $qa_i \to p$. According to a rule of the form $q \to p$, M only changes its current state q to p. According to a rule of the form $qa_i \to p$, M changes q to p and, in addition, reads a_i, so a_{i+1} becomes the current input symbol in the next move. If in this way M reads $a_1 \ldots a_n$ by making a sequence of moves from the start state to a final state, M accepts $a_1 \ldots a_n$; otherwise, M rejects $a_1 \ldots a_n$. $L(M)$ consists of all strings M accepts in this way.

Convention 2.4. In examples, we often describe a finite automaton by simply listing its rules. If we want to explicitly state that s is the start state in such a list of rules, we mark it with ‣ as ‣s. To express that f is a final state, we mark f with ▪ as ▪f.

■

Example 2.1 ***Finite Automaton and its Language.*** Let us first informally describe a finite automaton M so that $L(M) = \{a\}^*(\{b\}^+ \cup \{c\}^+)\{a\}^*$. In other words, M accepts every input word of the form $w = w_1w_2w_3$, where $w_1, w_3 \in \{a\}^*$ and $w_2 \in \{b\}^+ \cup \{c\}^+$; for example, $abba \in L(M)$, but $abca \notin L(M)$. In essence, we construct M that works with $w_1w_2w_3$ as follows. In its start state s, M reads w_1. From s, M can enter either state p or q without reading any input symbol. In p, M reads w_2 consisting of bs whereas in q, M reads w_2 consisting of cs. From p, M can go to its final state f with b while from q, M enters f with c. In f, M reads w_3 and, thereby, completes the acceptance of $w_1w_2w_3$. To summarize this construction, we see that M's states are s, p, q, and f, where s is the start state and f is the final state. Its input symbols are a, b, and c. Starting from s, M makes its steps according to the following eight rules.

rule 1: with a, stay in s
rule 2: without reading any symbol, change s to p
rule 3: with b, stay in p
rule 4: with b, change p to f
rule 5: without reading any symbol, change s to q
rule 6: with c, stay in q
rule 7: with c, change q to f
rule 8: with a, stay in f

By rule 1, M reads w_1. By rules 2 through 4, M reads $w_2 \in \{b\}^+$ while by rules 5 through 7, it reads $w_2 \in \{c\}^+$. Finally, by rule 8, M reads w_3.

Formally, $M = (\Sigma, R)$, where $\Sigma = Q \cup \Delta$, $\Delta = \{a, b, c\}$, $Q = \{s, f, p, q\}$, where s is the start state, and $\{f\}$ is the set of final states. Consider rule 1 above. Mathematically, we specify this rule as $sa \to s$. Analogously, we obtain $R = \{sa \to s, s \to p, pb \to p, pb \to f, s \to q, qc \to q, qc \to f, fa \to f\}$. Under Convention 2.4, we can succinctly specify M as

1: $\blacktriangleright sa \to \blacktriangleright s$
2: $\blacktriangleright s \to p$
3: $pb \to p$
4: $pb \to \blacksquare f$
5: $\blacktriangleright s \to q$
6: $qc \to q$
7: $qc \to \blacksquare f$
8: $\blacksquare fa \to \blacksquare f$

Consider $sabba$. By using rules 1, 2, 3, 4, and 8, M makes $\blacktriangleright sabba \Rightarrow^* \blacksquare f$, so M accepts $abba$. A rigorous verification that $L(M) = \{a\}^*(\{b\}^+ \cup \{c\}^+)\{a\}^*$ is left as an exercise.

■

Representations of Finite Automata

Tabularly, a finite automaton $M = (\Sigma, R)$ is represented by its *state table*, denoted by *M-state-table*. Its columns are denoted with the members of $\Delta \cup \{\varepsilon\}$, and the rows are denoted with the states of Q. For each $q \in Q$ and each $a \in \Delta \cup \{\varepsilon\}$, entry *M-state-table*$[q, a] = \{p \mid qa \to p \in R\}$. For brevity, we omit the braces in the sets of *M-state-table*'s entries; a blank entry means $\emptyset$.

Graphically, a finite automaton $M = (\Sigma, R)$ is represented by its *state diagram*, *M-state-diagram*. *M-state-diagram* is a labeled directed graph such that each node is labeled with a state $q \in Q$ and for two nodes $q, p \in Q$, there is an edge (q, p) labeled with $\{a \mid qa \to p \in R, a \in \Delta \cup \{\varepsilon\}\}$. For simplicity, we omit every edge labeled with $\emptyset$ in *M-state-diagram*.

Example 2.2 ***Tabular and Graphical Representation.*** Consider the finite automaton *M*, discussed in the previous parts of this example. *M-state-table* and *M-state-diagram* are in Figure 2.2 and Figure 2.3, respectively.

M-state-table	*a*	*b*	*c*	ε
▸*s*	▸*s*			*p*, *q*
p		*p*, ▪*f*		
q			*q*, ▪*f*	
▪*f*	▪*f*			

Figure 2.2 ***State Table.***

$$\begin{tikzcd}
{\blacktriangleright}s \arrow[loop left, "a"] \arrow[r, "\varepsilon"] \arrow[d, "\varepsilon"] & p \arrow[loop right, "b"] \arrow[d, "b"] \\
q \arrow[loop left, "c"] \arrow[r, "c"] & {\blacksquare}f \arrow[loop right, "a"]
\end{tikzcd}$$

Figure 2.3 ***State Diagram.***

■

Simplification

Next, we introduced some simplified versions of finite automata that are easy to implement and use in practice. As demonstrated in Section 2.3.2, every finite automaton can be algorithmically converted to any of the following simplified versions so that they are equivalent, meaning they accept the same language (see Section 1.3).

Elimination of ε*-moves*. In general, finite automata may contain some ε*-rules* that have only a state on their left-hand side. According to these rules, they make ε*-moves* during which they change states without reading any input symbol. As a result, with some strings, they may perform infinitely many sequences of moves. For example, consider the finite automaton having these two rules ▸*s* → ▸*s* and *sa* → ▪*f*. The language accepted by this automaton equals {*a*}; however, the automaton accepts *a* by infinitely many sequences of moves. The next definition rules out ε-rules in finite automata. As finite automata without ε-rules read an input symbol during every computational step, with any input string, *w*, they make a finitely many sequences of moves. In fact, each of these sequences is of length |*w*| or less, and this property obviously simplifies the use of finite automata.

Definition 2.5 ***Finite Automaton without*** **ε*-Rules.*** Let *M* = (Σ, *R*) be a finite automaton. *M* is a *finite automaton without* ε*-rule* if for every *r* ∈ *R*, ***lhs***(*r*) ∈ *Q*Δ.

■

Example 2.3 ***Finite Automaton without ε-Rules.*** The finite automaton in Example 2.2 has two ε-rules (see Figure 2.3). Figure 2.4 presents another finite automaton without ε-rules. As both automata accept $\{a\}^*(\{b\}^+ \cup \{c\}^+)\{a\}^*$, they are equivalent.

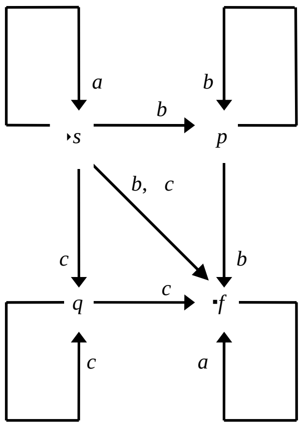


Figure 2.4 ***Finite automaton without ε-rules.***

■

Determinism. Even if finite automata contain no ε-rule, they can still make several different direct computations from the same configuration. As a result, with the same input string, they can make various sequences of moves, and this non-deterministic behavior naturally complicates their implementation. The next definition introduces the deterministic version of finite automata that make no more than one move from any configuration. As a result, with any input string, *w*, they make a unique sequence of moves.

Definition 2.6 ***Deterministic Finite Automaton.*** Let $M = (\Sigma, R)$ be a finite automaton without ε-rules. *M* is a *deterministic finite automaton* if for every $q \in Q$ and every $a \in \Delta$, $card(\{\textbf{rhs}(r) | r \in R, \textbf{lhs}(r) = qa\}) \leq 1$.

■

That is, a finite automaton without ε-rule $M = (\Sigma, R)$ is deterministic if for every $q \in Q$ and every $a \in \Delta$, there exists no more than one $p \in Q$ such that $qa \to p$ is in *R*.

Example 2.4 ***Deterministic Finite Automaton.*** Consider the finite automaton without ε-rules from Example 2.3 (see Figure 2.4). This automaton works non-deterministically: for instance, from configuration *sbb*, it can make a move to *pb* or to ▪*fb* with *b*. Notice that this automaton is equivalent to the deterministic finite automaton depicted in Figure 2.5, which also accepts $\{a\}^*(\{b\}^+ \cup \{c\}^+)\{a\}^*$.

■

Complete specification. A deterministic finite automaton may lack rules for some states or input symbols; at this point, it may not complete reading some input words. For instance, consider the deterministic finite automaton in Figure 2.5. Let *bc* be its input string. After reading *b*, the automaton enters *p* in which it gets stuck because it has no rule with *p* and *c* on the left-hand side.

Definition 2.7 ***Completely Specified Deterministic Finite Automaton.*** Let $M = (\Sigma, R)$ be a deterministic finite automaton. M is *completely specified* if for every $q \in Q$ and every $a \in \Delta$, $card(\{\mathbf{rhs}(r) | r \in R, \mathbf{lhs}(r) = qa\}) = 1$.

■

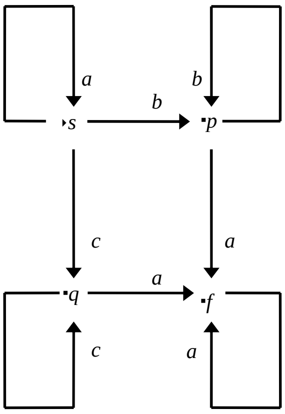


Figure 2.5 ***Deterministic finite automaton.***

Completely specified deterministic finite automata read all symbols of every input string, and this property is often appreciated when these automata are implemented. Indeed, we next present two algorithms that implement completely specified deterministic finite automata. The first algorithm actually implements the state table of a completely specified deterministic finite automaton based on its state table. The other algorithm is based on a nested **case** statement. Both algorithms assume that a given input string w is followed by ◀, which thus acts as an *input end marker*. As their output, the algorithm announces **ACCEPT** if $w \in L(M)$ and **REJECT** if $w \notin L(M)$ (see Convention 1.8).

Algorithm 2.8 ***Implementation of a Finite Automaton—Tabular Method.***

Input • a completely specified deterministic finite automaton, $M = (\Sigma, R)$ with $Q = \{1, \dots, n\}$, $\Delta = \{a_1, \dots, a_m\}$, and $s = 1$;
• w◀ with $w \in \Delta^*$.

Output • **ACCEPT** if $w \in L(M)$, and **REJECT** if $w \notin L(M)$.

Method

```
type
    States = 1..n;
    InputSymbols = 'a_1'.. 'a_m';
    Rules = array[States, InputSymbols] of States;
    StateSet = set of States;

var
    State: States;
    InputSymbol: InputSymbols;
    Rule: Rules;
    FinalStates: StateSet;
```

```
begin

    for $i$ := 1 to $n$ do                    {initialization of Rule according to $R$}
        for $j$ := 1 to $m$ do
            if $ia_j \to k \in R$ then Rule[$i$, '$a_j$'] := $k$;

    for $i$ := 1 to $n$ do                    {initialization of FinalStates according to $F$}
        if $i \in F$ then FinalStates := FinalStates $\cup$ {$i$};

    State := 1;
    read(InputSymbol);

    while InputSymbol $\neq$ ◀ do             {simulation of a move}
    begin
        State := Rule[State, InputSymbol];
        read(InputSymbol)
    end;

    if State $\in$ FinalStates then           {decision of acceptance}
        ACCEPT                                {$w \in L(M)$}
    else
        REJECT                                {$w \notin L(M)$}

end.
```

In Algorithm 2.8, a rule, $ia_j \to k \in R$, is represented by *Rule*[i, 'a_j'] := k. If *State* = i, *InputSymbol* = a_j, and *Rule*[i, 'a_j'] = k, the **while** loop of this algorithm sets *State* to k to simulate the application of $ia_j \to k$. When this loop reads ◀, it exits and the **if** statement tests whether *State* represents a final state. If so, $w \in L(M)$; otherwise, $w \notin L(M)$.

Example 2.5 ***Implementation of a Finite Automaton.*** Return to the automaton in Figure 2.5. Represent it as a table (see Figure 2.6). Rename its states s, p, q, and f to 1, 2, 3, and 4, respectively. By using a new non-final state 5, change this automaton to a completely specified deterministic finite automaton in Figure 2.7. This automaton is implemented by using Algorithm 2.8.

	a	*b*	*c*
‣*s*	‣*s*	▪*p*	▪*q*
▪*p*	▪*f*	▪*p*	
▪*q*	▪*f*		▪*q*
▪*f*	▪*f*		

Figure 2.6 ***Incompletely specified deterministic finite automaton.***

state-table	*a*	*b*	*c*
‣1	‣1	▪2	▪3
▪2	▪4	▪2	5
▪3	▪4	5	▪3
▪4	▪4	5	5
5	5	5	5

Figure 2.7 ***Completely specified deterministic finite automaton.***

```
program FiniteAutomaton(input, output);

{
Based on Algorithm 2.8, this program simulates the completely specified deterministic finite
automaton depicted in Figure 2.7.  It reads an input string from file input.  The input string is
terminated with eof.  Input characters out of range 'a'…'c' cause an error.
}

type
   States = 1..5;
   InputSymbols = 'a'..'c';
   Rules = array[States, InputSymbols] of States;
   StateSet = set of States;

var
   State: States;
   InputSymbol: InputSymbols;
   Rule: Rules;
   FinalStates: StateSet;

begin

   {Initialization of Rule according to the state table in Figure 2.7}
   Rule[1, 'a'] := 1; Rule[1, 'b'] := 2; Rule[1, 'c'] := 3;
   Rule[2, 'a'] := 4; Rule[2, 'b'] := 2; Rule[2, 'c'] := 5;
   Rule[3, 'a'] := 4; Rule[3, 'b'] := 5; Rule[3, 'c'] := 3;
   Rule[4, 'a'] := 4; Rule[4, 'b'] := 5; Rule[4, 'c'] := 5;
   Rule[5, 'a'] := 5; Rule[5, 'b'] := 5; Rule[5, 'c'] := 5;

   {Initialization of FinalStates};
   FinalStates := [2, 3, 4];

   State := 1;

   read(InputSymbol);

   while not eof do
   begin
      State := Rule[State, InputSymbol];
      read(InputSymbol)
   end;

   if State in FinalStates then {decision of acceptance}
      writeln('ACCEPT')
   else writeln('REJECT')

end.
```

With *abba eof* as its input, the **while** loop in the above program makes these four iterations. The first iteration begins with *State* = 1 and *InputSymbol* = *a*. Thus, it sets *State* := 1 because *Rule*[1, '*a*'] = 1. The second iteration has *State* = 1 and *InputSymbol* = *b*. At this point, *State* = 2 because *Rule*[1, '*b*'] = 2. The third iteration starts with *State* = 2 and *InputSymbol* = *b*, so it sets *State* := 2 because *Rule*[2, '*b*'] = 2. The fourth iteration begins with *State* = 2 and *InputSymbol* = *a*. Consequently, this iteration determines *State* = 4 by *Rule*[2, '*a*'] = 4. The next symbol is *eof*, so the **while** loop exits and the **if** statement determines that *State* belongs to *FinalStates* because *State* = 4 and *FinalStates* = [2, 3, 4]. Therefore, this statement writes ACCEPT as desired.

■

The previous algorithm represents the input automaton’s rules by a two-dimensional array. The next algorithm is based upon a nested **case** statement, which frees this implementation from using any array.

Algorithm 2.9 ***Implementation of a Finite Automaton*—case-*Statement Method*.**

Input • a completely specified deterministic finite automaton, $M = (\Sigma, R)$ with $Q = \{1, \dots, n\}$, $\Delta = \{a_1, \dots, a_m\}$, and $s = 1$;
• w◀ with $w \in \Delta^*$.

Output • **ACCEPT** if $w \in L(M)$, and **REJECT** if $w \notin L(M)$.

Method

```
type
    States = 1..n;
    InputSymbols = 'a_1'.. 'a_m';
    StateSet = set of States;

var
    State: States;
    InputSymbol: InputSymbols;
    FinalStates: StateSet;

begin

    for i := 1 to n do                        {initialization of FinalStates according to F}
    if $i \in F$ then FinalStates := FinalStates $\cup$ {i}

    State := 1;
    read(InputSymbol);

    while InputSymbol $\neq$ ◀ do              {simulation of moves}
    begin
        case State of
        1: …
        ⋮
        i:  case InputSymbol of
            'a_1':…
            ⋮
            'a_j': State := k if $ia_j \rightarrow k \in R$;
            ⋮
            'a_m':…
            end {case corresponding to i};
        ⋮
        n: …
        ⋮
        end;
        read(InputSymbol)
    end;

    if State $\in$ FinalStates then           {decision of acceptance}
        ACCEPT                                {$w \in L(M)$}
    else REJECT                               {$w \notin L(M)$}
end.
```

Algorithm 2.9 underlies the design of a scanner given in the next section (see Case Study 4/35).

Finite Transducers

In essence, finite transducers are finite automata that can emit an output string during any move. Before we define them, let us point out that finite automata represent rewriting systems, and as such, the notions introduced for these systems straightforwardly apply to them (see Section 1.3). Specifically, for a finite automaton $M = (\Sigma, R)$, by $u \Rightarrow v$ $[r]$, where $u, v \in \Sigma^*$ and $r \in R$, we express that M directly rewrites u to v by r or, as we customarily say in terms of finite automata, that *M makes* a *move from u to v by r*. Furthermore, to express that M makes $u \Rightarrow^* w$ according to a sequence of rules, $r_1r_2\ldots r_n$, we write $u \Rightarrow^* v$ $[r_1r_2\ldots r_n]$, which is read as a *sequence of moves from u to v by using* $r_1r_2\ldots r_n$.

Definition 2.10 ***Finite Transducer and its Translation.*** Let $M = (\Sigma, R)$ be a finite automaton, O be an *output alphabet*, and o be a total *output function* from R to O^*. Then, $\Pi = (\Sigma, R, o)$ is a *finite transducer underlain by M*. Let $\Pi = (\Sigma, R, o)$ be a finite transducer, $v \in \Delta^*$, and $w \in O^*$. If *M accepts v according to* $r_1r_2\ldots r_n$, where $r_i \in R$, $1 \le i \le n$, for some $n \ge 0$, then Π *translates v* to $o(r_1)o(r_2)\ldots o(r_n)$ *according to* $r_1r_2\ldots r_n$. The *translation of* Π, $T(\Pi)$, is defined as $T(\Pi) = \{(v, w) | v \in \Delta^*, w \in O^*, \Pi \text{ translates } v \text{ to } w\}$.

■

Convention 2.11. For any finite transducer, $M = (\Sigma, R)$, we automatically assume that O denotes its output alphabet; otherwise, we use the same conventions as for any finite automaton. For brevity, we often express $r \in R$ with $o(r) = y$ as ry; that is, if $r\colon qa \to p \in R$ and $o(r) = y$, we write $r\colon qa \to py$.

■

Informally, a finite transducer, $\Pi = (\Sigma, R, o)$, translates an input string, $a_1\ldots a_n$, to an output string, $x_1\ldots x_n$, as follows. In essence, it works by analogy with its underlying automaton $M = (\Sigma, R)$; moreover, Π writes a suffix of the produced output word during every step of this translation process. More exactly, assume that Π has already translated $a_1\ldots a_{i-1}$ to $x_1\ldots x_{i-1}$, so Π's current input symbol is a_i. At this point, Π can make a translation step according to a rule of the form $q \to px_i$ or $qa_i \to px_i$, where q is Π's current state. According to $q \to px_i$, Π changes q to p and writes x_i behind $x_1\ldots x_{i-1}$, so it actually extends $x_1\ldots x_{i-1}$ to $x_1\ldots x_{i-1}x_i$. According to $qa_i \to px_i$, Π does the same; in addition, however, it reads a_i, so a_{i+1} becomes the input symbol in the next step. If in this way Π translates $a_1\ldots a_n$ to $x_1\ldots x_n$ and M accepts $a_1\ldots a_n$, $(a_1\ldots a_n, x_1\ldots x_n) \in T(\Pi)$, which denotes the translation defined by Π.

A finite transducer $\Pi = (\Sigma, R, o)$ is graphically represented by its *state-translation diagram* constructed as follows. Draw a *M-state-diagram*, where $M = (\Sigma, R)$, which is the finite automaton underlying Π. For every $r\colon qa \to p \in R$, write $o(r)$ under the edge from q to p at the place where a occurs above this edge (see Figure 2.8).

$$\begin{tikzcd} q \arrow[rr, "a", "o(r)"'] & & p \end{tikzcd}$$

Figure 2.8 ***Graphical Representation of a Finite Transducer's Rule.***

The next section illustrates how the finite transducers and their state-translation diagrams are applied in the scanner design (see Convention 2.13 and Figures 2.19 and 2.20).

2.2 Methods

In this section, we first explain how to represent lexemes by their tokens. Then, we describe how to construct a lexical analyzer that makes this representation.

Lexemes and Tokens

To allow programmers to denote their lexical units as flexibly as possible, every high-level programming language offers them a wide variety of lexemes, such as identifiers or integers. As already noted in Section 1.2, these lexemes are classified into finitely many lexical types, each of which the lexical analyzer represents by its token. By *t-lexemes*, where *t* is a token, we denote the language consisting of all lexemes represented by *t*. Although there always exist finitely many tokens regarding, *t-lexemes* may contain infinitely many lexemes, and this case deserves our special attention.

Let *t-lexemes* be infinite. At this point, we introduce a regular expression, denoted by *t-expression*, that specifies *t-lexemes*. For instance, suppose that *i* is the token denoting the identifiers defined alphanumeric strings that start with a letter. As the set of all these strings, denoted by *i-lexemes*, is infinite, we specify *i-lexemes* by *i-expression* defined as *letter*(*letter*|*digit*)*, where *letter* and *digit* denote the set of all letters and the set of all digits, respectively (see Case Study 2/35). Furthermore, to describe the infinitely many lexemes in *t-lexemes*, the lexical analyzer represents each of them by *t* with an attribute that further specifies the lexeme. That is, the lexical analyzer converts every particular lexeme $l \in$ *t-lexemes* to an *attributed token* of a fixed-sized uniform form, denoted by *t*{*a*}, where *a* represents *t*'s attribute that provides further information about *l*. From an implementation point of view, *a* is often realized as a pointer to a symbol-table entry that records all the necessary information about *l*. For instance, consider *x* as an identifier, and let *i* have the same meaning as above. Then, *x* is represented by *i*{☞*x*}, where ☞*x* is a pointer to the symbol-table entry that records all needed information about this identifier, such as its type and its name *x*. Of course, whenever possible, the lexical analysis simplifies the attributed token as much as possible. In some cases, the attached attribute is not a pointer but a numeric value. Specifically, a token that specifies an integer number usually has the form #{*n*}, where # is the token that specifies integers and the attribute *n* is the numeric value of the integer number, which is the only information needed regarding this lexeme. For example, 009 is represented as #{9}. In most cases, there is no need for any attribute at all. Consider, for instance an addition operator +. The lexical analyzer represents this lexeme by itself as + because no additional attribute is necessary.

Case Study 3/35 ***Lexemes and Tokens.*** In this part of the case study, we represent the FUN lexemes by their tokens.

The first five types of FUN lexemes are

- identifiers
- labels
- text literals
- integers
- real numbers

There exist infinitely many FUN lexemes of each of these five types. Therefore, they are specified by regular expressions as demonstrated next in detail.

Identifiers. FUN identifiers are formed by alphanumeric strings that begin with a letter. Except for the name of the program, every FUN identifier is declared as a ***real*** or ***integer*** variable. The next table gives their token and expressions specifying them; in addition, the table illustrates them by some examples.

token	*i*
i-expression	*letter*(*letter*|*digit*)*
examples	factorial, n, iden99

Figure 2.9 *Identifiers.*

Labels. FUN labels have the form @*x*, where *x* is a non-empty alphanumeric word. The following table presents their token, expression, and some examples.

token	*l*
l-expression	@(*letter*|*digit*)$^+$
examples	@99, @abc1

Figure 2.10 *Labels.*

Text Literals. Text literals have the form '*x*', where *x* is any string in (*alph*(FUN) – {'})*, where *alph*(FUN) is the alphabet consisting of all printable characters and, in addition, a white space. In the next table, α is a regular expression that specifies (*alph*(FUN) – {'}).

token	*t*
t-expression	'α'
examples	'RESULT = ', 'hi'

Figure 2.11 *Text Literals.*

Integer Numbers. Integers in FUN correspond to the integers in mathematics.

token	#
#-expression	*digit*$^+$
examples	0, 987654321, 1957

Figure 2.12 *Integer Numbers.*

Real Numbers. In essence, real numbers in FUN also correspond to the real numbers in mathematics. That is, they have the form *ax.y*, where *x* is a non-empty numeric string and *y* is a numeric string; the case when $y = \varepsilon$ means $y = 0$. The symbol *a* denotes a plus or minus sign; the case when $a = \varepsilon$ means that the number is positive. For instance, 2. have the same meaning as +2.0.

token	¢
¢-expression	*digit*$^+$. *digit**
examples	0., -3.14, +0.2121

Figure 2.13 *Real Numbers.*

As the sets of the above FUN lexemes are infinite, they all are represented by attributed tokens, given in the next table.

Lexeme Type	*Token*	*Attribute*	*Example*
identifiers	*i*	pointer to a symbol-table entry	*i*{☞*x*}
labels	*l*	pointer to a label-table entry	*i*{☞*@lab*}
text literals	*t*	pointer to a text-literal-table entry	*t*{☞'text'}
integer numbers	#	integer numeric value	#{21}
real numbers	¢	real numeric value	¢{0.8}

Figure 2.14 ***Attributed Tokens.***

Relational Operators. Although the set of relational operators is finite, we represent them by the same token *r* with an attribute specifying each of them (see Figure 2.15).

Relational Operators	*Attributed Token*	*Meaning*
gr	*r*{>}	*gr*eater than
ge	*r*{≥}	*g*reater than or *e*qual to
ls	*r*{<}	*l*e*s*s than
le	*r*{≤}	*l*ess than or *e*qual to
ne	*r*{≠}	*n*ot *e*qual to
eq	*r*{=}	*eq*ual to

Figure 2.15 ***Relational Operators with Attributes.***

The remaining FUN lexemes are represented by non-attributed tokens. In fact, many of them are simply represented by themselves; for instance, + is the token representing + (arithmetic addition).

Logical and Arithmetic Operators. The next table presents all arithmetic and logical operators together with corresponding tokens representing them; their mathematical meaning is given, too.

Operator	*Token*	*Meaning*
+	+	arithmetic addition
-	-	arithmetic subtraction
*	*	arithmetic multiplication
/	/	arithmetic division
&	∧	logical *and*
|	∨	logical *or*
!	¬	logical *not*

Figure 2.16 ***Arithmetic and Logical Operators.***

Separators, Parentheses, and Assignment. As separators, FUN contains comma (,) and semicolon (;). FUN uses the usual parentheses (and), and its FUN assignment symbol is =. The following table presents these lexemes together with the tokens representing them. Notice that all of them are represented by themselves, which is the most natural representation concerning one-symbol lexemes.

Lexeme	*Token*
,	,
;	;
(	(
)	)
=	=

Figure 2.17 ***Punctuation.***

Keywords. Each FUN keyword is represented by itself as well; for instance, ***begin*** is the token representing the reserved word ***begin***. FUN contains these keywords:

begin, ***declaration***, ***else***, ***end***, ***execution***, ***for***, ***goto***, ***if***, ***integer***, ***iterate***, ***label***, ***program***, ***provided***, ***read***, ***real***, ***then***, ***through***, ***write***.

The FUN comments are enclosed in curly braces. They just make programs easier to understand; however, they do not represent any lexeme. Therefore, there is no need to represent them by any token.

Like most programming languages, FUN is case-insensitive; for instance, ***bEgIn*** and ***BEGIN*** are synonymous to ***begin***. Therefore, except for the text literals, the FUN lexical analyzer converts all letters to lowercases.

Case Study 1/35 has given an example of a complete FUN source program together with its tokenized version.

■

Lexical analyzer

Consider a file containing a string of characters, and let this string represent a source program written in a source language L. Upon receiving a request for the next token from the parser, the scanner of a lexical analyzer reads this string from left to right in order to recognize the next lexeme and represent it by a token. Suppose that L's lexemes are classified into n lexical types represented by tokens t_1 through t_n, for some $n \geq 1$. At this point, the scanner of L is based on a deterministic finite automaton $M = (\Sigma, R)$ with its set of final states $F = \{g_1, \ldots, g_n\}$ so that for all $k = 1, \ldots, n$, $sw \Rightarrow^* g_k$ in M if and only if $w \in t_k$*-lexemes*. Conceptually, M consists of several subautomata, each of which may be further divided into smaller subautomata, and so on. Depending on the scanned input symbol, M enters a subautomaton. If this subautomaton has two or more final states, it again uses the input symbols to enter a smaller subautomaton, and M repeats this process until it enters a subautomaton with a single final state, g_k. If M eventually reaches g_k, it recognizes a lexeme from g_k*-lexemes* represented by token t_k and, thereby, successfully completes its task. Before we describe M in greater detail, we introduce some notions and conventions that simplify this description.

Definition 2.12 ***Subautomaton.*** Second, let $A = ({}_A\Sigma, {}_AR)$ and $B = ({}_B\Sigma, {}_BR)$ be two finite automata such that ${}_B\Sigma \subseteq {}_A\Sigma$ and ${}_BR \subseteq {}_AR$. Then, B is a *subautomaton of* A.

■

Convention 2.13. In the state diagram of an incomplete deterministic finite automaton, $C = ({}_C\Sigma, {}_CR)$, we sometimes introduce an edge (q, p) labeled with **others**, where $q, p \in Q$ and **others** = $({}_C\Delta - \{a| \ qa \rightarrow p \in R, a \in {}_C\Delta \cup \{\varepsilon\}\})$. In other words, **others** denote all symbols $b \in {}_C\Delta$ satisfying $qb \rightarrow p \notin R$. Consider the state-translation diagram of a finite transducer $\Pi = ({}_D\Sigma, {}_DR, o)$ underlain by a finite automaton, $D = ({}_D\Sigma, {}_DR)$. Let (q, p) be an edge in this diagram, $q, p \in {}_DQ$, and let Ψ be the set of all its labels, written above this edge, where $\Psi \subseteq {}_D\Sigma$. If each r: $qa \rightarrow p \in {}_DR$ with $a \in \Psi$, satisfies $o(r) = a$, Π actually translates every element of Ψ to the same element during a move from q to p; at this point, we write Ψ above the edge (q, p) and **same** under (q, p) at the place where Ψ occurs above this edge (see Figure 2.19 for an illustration).

■

Convention 2.14. For any set of tokens, $\upsilon \subseteq \{t_1, \ldots, t_n\}$, set υ*–lexemes* = $\{l| \ l \in t$*-lexemes* for some $t \in \upsilon\}$.

■

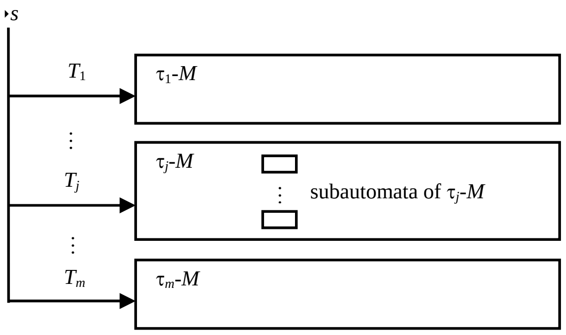


Figure 2.18 ***Decomposition of a Scanner.***

The construction of M is schematically described in Figure 2.18, where $m \leq n$, T_1 through T_m represent pairwise disjoint subalphabets of $alph(L)$ satisfying

- $\tau_j \subseteq \{t_1, \ldots, t_n\}$ and $T_j = \{a |\ a \in symbol(\xi\text{–}lexemes, 1), \xi \in \tau_j\}$, for $1 \leq j \leq m$;
- $T_1 \cup \ldots \cup T_m = \{b |\ b \in symbol(t_k\text{-}lexemes, 1), 1 \leq k \leq n\}$.

M contains subautomata τ_1-M through τ_m-M, whose sets of states are pairwise disjoint (see Figure 2.18). On the other hand, the union of all these sets equals ${}_MQ - \{{}_Ms\}$. With any symbol from T_j, M enters τ_j-M, which contains $card(\tau_j)$ final states. If $card(\tau_j) = 1$, τ_j contains a single token t_j, and at this point, we write t_j-M instead of $\{t_j\}$-M for brevity. If $card(\tau_j) \geq 2$, τ_j-M itself is analogously decomposed into several subautomata, and this nested decomposition ends when each subautomaton contains precisely one final state g_k, in which M accepts the lexemes from t_k-*lexemes*.

As a whole, the lexical analyzer represents a finite transducer Π rather than a finite automaton because apart from the recognition of lexemes, it produces their tokens as output. From M constructed above, we straightforwardly obtain Π so that we

- change final states $\blacksquare g_1, \ldots, \blacksquare g_n$ to non-final states $g_1, \ldots, g_n$, respectively;
- add new final states $\blacksquare f_1, \ldots, \blacksquare f_n$ and rules $g_j \rightarrow \blacksquare f_j$ for $1 \leq j \leq n$;
- define the output alphabet $O = \{t_1, \ldots, t_n\}$ and the output function o from the set of rules so that for each new rule $g_j \rightarrow \blacksquare f_j$ added in previous step, $o(g_j \rightarrow \blacksquare f_j) = t_j$, and map all the other rules to ε by o.

Observe that the translation of Π, $T(\Pi)$, is defined as $T(\Pi) = \{(v, t_k) |\ v \in t_k\text{-}lexemes, k = 1, \ldots, n\}$ as desired. A rigorous verification of $T(\Pi)$ is left as an exercise.

In Figure 2.19, we illustrate Π by a finite transducer underlain by {#, ¢}-M, which represents a finite automaton that recognizes the FUN integer and real numbers. Notice that this transducer enters #-g or ¢-g with a non-digital character that actually follows the recognized number; later in this section, we explain how to return this character.

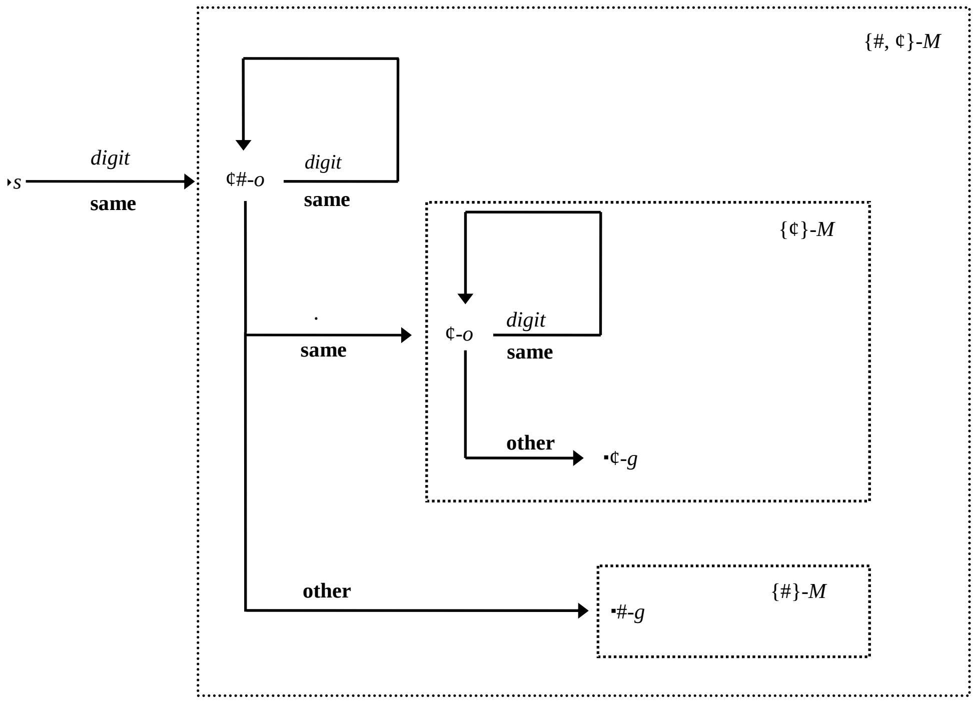


Figure 2.19 ***A Lexical Analyzer that Recognizes Numbers.***

Compared to the previous parts of this section, we approach the lexical analyzer from a more practical point of view in the rest of this section.

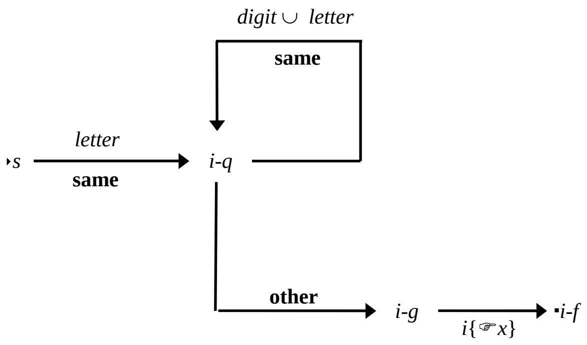


Figure 2.20 ***Lexical Analyzer for Identifiers.***

Attributes. As explained in the beginning of this section, the lexical analyzer often works with attributed tokens of the form *t*{*a*}, where *t* is a token and *a* represents *t*'s attribute that provides further information about *t*. If there exist finitely many values of *a*, we might restructure *M* so that the attribute is defined by a specific final state in which *M* ends up its acceptance of the recognized lexeme. However, if there exist infinitely many possible values of *a*, any modification of this kind is ruled out. At this point, the lexical analyzer is usually extended so it produces the value of *a* during the recognition of a lexeme from *t-lexemes*. To illustrate, consider the FUN identifiers, which represent an infinite set (see Case Study 3/35). Its attributed token has the form *i*{☞*x*}, where ☞*x* is the symbol-table address pointing to the entry that records information about the recognized identifier whose name is *x* (see Convention 1.4). Figure 2.20 describes how the lexical analyzer reads a character and, simultaneously, records the character as output. Entering its final state, the analyzer has recorded *x*, so it produces *i*{☞*x*} as the resulting token.

Lexical Analyzer and Symbol-Table Handler. The lexical analyzer closely cooperates with the symbol-table handler. Most importantly, this cooperation takes place during the analysis of identifiers and keywords.

- *Identifiers* are stored and found in the symbol table by the symbol-table handler based upon requests from the lexical analyzer. More precisely, when an *identifier* is recognized, the symbol-table handler provides the lexical analyzer with the symbol-table address pointing to the entry that stores the recognized identifier. If the identifier already exists in the table, the symbol-table handler finds it and reports its entry address. If the first occurrence of an identifier is recognized and, therefore, this identifier does not appear in the table, the symbol-table handler creates a new table entry for it, stores the identifier, and sends the address of this newly created entry to the lexical analyzer. This address becomes the attribute of the token representing identifiers, which is sent to the syntax analyzer. In addition, the lexical analyzer informs the syntax analyzer whether this identifier has already occurred in the table because this information may imply an announcement of an error. For instance, the absence of this occurrence may imply that an identifier is used before its declaration, and if this kind of use is illegal, an error is announced.

- *Keywords* usually represent alphabetic strings, so they can be temporarily seen as exceptions to the identifiers. Therefore, when the final state corresponding to identifiers is reached (see Figure 2.20), the lexical analyzer asks the symbol-table handler to decide whether the recognized string is a keyword or an identifier. To make this decision, the symbol table is preloaded with a list of keywords, which the symbol-table handler checks. If it finds the recognized string in this list, it tells the keyword to the lexical analyzer; otherwise, the recognized string is an identifier. In the same way, the lexical analyzer distinguishes any other alphanumeric strings from identifiers. For instance, *relational operators* are often written as short alphabetic strings; in FUN, the relational operators ***gr***, ***ge***, ***ls***, ***le***, ***ne***, and ***eq*** denote >, ≥, <, ≤, ≠, and =, respectively.

Return in the Source Program. The lexical analyzer always recognized a lexeme as the longest possible string denoted by the corresponding expression. For example, *xy* represents a single identifier rater than two identifiers, *x* and *y*. In many cases, it makes a decision regarding the recognized lexeme when it comes to its very end, and to recognize this end, it actually needs to read the symbol that follows the complete lexeme. Therefore, in these cases, the lexical analyzer has to return the symbol following the lexeme in order to reread it when searching for the next lexeme in the source program. For instance, *M* recognizes identifier *xy* in *xy*+*uv* after reading +; at this point, however, it has to return + so that it correctly recognizes this arithmetic operator as the lexeme when called by the parser next time. To give another example, return to Figure 2.19, in which the scanner *M* enters a final state with a non-digital character following the recognized number, so the lexical analyzer has to return this character so *M* starts its scanning from this returned character when searching for the next lexeme in the source program.

Source-Program Simplification. Frequently, the lexical analyzer performs some other minor tasks simplifying the source-program text, including the following changes.

- *Case Conversion*. Capitalization is ignored in many programming languages, so the lexical analyzer converts all the uppercase letters to the lowercase letters.

- *Text Removal*. The lexical analyzer removes all parts of the source-program text irrelevant to the translation. Specifically, it removes all comments. Furthermore, it takes away any character that results in empty space when printed; for instance, these white-space characters include blanks, carriage returns, and line feeds.

Note, however, that within text literals, these changes are turned off; for instance, we want to keep 'Albert Einstein' unchanged rather than modify it to 'alberteinstein'.

Convention 2.15. We denote the operation concatenation by · in the procedures given in this book.

■

Case Study 4/35 *Scanner.* In this part of the case study, we design the FUN scanner. First, however, we define the following variables and sets this scanner makes use of.

• *character*	global variable that stores a single character;
• *lexeme*	global linked list that stores a string of characters;
• *token*	global variable that stores any FUN token;
• *letter*	set of all letters (see Figure 2.1);
• *digit*	set of all digits (see Figure 2.1).

The scanner also makes these trivial operations:

- **INPUT-CHARACTER** reads the next input character and places it into *character* (this routine thus contains all necessary subroutines that the reading from the standard input involves, such as skipping the end-of-line symbol);

- **RETURN-CHARACTER** pushes back the current character in *character* onto the standard input.

In the end of Section 2.1, we have described two common methods of implementing a finite automaton—the tabular method (Algorithm 2.8) and the case-statement method (Algorithm 2.9). We next implement the FUN scanner based upon a case statement, leaving its implementation by the tabular method as an exercise. Receiving a request from the FUN syntax analyzer for the next token, the FUN scanner is schematically implemented as follows:

```
procedure FUN-scanner;
begin
    set lexeme to $\varepsilon$;
    repeat
        INPUT-CHARACTER
    until character contains a non-white character;
    case character of
        letter  :  i-P;
        @       :  l-P;
        '       :  t-P;
        digit   :  {#, ¢}-P;
        +       :  +-P;
        ⋮
        otherwise: ☹-P
    end
end
```

where *i-P* through *+-P* are the procedures described next. Notice that each of them sets *token* to the token that specifies the recognized lexeme.

i-P. Procedure *i-P*, given next, represents the part of the scanner that recognizes any FUN identifier *x* and represents them by their tokens of the form *i*{☞*x*} (see Figure 2.20). As explained above, the FUN symbol-table handler provides the scanner with the symbol-table address ☞*x*. Notice that **procedure** FUN-scanner has already a letter in *character* when it calls *i-P*, whose description follows next.

```
procedure i-P;
begin
    repeat
        lexeme := lexeme·character;    {concatenate the string in lexeme and the character in
                                          character}
        INPUT-CHARACTER
    until character $\notin$ letter $\cup$ digit; {exit when a non-alphanumeric symbol is read}
    RETURN-CHARACTER; {the non-alphanumeric symbol in character is pushed back onto
                              the standard input}
    if lexeme contains a FUN keyword {the symbol-table handler determines this} then
        token := lexeme {the token of every keyword is the keyword itself; for instance, if is the
                              token of if}
    else
    begin
        token := i{☞x}, where x is the identifier contained in lexeme;
        report whether x already existed in the symbol table to the FUN parser
        {the symbol-table handler provides ☞x as well as the information whether x was previously
          stored in the symbol table}
    end
end
```

l-P. If *l-P* recognizes a well-formed label *@y*, where *y* is a non-empty alphanumeric word, *l-P* represents the recognized label by its attributed token of the form *l*{*l-address*(*@y*)}, where *l-address*(*@y*) points to *@y* stored, for instance, in a special table of labels. Recall that *character* already contains @ when the scanner enters procedure *l-P*, whose description follows next.

```
procedure l-P;
begin
    repeat
        lexeme := lexeme·character; {concatenate the string in lexeme and the character in symbol}
        INPUT-CHARACTER
    until character $\notin$ letter $\cup$ digit; {exit when a non-alphanumeric symbol is read}
    RETURN-CHARACTER; {the non-alphanumeric symbol in character is pushed back onto
                              the standard input}
    if lexeme contains only @ then
        lexical error "invalid label—no letter or digit follows @" is issued
    else
        token := l{l-address(@y)}, where @y is the recognized label contained in lexeme
end
```

t-P. This procedure recognizes the FUN text literals. If *t-P* recognizes a well-formed text literal 'z', where *z* is any string that does not contain a quotation mark ('), *t-P* represents the recognized text literal by its token of the form *t*{*t-address*(z)}, where *t-address*(z) is the address of *z* stored, for example, in a special linked list created for this purpose. Recall that *character* contains the first quotation mark during the entrance into *t-P*, whose description follows next.

```
procedure t-P;
begin
    INPUT-CHARACTER;
    while character ≠ ' do
    begin {exit when ' is read}
        lexeme := lexeme·character; {concatenate the string in lexeme and the character in
                                     character}
        INPUT-CHARACTER
    end
    if lexeme = ε then
        warning "empty text literal" is issued
    else
        token := t{t-address(z)}, where z is the text contained in lexeme
end
```

In reality, *t-P* should take care of some special cases that may occur when text literals are recognized. Specifically, if *t-P* reads the *end-of-file* character of the source-program file, the source program ends with an unfinished text literal, so this error is announced.

#-¢-*P*. Figure 2.19 depicts {#, ¢}-*M* that recognizes {#, ¢}-*lexemes*. Based on {#, ¢}-*M*, we next construct procedure #-¢-*P* that recognizes {#, ¢}-*lexemes* and, in addition, represents these lexemes by their tokens. To describe these tokens in greater detail, if the string in *lexeme* represents a FUN integer number, it makes the attributed token of the form #{#-*value*(*lexeme*)}, where #-*value*(*lexeme*) denotes the numeric value of the string in *lexeme*. Similarly, if the string in *lexeme* represents a FUN real number, it makes the attributed token of the form ¢{¢-*value*(*lexeme*)}, where ¢-*value*(*lexeme*) denotes the numeric value of the string in *lexeme*.

```
procedure #-¢-P;
begin
    repeat
        lexeme := lexeme·character; {concatenate the string in lexeme and the character in
                                     character}
        INPUT-CHARACTER
    until character ∉ digit; {exit when a non-numeric symbol is read}
    if character ≠ . then
    begin
        RETURN-CHARACTER; {the non-numeric symbol in character is pushed back onto the
                           standard input}
        token := #{#-value(lexeme)};
    end
    else
    begin
        repeat
            lexeme := lexeme·character; {concatenate the string in lexeme and the character in
                                         character}
            INPUT-CHARACTER
        until character ∉ digit; {exit when a non-numeric symbol is read}
        RETURN-CHARACTER; {the non-numeric symbol in character is pushed back onto the
                           standard input}
        token := ¢{¢-value(lexeme)}
    end
end
```

+-P. The remaining procedures are simple. For instance, *+-P* is described next.

```
procedure +-P;
begin
    token := +
end
```

If the FUN scanner reads a non-white symbol that begins no FUN lexeme, it enters ☹-*P* that handles lexical errors of this kind. As an exercise, we further discuss ☹-*P*.

■

Additional Tasks

Interpretation of Compiler Directives. A good programming-language compiler provides its user with a large variety of directives specifying the way the compiler should operate with the source program in detail. These directives may require an inclusion of some file into the source program or an assistance in debugging the program. The lexical analyzer interprets these directives and takes an appropriate action to fulfill them.

Output Listing. The lexical analyzer creates a file that contains an annotated version of the source program. This version contains the numbers of source-program lines, error messages, warnings, and a protocol summarizing the translation of the source program.

2.3 Theory

In this section, we discuss the theory of regular languages and their models, concentrating our attention on the properties and results relevant to the lexical analysis. First, we discuss transformations of models for regular languages. We give algorithms that transform regular expressions to equivalent deterministic finite automata (Section 2.3.1). By using these automata, we can thus automatically turn a set of lexemes specified by regular expressions to the lexical analyzer based on finite automata for these lexemes. In fact, some real software units, such as *lex* described in Section 6.5, perform exactly this transformation. Then, we simplify finite automata (Section 2.3.2). We also explain how to demonstrate that some languages are not regular and, thereby, unsuitable to represent programming language lexemes (Section 2.3.3). Finally, we give algorithms that decide some crucial problems about finite automata (Section 2.3.4). We frequently take advantage of these algorithms to verify that a scanner is correctly designed.

2.3.1 Transformation of Regular Expressions to Finite Automata

In this section, we explain how to transform regular expressions to equivalent finite automaton, which is a task crucial to the lexical analysis. Indeed, recall that regular expressions specify lexemes while a scanner uses several finite automata as its parts to recognize these lexemes (see Section 2.2). Thus, this transformation actually produces the scanner parts from the regular expressions that specify lexemes, and this algorithmic production obviously represents an invaluable help to a lexical-analyzer designer.

Consider the definition of regular expressions over an alphabet Δ and the languages that these expressions denote (see Definition 2.1). We next consider the well-created regular expressions according to this definition and construct equivalent finite automata for them. By this definition,

• $\emptyset$ is a regular expression denoting the empty set;
• ε is a regular expression denoting {ε};
• *a*, where a $\in \Delta$, is a regular expression denoting {*a*}.

Lemma 2.16. There exists a finite automaton, *M*, such that *L*(*M*) equals the empty set.

Proof. Any finite automaton with no final state accepts the empty set.

■

Lemma 2.17. There exists a finite automaton, *M*, such that *L*(*M*) = {ε}.

Proof. Consider a finite automaton that has no rule and only the start state that is also a final state. This automaton accepts {ε}.

■

Lemma 2.18. Let $a \in \Sigma$. There exists a finite automaton, *M*, such that *L*(*M*) = {*a*}.

Proof. Let $a \in \Delta$. Consider a finite automaton defined by a single rule of the form ‣$sa \rightarrow$ ▪f; that is, *s* is the start non-final state and *f* is a final state. At this point, *L*(*M*) = {*a*}.

■

Union. By the definition of regular expressions, if *x* and *y* are regular expressions denoting languages *X* and *Y*, respectively, then $x|y$ is a regular expression denoting $X \cup Y$. Therefore, in terms of finite automata, we next demonstrate that if *M* and *N* are finite automata that accept languages *X* and *Y*, respectively, then there is a finite automaton that accepts $X \cup Y$.

Goal. Convert any two finite automata, *M* and *N*, to a finite automaton, $U = ({}_U\Sigma, {}_UR)$.

Gist (Figure 2.21). Consider any two finite automata, $M = ({}_M\Sigma, {}_MR)$ and $N = ({}_N\Sigma, {}_NR)$. If *M* and *N* contain some states in common, we rename states in either of these automata so that they have no state in common; in other words, without any loss of generality, we assume that *M* and *N* have disjoint sets of states. Construct *U* so that from its start state ${}_Us$, it enters ${}_Ms$ or ${}_Ns$ by an ε-move. From ${}_Ms$, *U* simulates *M*, and from ${}_Ns$, it simulates *N*. Whenever occurring in a final state of *M* or *N*, *U* can enter its only final state ${}_Uf$ by an ε-move and stop its computation. Thus, $L(U) = L(N) \cup L(M)$.

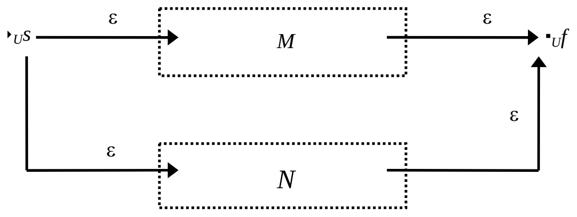


Figure 2.21 ***Finite Automaton for Union.***

Algorithm 2.19 ***Finite Automaton for Union*****.**

Input • two finite automata, $M = ({}_M\Sigma, {}_MR)$ and $N = ({}_N\Sigma, {}_NR)$, such that ${}_MQ \cap {}_NQ = \emptyset$.

Output • finite automaton $U = ({}_U\Sigma, {}_UR)$ such that $L(U) = L(N) \cup L(M)$.

Method

```
begin
    $_U\Delta := {}_M\Delta \cup {}_N\Delta$;
    $_UQ := {}_MQ \cup {}_NQ \cup \{_Us, {}_Uf\}$;        {$_Us$ and $_Uf$ are $U$'s start state and $U$'s only final state;
    $_Us$ and $_Uf$ are new states, which are not in $_MQ \cup {}_NQ$}
    $_UF := \{_Uf\}$;
    $_UR := {}_MR \cup {}_NR \cup \{_Us \rightarrow {}_Ms, {}_Us \rightarrow {}_Ns\} \cup \{p \rightarrow {}_Uf \mid p \in {}_MF \cup {}_NF\}$
end.
```

Lemma 2.20. Let $M = ({}_M\Sigma, {}_MR)$ and $N = ({}_N\Sigma, {}_NR)$ be two finite automata such that ${}_MF \cap {}_NF = \emptyset$. With M and N as its input, Algorithm 2.19 constructs a finite automaton $U = ({}_U\Sigma, {}_UR)$ such that $L(U) = L(N) \cup L(M)$.

Proof. To establish $L(U) = L(N) \cup L(M)$, we first prove the following claim.

Claim. For every $w \in {}_U\Delta^*$, $qw \Rightarrow^* p$ in U if and only if $qw \Rightarrow^* p$ in O, where $O \in \{M, N\}$, $q, p \in {}_OQ$.

Proof of the Claim.

Only if. By induction on $i \geq 0$, we prove for every $w \in {}_U\Delta^*$ and every $i \geq 0$, $qw \Rightarrow^i p$ in U implies $qw \Rightarrow^i p$ in O, where $O \in \{M, N\}$ and $q, p \in {}_OQ$. As obvious, the only-if part follows from this implication.

Basis. Let $i = 0$, so $qw \Rightarrow^0 p$ in U, where $O \in \{M, N\}$ and $q, p \in {}_OQ$. Then, $q = p$ and $w = \varepsilon$. Clearly, $q \Rightarrow^0 q$ in O, and the basis holds.

Induction Hypothesis. Assume that the implication holds for all $i \leq n$, where n is a non-negative integer.

Induction Step. Let $qw \Rightarrow^{n+1} p$ in U, where $q, p \in {}_OQ$ for some $O \in \{M, N\}$ and $w \in {}_U\Delta^*$. Because $n + 1 \geq 1$, express $qw \Rightarrow^{n+1} p$ as $qva \Rightarrow^n oa \Rightarrow p$ in U, where $o \in {}_UQ$, $w = va$ with $a \in \{\textbf{suffix}(w, 1), \varepsilon\}$ and $v = \textbf{prefix}(w, |w| - |a|)$. Recall that $o \notin \{_Us, {}_Uf\}$. Since $p \in {}_OQ$ and ${}_MQ \cap {}_NQ = \emptyset$, $o \in {}_OQ$ as well. As $oa \Rightarrow p$ in U and $o, p \in {}_OQ$, $oa \rightarrow p \in {}_OR$, so $oa \Rightarrow p$ in O. By the inductive hypothesis, $qv \Rightarrow^n o$ in O. Putting $qv \Rightarrow^n o$ and $oa \Rightarrow p$ in O together, $qva \Rightarrow^n oa \Rightarrow p$ in O. Because $va = w$, $qw \Rightarrow^{n+1} p$ in O, which completes the induction step.

If. The if-part of the claim is left as an exercise.

As a special case of the previous claim, for every $w \in {}_U\Delta^*$, ${}_Osw \Rightarrow^* p$ in U if and only if ${}_Osw \Rightarrow^* p$ in O, where $O \in \{M, N\}$ and $p \in {}_OF$. Since $\{_Us \rightarrow {}_Ms, {}_Us \rightarrow {}_Ns\} \cup \{p \rightarrow {}_Uf \mid p \in {}_MF \cup {}_NF\}$ is the set of all rules that contain ${}_Us$ or ${}_Uf$ in U, the previous equivalence implies ${}_Usw \Rightarrow {}_Osw \Rightarrow^* p \Rightarrow {}_Uf$ in U if and only if ${}_Osw \Rightarrow^* p$ in O, where $O \in \{M, N\}$ and $p \in {}_OF$, so $w \in L(U)$ if and only if $w \in$

$L(O)$. In other words, $L(U) = \{w \in L(O) | O \in \{M, N\}\}$. That is, $L(U) = L(N) \cup L(M)$, and the lemma holds.

■

In a finite automaton, a *stop state* is a state that does not occur on the left-hand side of any rule, so the automaton can never leave it. By the next corollary, without any loss of generality, we can always assume that a finite automaton has only one final state, which is also a stop state. Later in this section, we make use of this assumption.

Corollary 2.21. For any finite automaton $M = ({}_M\Sigma, {}_MR)$, there exists an equivalent finite automaton $U = ({}_U\Sigma, {}_UR)$ in which there is only one final state, ${}_Uf$, such that ${}_Uf$ is a stop state.

Proof. Let $M = ({}_M\Sigma, {}_MR)$ be a finite automaton, and let $N = ({}_N\Sigma, {}_NR)$ be any finite automata that accepts $\emptyset$ (see Lemma 2.16). Use Algorithm 2.19 to construct a finite automaton, $U = ({}_U\Sigma, {}_UR)$, such that $L(U) = L(N) \cup L(M) = L(M) \cup \emptyset = L(M)$. Observe that U constructed in this way has a single final state, ${}_Uf$, such that ${}_Uf$ is a stop state, so this corollary holds.

■

Concatenation. If x and y are regular expressions denoting languages X and Y, respectively, then xy is a regular expression denoting XY. Therefore, we next demonstrate that if M and N are finite automata that accept languages X and Y, respectively, then there exists a finite automaton that accepts XY.

Goal. Convert any two finite automata, M and N, to a finite automaton, C, such that $L(C) = L(M)L(N)$.

Gist (Figure 2.22). Consider any two finite automata, $M = ({}_M\Sigma, {}_MR)$ and $N = ({}_N\Sigma, {}_NR)$, such that ${}_MQ \cap {}_NQ = \emptyset$. In addition, without any loss of generality, suppose that M has a single final state, ${}_Mf$, such that ${}_Mf$ is also a stop state, and N has also only one final state, ${}_Nf$, and this state is a stop state. Construct C as follows. Its start state is ${}_Ms$ from which C simulates until it ends up in ${}_Mf$. From ${}_Mf$, C enters ${}_Ns$ by an ε-move. From ${}_Ms$, C simulates N, and this simulation ends when C enters ${}_Nf$. Thus, $L(C) = L(M)L(N)$.

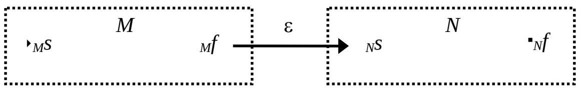


Figure 2.22 ***Finite Automaton for Concatenation.***

Algorithm 2.22 ***Finite Automaton for Concatenation.***

Input • two finite automata, $M = ({}_M\Sigma, {}_MR)$ and $N = ({}_N\Sigma, {}_NR)$, such that ${}_MQ \cap {}_NQ = \emptyset$, ${}_MF = \{{}_Mf\}$, ${}_NF = \{{}_Nf\}$, ${}_Mf$ and ${}_Nf$ are both stop states.

Output • finite automaton $C = ({}_C\Sigma, {}_CR)$ such that $L(C) = L(M)L(N)$.

Method

```
begin
    ${}_C\Delta := {}_M\Delta \cup {}_N\Delta$;
    ${}_CQ := {}_MQ \cup {}_NQ$;
    $_Cs := {_Ms}$;
    $_CF := \{_Nf\}$;
    $_CR := {_MR} \cup {_NR} \cup \{_Mf \rightarrow {_Ns}\}$
end.
```

Lemma 2.23. Let $M = ({_M\Sigma}, {_MR})$ and $N = ({_N\Sigma}, {_NR})$ be two finite automata satisfying such that ${_MQ} \cap {_NQ} = \emptyset$, ${_MF} = \{_Mf\}$, ${_NF} = \{_Nf\}$, $_Mf$ and $_Nf$ are both stop states. With M and N as its input, Algorithm 2.22 constructs a finite automaton, $C = ({_C\Sigma}, {_CR})$, such that $L(C) = L(M)L(N)$.

Proof. To see the reason why $L(C) = L(M)L(N)$, observe that C accepts every $w \in L(C)$ as ${_Ms}uv \Rightarrow^* {_Mf}v \Rightarrow {_Ns}v \Rightarrow^* {_Nf}$, where $w = uv$. Thus, ${_Ms}u \Rightarrow^* {_Mf}$ in M, and ${_Ns}v \Rightarrow^* {_Nf}$ in N. Therefore, $u \in L(M)$ and $v \in L(N)$, so $L(C) \subseteq L(M)L(N)$. In a similar way, demonstrate $L(M)L(N) \subseteq L(C)$. These two inclusions imply $L(C) = L(M)L(N)$. A rigorous version of this proof is left as an exercise.

■

Iteration. We now prove that if M is a finite automaton, then there exists a finite automaton N such that $L(N) = L(M)^*$.

Goal. Convert any finite automaton M to a finite automaton N such that $L(N) = L(M)^*$.

Gist (Figure 2.23). Consider a finite automaton, M. Without any loss of generality, suppose that M has a single final state $_Mf$ such that $_Mf$ is also a stop state (see Corollary 2.21). Besides M's states, N has two new states, $_Ns$ and $_Nf$, where $_Ns$ is its start state and $_Nf$ is its only final state. From $_Ns$, N can enter $_Nf$ without reading any symbol, thus accepting ε. In addition, it can make an ε-move to $_Ms$ to simulate M. Occurring in $_Mf$, N can enter $_Ms$ or $_Nf$; in the former case, it starts simulating another sequence of moves in M, and in the latter case, it successfully completes its computation. Therefore, $L(N) = L(M)^*$.

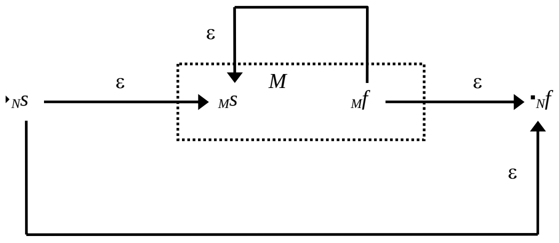


Figure 2.23 ***Finite Automaton for Iteration.***

Algorithm 2.24 ***Finite Automaton for Iteration.***

Input • finite automaton $M = ({_M\Sigma}, {_MR})$ such that ${_MF} = \{_Mf\}$ and $_Mf$ is a stop state.

Output • finite automaton $N = ({_N\Sigma}, {_NR})$ such that $L(N) = L(M)^*$.

Method

```
begin
    $_N\Delta := {_M\Delta}$;
    $_NQ := {_MQ} \cup \{_Ns, {_Nf}\}$;                {$_Ns$, $_Nf$ are new states}
    ${}_NF := \{{}_Nf\}$;
    ${}_NR := {}_MR \cup \{{}_Ns \to {}_Nf, {}_Ns \to {}_Ms, {}_Mf \to {}_Ms, {}_Mf \to {}_Nf\}$
end.
```

Lemma 2.25. Let $M = ({}_M\Sigma, {}_MR)$ be a finite automaton such that ${}_MF = \{{}_Mf\}$, and ${}_Mf$ is a stop state. With M as its input, Algorithm 2.24 correctly constructs a finite automaton $N = ({}_N\Sigma, {}_NR)$ such that $L(N) = L(M)^*$.

Proof. To see the reason why $L(N) = L(M)^*$, observe that N accepts every $w \in L(N) - \{\varepsilon\}$ as ${}_Nsv_1v_2\ldots v_n \Rightarrow^* {}_Msv_1v_2\ldots v_n \Rightarrow^* {}_Mfv_2\ldots v_n \Rightarrow {}_Msv_2\ldots v_n \Rightarrow^* {}_Mfv_3\ldots v_n \Rightarrow \ldots \Rightarrow {}_Msv_n \Rightarrow^* {}_Mf \Rightarrow^* {}_Nf$, where n is a positive integer, $w = v_1v_2\ldots v_n$ with $v_i \in L(M)$ for all $i = 1, \ldots, n$. Furthermore, N accepts ε as ${}_Ns\varepsilon \Rightarrow {}_Nf$. Therefore, $L(M)^* \subseteq L(N)$. Similarly, prove $L(N) \subseteq L(M)^*$. As $L(M)^* \subseteq L(N)$ and $L(N) \subseteq L(M)^*$, $L(N) \subseteq L(M)^*$. A rigorous version of this proof is left as an exercise.

■

Transformation of Regular Expressions to Finite Automata. We next explain how to turn any regular expression to an equivalent finite automaton.

Goal. Convert a regular expression t to a finite automaton M such that $L(t) = L(M)$.

Gist. Let t be a regular expression over an alphabet Δ. Consider its fully parenthesized version (see Definition 2.1 and the notes following this definition). Processing from the innermost parentheses out, determine how the expression is constructed by Definition 2.1. Follow this construction step by step and simultaneously create the equivalent finite automata by the algorithms given earlier in this section. That is, suppose that r and s be two regular expressions obtained during the construction.

- If r is a regular expression of the form $\emptyset$, ε, and a, where a $\in \Delta$, then r is equivalent to the finite automaton constructed in the proof of Lemmas 2.16, 2.17, and 2.18, respectively;

- Let r and s denote the languages R and S, respectively, and let M and N be two finite automata accepting R and S, respectively. Then,
  $r|s$, which denotes $R \cup S$, is equivalent to the finite automaton constructed by Algorithm 2.20;
  rs, which denotes RS, is equivalent to the finite automaton constructed by Algorithm 2.22;
  r^*, which denotes R^*, is equivalent to the finite automaton constructed by Algorithm 2.24.

We leave a rigorous description of the algorithm that makes the above conversion as an exercise.

Example 2.6. Consider the regular expression $a(b|c)^*$, whose fully parenthesized version is $((a)(((b)|(c)))^*)$. This expression is constructed as follows:

$$a, b, c, b|c, (b|c)^*, a(b|c)^*$$

Consider the first three elementary subexpressions a, b, and c that denote languages $\{a\}$, $\{b\}$, and $\{c\}$, respectively. Based on the construction given in the proof of Lemma 2.18, we construct finite automata ${}_aM$, ${}_bM$, and ${}_cM$ that accept $\{a\}$, $\{b\}$, and $\{c\}$, respectively. From expressions b and c, we make expression $b|c$ that denotes $\{b\} \cup \{c\}$. Recall that ${}_bM$ and ${}_cM$ are equivalent to expressions b and c, respectively; that is, $L({}_bM) = \{b\}$ and $L({}_cM) = \{c\}$. Thus, with ${}_bM$ and ${}_cM$ as the input of Algorithm 2.20, we construct ${}_{b|c}M$ that accept $L({}_bM) \cup L({}_cM) = \{b\} \cup \{c\} = \{b, c\}$. From $b|c$, we can make $(b|c)^*$ that denotes $(\{b\} \cup \{c\})^* = \{b, c\}^*$. Recall that ${}_{b|c}M$ is equivalent to $b|c$. Therefore, with ${}_{b|c}M$ as the input of Algorithm 2.24, we construct a finite automaton ${}_{(b|c)^*}M$ equivalent to $(b|c)^*$. From a and $(b|c)^*$, we make $a(b|c)^*$ that denotes $\{a\}(\{b\} \cup \{c\})^* = \{a\}\{b,$

$c\}^*$. Automata ${}_aM$ and ${}_{(b|c)^*}M$ are equivalent to expressions a and $(b|c)^*$, respectively. Therefore, with ${}_aM$ and ${}_{(b|c)^*}M$ as the input of Algorithm 2.22, we construct a finite automaton ${}_{a(b|c)^*}M$ equivalent to $a(b|c)^*$ as desired. Figures 2.24 through 2.27 summarize this construction.

$$\begin{tikzcd}
\blacktriangleright {}_as \arrow[r, "a"] & \blacksquare {}_af & \blacktriangleright {}_bs \arrow[r, "b"] & \blacksquare {}_bf & \blacktriangleright {}_cs \arrow[r, "c"] & \blacksquare {}_cf
\end{tikzcd}$$

Figure 2.24 ***${}_aM$, ${}_bM$, and ${}_cM$.***

$$\begin{tikzcd}
\blacktriangleright {}_{b|c}s \arrow[r, "\varepsilon"] \arrow[dr, "\varepsilon"'] & {}_bs \arrow[r, "b"] & {}_bf \arrow[r, "\varepsilon"] & \blacksquare {}_{b|c}f \\
 & {}_cs \arrow[r, "c"] & {}_cf \arrow[ur, "\varepsilon"'] &
\end{tikzcd}$$

Figure 2.25 ***${}_{b|c}M$.***

$$\begin{tikzcd}
{}_{(b|c)^*}s \arrow[r, "\varepsilon"] \arrow[rrrrrr, bend left, "\varepsilon"] & {}_{b|c}s \arrow[r, "\varepsilon"] \arrow[dr, "\varepsilon"'] & {}_bs \arrow[r, "b"] & {}_bf \arrow[r, "\varepsilon"] & {}_{b|c}f \arrow[r, "\varepsilon"] \arrow[llll, bend left=60, "\varepsilon"] & \blacksquare {}_{(b|c)^*}f \\
 & & {}_cs \arrow[r, "c"] & {}_cf \arrow[ur, "\varepsilon"'] & &
\end{tikzcd}$$

Figure 2.26 ***${}_{(b|c)^*}M$.***

$$\begin{tikzcd}
\blacktriangleright {}_{a(b|c)^*}s \arrow[d, "a"] & & & & & & & \\
{}_af \arrow[r, "\varepsilon"] & {}_{(b|c)^*}s \arrow[r, "\varepsilon"] \arrow[rrrrrr, bend left, "\varepsilon"] & {}_{b|c}s \arrow[r, "\varepsilon"] \arrow[dr, "\varepsilon"'] & {}_bs \arrow[r, "b"] & {}_bf \arrow[r, "\varepsilon"] & {}_{b|c}f \arrow[r, "\varepsilon"] \arrow[lll, bend left=60, "\varepsilon"] & \blacksquare {}_{(b|c)^*}f \\
 & & & {}_cs \arrow[r, "c"] & {}_cf \arrow[ur, "\varepsilon"'] & &
\end{tikzcd}$$

Figure 2.27 ***${}_{a(b|c)^*}M$.***

■

Lemma 2.26. For any regular expression t, there exists an equivalent finite automaton.

Proof. Formalize the above conversion of a regular expression t to an equivalent finite automaton. By induction on the number of operations occurring in t, prove the equivalence. We leave this formalization and proof as an exercise.

■

There also exists a converse transformation that turns any finite automaton to an equivalent regular expression. However, during the design of a lexical analyzer, we need just the opposite transformation, which turn any regular expressions to an equivalent finite automata as already described. Therefore, we leave the proof of the next lemma as an exercise.

Lemma 2.27. For every finite automaton, there exists an equivalent regular expression.

We conclude this section by the next crucial result concerning the characterization of the family of regular languages.

Theorem 2.28. The regular expressions and the finite automata are equally powerful. They both characterize the family of regular languages.

Proof. This theorem follows from Lemmas 2.26 and 2.27.

■

2.3.2 Simplification of Finite Automata

The present section simplifies finite automata so they are easy to implement as the base of lexical analyzer.

Elimination of ε-rules

In general, finite automata may contain some ε-rules that have no input symbol on their left-hand side. According to these rules, they make ε-moves during which they change states without reading any input symbol. The theory of finite automata takes a great advantage of these steps very often; as a matter of fact, almost every algorithm given in Section 2.3.1 makes use of them. However, if a finite automaton can make ε-moves, then with some words, it may perform infinitely many valid sequences of moves, and this property complicates its implementation and use in practice. To illustrate, consider the finite automaton having these two rules $\blacktriangleright s \rightarrow \blacktriangleright s$ and $\blacktriangleright sa \rightarrow \blacksquare f$. The trivial language accepted by this automaton equals $\{a\}$, yet it accepts a by infinitely many accepting sequences of moves. Therefore, this section next explains how to remove ε-rules from a finite automaton without disturbing its language. Before this removal, however, we need to determine whether a finite automaton can make a sequence of moves from a state to another state without reading any symbol.

Goal. Given a finite automaton, $M = (\Sigma, R)$, and a set of states, $O \subseteq Q$, determine $\varepsilon\text{-}O = \{q \mid o \Rightarrow^* q$, where $o \in O, q \in Q\}$.

Gist. In other words, $\varepsilon\text{-}O$ contains a state $q \in Q$ if and only if M can reach q from a state in O without reading any input symbol. Initially, set $\varepsilon\text{-}O := O$ because M can reach any state in O without reading any input symbol by a sequence consisting of zero moves. If Q contains a state p such that $q \rightarrow p \in R$ with $q \in \varepsilon\text{-}O$, add p to $\varepsilon\text{-}O$. Repeat the extension of $\varepsilon\text{-}O$ in this way until this process does not change $\varepsilon\text{-}O$ at all. The resulting set $\varepsilon\text{-}O$ satisfies $\varepsilon\text{-}O = \{q \mid o \Rightarrow^* q$, where $o \in O$, $q \in Q\}$.

Next, we give Algorithm 2.29, which describes the construction of $\varepsilon\text{-}O$ formally. In addition, this algorithm is followed by Lemma 2.30, which rigorously verifies that the construction is correct.

Algorithm 2.29 ***States Reachable without Reading.***

Input • a finite automaton $M = (\Sigma, R)$;
• $O \subseteq {}_M Q$.

Output • $\varepsilon\text{-}O = \{q \mid o \Rightarrow^* q$, where $o \in O, q \in Q\}$.

Method

```
begin
    $\varepsilon\text{-}O := O$;
    repeat
        include $\{p \mid q \rightarrow p \in R$ and $q \in \varepsilon\text{-}O\}$ into $\varepsilon\text{-}O$
    until no change; {$\varepsilon\text{-}O$ cannot be further extended}
end.
```

Lemma 2.30. Let $M = (\Sigma, R)$ be a finite automaton and $O \subseteq Q$. Then, Algorithm 2.29 correctly determines $\varepsilon\text{-}O = \{q \mid o \Rightarrow^* q$, where $o \in O, q \in Q\}$.

Proof. Let $\varepsilon\text{-}O_i$ denote the set of states $\varepsilon\text{-}O$ contains after the ith iteration of the **repeat** loop, where $i = 0, 1, \ldots h$, for some $h \leq card(Q)$. To establish this lemma, we prove Claims A and B.

Claim A. For every $j \geq 0$ and every $p \in Q$, if $p \in \varepsilon\text{-}O_j$, then there exists $q \in O$ such that $q \Rightarrow^* p$ in M.

Proof of Claim A (by induction on $j \geq 0$).

Basis. Let $j = 0$. Observe that $p \in \varepsilon\text{-}O_0$ if and only if $p \in O$. As $p \Rightarrow^0 p$, the basis holds.

Induction Hypothesis. Assume that Claim A holds for all $j = 0, \ldots, i$, where i is a non-negative integer.

Induction Step. Let $p \in \varepsilon\text{-}O_{i+1}$. Distinguish these two cases—$p \in \varepsilon\text{-}O_j$, for some $j \leq i$ and $p \in \varepsilon\text{-}O_{i+1} - \varepsilon\text{-}O_i$

- Let $p \in \varepsilon\text{-}O_j$, for some $j \leq i$. By the induction hypothesis, there exists $q \in O$ such that $q \Rightarrow^j p$ in M, so the inductive step holds in this case.
- Let $p \in \varepsilon\text{-}O_{i+1} - \varepsilon\text{-}O_i$. Examine the **repeat** loop. Observe that there exists $o \rightarrow p \in R$ for some $o \in \varepsilon\text{-}O_i$. By the induction hypothesis, $q \Rightarrow^* o$ in M for some $q \in O$, so $q \Rightarrow^* o \Rightarrow p$ in M. Thus, $q \Rightarrow^* p$ in M, and the inductive step holds in this case as well.

Claim B. For all $j \geq 0$, if $q \Rightarrow^j p$ in M with $q \in O$, then $p \in \varepsilon\text{-}O$.

Proof of Claim B (by induction on $j \geq 0$).

Basis. Let $j = 0$. That is, $q \Rightarrow^0 p$ in M with $q \in O$, so $q = p$. Then, the algorithm includes p into $\varepsilon\text{-}O$ even before the first iteration of the **repeat** loop; formally, $p \in \varepsilon\text{-}O_0$. Thus, the basis holds.

Induction Hypothesis. Assume that Claim B holds for all $j = 0, \ldots, i$, where i is a non-negative integer.

Induction Step. Let $q \Rightarrow^{i+1} p$ in M with $q \in O$. We distinguish these two cases—$p \in \varepsilon\text{-}O_j$, for some $j \leq i$ and $p \notin \varepsilon\text{-}O_j$, for all $j \leq i$.

- Let $p \in \varepsilon\text{-}O_j$, for some $j \leq i$. Recall that no iteration of the **repeat** loop removes any states from $\varepsilon\text{-}O$. Therefore, $p \in \varepsilon\text{-}O$.
- Let $p \notin \varepsilon\text{-}O_j$, for all $j \leq i$. As $i + 1 \geq 1$, we can express $q \Rightarrow^{i+1} p$ in M as $q \Rightarrow^{i} o \Rightarrow p$, and the induction hypothesis implies $o \in \varepsilon\text{-}O$. R contains $o \rightarrow p$ because $o \Rightarrow p$ in M. As $o \in \varepsilon\text{-}O$ and $o \rightarrow p \in R$, the **repeat** loop adds p to O during iteration $i + 1$, so $p \in \varepsilon\text{-}O$.

By Claims A and B, $q \Rightarrow^{*} p$ in M with $q \in O$ if and only if $p \in \varepsilon\text{-}O$. Hence, Algorithm 2.29 correctly determines $\varepsilon\text{-}O = \{q |\ o \Rightarrow^{*} q$, where $o \in O, q \in Q\}$, so the lemma holds.

■

Example 2.7 ***States Reachable without Reading.*** Return to the finite automaton discussed in Example 2.2 (see Figure 2.3). Denote this automaton by U. U is thus defined as ▸$sa \rightarrow$ ▸s, ▸$s \rightarrow p$, $pb \rightarrow p$, $pb \rightarrow$ ▪f, ▸$s \rightarrow q$, $qc \rightarrow q$, $qc \rightarrow$ ▪f, ▪$fa \rightarrow$ ▪f. Consider $O = \{s\}$. With U and O, Algorithm 2.29 initially sets $\varepsilon\text{-}O = \{s\}$. The first iteration of the **repeat** loop adds p and q to $\varepsilon\text{-}O$ because ▸$s \rightarrow p$ and ▸$s \rightarrow q$ are the ε-rules with s on the left-hand sides, so $\varepsilon\text{-}O = \{s, q, p\}$ after this iteration. As q and p represent new states in $\varepsilon\text{-}O$, the algorithm performs the second iteration of the **repeat** loop. However, this iteration adds no new state to $\varepsilon\text{-}O$, so the **repeat** loop exits. Figure 2.28 summarizes these iterations of Algorithm 2.29.

Iteration	$\varepsilon\text{-}\{s\}$
0	s
1	s, p, q
2	s, p, q (**no change**)

Figure 2.28 ***Summary of Example*** **2.7.**

■

We are now ready to convert any finite automaton to an equivalent finite automaton without ε-rules.

Goal. Convert any finite automaton $M = ({}_M\Sigma, {}_MR)$ to an equivalent finite automaton $N = ({}_N\Sigma, {}_NR)$ without ε-rules.

Gist. Use Algorithm 2.29 to find out whether $\varepsilon\text{-}\{{}_Ms\} \cap {}_MF \neq \emptyset$, which implies that M accepts ε; if so, include ${}_Ms$ into N's final states in order that N accepts ε as well. Furthermore, set ${}_NR$ to $\{qa \rightarrow p|\ q \in {}_MQ, a \in {}_M\Delta, oa \rightarrow p$ in M for some $o \in \varepsilon\text{-}\{q\}\}$, where $\varepsilon\text{-}\{q\}$ is constructed by Algorithm 2.29. In this way, $qa \rightarrow p \in {}_NR$ simulates M's sequence of moves of the form $qa \Rightarrow^{*} oa \Rightarrow p$.

Algorithm 2.31 ***Removal of*** **ε*-Rules.***

Input • a finite automaton $M = ({}_M\Sigma, {}_MR)$.

Output • a finite automaton $N = ({}_N\Sigma, {}_NR)$ without ε-rules such that $L(N) = L(M)$.

Method

```
begin
    $_N\Sigma := {}_M\Sigma$ with $_N\Delta := {}_M\Delta$, $_NQ := {}_MQ$, $_Ns := {}_Ms$, and
```

$\quad {}_NF := \{q \mid q \in {}_MQ, \varepsilon\text{-}\{q\} \cap {}_MF \neq \emptyset\}$;
$\quad {}_NR := \{qa \rightarrow p \mid q \in {}_MQ, a \in {}_M\Delta, oa \rightarrow p$ in M for some $o \in \varepsilon\text{-}\{q\}\}$
end.

Note that we can find out whether ${}_Ms \Rightarrow^* f$ in M for any final state $f \in {}_MF$ and $q \Rightarrow^* o$ and $oa \Rightarrow^* p$ in M for any $o, p, q \in {}_MQ$ and $a \in {}_M\Delta$ by using Algorithm 2.29.

Lemma 2.32. Let $M = ({}_M\Sigma, {}_MR)$ be a finite automaton. Then, Algorithm 2.31 correctly constructs a finite automaton $N = ({}_N\Sigma, {}_NR)$ without ε-rules such that $L(N) = L(M)$—that is, N and M are equivalent.

Proof. Notice that Algorithm 2.31 produces ${}_N\Delta = {}_M\Delta$, ${}_NQ = {}_MQ$, and ${}_Ns = {}_Ms$, so in the rest of the proof, we simply use Δ, Q, and s to denote the set of input symbols, states, and the start state in both automata, respectively. However, we have to distinguish ${}_MF$ from ${}_NF$ because ${}_MF \subseteq {}_NF \subseteq {}_MF \cup \{s\}$, so they may be different in containing s.

Clearly, ${}_NR$ contains no ε-rules. To establish $L(N) = L(M)$, we first prove the following claim for all non-empty strings.

Claim. For every $q, p \in Q$ and $w \in \Delta^+$, $qw \Rightarrow^* p$ in N if and only if $qw \Rightarrow^* p$ in M.

Proof of the Claim.

Only if. The only-if part says that for every $q, p \in Q$ and $w \in \Delta^+$, $qw \Rightarrow^* p$ in N implies $qw \Rightarrow^* p$ in M. By induction on $|w| \geq 1$, we next prove this implication.

Basis. Let $|w| = 1$ and $qw \Rightarrow^* p$ in N. Since $|w| = 1$, w is a single symbol from Δ. As R contains no ε-rules, $qw \Rightarrow^* p$ actually means $qw \Rightarrow p$ in N, so $qw \rightarrow p$ in R. By the definition of R, $qw \rightarrow p \in R$ for some $o \in \varepsilon\text{-}\{q\}$, so $q \Rightarrow^* o$ and $ow \Rightarrow p$ in M. Consequently, $qw \Rightarrow^* p$ in M.

Induction Hypothesis. Assume that the only-if part of the claim holds for all $w \in {}_M\Delta^+$ with $|w| \leq n$, where n is a positive integer.

Induction Step. Let $qw \Rightarrow^* p$ in N, where $q, p \in Q$ and $w \in \Delta^+$ with $|w| = n + 1$. Because $|w| \geq 1$ and R contains no ε-rules, we can express $qw \Rightarrow^* p$ as $qva \Rightarrow^* ha \Rightarrow p$, where $h \in Q$, $a \in \Delta$, $v =$ **prefix**(w, n). As $qv \Rightarrow^* h$ and $|v| = n$, $qv \Rightarrow^* h$ in M by the inductive hypothesis. Since $ha \Rightarrow p$ in N, $ha \rightarrow p \in R$. By the definition of R, $ha \Rightarrow^* oa \Rightarrow p$ in M, where $o \in Q$. Putting $qv \Rightarrow^* h$ and $ha \Rightarrow^* oa \Rightarrow p$ in M together, $qva \Rightarrow^* ha \Rightarrow^* h$ in M. Because $va = w$, $qw \Rightarrow^* p$ in M. Thus, the only-if part of the claim holds.

If. The if-part of the claim is left as an exercise.

Next, we prove $w \in L(N)$ if and only if $w \in L(M)$, for all $w \in \Delta^*$. Consider the above claim for $q = s$. That is, $sw \Rightarrow^* p$ in N if and only if $sw \Rightarrow^* p$ in M, where $p \in Q$. Recall that ${}_NF = \{q \mid q \in {}_MQ, \varepsilon\text{-}\{q\} \cap {}_MF \neq \emptyset\}$. By this definition of ${}_NF$, $sw \Rightarrow^* p$ in N with $p \in {}_NF$ if and only if $sw \Rightarrow^* p \Rightarrow^* f$ in M with $f \in {}_MF$. Therefore, $w \in L(N)$ if and only if $w \in L(M)$, for all $w \in \Delta^*$, so the lemma holds.
■

Example 2.8 ***Removal of ε-Rules.*** Return to the finite automaton U discussed in Example 2.7. Consider $O = \{s\}$. With U as its input, Algorithm 2.31 produces an equivalent finite automaton V without ε-rules as follows. As $\varepsilon\text{-}\{{}_Us\} \cap {}_UF = \emptyset$, ${}_UF = {}_VF = \{\blacksquare f\}$. Since $pb \rightarrow \blacksquare f \in {}_UR$ and $p \in$

ε-$\{{}_Us\}$, $sb \to {\blacksquare}f \in {}_VR$. As $pb \to p \in {}_UR$ and $p \in \varepsilon$-$\{{}_Us\}$, $sb \to p$ belongs to ${}_VR$ as well. As a result, we obtain V defined as

$${\blacktriangleright}sa \to {\blacktriangleright}s,\ {\blacktriangleright}sb \to p,\ {\blacktriangleright}sb \to {\blacksquare}f,\ pb \to p,\ pb \to {\blacksquare}f,\ {\blacktriangleright}sc \to q,\ {\blacktriangleright}sc \to {\blacksquare}f,\ qc \to q,\ qc \to {\blacksquare}f, \text{ and } {\blacksquare}fa \to {\blacksquare}f$$

Figure 2.4 describes V graphically. A quite rigorous discussion of the construction of V by Algorithm 2.31 is left as an exercise.

■

Theorem 2.33. The finite automata without ε-rules characterize the family of regular languages.

Proof. By Algorithm 2.31 and Lemma 2.32, for every finite automaton, there exists an equivalent finite automaton without ε-rules. On the other hand, any finite automaton without ε-rules represents a special case of a finite automaton. Thus, this theorem follows from Theorem 2.28.

■

Determinism

A deterministic finite automaton makes no more than one move from any configuration. As a result, with any input string, it makes a unique sequence of moves, and this property is obviously highly appreciated in practice. Next, we explain how to make any finite automaton deterministic.

Goal. Convert any finite automaton $M = ({}_M\Sigma, {}_MR)$ without ε-rules to an equivalent deterministic finite automaton $N = ({}_N\Sigma, {}_NR)$.

Gist. M and N use the same input alphabet, Δ. For any $O \subseteq {}_MQ$, introduce $\langle O\rangle$ as a state of ${}_NQ$; in addition, if $O \cap {}_MF \neq \emptyset$, include $\langle O\rangle$ into ${}_NF$. Set ${}_NR$ to $\{\langle O\rangle a \to \langle P\rangle |\ \langle O\rangle, \langle P\rangle \in {}_MQ, a \in \Delta\ , p \in P$ if and only if $oa \to p \in {}_MR$ for some $o \in O\}$. At this point, $\langle O\rangle a \Rightarrow \langle P\rangle$ in N if and only if $oa \Rightarrow p$ for some $o \in O, p \in P$. Consequently, $\langle\{{}_Ms\}\rangle w \Rightarrow^* \langle D\rangle$ in N if and only if ${}_Msw \Rightarrow^* p$, where $p \in D$ and $w \in \Delta^*$. Specifically, for every $\langle E\rangle \in {}_NF$, $\langle\{{}_Ms\}\rangle w \Rightarrow^* \langle E\rangle$ in N if and only if ${}_Msw \Rightarrow^* p$ in M, for some $p \in E \cap {}_MF$. In other words, N accepts w if and only if M accepts w, so $L(N) = L(M)$.

Algorithm 2.34 ***Determinism.***

Input • a finite automaton $M = ({}_M\Sigma, {}_MR)$ without ε-rules.

Output • a deterministic finite automaton $N = ({}_N\Sigma, {}_NR)$ such that $L(N) = L(M)$.

Method

```
begin
    $_N\Delta := {}_M\Delta$;
    $_NQ := \{\langle O\rangle |\ \emptyset \subset O \subseteq {}_MQ\}$;
    $_NF := \{\langle O\rangle |\ \langle O\rangle \in {}_NQ$ and $O \cap {}_MF \neq \emptyset\}$;
    $_Ns := \langle\{{}_Ms\}\rangle$;
    $_NR := \{\langle O\rangle a \to \langle P\rangle |\ \langle O\rangle, \langle P\rangle \in {}_NQ, a \in {}_N\Delta$, and $p \in P$ if and only if $oa \to p \in {}_MR$
            for some $o \in O\}$
end.
```

Lemma 2.35. Algorithm 2.34 correctly converts a finite automaton $M = ({}_M\Sigma, {}_MR)$ without ε-rules to a deterministic finite automaton $N = ({}_N\Sigma, {}_NR)$ such that $L(N) = L(M)$.

Proof. Notice that Algorithm 2.34 produces ${}_N\Delta = {}_M\Delta$, so in the rest of the proof, we simply use Δ to denote the set of input symbols. Clearly, for any non-empty subset $O \subseteq {}_MQ$ and any input symbol $a \in \Delta$, there exists a single set, $P \subseteq {}_MQ$, satisfying the equivalence $p \in P$ if and only if $oa \rightarrow p \in {}_MR$ for some $o \in O$. Consequently, for any $\langle O \rangle \in {}_MQ$ and $a \in \Delta$, there exists a single state, $\langle P \rangle \in {}_NQ$, such that $\langle O \rangle a \rightarrow \langle P \rangle \in {}_NR$, so N is deterministic. A proof that $L(N) = L(M)$ is sketched in the notes preceding Algorithm 2.34, and its rigorous version is left as an exercise.

■

Example 2.9 ***Determinism.*** Consider the finite automaton V without ε-rules constructed in Example 2.8. Observe that this automaton is non-deterministic; for instance, $sb \rightarrow p$ and $sb \rightarrow f$ violate the requirement of determinism. With V as its input, Algorithm 2.34 produces an equivalent deterministic finite automaton W as follows. W's states are

$$\langle s \rangle, \langle p \rangle, \langle q \rangle, \langle f \rangle, \langle s, p \rangle, \langle s, q \rangle, \langle s, f \rangle, \langle p, q \rangle, \langle p, f \rangle, \langle q, f \rangle, \langle s, p, q \rangle, \langle s, p, f \rangle, \langle s, q, f \rangle, \langle p, q, f \rangle, \langle s, p, q, f \rangle$$

Consider $\langle s \rangle$ and b. In V, $\{sb \rightarrow p, sb \rightarrow f\}$ is the set of all rules with sb on the left-hand side, so $\langle s \rangle b \rightarrow \langle p, f \rangle$ is a rule in W. Consider state $\langle q, f \rangle$ in W and b. In V, $\{qc \rightarrow q, qc \rightarrow f\}$ is the set of all rules with the left-hand side equal to qc or fc (in fact, there is no rule with fc on its left-hand side). Thus, the algorithm introduces $\langle q, f \rangle c \rightarrow \langle q, f \rangle$ into W's rules. The resulting deterministic finite automaton W is given in Figure 2.5.

■

Algorithm 2.34 works simply, but its exponential increase in the number of states represents a drawback. Some of the introduced states may be obviously completely irrelevant regarding the accepted language, so they are introduced uselessly. These useless states obviously contain the states that can never be reached. To put it more precisely, in a finite automaton $M = (\Sigma, R)$, a state $q \in Q$ is *reachable* if there exists $w \in \Delta^*$ such that $sw \Rightarrow^* q$ in M; otherwise, q is *unreachable* in M. Next, we transform any finite automaton without ε-rules to an equivalent deterministic finite automaton that contains only reachable states because the unreachable states cannot affect the accepted language at all.

Goal. Convert any finite automaton $M = ({}_M\Sigma, {}_MR)$ without ε-rules to a deterministic finite automaton $N = ({}_N\Sigma, {}_NR)$ such that $L(M) = L(N)$ and ${}_NQ$ contains only reachable states.

Gist. The basic idea behind the next algorithm resembles the idea behind the previous algorithm except that a new state is introduced into ${}_NQ$ only if it is reachable. We start with ${}_NQ$ containing the start state $\langle \{{}_Ms\} \rangle$. Then, only if $P = \{p | \ oa \Rightarrow p \text{ some } o \in O\}$ is non-empty, where ${}_NQ$ already contains $\langle O \rangle$, $O \subseteq {}_MQ$, and $a \in \Delta$, we add $\langle P \rangle$ to ${}_NQ$ and include $\langle O \rangle a \rightarrow \langle P \rangle$ into ${}_NR$.

Algorithm 2.36 ***Determinism with Reachable States.***

Input • a finite automaton $M = ({}_M\Sigma, {}_MR)$ without ε-rules.

Output • a deterministic finite automaton $N = ({}_N\Sigma, {}_NR)$ such that $L(N) = L(M)$ and ${}_NQ$ contains only reachable states.

Method

```
begin
    $_N\Delta := {}_M\Delta$;
    $_NQ := \{\langle\{_Ms\}\rangle\}$;
    $_Ns := \langle\{_Ms\}\rangle$;
    if $_Ms \in {}_MF$ then
        $_NF := \{\langle\{_Ms\}\rangle\}$
    else $_NF := \emptyset$;
    $_NR := \emptyset$;
    repeat
        if $a \in \Delta$, $\langle O\rangle \in {}_NQ$, $O \subseteq {}_MQ$, and $P = \{p \mid oa \Rightarrow p$ some $o \in O\} \neq \emptyset$ then
        begin
            add $\langle O\rangle a \rightarrow \langle P\rangle$ to $_NR$ and add $\langle P\rangle$ to $_NQ$;
            if $P \cap {}_MF \neq \emptyset$ then
                add $\langle P\rangle$ to $_NF$
        end
    until no change
end.
```

Example 2.10 ***Determinism with Reachable States.*** Consider the finite automaton V without ε-rules constructed in Example 2.8. With this automaton as its input, Algorithm 2.36 produces an equivalent deterministic finite automaton that coincides with the automaton in Figure 2.5 except for the names of the states. Observe that the resulting automaton has only reachable states. A detailed discussion of this example is left as an exercise.

■

The useless states of finite automata also contain the states from which no final state can be reached. More exactly, in a finite automaton $M = (\Sigma, R)$, a state $q \in Q$ is *terminating* if there exists $w \in \Delta^*$ such that $qw \Rightarrow^* f$ in M with $f \in {}_MF$; otherwise, q is *non-terminating*. The transformation of any finite automaton without ε-rules to an equivalent deterministic finite automaton that contains only reachable and terminating states is left as an exercise.

Complete Specification

Recall that a deterministic finite automaton $M = (\Sigma, R)$ is completely specified if for every state $q \in Q$ and every input symbol $a \in \Delta$, there is exactly one rule with qa on the left-hand side. As a result, a completely specified deterministic finite automaton cannot get stuck during reading an input string because it always reads all symbols of every input string. In practice, we often want to implement automata satisfying this property (see Algorithms 2.8 and 2.9), and that is why we describe the next conversion.

Goal. Convert any deterministic finite automaton $M = ({}_M\Sigma, {}_MR)$ to an equivalent completely specified deterministic finite automaton $N = ({}_N\Sigma, {}_NR)$. In addition, make this conversion so that if M contains only reachable states, so does N.

Gist. Set $_NQ$ to $_MQ \cup \{o\} = \emptyset$, where o is a new non-final state. Set $_NR$ to $_MR \cup \{qa \rightarrow o \mid q \in {}_NQ$, $a \in {}_M\Delta$, and $qa \neq$ ***lhs***(r) for any $r \in {}_MR\}$. Observe that o is a non-terminating state. Furthermore, notice that $_NR$ contains $oa \rightarrow o$ for every $a \in {}_M\Delta$. Obviously, N is a completely specified deterministic finite automaton. Consider an input string, w. If M can make a move from a configuration, N simulates the move, so M accepts w if and only if N accepts w. If M reaches a configuration in which it gets stuck because it has no rule to make another move, N enters o. Remaining in o, N reads the rest of an input string w until it reaches its end. Thus, N rejects w if and only if M rejects w. In other words, N is equivalent to M. In addition, observe that if M has only reachable states, so does N.

A precise algorithm making this conversion of M to N is left as an exercise. Example 2.5 discusses a completely specified deterministic finite automaton and its implementation.

Recall that some variants of finite automata are *equally powerful* if they all accept the same family of languages (see Section 1.3). Consequently, for any two equally powerful variants of finite automata, Ξ and Ψ, Ξ contains an automaton that accepts a language L if and only if Ψ contains an automaton that accepts L. The next theorem summarizes the results achieved earlier in this section.

Theorem 2.37. The following variants of finite automata are equally powerful:

- finite automata
- finite automata without ε-rules
- deterministic finite automata
- deterministic finite automata with only reachable and terminating states
- completely specified deterministic finite automata
- completely specified deterministic finite automata with only reachable states

2.3.3 Non-Regular Lexical Constructs

The lexical analysis techniques described in the previous chapter are based upon the regular expressions and finite automata, whose power is limited to the family of regular languages. Therefore, if some lexemes represent non-regular languages, these techniques are inapplicable, so a regular redesign of these lexemes is highly recommendable. Thus, the task of demonstrating that a language L is not regular obviously plays an important role in the lexical analysis theory. Strictly speaking, this demonstration requires to prove that none of all possible finite automata accepts L. Frequently, we can simplify such a proof by demonstrating that L does not satisfy some conditions that all regular languages satisfy, so L cannot be regular. This section presents some conditions of this kind. First, it gives conditions of the following pumping lemma for regular languages, then it presents some useful closure properties to prove that a language is not regular.

Pumping Lemma

This lemma says that for every regular language L, there is a constant $c \geq 1$ such that every word $z \in L$ with $|z| \geq c$ can be expressed as $z = uvw$ with $v \neq \varepsilon$ so that L also contains $uv^m w$, for every non-negative integer m. This crucial lemma results from a relationship between $|z|$ and the sequence of steps that a deterministic finite automaton $M = (\Sigma, R)$ makes with z on the input tape. More precisely, let $L = L(M)$ and $z \in L(M)$ with $|z| \geq card(Q)$. As M reads a symbol during every move, M accepts z by making a sequence of $|z|$ moves; therefore, M visits a state $q \in Q$ twice when accepting z. These two visits of q determine the decomposition of z as $z = uvw$ so that

A. u takes M from the start state to the first visit of q;
B. v takes M from the first visit of q to the second visit of q;
C. w takes M from the second visit of q to a final state f.

M can obviously iterate the sequence of moves between the two visits of q in B m times, for any $m \geq 0$, and during each of these iteration, M reads v. Consequently, M accepts every string of the form $uv^m w$.

Lemma 2.38 ***Pumping Lemma for Regular Languages.*** Let L be an infinite regular language. Then, there exists a positive integer $k \geq 1$ such that every word $z \in L$ satisfying $|z| \geq k$ can be expressed as $z = uvw$, where $0 < |v| \leq |uv| \leq k$, and $uv^m w \in L$, for all $m \geq 0$.

Proof. Let L be a regular language, and let $M = (\Sigma, R)$ be a deterministic finite automaton such that $L(M) = L$. Set $k = card(Q)$. Consider any string $z \in L$ with $|z| \geq k$. As $z \in L$, $sz \Rightarrow^* f\ [\rho]$, where $\rho \in R^*$ is a sequence of rules from R. Because M makes no ε-moves and, thus, reads a symbol on every step, $|\rho| = |z|$. As $|z| \geq k$ and $k = card(Q)$, $|\rho| \geq card(Q)$. Therefore, during $sz \Rightarrow^* f\ [\rho]$, M enters some states more than once. Consider the shortest initial part of this sequence of moves during which M visits the same state, $q \in Q$. More formally, express $sz \Rightarrow^* f\ [\rho]$ as

$$\begin{array}{lll} suvw & \Rightarrow^* qvw & [\sigma] \\ & \Rightarrow^* qw & [\tau] \\ & \Rightarrow^* f & [\upsilon] \end{array}$$

where $\rho = \sigma\tau\upsilon$, $q \in Q$, $|\tau| \geq 1$, and M visits any state from $(Q - \{q\})$ no more than once during $suvw \Rightarrow^* qw$.

The following claims verify that uvw satisfies the conditions stated in the lemma.

Claim A. $0 < |v| \leq |uv| \leq k$.

Proof of Claim A. As $|v| = |\tau|$ and $|\tau| \geq 1$, $0 < |v|$. Clearly, $|v| \leq |uv|$. During $suvw \Rightarrow^* qw\ [\sigma\tau]$, M visits q twice and any other state no more than once. Consequently, $|\sigma\tau| \leq card(Q)$. As $|uv| = |\sigma\tau|$ and $card(Q) = k$, $|uv| \leq k$. Thus, $0 < |v| \leq |uv| \leq k$.

Claim B. For all $m \geq 0$, $uv^mw \in L$.

Proof of Claim B. Let $m \geq 1$. M accepts uv^mw by repeating the sequence of moves according to τ m times; formally,

$$\begin{array}{lll} suv^mw & \Rightarrow^* qv^mw & [\sigma] \\ & \Rightarrow^* qv^{m-1}w & [\tau] \\ & \vdots & \\ & \Rightarrow^* qvw & [\tau] \\ & \Rightarrow^* qw & [\tau] \\ & \Rightarrow^* f & [\upsilon] \end{array}$$

In brief,

$$\begin{array}{lll} suv^mw & \Rightarrow^* qv^mw & [\sigma] \\ & \Rightarrow^* qw & [\tau^m] \\ & \Rightarrow^* f & [\upsilon] \end{array}$$

Thus, for all $m \geq 1$, $uv^mw \in L$. Consider uv^mw with $m = 0$; that is, $uv^mw = uw$. M accepts uw so it completely omits the sequence of moves according to τ; that is,

$$\begin{array}{lll} suw & \Rightarrow^* qw & [\sigma] \\ & \Rightarrow^* f & [\upsilon] \end{array}$$

Consequently, for all $m \geq 0$, $uv^mw \in L$.

■

Example 2.11 ***Proof of the Pumping Lemma.*** To illustrate the technique used in the previous proof, consider the regular language $L = \{a\}\{bc\}^*\{b\}$ and the deterministic finite automaton M

defined by rules 1: $\blacktriangleright sa \rightarrow p$, 2: $pb \rightarrow \blacksquare f$, and 3: $\blacksquare fc \rightarrow p$. Observe that $L = L(M)$. As M has three states, set $k = 3$. Consider $z = abcb \in L$. As $|z| = 4$ and $k = 3$, z satisfies $|z| \geq k$. M accepts z as $\blacktriangleright sabcb \Rightarrow^* \blacksquare f$ [1232], which can be expressed move by move as

$$\begin{array}{lll} \blacktriangleright sabcb & \Rightarrow pbcb & [1] \\ & \Rightarrow \blacksquare fcb & [2] \\ & \Rightarrow pb & [3] \\ & \Rightarrow \blacksquare f & [2] \end{array}$$

The shortest initial part of this sequence of moves that contains two occurrences of the same state

$$\begin{array}{lll} \blacktriangleright sabcb & \Rightarrow pbcb & [1] \\ & \Rightarrow \blacksquare fcb & [2] \\ & \Rightarrow pb & [3] \end{array}$$

where p is the state M visits twice. Following the above proof, we express $\blacktriangleright sabcb \Rightarrow^* \blacksquare f$ [1232] as

$$\begin{array}{lll} \blacktriangleright suvw & \Rightarrow^* qvw & [1] \\ & \Rightarrow^* qw & [23] \\ & \Rightarrow^* \blacksquare f & [2] \end{array}$$

with set $u = a$, $v = bc$, and $w = b$. Observe that $0 < |v| = 1 \leq |uv| = 2 \leq k = 3$, so the first two conditions of Lemma 2.38 hold. As $a(bc)^m b = u(v)^m w \in L$, for all $m \geq 0$, the other condition holds as well. To illustrate this condition in a more specific way, consider $m = 2$ in $a(bc)^m b$—that is, $a(bc)^2 b = abcbcb$. M accepts this string by iterating the partial computation according to 23 twice as

$$\begin{array}{lll} \blacktriangleright sabcbcb & \Rightarrow^* pbcbcb & [1] \\ & \Rightarrow^* pbcb & [23] \\ & \Rightarrow^* pb & [23] \\ & \Rightarrow^* \blacksquare f & [2] \end{array}$$

■

Applications of the Pumping Lemma

As already noted, we mainly use Lemma 2.38 to prove that a given language L is not regular. Creating a proof like this always requires some ingenuity. Nevertheless, in principle, it is usually made by contradiction as follows:

A. Assume that L is regular;
B. Consider the constant k from the pumping lemma, and select a word $z \in L$ whose length depends on k so $|z| \geq k$ is surely true;
C. Consider all possible decompositions of z into uvw, satisfying $|uv| \leq k$ and $v \neq \varepsilon$; for each of these decompositions, demonstrate that there exists $m = 0$ such that $uv^m w \notin L$, which contradicts the third condition in Lemma 2.38;
D. The contradiction obtained in C implies that the assumption in A was incorrect, so L is not regular.

Example 2.12 ***Non-Regular Identifiers.*** Let I be the language containing all identifiers defined as alphanumeric strings starting with a letter and containing the same number of letters and digits. At a glance, these identifiers resemble the common definition of identifiers in such languages as

Pascal. As opposed to I, however, the common definition does not require the same number of letter and digits, and this difference is more than significant. Indeed, while the set of Pascal identifiers obviously represent a regular language, I is not regular as demonstrated by the pumping lemma next.

A. Assume that I is regular.
B. As I is regular, there exists a natural number k satisfying Lemma 2.38. Set $z = a^k1^k$. Notice that $z \in I$. As $|z| = 2k \geq k$, $|z| \geq k$.
C. By Lemma 2.38, z can be written as $z = uvw$ so the conditions of the pumping lemma hold. As $0 < |v| \leq |uv| \leq k$, $v \in \{a\}^+$. Consider $uv^0w = uw$. Then, $uw = a^j1^k$ with $j = k - |v|$, so $uw \notin I$. However, from the pumping lemma, $uv^mw \in I$, for all $m \geq 0$, so $uv^0w = uw \in I$—a contradiction.
D. By the contradictions in C, I is not regular.

■

The previous example has represented a straightforward application of the pumping lemma. The following example uses this lemma in a more ingenious way.

Example 2.13 ***Primes are Non-Regular.*** Clearly, $\{a^n| \ n$ is a positive integer$\}$ is regular. However, as we next demonstrate, its sublanguage $P = \{a^n| \ n$ is prime$\}$ is not (a positive integer n is prime if its only positive divisors are 1 and n).

Assume that P is regular. Then, there exists a positive integer k satisfying the pumping lemma. As P is infinite, there surely exists a string $z \in P$ such that $|z| \geq k$. By Lemma 2.38, z can then be written as $z = uvw$, so the three conditions of the pumping lemma hold. Consider uv^mw with $m = |uw| + 2|v| + 2$. As $|uv^mw| = |uw| + m|v| = |uw| + (|uw| + 2|v| + 2)|v| = (|uw| + 2|v|) + (|uw| + 2|v|)|v| = (|uw| + 2|v|)(1 +|v|)$, integer $|uv^mw|$ is not prime and, thus, $uv^mw \notin P$. By the pumping lemma, however, $uv^mw \in P$—a contradiction. Thus, P is not regular. As P is non-regular, no finite automaton accepts P (see Theorem 2.28). Consequently, the finite automata are unable to decide whether a positive integer is prime; simply stated, they are not strong enough to handle the primes.

■

Inapplicability of the Pumping Lemma. As the previous two examples have illustrated, we find Lemma 2.38 useful when disproving that a language is regular. On the other hand, we cannot use this lemma to prove that a language is regular because some non-regular languages satisfy the pumping-lemma conditions as well. That is, proving that a language satisfies these conditions does not necessarily imply that the language is regular. In the next example, we present a language of this kind.

Example 2.14 ***Non-Regular Language that Satisfies the Pumping Lemma Conditions.*** We demonstrate that $L = \{a^mb^nc^n| \ m \geq 1$ and $n \geq 0\} \cup \{b^mc^n| \ m, n \geq 0\}$ is a non-regular language that satisfies the conditions of the pumping lemma for regular languages.

To demonstrate that L satisfies the pumping lemma conditions, set $k = 1$. Consider any string $z \in L$ and $|z| = k$. As $z \in L$, either $z \in \{a^mb^nc^n| \ m \geq 1$ and $n \geq 0\}$ or $z \in \{b^mc^n| \ m, n \geq 0\}$.

Let $a^mb^nc^n$, for some $m \geq 1$ and $n \geq 0$. Express z as $z = uvw$ with $u = \varepsilon$, $v = a$, and $w = a^{m-1}b^nc^n$. As $v \neq \varepsilon$ and $|uv| = k = 1$, the first two conditions of the pumping lemma hold. Notice that $a^hb^nc^n$, for all $h \geq 1$. In other words, $uv^hw \in L$, for all $h \geq 0$, so the third condition holds as well.

Let $z = b^mc^n$, for some $m, n \geq 0$. Express z as $z = uvw$ with $u = \varepsilon$, v is the leftmost symbol of b^mc^n, and w is the remaining suffix of z. For $m = 0$, $v = c$ and $w = c^{n-1}$, and all the three conditions hold. For $m \geq 1$, $v = b$ and $w = b^{m-1}c^n$, and all the three conditions hold in this case as well.

Thus, L satisfies the conditions of the pumping lemma for regular languages. To see the reason why L is a non-regular language, consider its sublanguage $\{a^m b^n c^n \mid m \geq 1 \text{ and } n \geq 0\}$. To accept $\{a^m b^n c^n \mid m \geq 1 \text{ and } n \geq 0\}$, a finite automaton needs to record the number of *a*s before it reads the *b*s. With a finite number of states, it cannot make this record because n runs to infinity. Thus, no finite automaton accepts L, so L is not regular by Theorem 2.28. A rigorous proof that L is non-regular is left as an exercise.

■

Closure properties

The family of regular languages is *closed* under a language operation o if this family contains every language that results from o applied to any regular languages; at this point, we also say that *o preserves* the family of regular languages. Combined with the pumping lemma, the closure properties of regular languages are often very helpful in proofs that a language L is non-regular. Indeed, assuming that L is regular, we first transform this language to a simpler language K by using some operations under which the family of regular languages is closed. Then, by the pumping lemma, we prove that K is non-regular, so L is not regular either.

This section proves that the family of regular languages is closed under these operations:

- union
- concatenation
- closure
- complement
- intersection
- regular substitution
- finite substitution
- homomorphism

The present section demonstrates most of these closure properties *effectively* in terms of finite automata. That is, to prove that a given operation o preserves the family of regular languages, this section presents an algorithm that converts any finite automata to a finite automaton that accepts the language resulting from o applied to the languages accepted by the input automata.

Union, concatenation, and closure. The three closure properties stated in the next theorem immediately follow from Definition 2.1. Algorithms 2.19, 2.21, and 2.23 demonstrate these closure properties effectively.

Theorem 2.39. The family of regular languages is closed under union, concatenation, and closure.

Complement and Intersection. To prove that the family of regular languages is closed under complement, we demonstrate that ${}_M\Delta^* - L(M)$ is regular, for every finite automaton $M = ({}_M\Sigma, {}_MR)$.
Goal. Convert any finite automaton M to a finite automaton N such that $L(N) = {}_M\Delta^* - L(M)$.

Gist. Consider any finite automaton $M = ({}_M\Sigma, {}_MR)$. Without any loss of generality, suppose that M is a completely specified deterministic finite automaton (see Theorem 2.37). Change M by making every non-final state final and every final state non-final. Let N be the finite automaton resulting from this change. As M is completely specified, so is N. Thus, N completely reads every string from ${}_M\Delta^*$. N accepts an input string if and only if M rejects the string. Thus, $L(N) = {}_M\Delta^* - L(M)$.

The following algorithm formally describes the trivial conversion of M to N, which accepts the complement of $L(M)$.

Algorithm 2.40 ***Finite Automaton for Complement.***

Input • a completely specified deterministic finite automaton $M = ({}_M\Sigma, {}_MR)$.

Output • a completely specified deterministic finite automaton $N = ({}_N\Sigma, {}_NR)$ such that $L(N) = {}_M\Delta^* - L(M)$.

Method

```
begin
    $_N\Delta := {}_M\Delta$;
    $_NQ := {}_MQ$;
    $_Ns := {}_Ms$;
    $_NR := {}_MR$;
    $_NF := {}_MQ - {}_MF$
end.
```

Lemma 2.41. Let $M = ({}_M\Sigma, {}_MR)$ be a completely specified deterministic finite automaton. With M as its input, Algorithm 2.40 produces a completely specified deterministic finite automaton $N = ({}_N\Sigma, {}_NR)$ such that $L(N) = {}_M\Delta^* - L(M)$.

Proof. As M is completely specified, so is N. Furthermore, M's complete specification implies that with every $w \in {}_M\Delta^*$, $sw \Rightarrow^* p$ in M, for some $p \in Q$. Thus, w $\notin L(M)$ if and only if $sw \Rightarrow^* p$ in M, for some $p \in {}_MQ - {}_MF$. Since ${}_NF = {}_MQ - {}_MF$, $sw \Rightarrow^* p$ in M with $p \in {}_MQ - {}_MF$ if and only if $sw \Rightarrow^* p$ in N with $p \in {}_NF$. Obviously, $sw \Rightarrow^* p$ in N with $p \in {}_NF$ if and only if $w \in L(N)$. From these equivalences, w $\notin L(M)$ if and only if $w \in L(N)$, so $L(N) = {}_M\Delta^* - L(M)$.

■

The next theorem follows from Algorithm 2.40 and Lemma 2.41.

Theorem 2.42. The family of regular languages is closed under complement.

As the complement and the union preserve the family of regular languages, the next theorem follows directly from DeMorgan's law (see Section 1.1).

Theorem 2.43. The family of regular languages is closed under intersection.

Proof. Let K and L be two regular languages. By DeMorgan's law, *complement*(*complement*(K) $\cup$ *complement*(L)) $= K \cap L$, so Theorem 2.43 follows from Theorems 2.39 and 2.42.

■

Let a family of languages be closed under union, intersection, and complement; then, this family is a *Boolean algebra of languages*.

Corollary 2.44. The family of regular languages is a Boolean algebra of languages.

Proof. Theorems 2.39, 2.42 and 2.43 imply Corollary 2.44.

■

Regular substitution. As a special case of substitution, we next define the regular substitution and prove that the family of regular languages is closed under this substitution. This important closure property represents a powerful result, which implies several other closure properties as demonstrated later in this section. To establish this result, we first prove Lemma 2.45, which represents a reformulation of Corollary 2.21 in terms of finite automata without ε-rules.

Lemma 2.45. Let K be a regular language such that $\varepsilon \notin K$. Then, $K = L(N)$, where $N = ({}_N\Sigma, {}_NR)$ is a finite automaton without ε-rules such that ${}_NF = \{{}_Nf\}$, where ${}_Nf$ is a stop state.

Proof. Let K be a regular language such that $\varepsilon \notin K$. By Theorem 2.28, K is accepted by a finite automaton, V, without ε-rules. As $\varepsilon \notin K$, V's start state is not final. By analogy with the proof of Corollary 2.21, convert V to a finite automaton, U, in which there is only one final state, f, such that f is a stop state. Notice that f produced in this way is not the start state in U. As this conversion has produced some ε-rules in U, use Algorithm 2.31 *Removal of ε-Rules* to transform this automaton to an equivalent finite automaton, N, without ε-rules. Observe that in N, f is again the only final state such that f is also a stop state, so Lemma 2.45 holds. A rigorous version of this proof is left as an exercise.

■

We are now ready to prove that the family of regular languages is closed under the regular substitution, defined next.

Definition 2.46. Let V and W be two alphabets, and let h be a substitution from V^* and W^* such that for all $a \in V$, $h(a)$ is a regular language. Then, h is a regular substitution.

■

Goal. For any regular substitution h and any regular language K such that $\varepsilon \notin K$, construct a finite automaton M such that $L(M) = h(K)$.

Gist. Let K be a regular language such that $\varepsilon \notin K$. Without any loss of generality, we assume that K is accepted by a finite automaton N without ε-rules such that ${}_NF = \{{}_Nf\}$, where ${}_Nf$ is a stop state (see Lemma 2.45). Let h be a regular substitution such that for every $a \in {}_N\Delta$, $h(a) = L(a\text{-}M)$, where $a\text{-}M$ is a finite automaton. Observe that for every non-empty string $a_1a_2\ldots a_n \in L(N)$, where each $a_i \in alph(K)$, $w_1w_2\ldots w_n \in h(a_1a_2\ldots a_n)$ if and only if $w_i \in h(a_i)$ for all $1 \leq i \leq n$. Next, we construct a finite automaton M satisfying $a_1a_2\ldots a_n \in L(N)$ if and only if $w_1w_2\ldots w_n \in L(M)$ for every $w_i \in L(a_i\text{-}M)$, so $L(M) = h(L(N)) = h(K)$. M has a single final state, ${}_Nf$, which thus coincides with N's final state. M works as follows. First, M simulates $a_1\text{-}M$. When M enters a final state of $a_1\text{-}M$ and, thereby, accepts w_1, M starts the simulation $a_2\text{-}M$. After simulating all n automata $a_1\text{-}M$ through $a_n\text{-}M$ in this way, M has read $w_1w_2\ldots w_n$ with $w_i \in h(a_i)$. As $a_1a_2\ldots a_n \in L(N)$, $a_1a_2\ldots a_n$ takes M to ${}_Nf$, and at this point, M accepts $w_1w_2\ldots w_n$.

Note that the following algorithm requires that the state sets of N, $a_1\text{-}M$, …, $a_n\text{-}M$ are pairwise disjoint. This requirement is obviously without any loss of generality; indeed, if two automata contain some states in common, we rename states in either of them to obtain two disjoint sets.

Algorithm 2.47 ***Finite Automaton for Regular Substitution.***

Input
- a finite automaton $N = ({}_N\Sigma, {}_NR)$ without ε-rules such that ${}_NF = \{{}_Nf\}$ and ${}_Nf$ is a stop state;
- $\Phi = \{a\text{-}M \mid a\text{-}M = ({}_{a\text{-}M}\Sigma, {}_{a\text{-}M}R)$ is a finite automaton, $a \in {}_N\Delta\}$ such that all the automata in $\Phi \cup \{N\}$ have pairwise disjoint sets of states.

Output • a finite automaton $M = ({}_M\Sigma, {}_MR)$ such that $w_1w_2\ldots w_n \in L(M)$ if and only if $w_1w_2\ldots w_n \in L(a_1\text{-}M)L(a_2\text{-}M)\ldots L(a_n\text{-}M)$, where $a_1a_2\ldots a_n \in L(N)$, $a_i \in {}_N\Delta$, $a_i\text{-}M \in \Phi$, $w_i \in L(a_i\text{-}M)$, $1 \le i \le n$, where n is a non-negative integer ($n = 0$ means $w_1w_2\ldots w_n = a_1a_2\ldots a_n = \varepsilon$).

Method

```
begin
    ${}_MQ := {}_NQ \cup \{\langle qo\rangle \mid q \in {}_NQ, o \in {}_OQ$ with $O \in \Phi\}$;
    ${}_Ms := {}_Ns$;
    ${}_MF := \{{}_Nf\}$;
    ${}_M\Delta := \{a \mid a \in {}_O\Delta$ with $O \in \Phi\}$;
    ${}_MR := \{q \to \langle q{}_Os\rangle \mid q \in {}_NQ, O \in \Phi\}$
            $\cup \{\langle qp\rangle a \to \langle qo\rangle \mid q \in {}_NQ, pa \to o \in {}_OR$ with $a \in {}_O\Delta \cup \{\varepsilon\}$ and $O \in \Phi\}$
            $\cup \{\langle qf\rangle \to p \mid q, p \in {}_NQ, qb \to p \in {}_NR$ with $b \in {}_N\Delta$, $f \in {}_{b\text{-}M}F$ with $b\text{-}M \in \Phi\}$
end.
```

Lemma 2.48. Algorithm 2.47 is correct.

Proof. Let N and Φ have the same meaning as in Algorithm 2.47. We need to demonstrate that Algorithm 2.47 correctly constructs a finite automaton $M = ({}_M\Sigma, {}_MR)$ such that $w_1w_2\ldots w_n \in L(M)$ if and only if $w_1w_2\ldots w_n \in L(a_1\text{-}M)L(a_2\text{-}M)\ldots L(a_n\text{-}M)$, where $a_1a_2\ldots a_n \in L(N)$, $a_i \in {}_N\Delta$, $a_i\text{-}M \in \Phi$, $1 \le i \le n$, where n is a positive integer. We begin with this important claim:

Claim. Let $q \in {}_NQ$, $a_1a_2\ldots a_n \in {}_N\Delta^*$, $a_i \in {}_N\Delta$, $1 \le i \le n$, where n is a positive integer. Then, ${}_Nsa_1a_2\ldots a_n \Rightarrow^* q$ in N if and only if ${}_Msw_1w_2\ldots w_n \Rightarrow^* q$ in M with $w_i \in L(a_i\text{-}M)$, $a_i\text{-}M \in \Phi$, $1 \le i \le n$.

Only if. By induction on $n \ge 0$, we next prove that for every $q \in {}_NQ$, $a_1a_2\ldots a_n \in {}_N\Delta^*$, $a_i \in {}_N\Delta$, $1 \le i \le n$, ${}_Nsa_1a_2\ldots a_n \Rightarrow^* q$ in N implies ${}_Msw_1w_2\ldots w_n \Rightarrow^* q$ in M with $w_i \in L(a_i\text{-}M)$, where $a_i\text{-}M \in \Phi$, $1 \le i \le n$.

Basis. Let $n = 0$, so $w_1w_2\ldots w_n = a_1a_2\ldots a_n = \varepsilon$. As N is without ε-rules, N accepts ε as ${}_Ns \Rightarrow^0 {}_Nf$ in N, so ${}_Ns = {}_Nf$. Since ${}_Ms = {}_Ns$ and ${}_MF = \{{}_Nf\}$, ${}_Ms \Rightarrow^0 {}_Mf$ in M, so ${}_Ms \Rightarrow^* {}_Mf$ in M.

Induction Hypothesis. Assume that the only-if part of the claim holds for all strings of length k, $1 \le k \le n$, where n is a non-negative integer.

Induction Step. Let $q \in {}_NQ$, $a_1a_2\ldots a_na_{n+1} \in {}_N\Delta^*$, $a_i \in {}_N\Delta$, $1 \le i \le n + 1$, and ${}_Nsa_1a_2\ldots a_{n+1} \Rightarrow^* q$ in N. As N is without ε-rules, express ${}_Nsa_1a_2\ldots a_n \Rightarrow^* q$ as

$$\begin{aligned} {}_Nsa_1a_2\ldots a_na_{n+1} &\Rightarrow^* pa_{n+1} \\ &\Rightarrow q \qquad [pa_{n+1} \to q] \end{aligned}$$

in N, where $pa_{n+1} \to q \in {}_NR$ with $p \in {}_NQ$. Thus, ${}_Nsa_1a_2\ldots a_n \Rightarrow^* p$ in N. By the induction hypothesis, ${}_Msw_1w_2\ldots w_n \Rightarrow^* p$ in M with $w_i \in L(a_i\text{-}M)$, $a_i\text{-}M \in \Phi$, $1 \le i \le n$. For every $w_{n+1} \in L(a_{n+1}\text{-}M)$, $\sigma w_{n+1} \Rightarrow^* \phi$ in $a_{n+1}\text{-}M$, where σ is the start state of $a_{n+1}\text{-}M$ and ϕ is a final state in $a_{n+1}\text{-}M$. At this point, Algorithm 2.46 introduces $p \to \langle p\sigma\rangle$ and $\langle p\phi\rangle \to q$ to ${}_MR$. Furthermore, it puts $a_{n+1}\text{-}M$'s rules to ${}_MR$ as well, so $\sigma w_{n+1} \Rightarrow^* \phi$ in M. Thus,

$$\begin{aligned} pw_{n+1} &\Rightarrow \langle p\sigma\rangle w_{n+1} \quad [p \to \langle p\sigma\rangle] \\ &\Rightarrow^* \langle p\phi\rangle \end{aligned}$$

$$\Rightarrow q \qquad [\langle p\phi\rangle \to q]$$

in M. Consequently, ${}_Msw_1w_2\ldots w_nw_{n+1} \Rightarrow^* pw_{n+1} \Rightarrow \sigma w_{n+1} \Rightarrow^* \phi \Rightarrow q$ in M. As ${}_Msw_1w_2\ldots w_nw_{n+1} \Rightarrow^* q$, the induction step is completed, and the only-if part of the proof holds.

If. A proof of this part of the claim is left as an exercise.

Consider the above claim for $q = {}_Nf$. That is, for every $a_1a_2\ldots a_n \in {}_N\Delta^*$, $a_i \in {}_N\Delta$, $1 \leq i \leq n$, where n is a positive integer, ${}_Nsa_1a_2\ldots a_n \Rightarrow^* {}_Nf$ in N if and only if ${}_Msw_1w_2\ldots w_n \Rightarrow^* {}_Nf$ in M with $w_i \in L(a_i\text{-}M)$, $a_i\text{-}M \in \Phi$, $1 \leq i \leq n$. Observe that ${}_MF = \{{}_Nf\}$. Therefore, $a_1a_2\ldots a_n \in L(N)$ if and only if $w_1w_2\ldots w_n \in L(M)$, so the lemma holds.

■

Theorem 2.49. The family of regular languages is closed under regular substitution.

Proof. Let K be a regular language. As K is regular, by Theorem 2.28, there exists a finite automaton N such that $K = L(N)$ with ${}_N\Delta = \boldsymbol{alph}(K)$.

Let $\varepsilon \notin K$. Without any loss of generality, assume that N is without ε-rules, ${}_NF = \{{}_Nf\}$ and ${}_Nf$ is a stop state (see Lemma 2.45). Let h be a regular substitution from ${}_N\Delta^*$ to V^*. That is, for every $a \in alph(K)$, $h(a)$ is regular, so there exists a finite automaton $a\text{-}M$ such that $h(a) = L(a\text{-}M)$. By Algorithm 2.46 and Lemma 2.47, there exists a finite automaton M satisfying $w_1w_2\ldots w_n \in L(M)$ if and only if $w_1w_2\ldots w_n \in L(a_1\text{-}M)L(a_1\text{-}M)\ldots L(a_n\text{-}M)$, where $a_1a_2\ldots a_n \in L(N)$, $a_i \in {}_N\Delta$, $a_i\text{-}M \in \{a\text{-}M |\ a\text{-}M = ({}_{a\text{-}M}\Sigma, {}_{a\text{-}M}R)$ is a finite automaton, $a \in alph(K)\}$, $w_i \in L(a_i\text{-}M)$, $1 \leq i \leq n$, where n is a non-negative integer ($n = 0$ means $w_1w_2\ldots w_n = a_1a_2\ldots a_n = \varepsilon$). Thus, $L(M) = h(L(N)) = h(K)$. Since M is a finite automaton, $h(K)$ is regular.

Let $\varepsilon \in K$. By analogy with the previous part of this proof, demonstrate that $h(K - \{\varepsilon\})$ is regular. Recall that $h(\varepsilon) = \{\varepsilon\}$ (see Section 1.1). Express $h(K) = h(K - \{\varepsilon\}) \cup \{\varepsilon\}$. By Lemma 2.17, Theorem 2.28, and Theorem 2.39, $h(K)$ is regular.

■

Corollary 2.50. The family of regular languages is closed under finite substitution and homomorphism.

Proof. Every finite language and homomorphism is regular. Therefore, this theorem follows from Theorem 2.49.

■

Applications of Closure Properties

As already pointed out, together with the pumping lemma for regular languages, the closure properties are frequently used to prove that a language, L, is not regular. Typically, a proof of this kind is made by contradiction in this way:

A. Assume that L is regular.
B. By using operations preserving the family of regular languages, construct a new language K from L so that the following proof in C is as simple as possible.
C. By using the pumping lemma, prove that K is not regular, which contradicts that K is regular by the closure properties concerning the operations used in B.
D. The contradiction obtained in C implies that K is not regular, so L is not regular either.

Example 2.15 ***Use of Closure Properties to Disprove that a Language is Regular.*** Let L be the language consisting of all alphanumeric strings containing a different number of letters and digits; for instance, *ab*0*c*1 is in this language but *ab*01 is not. Next, we prove that this language is non-regular.

A. Assume that L is regular.
B. Consider $K = complement(L) \cap \{0\}^*\{a\}^* = \{0^n a^n | \ n \geq 0\}$ (notice that $\{0\}^*\{a\}^*$ is regular).
C. By analogy with Example 2.12, prove that $K = \{0^n a^n | \ n \geq 0\}$ is not regular by using the pumping lemma; however, Theorems 2.42 and 2.43 imply that K is regular—a contradiction;
D. The contradiction obtained in C implies that K is not regular, so neither is L.

■

While the pumping lemma is of no use in proving that a language is regular (see Example 2.14), closure properties are helpful in proofs like these. Indeed, suppose a direct proof that a language L is regular leads to a design of an enormously complicated finite automaton, followed by a complex proof that this automaton accepts L. At this point, we can simplify this direct proof so we define L by several simple regular languages combined by operations under which the family of regular languages is closed.

Example 2.16 ***Use of Closure Properties to Prove that a Language is Regular.*** Consider $L = \{a^i b^j c^k d^l | \ i, k \geq 0 \text{ and } j, l \geq 1\}$. To verify that L is regular, express this language as $L = h(\{a\})\{b\}h(\{c\})\{d\}$, where h is a regular substitution defined as $h(a) = \{a\}^*\{b\}^*$ and $h(c) = \{c\}^*\{d\}^*$. As $\{a\}$, $\{b\}$, $\{c\}$, and $\{d\}$ are obviously regular languages, so is $L = h(\{a\})\{b\}h(\{c\})\{d\}$ by Theorems 2.39 and 2.49.

■

2.3.4 Decidable problems

In this book, a *problem* is specified by a *question* together with the set of problem *instances*. The question formulates the problem. The set of instances is the collection of all allowable values for the unknowns of the problem. A problem is *decidable* if there exists an algorithm that decides it for any instance. The present section discusses problems relevant to lexical analysis; specifically,

- membership problem
- emptiness problem
- infiniteness problem
- finiteness problem
- equivalence problem

As finite automata are crucial to the lexical analysis, we describe algorithms that decide these problems in terms of these automata. Each algorithm contains a boolean variable *answer*. With one of the allowable instances as its input, the algorithm sets *answer* to **true** to specify the affirmative answer to the question in this instance; otherwise, it sets *answer* to **false**. Since all these algorithms are simple, we present them less formally than the algorithms given in the previous parts of this chapter.

Convention 2.51. Throughout the rest of this section, we automatically assume that each automaton $M = ({}_M\Sigma, {}_MR)$ represents a completely specified deterministic finite automata with only reachable states (see Theorem 2.37).

■

Membership problem. Deciding whether a string is a member of the language that a finite automaton accepts is obviously the crucial task a scanner needs to handle.

Membership problem for finite automata.
Instance: $M = ({}_M\Sigma, {}_MR)$ and a string, $w \in {}_M\Delta^*$.
Question: Is w a member of $L(M)$?

Fortunately, a decision algorithm for this important problem is trivial.

Algorithm 2.52 ***Decision of Membership Problem for Finite Automata.*** Run M on w. If M accepts w, set *answer* to **true**; if M rejects w, set *answer* to **false**.

Emptiness problem. Consider a token t and the regular language, *t-lexemes*, which represents the regular language of all lexemes denoted by t (see Section 2.2). If no lexeme is in *t-lexemes*, t is uselessly introduced as a token. In reality, this situation suggests that the lexical analyzer is improperly designed. As a result, an algorithm that decides whether a regular language equals $\emptyset$ is highly relevant to the lexical analysis.

Emptiness problem for finite automata.
Instance: $M = ({}_M\Sigma, {}_MR)$.
Question: Is $L(M)$ empty?

Algorithm 2.53 ***Decision of Emptiness Problem for Finite Automata.*** If M has no final state, $L(M) = \emptyset$, so set *answer* to **true**; otherwise, $L(M) \neq \emptyset$, so set *answer* to **false**.

Lemma 2.54. Algorithm 2.53 correctly decides the emptiness problem for M.

Proof. According to Convention 2.51, M is a completely specified deterministic finite automaton in which every state is reachable. Observe that this property implies that $L(M) = \emptyset$ if and only if ${}_MF = \emptyset$.

■

An algorithmic decision of the emptiness problem is less trivial if we relax our assumption that M is a completely specified deterministic finite automaton with all state reachable. A discussion of this case is left as an exercise.

Infiniteness problem. Consider a token t and the corresponding regular language *t-lexemes* contain all lexemes denoted by t (see Section 2.2). If *t-lexemes* is infinite, the lexical analyzer handles these lexemes with special care as already explained in Section 2.2. Indeed, it specifies these lexemes by a regular expression *t-expression* and converts them to attributed tokens. Therefore, an algorithm that decides whether a regular language is infinite plays a useful role in the lexical analysis.

Infiniteness problem for finite automata.
Instance: $M = ({}_M\Sigma, {}_MR)$.
Question: Is $L(M)$ infinite?

Next, we informally sketch two methods of deciding this problem.

State-Diagram-Based Method. Recall that M represents a completely specified deterministic finite automaton. Assume that $L(M) \neq \emptyset$; otherwise, $L(M)$ is obviously finite because it is empty.

Observe that $L(M)$ is infinite if and only if its state diagram contains a cycle. We further discuss this method as an exercise.

Pumping-Lemma-Based Method. Consider the pumping lemma for regular languages (see Lemma 2.38). Let k be the pumping-lemma constant for $L(M)$. Set

$$M\text{-}infiniteness\text{-}test = \{x \mid x \in {}_N\Delta^*, x \in L(M), \text{ and } k \leq |x| < 2k\}$$

Algorithm 2.55 ***Decision of Infiniteness Problem for Finite Automata.*** If M-*infiniteness-test* $\neq \emptyset$, set *answer* to **true**; otherwise, set *answer* to **false**.

Lemma 2.56. Algorithm 2.55 correctly decides the infiniteness problem for M.

Proof. We prove this lemma by demonstrating that M-*infiniteness-test* $\neq \emptyset$ if and only if $L(M)$ is infinite.

Only If. Let M-*infiniteness-test* $\neq \emptyset$. Consider any $z \in M$-*infiniteness-test*. By Lemma 2.38, $z = uvw$, where $0 < |v| \leq |uv| \leq k$ (k is the pumping lemma constant), and $uv^mw \in L$, for all $m \geq 0$. Thus, $L(M)$ is infinite.

If. We prove that $L(M)$ is infinite implies M-*infiniteness-test* $\neq \emptyset$ by contradiction. Assume that L is infinite and M-*infiniteness-test* $= \emptyset$. Let z be the shortest string such that $z \in L(M)$ and $|z| \geq 2k$. As $|z| \geq 2k \geq k$, Lemma 2.38 implies that $z = uvw$, where $0 < |v| \leq |uv| \leq k$, and $uv^mw \in L$, for all $m \geq 0$. Take $uv^0w = uw \in L$. Observe that $2k > |uw|$ because $2k \leq |uw| < |z|$ would contradict that z is the shortest string satisfying $z \in L(M)$ and $|z| \geq 2k$. As $0 < |v| \leq k$, $k \leq |uw| < 2k \leq |z|$, so $uw \in M$-*infiniteness-test* $\neq \emptyset$, which contradicts M-*infiniteness-test* $= \emptyset$. Thus, if $L(M)$ is infinite, then M-*infiniteness-test* $\neq \emptyset$.

■

As we can decide the infiniteness problem for finite automata, we can also algorithmically decide

Finiteness problem for finite automata.
Instance: $M = ({}_M\Sigma, {}_MR)$.
Question: Is $L(M)$ finite?

Algorithm 2.57 ***Decision of Finiteness Problem for Finite Automata.*** If M-*infiniteness-test* $\neq \emptyset$, set *answer* to **false**; otherwise, set *answer* to **true**.

Equivalence problem. Consider two tokens, t and u, and their corresponding regular language, *t-lexemes* and *u-lexemes*, respectively. Furthermore, assume that *t-lexemes* and *u-lexemes* are specified by finite automata, M and N, respectively; formally, *t-lexemes* $= L(M)$ and *u-lexemes* $= L(N)$. If M is equivalent to N, then *t-lexemes* = *u-lexemes*. At this point, the lexical analyzer probably contains some incorrect, duplicates, or superfluous parts, so its designer should carefully revise its construction. As a result, an algorithm that decides the equivalence of two finite automata is surely relevant to the lexical analysis.

Equivalence problem for finite automata.
Instance: Finite automata $M = ({}_M\Sigma, {}_MR)$ and $N = ({}_M\Sigma, {}_MR)$.
Question: Is M equivalent to N?

Let M and N be two completely specified deterministic finite automata. As an exercise, prove that $L(M) = L(N)$ if and only if $H = \emptyset$, where $H = (L(M) \cap complement(L(N))) \cup (L(N) \cap$

complement($L(M)$)). Design an algorithm that converts any finite automata M and N to a finite automaton O such that $L(O) = H$. By using the algorithms given in Section 2.3.3, convert O to an equivalent completely specified deterministic finite automata, P. By Algorithm 2.53, decide $L(P) = \emptyset$. If $L(P) = \emptyset$, M is equivalent to N; otherwise, M is not equivalent to N. A precise description of an algorithm that decides this problem is left as an exercise.

Exercises

2.1. Consider the language consisting of all identifiers defined as alphanumerical strings ending with a letter. Construct a finite automaton that accepts this language. Construct a regular expression that denotes this language.

2.2. Consider a language of all Pascal identifiers of even length. Construct a finite automaton that accepts this language and a regular expression that denotes it.

2.3. Let $L = \{x|\ x \in \{a, b\}^*, ab \notin substring(x)$, and $|x|$ is divisible by 3$\}$ be the language over $\Sigma = \{a, b\}$. Construct a finite automaton that accepts this language and a regular expression that denotes it.

2.4. Let $L = \{x|\ x \in \{a, b\}^*, aa \notin suffixes(x)\}$. Construct a regular expression that denotes this language. Explain why this construction is more difficult than the construct of a finite automaton that accepts this language.

2.5. Consider the following Pascal program. Does the program contain any lexical errors? If so, specify them. If not, what is the token sequence produced by a Pascal scanner working with this text? Which of these tokens are attributed?

```
program trivialprogram(input, output);
var i : integer;

   function result(k: integer): integer;
   begin
      result := k * k + k
   end;

begin
   read(i);
   writeln('For  ', i, ', the result is ', result(i))
end.
```

2.6. Detect all lexical errors, if any, appearing in the following Pascal fragment. How should a Pascal lexical analyzer handle each of these errors?

```
while a <> 0.2+E2 do writeln("   ');

if a > 24. than
   writewrite('hi')
else
   write ('bye);

fori := 1 through 10 todo writeln(Iteration: ', i:1);
```

2.7. Write a regular expression that defines programming language comments delimited by { and } in a Pascal-like way. Suppose that such a comment may contain any number of either {s or }s; however, it does not contain both { and }. Write a program that implements a finite automaton equivalent to the expression.

2.8. Design a regular expression that defines Pascal-like real numbers without any superfluous leading or trailing 0s; for instance, 10.01 and 1.204 are legal, but 01.10 and 01.240 are not.

2.9. FORTRAN ignores blanks. Explain why this property implies that a FORTRAN lexical analyzer may need an extensive lookahead to determine how to scan a source program. Consider `DO 9 J = 1, 9` and `DO 10 J = 1. 9`. Describe the lexical analysis of both fragments in detail. Are both correct in FORTRAN? If not, justify your answer in detail. If so, how many lexemes are in the former fragment? How many lexemes are in the other?

2.10. Modify FUN lexemes by reversing them; for instance, identifiers are alphanumerical strings that end with a letter. Which lexemes remain unchanged after this modification? Write a program to implement a FUN scanner that recognizes these modified lexemes.

2.11. Consider the next Ada source text. What are the Ada lexemes contained in this text? Translate these lexemes to the corresponding tokens.

```
with Ada.Text_IO;

procedure Lexemes is
begin
   Ada.Text_IO.Put_Line("Find lexemes!");
end Lexemes;
```

2.12. Design a data structure for representing tokens, including attributed tokens. Based on this structure, implement the tokens given in Case Study 3/35 *Lexemes and Tokens*.

2.13. In detail, design a cooperation between a lexical analyzer and a symbol-table handler. Describe how to store and find identifiers in the table. Furthermore, explain how to distinguish keywords from identifiers.

2.14. Write a program to simplify the source-program text, including the conversion of the uppercase letters to the lowercase letters and the removal of all comments and blanks.

2.15. Write a program to implement the representations of finite automata described in Example 2.2 *Tabular and Graphical Representation*.

2.16. Write a program that transforms the tabular representation of any completely specified deterministic finite automaton to a C program that represents the implementation described in Algorithm 2.8 *Implementation of a Finite Automaton—Tabular Method*.

2.17. Write a program that transforms the tabular representation of any completely specified deterministic finite automaton to a Pascal program that represents the implementation described in Algorithm 2.9 *Implementation of a Finite Automaton—Case-Statement Method*.

2.18. Write a program to implement the scanner described in Case Study 4/35 *Scanner*.

2.19. Write a program to implement

(a) Algorithm 2.19 *Finite Automaton for Union*;
(b) Algorithm 2.22 *Finite Automaton for Concatenation*;
(c) Algorithm 2.24 *Finite Automaton for Iteration*.

2.20. Describe the transformation of regular expressions to finite automata formally as an algorithm. Write a program to implement this algorithm.

2.21. Give a fully rigorous proof that for any regular expression, there exists an equivalent finite automaton (see Lemma 2.26).

2.22. Give a fully rigorous proof that for every finite automaton, there exists an equivalent regular expression (see Lemma 2.27).

2.23. Consider each of the following languages. By the pumping lemma for regular languages (Lemma 2.38), demonstrate that the language is not regular.

(a) $\{a^iba^i \mid i \geq 1\}$
(b) $\{a^ib^j \mid 1 \leq i \leq j\}$
(c) $\{a^{2^i} \mid i \geq 0\}$
(d) $\{a^ib^jc^k \mid i, j, k \geq 0 \text{ and } i = k + j\}$
(e) $\{a^ib^ic^j \mid i, j \geq 0 \text{ and } i \leq j \leq 2i\}$
(f) $\{a^ib^jc^k \mid i, j, k \geq 0, i \neq j, k \neq i, \text{ and } j \neq k\}$
(g) $\{a^ib^jc^jd^i \mid i, j \geq 0 \text{ and } j \leq i\}$
(h) $\{a^ib^{2i} \mid i \geq 0\}$

2.24. Introduce a set of ten regular languages such that each of them contains infinitely many subsets that represent non-regular languages. For each language, L, in this set, define a non-regular language, K, such that $K \subseteq L$ and prove that K is not regular by the regular pumping lemma. For instance, if $L = \{a,b\}^*$, $K = \{a^nb^n \mid n \geq 0\}$ represents one of infinitely many non-regular languages such that $K \subseteq L$.

2.25. Prove the pumping lemma for regular languages (Lemma 2.38) in terms of regular expressions.

2.26. Prove the following modified version of the pumping lemma for regular languages.

Lemma 2.58. Let L be a regular language. Then, there is a natural number, k, such that if $xzy \in L$ and $|z| = k$, then z can be written as $z = uvw$, where $|v| \geq 1$ and $xuv^mwy \in L$, for all $m \geq 0$.

2.27. Use Lemma 2.58, established in Exercise 2.26, to prove that $\{a^ib^jc^j \mid i, j \geq 1\}$ is not regular.

2.28. Prove that a language, L, over an alphabet Σ, is regular if and only if there exists a natural number, k, satisfying this statement: if $z \in \Sigma^*$ and $|z| \geq k$, then (1) $z = uvw$ with $v \neq \varepsilon$ and (2) $zx \in L$ if and only if $uv^mwx \in L$, for all $m \geq 0$ and $x \in \Sigma^*$. Explain how to use this lemma to prove that a language is regular. Then, explain how to use this lemma to prove that a language is not regular.

2.29. Let L be a regular language over an alphabet Σ. Are *complement*(L), *substrings*(L), *prefixes*(L), *suffixes*(L), *reversal*(L), L^*, and L^+ regular as well (see Section 1.1)?

2.30. Let L be a regular language over an alphabet Σ. Define the next language operations. For each of these operations, prove or disprove that the family of regular languages is closed under the operation.

(a) $min(L) = \{w \mid w \in L \text{ and } \{prefix(w) - \{w\}\} \cap L = \emptyset\}$
(b) $max(L) = \{w \mid w \in L \text{ and } \{w\}\Sigma^+ \cap L = \emptyset\}$
(c) $sqrt(L) = \{x \mid xy \in L \text{ for some } y \in \Sigma^*, \text{ and } |y| = |x|^2\}$
(d) $log(L) = \{x \mid xy \in L \text{ for some } y \in \Sigma^*, \text{ and } |y| = 2^{|x|}\}$
(e) $cycle(L) = \{vw \mid wv \in L \text{ for some } v, w \in \Sigma^*\}$
(f) $half(L) = \{w \mid wv \in L \text{ for some } v \in \Sigma^*, \text{ and } |w| = |v|\}$

(g) $inv(L) = \{xwy \mid xzy \in L$ for some $x, y, w, z \in \Sigma^*, z = reversal(w)\}$

2.31. For a language L over an alphabet Σ and a symbol, $a \in \Sigma$, $eraser_a(L)$ denotes the language obtained by removing all occurrences of a from the words of L. Formalize $eraser_a(L)$. Is the family of regular languages closed under this operation?

2.32*Solved*. Theorem 2.49 has demonstrated that the family of regular languages is closed under regular substitution in terms of finite automata. Prove this important closure property in terms of regular expressions.

2.33. Consider these languages. Use the closure properties of the regular languages and the regular pumping lemma to demonstrate that these languages are not regular.

(a) $\{w \mid w \in \{a, b\}^*$ and $occur(w, a) = 2occur(w, b)\}$
(b) $\{0^i10^i \mid i \geq 1\}$
(c) $\{wcv \mid w \in \{a, b\}^*$ and $v = reversal(w)\}$

2.34. In Section 2.3.4, we have described all algorithms rather informally. Give these algorithms formally and verify them in a rigorous way.

2.35. Reformulate all decision problems discussed in Section 2.3.4 in terms of regular expressions. Design algorithms that decide the problems reformulated in this way.

2.36. Consider *Equivalence problem for finite automata and regular expressions*.
• *Instance*: A finite automaton, M, and a regular expression, E.
• *Question*: Is M equivalent to E?
Design an algorithm that decides this problem.

2.37. Consider *Computational multiplicity problem for finite automata*.
• *Instance*: A finite automaton, $M = (\Sigma, R)$, and $w \in \Sigma^*$.
• *Question*: Can M compute $sw \Rightarrow^* f\,[\rho]$ and $sw \Rightarrow^* f'\,[\rho']$ so $f, f' \in F$ and $\rho \neq \rho'$?
Design an algorithm that decides this problem.

2.38*Solved*. Let $M = (\Sigma, R)$ be a deterministic finite automaton (M may not be completely specified, however) and $k = card(Q)$. Prove that $L(M) = \emptyset$ if and only if $\{x \mid x \in \Sigma^*, x \in L(M)$, and $|x| < k\} = \emptyset$. Notice that from this equivalence, it follows that the emptiness problem for finite automata is decidable.

2.39. Consider

Definition 2.59 ***Minimal Finite Automaton.*** Let $M = (\Sigma, R)$ be a deterministic finite automaton. M is a *minimal finite automaton* if every deterministic finite automaton that accepts $L(M)$ has no fewer states than M has.

To explain this definition, a deterministic finite automaton, M, may contain some redundant states that can be merged together without any change of the accepted language. For instance, introduce a deterministic finite automaton, M, with these six rules ▸*sa* → ▸*s*, ▸*sb* → ▪*q*, ▪*qa* → ▪*f*, ▪*qb* → ▪*f*, ▪*fa* → ▪*f*, and ▪*fb* → ▪*f*. Clearly, $L(M) = \{a\}^*\{b\}\{a, b\}^*$. Consider another deterministic finite automaton, N, obtained from M by merging states q and ▪*f* to a single state, ▪*g*, and replacing the six rules with these four rules ▸*sa* → ▸*s*, ▸*sb* → ▪*g*, ▪*ga* → ▪*g*, and ▪*gb* → ▪*g*. Clearly, both automata are equivalent. However, N has two states while M has three. In a minimal finite automaton, however, there exist no redundant states that can be merged together without changing the accepted language. For example, N is a minimal finite automaton while M is not. Design an algorithm that turns any deterministic finite automaton to an equivalent minimal finite automaton.

2.40$_{\textit{Solved}}$**.** Reconsider Example 2.13 *Primes are Non-Regular*, which demonstrates that $L = \{a^n |\ n$ is a prime$\}$ is not regular. Give an alternative proof of this result by using the pumping lemma.

2.41. Design tabular and graphical representations for finite-state transducers.

2.42. Write a program to implement

(a) Algorithm 2.29 *States Reachable without Reading*;
(b) Algorithm 2.31 *Removal of ε-Rules*;
(c) Algorithm 2.34 *Determinism*;
(d) Algorithm 2.36 *Determinism with Reachable States*.

2.43. Discuss Example 2.8 *Removal of ε-Rules* in detail.

2.44. Give a rigorous proof of Lemma 2.35.

2.45. Prove that Algorithm 2.36 *Determinism with Reachable States* correctly converts a finite automaton $M = ({}_M\Sigma, {}_MR)$ without ε-rules to a deterministic finite automaton $N = ({}_N\Sigma, {}_NR)$ such that $L(N) = L(M)$ and ${}_NQ$ contains only reachable states.

2.46. Consider

Definition 2.60 ***Lazy Finite Automaton and its Language*****.** A *lazy finite automaton* is a rewriting system, $M = (\Sigma, R)$, where

- Σ is divided into two pairwise disjoint subalphabets Q and Δ;
- R is a finite *set of rules* of the form $qx \rightarrow p$, where $q, p \in Q$ and $x \in \Delta^*$.

Q and Δ are referred to as the *set of states* and the *alphabet of input symbols*, respectively. Q contains a state called the *start state*, denoted by s, and a *set of final states*, denoted by F. Like in any rewriting, we define $u \Rightarrow v$, $u \Rightarrow^n v$ with $n \geq 0$, and $u \Rightarrow^* v$, where $u, v \in \Sigma^*$. If $sw \Rightarrow^* f$ in M, where $w \in \Delta^*$, *M accepts w*. The set of all strings that M accepts is the *language accepted by M*, denoted by $L(M)$.

■

Definition 2.60 generalizes finite automata (see Definition 2.2). Explain this generalization. Design an algorithm that turns any lazy finite automaton to an equivalent deterministic finite automaton.

2.47. Consider the language consisting of FUN keywords and identifiers. Design a lazy finite automaton that accepts this language (see Exercise 2.46). Then, convert this automaton to an equivalent deterministic finite automaton.

2.48 Consider

Definition 2.61 ***Loop-Free Finite Automaton*****.** Let $M = (\Sigma, R)$ be a deterministic finite automaton (see Definition 2.6). A state, $q \in {}_MQ$, is *looping* if there exists $x \in \Delta^+$ such that $qw \Rightarrow^* q$. M is *loop-free* if no state in ${}_MQ$ is looping.

■

Prove the next theorem.

Theorem 2.62. A language L is finite if and only if $L = L(M)$, where M is a loop-free finite automaton.

2.49. Consider

Definition 2.63 ***Two-Way Finite Automaton and its Language.*** A *two-way finite automaton* is a rewriting system, $M = (\Sigma, R)$, where

- Σ is divided into two pairwise disjoint subalphabets Q and Δ;
- R is a finite *set of rules* of the form $x \rightarrow y$, where either $x = qa$ and $y = ap$ or $x = aq$ and $y = pa$, for some $q, p \in Q$ and $a \in \Delta$.

Q and Δ are referred to as the *set of states* and the *alphabet of input symbols*, respectively. Q contains a state called the *start state*, denoted by s, and a *set of final states*, denoted by F. Like in any rewriting system, we define $u \Rightarrow v$, $u \Rightarrow^n v$ with $n \geq 0$, and $u \Rightarrow^* v$, where $u, v \in \Sigma^*$ (see Definition 1.5). If $qw \Rightarrow^* wf$ in M with $f \in F$, M *accepts* w. The set of all strings that M accepts is the *language accepted by* M, denoted by $L(M)$.

■

Informally, a two-way finite automaton makes moves right as well as left on the input string. Explain this extension of finite automata in greater detail. Prove that the two-way finite automata are as powerful as finite automata.

Solutions to Selected Exercises

2.32. Consider a regular substitution, η, and a regular language, L. To prove that $\eta(L)$ is regular, define L and η by regular expressions. More precisely, consider a regular language, L, such that $L = L(E_0)$, where E_0 is a regular expression over an alphabet Σ_0. Let η be a regular substitution η, such that for each $a \in \Sigma_0$, $\eta(a) = L(E_a)$, where E_a is a regular expression. Observe that for every word $a_1a_2\ldots a_n \in L(E_0)$, where $a_i \in \Sigma_0$, $i = 1, \ldots, n$, for some $n \geq 0$,

$$w_1w_2\ldots w_n \in \eta(a_1a_2\ldots a_n) \text{ if and only if } w_i \in L({}_iE_a)$$

(the case when $n = 0$ implies $a_1a_2\ldots a_n = w_1w_2\ldots w_n = \varepsilon$). Construct a regular expression E, satisfying $w \in L(E_0)$ if and only if $w' \in L(E)$, where $w = a_1a_2\ldots a_n \in \Sigma_0^*$; $a_i \in \Sigma_0$; $w' = w_1w_2\ldots w_n$ with $w_i \in L({}_iE_a)$; $i = 1, \ldots, n$; for some $n \geq 0$. As $\eta(a) = L(E_a)$, for all $a \in \Sigma_0$, $L(E) = \eta(L(E_0)) = \eta(L)$ and, thus, $\eta(L)$ is regular.

2.38. First, by contradiction, prove that $\{x|\ x \in \Sigma^*, x \in L(M), \text{ and } |x| < k\} = \emptyset$ implies $L(M) = \emptyset$. Assume that $\{x|\ x \in \Sigma^*, x \in L(M), \text{ and } |x| < k\} = \emptyset$ and $L(M) \neq \emptyset$; that is, all words of $L(M)$ are longer than $k - 1$. Let z be the shortest word in $L(M)$. As $z \in L(M)$ and $|z| \geq k$, by the pumping lemma for regular languages, $z = uvw$ so $v \neq \varepsilon$, $|uv| \leq k$, and uv^mw, for all $m \geq 0$. Take $z' = uw \in L(M)$. As $|z'| = |z| - |v|$ and $v \neq \varepsilon$, $|z| > |z'|$. Because $z' \in L(M)$ and $|z| > |z'|$, z is not the shortest word in $L(M)$—a contradiction. Thus, $L(M) = \emptyset$ if $\{x|\ x \in \Sigma^*, x \in L(M), \text{ and } |x| < k\} = \emptyset$. On the other hand, $L(M) = \emptyset$ implies $\{x|\ x \in \Sigma^*, x \in L(M), \text{ and } |x| < k\} = \emptyset$ because $\{x|\ x \in \Sigma^*, x \in L(M), \text{ and } |x| < k\} \subseteq L(M)$. Thus, $L(M) = \emptyset$ if and only if $\{x|\ x \in \Sigma^*, x \in L(M), \text{ and } |x| < k\} = \emptyset$.

2.40. Observe that $L = \{a^n | \ n$ is a prime$\}$ is infinite. Let z be any string in L such that $|z| \geq k$, where k is the constant from the regular pumping lemma (see Lemma 2.38). Express $z = uvw$, where u, v, and w satisfy the properties of the regular pumping lemma. If $|uw| \in \{0, 1\}$, then $|uv^0w| \notin L$—a contradiction. Let $|uw| \geq 2$. Set $m = |uw|$. By the pumping lemma $uv^mw \in L$. However, $|uv^mw| = |uw| + |v|^m = m + m|v| = m(1 + |v|)$, so $uv^mw \notin L$. Thus, L is no regular language.

CHAPTER 3

3 Syntax Analysis

The source language syntax is specified by a *grammar*, which is based upon finitely many rules. By these rules, the parser of a syntax analyzer verifies that a string of tokens, produced by the lexical analyzer, represents a syntactically well-formed source program. More precisely, reading the string in a left-to-right way, the parser derives the string by the grammatical rules, and if it finds this derivation, the source program syntax is correct. Graphically, this grammatical *derivation* is usually displayed by a *parse tree*, in which each parent-children portion represents a grammatical rule, so the parser's task can be also described as the construction of a parse tree for the string of tokens. This construction is made in a top-down or bottom-up way. A *top-down parser* builds the parse tree from the root and proceeds down toward the frontier while a *bottom-up parser* starts from the frontier and works up toward the root. By successfully completing the construction of this tree, the parser verifies that the source program is syntactically correct. Formally, a parser is usually specified by a *pushdown automaton*, which represents a finite automaton extended by a potentially infinite pushdown list.

As obvious, the syntax analysis of the tokenized source programs does not require any attributes attached to the tokens. Therefore, Chapters 3 through 5, which discuss syntax analysis, always deal with tokens without any attributes.

Synopsis. Section 3.1 introduces the fundamental language models underlying the syntax analysis. Specifically, it defines grammars, pushdown automata, and pushdown transducers. Making use of these models, Section 3.2 describes general parsing methods. Finally, Section 3.3 presents the theoretical foundations of parsing.

3.1 Models

Syntax analysis is based on these formal models

- grammars
- pushdown automata and pushdown transducers

Recall that as its main task the syntax analyzer verifies that the string of tokens produced by the lexical analyzer represents a syntactically well-formed source program. Since the tokens fulfill such a crucial role, it comes as no surprise that they are represented by special symbols in the above formal models. Specifically, in grammars, they are referred to as the *terminal symbols* because the grammatical derivation terminates at the string consisting of these symbols. In the pushdown automata and transducers, they are referred to as the *input symbols* because the syntax analyzer receives them as its input symbols from the lexical analyzer. Apart from these token-representing symbols, these formal models contain several other components, whose core is a finite set of rules according to which these models work as explained in the rest of this section.

Grammars

A grammar is based upon finitely many grammatical rules, which contain terminal and nonterminal symbols. As just pointed out, the terminal symbols represent tokens. The nonterminal symbols formalize more general syntactical entities, such as loops or arithmetic expressions. To verify that a string of terminal symbols represent the tokenized version of a syntactically well-formed program, we need to derive this string by the grammatical rules. If we find this derivation, the program is syntactically correct. If such a derivation does not exist, the program is syntactically incorrect. Next, we formalize this intuitive insight into grammars.

Definition 3.1 ***Grammar and its Language.*** A *grammar* is a rewriting system, $G = (\Sigma, R)$, where

- Σ is divided into two disjoint subalphabets, denoted by N and Δ;
- R is a finite *set of rules* of the form $A \rightarrow x$, where $A \in N$ and $x \in \Sigma^*$.

N and Δ are referred to as the *alphabet of nonterminal symbols* and the *alphabet of terminal symbols*, respectively. N contains a special nonterminal called the *start symbol*, denoted by S.

If $S \Rightarrow^* w$, where $w \in \Sigma^*$, G *derives* w, and w is a *sentential form*. $F(G)$ denotes the set of all sentential forms derived by G. The *language generated by G*, symbolically denoted by $L(G)$, is defined as $L(G) = F(G) \cap \Delta^*$, and its members are called *sentences*. If $S \Rightarrow^* w$ and w is a sentence, $S \Rightarrow^* w$ is a *successful derivation* in G.

■

In the literature, the notion of a grammar introduced in Definition 3.1 is often called, more precisely, a *context-free grammar* to point out that during any derivation step, a nonterminal is rewritten to a string regardless of the context surrounding the rewritten nonterminal as opposed to some other types of grammars that work in a context-dependent way. Accordingly, a language that is generated by a context-free grammar is called a *context-free language*, so the *family of context-free language* is the entire family of languages generated by the grammars.

Since grammars represent rewriting systems, all the notions introduced for these systems apply to them as well (see Section 1.3). That is, by $u \Rightarrow v$ [r], where $u, v \in \Sigma^*$ and $r \in R$, we express that G directly rewrites u to v by r or, as we usually say in terms of grammars, that G *makes* a *derivation step from u to v by r*. Furthermore, to express that G makes $u \Rightarrow^* w$ according to a sequence of rules, $r_1r_2\ldots r_n$, we write $u \Rightarrow^* v$ [$r_1r_2\ldots r_n$], which is read as a *derivation from u to v by using* $r_1r_2\ldots r_n$. On the other hand, whenever the information regarding the applied rules is immaterial, we omit these rules; in other words, when we simplify $u \Rightarrow^* v$ [$r_1r_2\ldots r_n$] and $u \Rightarrow v$ [r] to $u \Rightarrow^* v$ and $u \Rightarrow v$, respectively.

To explain Definition 3.1 informally, consider a string xAy and a rule r: $A \rightarrow u \in R$. By using this rule, G makes a derivation step from xAy to xuy by changing A to u in xAy, which is symbolically written as $xAy \Rightarrow xuy$. If G makes a sequence of derivation steps from S to a string $w \in \Sigma^*$, then w is a sentential form. Every sentential form from Δ^* is a sentence, and the set of all sentences is the language of G. As a result, every sentence w satisfies $S \Rightarrow^* w$ with $w \in \Delta^*$, so $L(G) = \{w \in \Delta^* | \, S \Rightarrow^* w\}$.

Convention 3.2. For any grammar G, we automatically assume that Σ, N, Δ, S, and R denote G's total alphabet, alphabet of nonterminal symbols, alphabet of terminal symbols, start symbol, and set of rules, respectively. If there exists a danger of confusion, we mark Σ, N, Δ, S, R with G as ${}_G\Sigma$, ${}_GN$, ${}_G\Delta$, ${}_GS$, ${}_GR$, respectively, in order to clearly relate these components to G (in particular, we make these marks when several grammars are simultaneously discussed). For brevity, we often call nonterminal symbols and terminal symbols *nonterminals* and *terminals*, respectively. Terminals and nonterminals are usually represented by early lowercase letters a, b, c, d and early uppercase letters A, B, C, D, respectively. Under these conventions, we often describe a grammar by simply listing its rules. If we want to express that a nonterminal, A, forms the left-hand side of a rule, we refer to this rule as an *A-rule*. If we want to specify that A is rewritten during a derivation step, $xAy \Rightarrow xuy$, we underline this A as $x\underline{A}y \Rightarrow xuy$.

■

Example 3.1 ***Grammar and its Language.*** Consider $L = \{a^k b^k | \ k \geq 1\}$. In principle, we generate the strings from L by a grammar, G, so G derives sentential forms

$$S, aSb, aaSbb, aaaSbbb, \dots$$

in which G rewrites S with ab during the very last derivation step. Formally, $G = (\Sigma, R)$, where $\Sigma = \{a, b, S\}$ and $R = \{S \rightarrow aSb, S \rightarrow ab\}$. In Σ, $\Delta = \{a, b\}$ is the alphabet of terminals and $N = \{S\}$ is the alphabet of nonterminals, where S is the start symbol of G. Under Convention 3.2, we can specify G simply as

1: $S \rightarrow aSb$
2: $S \rightarrow ab$

Consider $aaSbb$. By using rule 1, G rewrites S with aSb in this string, so $aaSbb \Rightarrow aaaSbbb$ [1]. By using 2, G rewrites S with ab, so $aaSbb \Rightarrow aaabbb$ [2]. By using the sequence of rules 112, G makes

$$\begin{aligned} aaSbb &\Rightarrow aaaSbbb && [1] \\ &\Rightarrow aaaaSbbbb && [1] \\ &\Rightarrow aaaaabbbbb && [2] \end{aligned}$$

Briefly, we write $aaSbb \Rightarrow^* aaaaabbbbb$ [112] or, even more simply, $aaSbb \Rightarrow^* aaaaabbbbb$.

To verify that G generates $\{a^k b^k | \ k \geq 1\}$, recall that by using rule 1, G replaces S with aSb, and by rule 2, G replaces S with ab. Consequently, every successful derivation has the form

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow \dots \Rightarrow a^k S b^k \Rightarrow a^{k+1} b^{k+1},$$

which G makes according to a sequence of rules of the form 1^k2, for some $k \geq 0$. In symbols, $S \Rightarrow^* a^{k+1} b^{k+1}$ $[1^k2]$. From these observations, we see $L(G) = \{a^k b^k | \ k \geq 1\}$; a detailed verification of this identity is left as an exercise.

■

Convention 3.3. Throughout this book, in the grammars for programming languages, nonterminals have the general angle-bracket form $\langle x \rangle$, where x is a name of the syntactical construct that this nonterminal defines, while the other symbols are terminals. For instance, in Case Study 5/35 *Grammar*, ⟨declaration part⟩ is a nonterminal that defines the syntactically well-formed declaration parts of the FUN programs whereas i is a terminal, which represents an identifier. In a list of rules that define a grammar, the left-hand side of the first rule represents the grammar's start symbol; for example, ⟨program⟩ is the start symbol of *FUN-GRAMMAR* defined in the next case study.

■

Case Study 5/35 ***Grammar.*** In this part of the case study, we design a grammar, *FUN-GRAMMAR*, which defines the syntax of the programming language FUN. The terminals of *FUN-GRAMMAR* coincide with the FUN tokens without attributes (see Case Study 3/35). Next, we introduce *FUN-GRAMMAR*'s nonterminals and, most importantly, the grammatical rules that rigorously define the FUN syntax.

We want every FUN program to start and end with keywords ***program*** and ***end***, respectively. Like in most programming languages, it consists of two essential parts—a declaration part and an execution part. Therefore, we introduce the start symbol ⟨program⟩ and this ⟨program⟩-rule (recall that according to Convention 3.2, a ⟨program⟩-rule means a rule whose left-hand side is ⟨program⟩)

⟨program⟩ → ***program*** ⟨declaration part⟩ ⟨execution part⟩ ***end***

The declaration part starts with keyword ***declaration*** followed by a declaration list of program variables. Therefore, we have this rule

⟨declaration part⟩ → ***declaration*** ⟨declaration list⟩

so that *FUN-GRAMMAR* makes ⟨declaration part⟩ $\Rightarrow^*$ ***declaration*** *x*, where *x* represents a list of declarations, derived by using these two rules

⟨declaration list⟩ → ⟨declaration⟩; ⟨declaration list⟩
⟨declaration list⟩ → ⟨declaration⟩

Indeed, these two rules derive a string that consists of some substrings separated by semicolons so that each of these substrings represents a declaration; formally,

⟨declaration list⟩ $\Rightarrow^* y_1; \ldots; y_n$

where ⟨declaration⟩ $\Rightarrow^* y_i$, $1 \leq i \leq n$, for some $n \geq 1$ ($n = 1$ means ⟨declaration list⟩ $\Rightarrow^* y_1$), and each y_i represents a declaration, which starts with ***integer***, ***real*** or ***label***. For this purpose, we add these three rules

⟨declaration⟩ → ***integer*** ⟨variable list⟩
⟨declaration⟩ → ***real*** ⟨variable list⟩
⟨declaration⟩ → ***label*** ⟨label list⟩

If a declaration starts with ***integer*** or ***real***, this keyword follows by a list of identifiers of the form *i*, …, *i*; recall that *i* is a token representing an identifier (see Section 2.2). Formally, *FUN-GRAMMAR* makes ⟨declaration⟩ ⇒ *a*⟨variable list⟩ $\Rightarrow^*$ *ay*, in which $a \in$ {***integer, real***} and *y* = *i*, …, *i*. To generate *y* of this form, we introduce these two ⟨variable list⟩-rules:

⟨variable list⟩ → *i*, ⟨variable list⟩
⟨variable list⟩ → *i*

If a declaration starts with ***label***, it follows by a list of labels, so we extend *FUN-GRAMMAR* by these rules

⟨label list⟩ → *l*, ⟨label list⟩
⟨label list⟩ → *l*

Return to the declaration part of the FUN source program given in Case Study 1/35 in Section 1.2. Omitting the attribute, this declaration part equals ***integer*** *i*;. Observe that

⟨declaration list⟩ $\Rightarrow^*$ ***integer*** *i*;

by the above rules in *FUN-GRAMMAR*.

The execution part of a *FUN* program starts with ***execution*** followed by a list of statements, so we introduce

$$\langle\text{execution part}\rangle \rightarrow \textbf{\textit{execution}}\ \langle\text{statement list}\rangle$$

The statement list that follows ***execution*** consists of a sequence of statements separated by semicolons. To generate a sequence like this, we introduce these two ⟨statement list⟩-rules

⟨statement list⟩ → ⟨statement⟩; ⟨statement list⟩
⟨statement list⟩ → ⟨statement⟩

Indeed, these two rules derive a string that consists of a sequence of FUN statements separated by semicolons; formally,

$$\langle\text{statement list}\rangle \Rightarrow^* y_1; \ldots; y_n$$

where ⟨statement ⟩ $\Rightarrow^* y_i$, $1 \leq i \leq n$, for some $n \geq 1$ ($n = 1$ means ⟨statement list⟩ $\Rightarrow^* y_1$), and each y_i represents a statement, defined by the following rules

⟨statement⟩ → *i* = ⟨expression⟩
⟨statement⟩ → *l* ⟨statement⟩
⟨statement⟩ → ***goto*** *l*
⟨statement⟩ → ***read***(⟨input list⟩)
⟨statement⟩ → ***write***(⟨output list⟩)
⟨statement⟩ → ***if*** ⟨condition⟩ ***then*** ⟨statement⟩
⟨statement⟩ → ***if*** ⟨condition⟩ ***then*** ⟨statement⟩ ***else*** ⟨statement⟩
⟨statement⟩ → ***provided*** ⟨condition⟩ ***iterate*** ⟨statement⟩
⟨statement⟩ → ***for*** *i* = ⟨expression⟩ ***through*** ⟨expression⟩ ***iterate*** ⟨statement⟩
⟨statement⟩ → ***begin*** ⟨statement list⟩ ***end***

Next, we interpret these rules one by one in a Pascal-like way.

Rule: ⟨statement⟩ → *i* = ⟨expression⟩
Example: $a = a + 1$
Interpretation: The variable specified by *i* is assigned the evaluated FUN expression right of =.

Rule: ⟨statement⟩ → *l* ⟨statement⟩
Example: @*lab*1 @*lab*2 $a = a + 1$
Interpretation: A sequence of labels may precede any statement. These labels are used as the addresses of ***goto*** instructions in FUN.

Rule: ⟨statement⟩ → ***goto*** *l*
Example: ***goto*** @*lab*1
Interpretation: Control is transferred to the statement denoted with the label following ***goto***.

Rule: ⟨statement⟩ → ***read***(⟨input list⟩)
Example: ***read***(*a*, *b*)
Interpretation: Data are entered from standard input into each variable specified in the input list.

Rule: ⟨statement⟩ → ***write***(⟨output list⟩)
Example: ***write***('RESULT = ', *a* + 2)
Interpretation: The characters specified by the output list are written to the end of the standard output.

Rule: ⟨statement⟩ → ***if*** ⟨condition⟩ ***then*** ⟨statement⟩
Example: ***if*** $a > 0$ ***then*** $b = a + 1$
Interpretation: If the condition evaluates to true, then the statement following ***then*** is executed.

Rule: ⟨statement⟩ → ***if*** ⟨condition⟩ ***then*** ⟨statement⟩ ***else*** ⟨statement⟩
Example: ***if*** $a > 0$ ***then*** $b = a + 1$ ***else*** $b = 12$
Interpretation: If the condition evaluates to true, then the statement between ***then*** and ***else*** is executed; otherwise, the statement behind ***else*** is executed.

Rule: ⟨statement⟩ → ***provided*** ⟨condition⟩ ***iterate*** ⟨statement⟩
Example: ***provided*** $a > 0$ ***iterate begin*** $b = b * a$; $a = a - 1$ ***end***
Interpretation: If the condition of this loop statement evaluates to true, then the statement behind ***iterate*** is repeated; otherwise, the loop exit occurs.

Rule: ⟨statement⟩ → ***for*** i = ⟨expression⟩ ***through*** ⟨expression⟩ ***iterate*** ⟨statement⟩
Example: ***for*** $q = 1$ ***to*** $o*2$ ***do*** $j = q * j$
Interpretation: The type of variable i is declared as **integer**. Let k and h be the integer values of the expressions preceding and following ***through***, respectively. The statement after ***iterate*** is repeated for $i = k, k + 1, \ldots, h$ provided that $h \geq k$; if $k > h$, the statement is repeated zero times.

Rule: ⟨statement⟩ → ***begin*** ⟨statement list⟩ ***end***
Example: ***begin*** $b = a$; $a = 0$ ***end***
Interpretation: The compound statement is bracketed by ***begin*** and ***end***.

Consider ⟨input list⟩ in rule ⟨statement⟩ → ***read***(⟨input list⟩). The input list consists of a sequence of identifiers separated by commas, so we define this list by these two ⟨input list⟩-rules:

⟨input list⟩ → i, ⟨input list⟩
⟨input list⟩ → i

Consider ⟨output list⟩ in rule ⟨statement⟩ → ***write***(⟨output list⟩). The output list consists of a sequence of text literals or expressions separated by commas, so we define this list by these rules:

⟨output list⟩ → ⟨write member⟩, ⟨output list⟩
⟨write member⟩ → ⟨expression⟩
⟨write member⟩ → t

Consider ⟨condition⟩, from which we want to derive conditions of the form

$$\iota_0 o_1 \iota_1 o_2 \iota_2 \ldots \iota_{n-1} o_n \iota_n$$

where os are logical operators, and ιs are arithmetic expressions or some other conditions enclosed by parentheses. We create a rule for the derivation of expressions simply as

⟨condition⟩ → ⟨expression⟩

Apart from expressions, however, ι may be a parenthesized condition—that is, any valid condition enclosed by parentheses; therefore, we also introduce

⟨condition⟩ → (⟨condition⟩)

The binary logical FUN operators are *r* (relational operator), ∨, and ∧, so we construct the rules for them as

⟨condition⟩ → ⟨condition⟩ *r* ⟨condition⟩
⟨condition⟩ → ⟨condition⟩ ∨ ⟨condition⟩
⟨condition⟩ → ⟨condition⟩ ∧ ⟨condition⟩

To include ¬ as the only legal unary logical FUN operator, we add

⟨condition⟩ → ¬ ⟨condition⟩

As a result, the FUN conditions are derived by

⟨condition⟩ → ⟨condition⟩ *r* ⟨condition⟩
⟨condition⟩ → ⟨condition⟩ ∧ ⟨condition⟩
⟨condition⟩ → ⟨condition⟩ ∨ ⟨condition⟩
⟨condition⟩ → ¬ ⟨condition⟩
⟨condition⟩ → (⟨condition⟩)
⟨condition⟩ → ⟨expression⟩

In a similar way, we construct the next rules that define the FUN expressions:

⟨expression⟩ → ⟨expression⟩ + ⟨term⟩
⟨expression⟩ → ⟨expression⟩ – ⟨term⟩
⟨expression⟩ → ⟨term⟩
⟨term⟩ → ⟨term⟩ * ⟨factor⟩
⟨term⟩ → ⟨term⟩ / ⟨factor⟩
⟨term⟩ → ⟨factor⟩
⟨factor⟩ → (⟨expression⟩)
⟨factor⟩ → *i*
⟨factor⟩ → #
⟨factor⟩ → ¢

Summarizing the previous construction, we obtain a complete version of *FUN-GRAMMAR* as

⟨program⟩ → ***program*** ⟨declaration part⟩ ⟨execution part⟩ ***end***
⟨declaration part⟩ → ***declaration*** ⟨declaration list⟩
⟨declaration list⟩ → ⟨declaration⟩; ⟨declaration list⟩
⟨declaration list⟩ → ⟨declaration⟩
⟨declaration⟩ → ***integer*** ⟨variable list⟩
⟨declaration⟩ → ***real*** ⟨variable list⟩
⟨declaration⟩ → ***label*** ⟨label list⟩
⟨variable list⟩ → *i*, ⟨variable list⟩
⟨variable list⟩ → *i*
⟨label list⟩ → *l*, ⟨label list⟩
⟨label list⟩ → *l*
⟨execution part⟩ → ***execution*** ⟨statement list⟩
⟨statement list⟩ → ⟨statement⟩; ⟨statement list⟩
⟨statement list⟩ → ⟨statement⟩
⟨statement⟩ → *i* = ⟨expression⟩

$\langle$statement$\rangle \rightarrow l\ \langle$statement$\rangle$
$\langle$statement$\rangle \rightarrow$ ***goto*** l
$\langle$statement$\rangle \rightarrow$ ***read***($\langle$input list$\rangle$)
$\langle$statement$\rangle \rightarrow$ ***write***($\langle$output list$\rangle$)
$\langle$statement$\rangle \rightarrow$ ***if*** $\langle$condition$\rangle$ ***then*** $\langle$statement$\rangle$
$\langle$statement$\rangle \rightarrow$ ***if*** $\langle$condition$\rangle$ ***then*** $\langle$statement$\rangle$ ***else*** $\langle$statement$\rangle$
$\langle$statement$\rangle \rightarrow$ ***provided*** $\langle$condition$\rangle$ ***iterate*** $\langle$statement$\rangle$
$\langle$statement$\rangle \rightarrow$ ***for*** $i = \langle$expression$\rangle$ ***through*** $\langle$expression$\rangle$ ***iterate*** $\langle$statement$\rangle$
$\langle$statement$\rangle \rightarrow$ ***begin*** $\langle$statement list$\rangle$ ***end***
$\langle$input list$\rangle \rightarrow i,\ \langle$input list$\rangle$
$\langle$input list$\rangle \rightarrow i$
$\langle$output list$\rangle \rightarrow \langle$write member$\rangle, \langle$output list$\rangle$
$\langle$write member$\rangle \rightarrow \langle$expression$\rangle$
$\langle$write member$\rangle \rightarrow t$
$\langle$condition$\rangle \rightarrow \langle$condition$\rangle\ r\ \langle$condition$\rangle$
$\langle$condition$\rangle \rightarrow \langle$condition$\rangle \wedge \langle$condition$\rangle$
$\langle$condition$\rangle \rightarrow \langle$condition$\rangle \vee \langle$condition$\rangle$
$\langle$condition$\rangle \rightarrow \neg \langle$condition$\rangle$
$\langle$condition$\rangle \rightarrow (\langle$condition$\rangle)$
$\langle$condition$\rangle \rightarrow \langle$expression$\rangle$
$\langle$expression$\rangle \rightarrow \langle$expression$\rangle + \langle$term$\rangle$
$\langle$expression$\rangle \rightarrow \langle$expression$\rangle - \langle$term$\rangle$
$\langle$expression$\rangle \rightarrow \langle$term$\rangle$
$\langle$term$\rangle \rightarrow \langle$term$\rangle * \langle$factor$\rangle$
$\langle$term$\rangle \rightarrow \langle$term$\rangle / \langle$factor$\rangle$
$\langle$term$\rangle \rightarrow \langle$factor$\rangle$
$\langle$factor$\rangle \rightarrow (\langle$expression$\rangle)$
$\langle$factor$\rangle \rightarrow i$
$\langle$factor$\rangle \rightarrow$ #
$\langle$factor$\rangle \rightarrow$ ¢

■

If a grammar generates every sentence by a unique derivation, such as the grammar discussed in Example 3.1, the analysis of the derivation process represents a relatively simple task. In a general case, however, a grammar can make several different derivations that lead to the same sentence as illustrated in Example 3.2. As this derivation multiplicity obviously makes any grammatical analysis more complicated, we next reduce it by introducing special types of *canonical derivations* that include leftmost and rightmost derivations.

Definition 3.4 ***Leftmost and Rightmost Derivations.*** Let $G = (\Sigma, R)$ be a grammar, $r: A \rightarrow x \in R$, $u \in \Delta^*$, $v \in \Sigma^*$; recall that Δ is G's alphabet of terminals (see Definition 3.1). Then, G makes a *leftmost derivation step* from uAv to uxv according to r, symbolically written as $uAv\ {}_{lm}\Rightarrow uxv\ [r]$ in G. G makes a *rightmost derivation step* from vAu to vxu according to r, symbolically written as $vAu\ {}_{rm}\Rightarrow vxu\ [r]$ in G.

Let $w_0, w_1, \ldots, w_n$ be a sequence, where $w_i \in \Sigma^*$, $1 \leq i \leq n$, for some $n \geq 1$. If $w_{j\text{-}1}\ {}_{lm}\Rightarrow w_j\ [r_j]$ in G, where $r_j \in R$ for all $1 \leq j \leq n$, then G makes a leftmost derivation from w_0 to w_n according to $r_1r_2\ldots r_n$, symbolically written as $w_0\ {}_{lm}\Rightarrow^* w_n\ [r_1r_2\ldots r_n]$. If $w_{j\text{-}1}\ {}_{rm}\Rightarrow w_j\ [r_j]$ in G, where $r_j \in R$ for all $1 \leq j \leq n$, then G makes a rightmost derivation from w_0 to w_n according to $r_1r_2\ldots r_n$, symbolically written as $w_0\ {}_{rm}\Rightarrow^* w_n\ [r_1r_2\ldots r_n]$. In addition, for every $w \in \Sigma^*$, we write $w\ {}_{lm}\Rightarrow^* w$

[ε] and $w \; {}_{rm}{\Rightarrow}^{*} \; w$ [ε], which represent the *zero-step leftmost and rightmost derivations*, respectively.

■

To rephrase the second part of Definition 3.4 less formally, let $u, v \in \Sigma^*$ and $\rho \in R^*$. If G makes $u \Rightarrow^* v$ [ρ] so that every step is leftmost in this derivation, then this derivation is *leftmost*, symbolically written $u \; {}_{lm}{\Rightarrow}^{*} \; v$ [ρ]. If G makes $u \Rightarrow^* v$ [ρ] so that every step is rightmost in this derivation, then this derivation is *rightmost*, $u \; {}_{rm}{\Rightarrow}^{*} \; v$ [ρ].

Apart from the canonical derivations, we often significantly simplify the discussion concerning parsing by introducing *parse trees*, which graphically represent derivations by displaying rules together with the nonterminals to which the rules are applied. On the other hand, they suppress the order of the applications of the rules, so we make use of these trees when this order is immaterial.

In the following definition of the parse trees, we use the terminology concerning trees introduced in Section 1.1. Let us note that the literature sometimes refers to these trees as *derivation trees* to express that they represent grammatical derivations.

Definition 3.5 ***Rule Trees and Parse Trees.*** Let $G = (\Sigma, R)$ be a grammar, and l: $A \rightarrow x \in R$. The *rule tree corresponding to l*, symbolically denoted by $rt(l)$, is a tree of the form $A\langle x\rangle$. A *parse tree* whose root is from Σ and each parent-children portion occurring in this tree represents $rt(l)$, for some rule $l \in R$.

The *correspondence between derivations and their parse trees* is defined recursively as follows.

- One-node tree X is the parse tree corresponding to $X \Rightarrow^0 X$ in G, where $X \in \Sigma$;
- Let t be the parse tree corresponding to $A \Rightarrow^* xBy$, $frontier(t) = xBy$, l: $B \rightarrow z \in R$. The parse tree corresponding to

$$\begin{array}{lll} A & \Rightarrow^* & x\underline{B}y \\ & \Rightarrow & xzy \quad [l] \end{array}$$

is obtained by replacing t's ($|x|$+1)st leaf, B, with $rt(l)$.

For $A \Rightarrow^* x$ in $G = (\Sigma, R)$, $pt(A \Rightarrow^* x)$ denotes the parse tree corresponding to this derivation.

■

Example 3.2 ***Canonical Derivations and Parse Trees.*** Consider L from Example 3.1. Let K be the set of all permutations of strings in L. That is, K contains all nonempty strings consisting of an equal number of as and bs, so we can equivalently define $K = \{w \mid w \in \{a, b\}^+$ and $occur(w, a) = occur(w, b)\}$. As proved in Section 3.3.2 (see Example 3.5), $K = L(G)$, where grammar G is defined as

1: $S \rightarrow aB$
2: $S \rightarrow bA$
3: $A \rightarrow a$
4: $A \rightarrow aS$
5: $A \rightarrow bAA$
6: $B \rightarrow b$
7: $B \rightarrow bS$
8: $B \rightarrow aBB$

Consider *aabbab* $\in$ *L*(*G*). *G* generates this sentence by derivations I through IV in Figure 3.1 (recall that we specify the rewritten symbols by underlining).

I		**II**		**III**		**IV**	
S		S		S		S	
$\Rightarrow a\underline{B}$	[1]	$\Rightarrow a\underline{B}$	[1]	$\Rightarrow a\underline{B}$	[1]	$\Rightarrow a\underline{B}$	[1]
$\Rightarrow aa\underline{B}B$	[8]	$\Rightarrow aaB\underline{B}$	[8]	$\Rightarrow aa\underline{B}B$	[8]	$\Rightarrow aa\underline{B}B$	[8]
$\Rightarrow aabS\underline{B}$	[7]	$\Rightarrow aa\underline{B}b$	[6]	$\Rightarrow aab\underline{S}B$	[7]	$\Rightarrow aab\underline{S}B$	[7]
$\Rightarrow aab\underline{S}b$	[6]	$\Rightarrow aab\underline{S}b$	[7]	$\Rightarrow aabbA\underline{B}$	[2]	$\Rightarrow aabb\underline{A}B$	[2]
$\Rightarrow aabb\underline{A}b$	[2]	$\Rightarrow aabb\underline{A}b$	[2]	$\Rightarrow aabb\underline{A}b$	[6]	$\Rightarrow aabba\underline{B}$	[3]
$\Rightarrow aabbab$	[3]	$\Rightarrow aabbab$	[3]	$\Rightarrow aabbab$	[3]	$\Rightarrow aabbab$	[6]

Figure 3.1 *Four Derivations for* *aabbab*.

Observe that in derivation IV, every step is leftmost, so $S\ {}_{lm}\Rightarrow^* aabbab$ [187236] while derivation II is rightmost, symbolically written as $S\ {}_{rm}\Rightarrow^* aabbab$ [186723].

Figure 3.2 presents the rule trees corresponding to *G*'s eight rules, *rt*(1) through *rt*(8). In addition, the first of these rule trees, *rt*(1), is pictorially shown in Figure 3.3.

Rule	***Rule Tree***
1: $S \rightarrow aB$	$S\langle aB\rangle$
2: $S \rightarrow bA$	$S\langle bA\rangle$
3: $A \rightarrow a$	$A\langle a\rangle$
4: $A \rightarrow aS$	$A\langle aS\rangle$
5: $A \rightarrow bAA$	$A\langle bAA\rangle$
6: $B \rightarrow b$	$B\langle b\rangle$
7: $B \rightarrow bS$	$B\langle bS\rangle$
8: $B \rightarrow aBB$	$B\langle aBB\rangle$

Figure 3.2 *Rules and the Corresponding Rule Trees*.

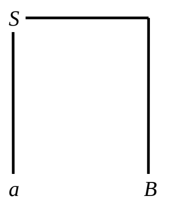


Figure 3.3 *Rule Tree* *rt*(1).

Consider, for instance, derivation IV in Figure 3.1. Next, Figure 3.4 presents this derivation together with its corresponding parse tree constructed in a step-by-step way.

Derivation		***Parse Tree***
S		S
$\Rightarrow a\underline{B}$	[1]	$S\langle aB\rangle$
$\Rightarrow aa\underline{B}B$	[8]	$S\langle aB\langle aBB\rangle\rangle$
$\Rightarrow aab\underline{S}B$	[7]	$S\langle aB\langle aB\langle bS\rangle B\rangle\rangle$
$\Rightarrow aabb\underline{A}B$	[2]	$S\langle aB\langle aB\langle bS\langle bA\rangle\rangle B\rangle\rangle$
$\Rightarrow aabba\underline{B}$	[3]	$S\langle aB\langle aB\langle bS\langle bA\langle a\rangle\rangle\rangle B\rangle\rangle$
$\Rightarrow aabbab$	[6]	$S\langle aB\langle aB\langle bS\langle bA\langle a\rangle\rangle\rangle B\langle b\rangle\rangle\rangle$

Figure 3.4 *Derivation and the Corresponding Parse Tree*.

■

As formally proved in Section 3.3.3 (see Corollary 3.24), without any loss of generality, we can represent every grammatical derivation by a canonical derivation or a parse tree, depending on what is more appropriate under given discussion. Unfortunately, even if we reduce our attention only to these derivations and trees, we do not always remove the undesirable derivation multiplicity of the same sentences. Indeed, there exist grammars that make several different canonical derivations of the same sentences, and this grammatical ambiguity obviously desires our special attention when discussing parsing.

Definition 3.6 ***Grammatical Ambiguity.*** Let $G = (\Sigma, R)$ be a grammar. If there exists a sentence $w \in L(G)$ and two different sequences $\rho, \sigma \in R^*$, $\rho \neq \sigma$, such that $S\ {}_{lm}\!\Rightarrow^* w\ [\rho]$ and $S\ {}_{lm}\!\Rightarrow^* w\ [\sigma]$, then G is *ambiguous*; otherwise, G is *unambiguous*.

■

Equivalently, in terms of rightmost derivations, G is ambiguous if in G, there exist $S\ {}_{rm}\!\Rightarrow^* w\ [\rho]$ and $S\ {}_{rm}\!\Rightarrow^* w\ [\sigma]$ with $\rho \neq \sigma$, for some $w \in L(G)$. In terms of parse trees, G is ambiguous if there exist $w \in L(G)$ and two different parse trees, t and u, such that $root(t) = root(u) = S$ and $frontier(t) = frontier(u) = w$.

Example 3.3 ***Ambiguous Grammar of a Programming Language.*** The following example discusses a well-known ambiguous specification of the **if-then-else** statement that occurred in the original description of the programming language ALGOL 60, which represented an important ancestor of Pascal. This statement was defined, in essence, by these grammatical rules

1: $S \rightarrow$ **if** b **then** S **else** S
2: $S \rightarrow$ **if** b **then** S
3: $S \rightarrow a$

Consider the grammar consisting of these three rules, containing a single nonterminal, S. Observe that this grammar generates **if** b **then if** b **then** a **else** a by the two different leftmost derivations given in Figure 3.5.

leftmost derivation 1		***leftmost derivation*** 2	
S		S	
$\Rightarrow$ **if** b **then** $\underline{S}$ **else** S	[1]	$\Rightarrow$ **if** b **then** $\underline{S}$	[2]
$\Rightarrow$ **if** b **then if** b **then** $\underline{S}$ **else** S	[2]	$\Rightarrow$ **if** b **then if** b **then** $\underline{S}$ **else** S	[1]
$\Rightarrow$ **if** b **then if** b **then** a **else** S	[3]	$\Rightarrow$ **if** b **then if** b **then** a **else** S	[3]
$\Rightarrow$ **if** b **then if** b **then** a **else** a	[3]	$\Rightarrow$ **if** b **then if** b **then** a **else** a	[3]

Figure 3.5 ***Different Leftmost Derivations of*** **if** b **then if** b **then** a **else** a**.**

As a consequence of this ambiguity, there exist two different interpretations of **if** b **then if** b **then** a **else** a. Although this undesirable consequence becomes quite clear after the explanation of the syntax directed translation in Chapter 6, we already intuitively see that the first interpretation associates the **else** part of this statement with the first **then** while the other interpretation makes this association with the second **then**. Obviously, this ambiguous interpretation has to be eliminated to translate the statement by a compiler. To do so, we can redefine the statement in an unambiguous way. Specifically, we define the **if-then-else** statement by the following equivalent unambiguous two-nonterminal five-rule grammar. This grammar always attaches each **else** to the last proceeded unmatched **then**, which is what most real programming languages do.

$S \rightarrow$ **if** b **then** S
$S \rightarrow$ **if** b **then** A **else** S

$S \to a$
$A \to$ **if** b **then** A **else** A
$A \to a$

■

As illustrated by the previous example, we often grammatically eliminate the ambiguity of some programming-language structures at the price of introducing several new nonterminals and rules, which increase and complicate the programming language description as a whole. Therefore, eliminating the ambiguity at the grammatical level may not be the best we can do in practice. Indeed, some parsing methods can be based on ambiguous grammars and still work in a completely unambiguous way. Specifically, grammar ${}_{cond}G$ from Case Study 6/35, given next, is ambiguous, yet an operator-precedence parsing method based on this grammar works in a quite unambiguous way by using the priority of operators as explained in Section 5.1. As a matter of fact, sometimes, the grammatical way of eliminating the ambiguity is just ruled out. Indeed, there exist *inherently ambiguous languages* that are generated only by ambiguous grammars; for instance, $\{a^ib^jc^k \mid i, j, k \geq 1$, and $i = j$ or $j = k\}$ is a language of this kind.

Case Study 6/35 ***Grammars for* FUN *expressions and conditions—Ambiguity vs. Size.*** Consider the language consisting of FUN arithmetic expressions that contain only variables as operands (in other words, no numeric operands are considered in them). This language is generated by grammars ${}_{expr}G$ and ${}_{expr}H$ in Figure 3.6. While ${}_{expr}H$ is ambiguous, ${}_{expr}G$ is not. On the other hand, ${}_{expr}H$ is smaller than ${}_{expr}G$. Indeed, ${}_{expr}H$ has a single nonterminal and six rules while ${}_{expr}G$ has three nonterminals and eight rules. From a broader perspective, this example illustrates a common process of designing an appropriate grammatical specification for some programming language constructs in practice. Indeed, in reality, we often design several equivalent grammars for these constructs, carefully consider their pros and cons, and based on this consideration, we select the grammar that is optimal for our purposes.

${}_{expr}G$	${}_{expr}H$
$E \to E + T$	$E \to E + E$
$E \to E - T$	$E \to E - E$
$E \to T$	$E \to E * E$
$T \to T * F$	$E \to E / E$
$T \to T / F$	$E \to (E)$
$T \to F$	$E \to i$
$F \to (E)$	
$F \to i$	

Figure 3.6 ${}_{expr}G$ **and** ${}_{expr}H$**.**

${}_{cond}G$	${}_{cond}H$
$C \to C \vee C$	$E \to E \vee T$
$C \to C \wedge C$	$E \to T$
$C \to (C)$	$T \to T \wedge F$
$C \to i$	$T \to F$
	$F \to (E)$
	$F \to i$

Figure 3.7 ${}_{cond}G$ **and** ${}_{cond}H$**.**

To give another example, consider the two equivalent grammars, ${}_{cond}G$ and ${}_{cond}H$, for simplified FUN conditions. These conditions have only binary operators $\vee$ and $\wedge$, and their operands are variables. Observe that the ambiguous grammar ${}_{cond}G$ has a single nonterminal and four rules while the unambiguous grammar ${}_{cond}H$ has three nonterminals and six rules.

Let us note that ${}_{expr}G$ and ${}_{cond}G$ are used in many subsequent examples and case studies of this book.

■

Pushdown automata

In essence, a pushdown automaton represents a finite automaton extended by a potentially infinite stack, commonly referred to as a *pushdown* in the theory of parsing.

Definition 3.7 ***Pushdown Automaton and its Language.*** A *pushdown automaton* is a rewriting system, $M = (\Sigma, R)$, where

- Σ is divided into subalphabets Q, Γ, Δ, and $\{\blacktriangleright, \blacktriangleleft\}$ such that $\{\blacktriangleright, \blacktriangleleft\} \cap (Q \cup \Gamma \cup \Delta) = \emptyset$ and $Q \cap (\Gamma \cup \Delta) = \emptyset$;
- R is a finite *set of rules* of the form $x \to y$, where $x \in \Gamma^*Q(\Delta \cup \{\varepsilon\})$ and $y \in \Gamma^*Q$.

Q, Γ, and Δ are referred to as the *set of states*, *alphabet of pushdown symbols*, and *alphabet of input symbols*, respectively. Q contains the *start state*, denoted by s, and a *set of final states*, denoted by F. Γ contains the *start symbol*, S. Symbols $\blacktriangleright$ and $\blacktriangleleft$ act as *pushdown-bottom* and *input-end markers*, respectively. Like in any rewriting system, we define $u \Rightarrow v$, $u \Rightarrow^n v$ for $n \geq 0$, and $u \Rightarrow^* v$, where $u, v \in \Sigma^*$ (see Section 1.3). If $\blacktriangleright Ssw\blacktriangleleft \Rightarrow^* \blacktriangleright f\blacktriangleleft$ in M with $f \in F$, *M accepts w*. The set of all strings that M accepts is the *language accepted by M*, denoted by $L(M)$.

■

In this book, a pushdown automaton M is a mathematical model used to formalize a parser as follows. As stated in the beginning of this chapter, the input symbols represent the tokens produced by a lexical analyzer, so a string w consisting of input symbols describes the tokenized version of a source program that is parsed by M. The set of rules is a highly stylized algorithm according to which the parser operates. The acceptance of w by M means a successful completion of the parsing process. In this way, we can express the basic ideas behind various parsing methods clearly and precisely as frequently demonstrated in Chapters 4 and 5.

Convention 3.8. As pushdown automata are defined as special cases of rewriting systems, all the notions introduced for the rewriting systems are straightforwardly applied to these automata as well (see Section 1.3). In this book, all the pushdown automata have a single state denoted by $\blacklozenge$. As $\blacklozenge$ is always the only state in these automata, we automatically assume that this state is the start state and, at the same time, a final state. For any pushdown automaton $M = (\Sigma, R)$, Δ, Γ, and S, always denote M's input alphabet, pushdown alphabet, and start symbol, respectively. If necessary, we mark Δ, Γ, and S with M as ${}_M\Delta$, ${}_M\Gamma$, and ${}_MS$, respectively, in order to clearly relate these components to M; these marks are particularly useful when several automata are simultaneously discussed. As a rule, input symbols are represented by early lowercase letters a, b, c, and d, and pushdown non-input symbols are represented by early uppercase letters A, B, C, and D with the exception of S as the start symbol. By analogy with the way we specify a grammar, we often describe a pushdown automaton simply by listing its labeled rules.

■

Taking advantage of Convention 3.8, we now explain Definition 3.7 informally. Consider a *configuration*, $\blacktriangleright u\blacklozenge v\blacktriangleleft$, where $u \in \Gamma^*$, $v \in \Delta^*$. This configuration actually represents an instantaneous description of M. Indeed, $\blacktriangleright u$ represents the contents of the *p*ush*d*own, symbolically denoted by *pd*, specified in a right-to-left order, so the topmost pushdown symbol occurs as the rightmost symbol of *pd*. Furthermore, $v\blacktriangleleft$ is the remaining suffix of the *in*put *s*tring, symbolically denoted by *ins*, specified in an ordinary left-to-right order, so the current input symbol occurs as the leftmost symbol of $v\blacktriangleleft$. Let $\blacktriangleright xz\blacklozenge ay\blacktriangleleft$ be a configuration and r: $z\blacklozenge a \rightarrow u\blacklozenge \in R$, where $x, z, u \in \Gamma^*$, $a \in \Delta \cup \{\varepsilon\}$, $y \in \Delta^*$. By using this rule, M directly rewrites $\blacktriangleright xz\blacklozenge ay\blacktriangleleft$ to $\blacktriangleright xu\blacklozenge y\blacktriangleleft$, which is usually referred to as a *move* that M makes from $\blacktriangleright xA\blacklozenge ay\blacktriangleleft$ to $\blacktriangleright xu\blacklozenge y\blacktriangleleft$. If in this way M makes a sequence of moves from $\blacktriangleright S\blacklozenge w\blacktriangleleft$ to $\blacktriangleright\blacklozenge\blacktriangleleft$, where $w \in \Delta^*$, then M accepts w. The set of all strings accepted by M is the *language of M*.

Example 3.4. Return to the language $L = \{a^kb^k|\ k \geq 1\}$ discussed in Example 3.1. Next, we design a pushdown automaton M that accepts L, in principle, as follows. First, it pushes down *a*s. When *b* appears, M begins to pop up *a*s and pair them off with *b*s. If the number of *a*s and *b*s coincides, M accepts the input string. Formally, $M = (\Sigma, R)$, where $\Sigma = \Gamma \cup \Delta \cup \{\blacklozenge, \blacktriangleright, \blacktriangleleft\}$, $\Gamma = \{S, C\}$, $\Delta = \{a, b\}$, $Q = F = \{\blacklozenge\}$, $\blacklozenge$ is the start state, and

$$R = \{S\blacklozenge a \rightarrow Sa\blacklozenge,\ a\blacklozenge a \rightarrow aa\blacklozenge,\ a\blacklozenge b \rightarrow C\blacklozenge,\ aC\blacklozenge b \rightarrow C\blacklozenge,\ SC\blacklozenge \rightarrow \blacklozenge\}$$

Next, under Convention 3.8, we succinctly specify M by listing its rules labeled by 1 through 5 as

1: $S\blacklozenge a \rightarrow Sa\blacklozenge$
2: $a\blacklozenge a \rightarrow aa\blacklozenge$
3: $a\blacklozenge b \rightarrow C\blacklozenge$
4: $aC\blacklozenge b \rightarrow C\blacklozenge$
5: $SC\blacklozenge \rightarrow \blacklozenge$

For instance, with *aaabbb*, M makes this sequence of moves

$$\begin{array}{llll}
\blacktriangleright S\blacklozenge aaabbb\blacktriangleleft & \Rightarrow & \blacktriangleright Sa\blacklozenge aabbb\blacktriangleleft & [1] \\
 & \Rightarrow & \blacktriangleright Saa\blacklozenge abbb\blacktriangleleft & [2] \\
 & \Rightarrow & \blacktriangleright Saaa\blacklozenge bbb\blacktriangleleft & [2] \\
 & \Rightarrow & \blacktriangleright SaaC\blacklozenge bb\blacktriangleleft & [3] \\
 & \Rightarrow & \blacktriangleright SaC\blacklozenge b\blacktriangleleft & [4] \\
 & \Rightarrow & \blacktriangleright SC\blacklozenge\blacktriangleleft & [4] \\
 & \Rightarrow & \blacktriangleright\blacklozenge\blacktriangleleft & [5]
\end{array}$$

As $\blacklozenge$ is final, M accepts *aaabbb*.

In general, observe that M makes every acceptance according to a sequence of rules 12^n34^n5, for some $n \geq 0$; more specifically, by 12^n34^n5, M accepts $a^{n+1}b^{n+1}$. Consequently, $L(M) = \{a^kb^k|\ k \geq 1\}$; a rigorous proof of this identity is left as an exception.

■

Determinism. In general, from the same configuration, a pushdown automata can make several moves and, thereby, enter many different configurations. As a result, with the same input string, they can make various sequences of moves. This nondeterministic behavior is usually convenient in theory, but it is obviously undesirable in practice. Therefore, the next definition introduces deterministic pushdown automata that make no more than one move from any configuration. As a result, with any input string, they make a unique valid sequence of moves, which is a feature highly appreciated when these automata are implemented.

Definition 3.9 *Deterministic Pushdown Automaton*. Let $M = (\Sigma, R)$ be a pushdown automaton. M is *deterministic* if each $r \in R$ with $\textbf{\textit{lhs}}(r) \in x\blacklozenge a$, where $x \in \Gamma^*$ and $a \in \Delta \cup \{\varepsilon\}$, satisfies

$$\{r\} = \{p \mid p \in R \text{ with } \boldsymbol{lhs}(p) \in suffixes(\Gamma^*\{x\})\{\blacklozenge\}\{a, \varepsilon\}\}.$$

■

In essence, a pushdown transducer is a pushdown automaton that can emit an output string during every move.

Definition 3.10 ***Pushdown Transducer and its Translation.*** Let $M = (\Sigma, R)$ be a pushdown automaton, O be an *output alphabet*, and o be a total *output function* from R to O^*. Then, $\Pi = (\Sigma, R, o)$ is a *pushdown transducer underlain by M*. If M is deterministic, Π is a *deterministic pushdown transducer*.

Let $\Pi = (\Sigma, R, o)$ be a pushdown transducer, $v \in \Delta^*$, and $w \in O^*$. If *M accepts v according to* $r_1r_2\ldots r_n$, where $r_i \in R$, $1 \leq i \leq n$, for some $n \geq 0$, then Π *translates v to* $o(r_1)o(r_2)\ldots o(r_n)$ *according to* $r_1r_2\ldots r_n$. The *translation of* Π, $\tau(\Pi)$, is defined as $\tau(\Pi) = \{(v, w) \mid v \in \Delta^*, w \in O^*, \Pi \text{ translates } v \text{ to } w\}$.

■

Convention 3.11. For any pushdown transducer, $\Pi = (\Sigma, R, o)$, we automatically assume that O denotes its output alphabet; otherwise, we use the same notation as for any pushdown automaton (see Convention 3.8). For brevity, we often express $r \in R$ with $o(r) = y$ as ry; that is, if r: $u\blacklozenge a \to v\blacklozenge \in R$ and $o(r) = y$, we write r: $u\blacklozenge a \to v\blacklozenge y$.

■

3.2 Methods

The parser verifies that the tokenized version of a source program, w, produced by the lexical analyzer, is syntactically correct by using a grammar, $G = (\Sigma, R)$, that specifies the source program syntax. In terms of the previous section, G's terminal alphabet, Δ, coincides with the set of tokens, so $w \in \Delta^*$, and $L(G)$ contains the set of all correct tokenized source programs. The parser has to find a derivation of the form $S \Rightarrow^* w$, where S is G's start symbol, by using rules from R to demonstrate that $w \in L(G)$ and, thereby, verify that w is syntactically well-designed.

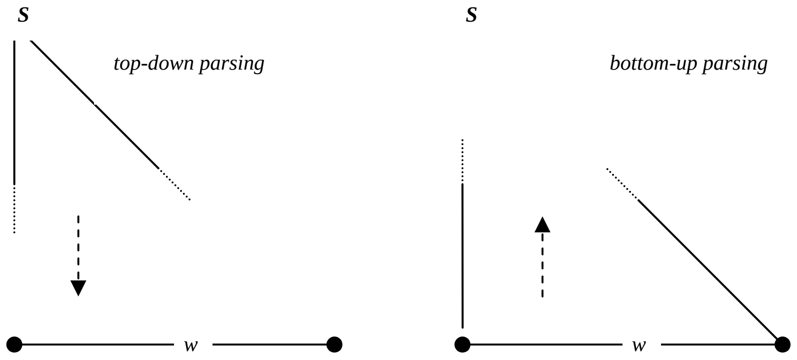


Figure 3.8 ***Top-Down and Bottom-Up Parsing.***

We usually express the construction of ${}_GS \Rightarrow^* w$ by building up a parse tree for w, $pt(S \Rightarrow^* w)$. Of course, initially, the parser has only the tree's root and frontier denoted by S and w, respectively. Starting from these two parts of the tree, the parser reads w from left to right and, simultaneously, tries to complete the construction of $pt(S \Rightarrow^* w)$ by composing various rule trees corresponding to R's rules. In essence, there exist two fundamental approaches to this construction. First, a *top-down parser* builds the tree starting from its root and proceeds down toward the frontier; alternatively, a *bottom-up parser* starts from the frontier and works up toward the root (see Figure 3.8). If the parser completes the construction of $pt(S \Rightarrow^* w)$, w is syntactically correct; otherwise, it is incorrect, and the parser has to take appropriate actions to handle the syntax errors.

Top-Down Parsing

Consider a grammar, $G = ({}_G\Sigma, {}_GR)$. The *G-based top-down parser* is represented by a pushdown automaton, $M = ({}_M\Sigma, {}_MR)$, which is not only equivalent to G but also makes its computation in a specific way closely related to the way G works. In terms of parse trees, M works so it uses its pushdown to construct G's parse trees from their roots down towards the frontiers. To rephrase this in terms of derivations, M actually models G's leftmost derivations by the pushdown. To do that, ${}_MR$ has a rule corresponding to each rule in ${}_GR$ by which M models a leftmost derivation step on its pushdown top. In addition, when a terminal symbol appears on the pushdown top, ${}_MR$ has an auxiliary rule by which M pops this symbol from the pushdown while reading the same symbol on the input tape to verify they both coincide. The *M-G parsing correspondence* between M's computation and G's leftmost derivations is described by a function o from ${}_MR^*$ to ${}_GR^*$ that erases the auxiliary rules and maps all the other rules in M to their corresponding grammatical rules. Consider ${}_GS\ {}_{lm}\Rightarrow^* w\ [\pi]$ in G, where $w \in {}_G\Sigma^*$ and $\pi \in {}_GR^*$; π is called a *left parse* of w. M models this leftmost derivation by making a sequence of moves according to $\rho \in {}_GR^*$ so o maps ρ precisely to the left parse π. Observe that at this point, with w as its input, the *G-based top-down transducer* $\Pi = ({}_M\Sigma, {}_MR, o)$ underlain by $M = ({}_M\Sigma, {}_MR)$ produces π as its output.

Goal. Given a grammar, $G = ({}_G\Sigma, {}_GR)$, construct the G-based top-down parser, M.

Gist. M uses its pushdown to simulate every leftmost derivation in G as follows. If a terminal a occurs as the pushdown top symbol, M *pops* a from the pushdown and, simultaneously, reads a on the input tape; in this way, it verifies their coincidence with each other. If a nonterminal A occurs as the pushdown top symbol, M simulates a leftmost derivation step made by a rule $r \in {}_GR$ with $\textbf{lhs}(r) = A$ by an *pd expansion* according to r so that it replaces the pushdown top A with $reversal(\textbf{rhs}(r))$. In somewhat greater detail, consider ${}_GS\ {}_{lm}\Rightarrow^* w$ in G, where $w \in {}_G\Delta^*$, expressed as

$$\begin{array}{llll} S & {}_{lm}\Rightarrow^* & vAy & [\rho] \\ & {}_{lm}\Rightarrow & vX_1X_2\ldots X_ny & [r: A \to X_1X_2\ldots X_n] \\ & {}_{lm}\Rightarrow^* & vu & \end{array}$$

where $w = vu$, so $v, u \in {}_G\Delta^*$. Suppose that M has just simulated the first portion of this derivation, $S\ {}_{lm}\Rightarrow^* vAy$; at this point, M's pushdown contains Ay in reverse while having u as the remaining input to read. In symbols, M occurs in the configuration $\blacktriangleright reversal(y)A\blacklozenge u\blacktriangleleft$, from which it simulates $vAy\ {}_{lm}\Rightarrow vX_1X_2\ldots X_ny\ [r: A \to X_1X_2\ldots X_n]$ as

$$\blacktriangleright reversal(y)A\blacklozenge u\blacktriangleleft \Rightarrow \blacktriangleright reversal(y)X_n\ldots X_1\blacklozenge u\blacktriangleleft$$

by using $A\blacklozenge \to X_n\ldots X_1\blacklozenge$ from ${}_MR$; notice that $reversal(y)X_n\ldots X_1 = reversal(X_1\ldots X_ny)$. In this way, M simulates $S\ {}_{lm}\Rightarrow^* w$ step by step and, thereby, acts as the G-based top-down parser.

Next, we give Algorithm 3.12, which describes the construction of M from G in a rigorous way.

Algorithm 3.12 ***Top-Down Parser for a Grammar*.**

Input • a grammar $G = ({}_G\Sigma, {}_GR)$.

Output • a G-based top-down parser represented by a pushdown automaton $M = ({}_M\Sigma, {}_MR)$ that accepts $L(G)$.

Method

```
begin
    $_M\Sigma := {}_G\Sigma$ with $_MN = {}_GN, {}_M\Delta = {}_G\Delta, {}_MS = {}_GS$;
    $_MR := \emptyset$;
    for each $r: A \rightarrow x \in {}_GR$, where $A \in_G N$ and $x \in {}_G\Sigma^*$ do
        add $A\blacklozenge \rightarrow reversal(x)\blacklozenge$ to $_MR$; {expansion rule}
    for each $a \in {}_G\Delta$ do
        add $a\blacklozenge a \rightarrow \blacklozenge$ to $_MR$;
end.
```

Crucially, the G-based top-down parser $M = ({}_M\Sigma, {}_MR)$ constructed by Algorithm 3.12 satisfies

$$A \rightarrow x \in {}_GR \text{ if and only if } A\blacklozenge \rightarrow reversal(x)\blacklozenge \in {}_MR$$

Introduce the M-G parsing correspondence o as a mapping from ${}_MR^*$ to ${}_GR^*$ defined as $o(A\blacklozenge \rightarrow reversal(x)\blacklozenge) = A \rightarrow x$, for every $A \rightarrow x \in {}_GR$, and $o(a\blacklozenge a \rightarrow \blacklozenge) = \varepsilon$, for each $a \in {}_G\Delta$. Observe that for every $\pi \in {}_GR^*$ such that $u\ {}_{lm}\Rightarrow^* yv\ [\pi]$, $y \in {}_G\Delta^*$, $u \in {}_GN_G\Sigma^*$, $v \in {}_GN_G\Sigma^* \cup \{\varepsilon\}$, there exists precisely one $\rho \in {}_GR^*$ such that $o(\rho) = \pi$ and

$$\blacktriangleright reversal(u)\blacklozenge y\blacktriangleleft \Rightarrow^* \blacktriangleright reversal(v)\blacklozenge\blacktriangleleft \quad [\rho] \text{ in } M \text{ if and only if } u\ {}_{lm}\Rightarrow^* yv\ [o(\rho)] \text{ in } G$$

Notice that for $u = {}_GS = {}_MS$ and $v = \varepsilon$, this equivalence states that

$$\blacktriangleright {}_MS\blacklozenge y\blacktriangleleft \Rightarrow^* \blacktriangleright\blacklozenge\blacktriangleleft \quad [\rho] \text{ in } M \text{ if and only if } {}_GS\ {}_{lm}\Rightarrow^* y\ [o(\rho)] \text{ in } G,$$

so $L(M) = L(G)$. The *G-based top-down transducer* is defined simply as the pushdown transducer $\Pi = ({}_M\Sigma, {}_MR, o)$ underlain by M. Observe that this transducer translates every $x \in L(G)$ to its left parse; more formally,

$$\tau(\Pi) = \{(x, \pi) \mid x \in L(G) \text{ and } \pi \text{ is a left parse of } x \text{ in } G\}$$

Lemma 3.13. Let $G = ({}_G\Sigma, {}_GR)$ be a grammar. With G as its input, Algorithm 3.12 correctly constructs the G-based top-down parser $M = ({}_M\Sigma, {}_MR)$ such that $L(M) = L(G)$.

Proof. Based on the notes preceding the algorithm, prove this lemma as an exercise by analogy with the proof of Lemma 3.17, which is more complicated and, therefore, presented in detail later in this section.

■

Case Study 7/35 ***Top-Down Parser*.** Consider the set of all well-formed tokenized FUN programs. Reduce this set by erasing everything but ***begin***s and ***end***s from it. The language resulting from this reduction thus contains all correctly balanced strings of ***begin***s and ***end***s; for instance, ***begin begin end begin end end*** is in this language, but ***begin begin end*** is not. For brevity, replace each ***begin*** and each ***end*** with *a* and *b*, respectively, in this language, which is generated by the following grammar, *G*:

1: $S \rightarrow SS$
2: $S \rightarrow aSb$
3: $S \rightarrow \varepsilon$

Algorithm 3.12 turns *G* to this *G*-based top-down parser, *M*,

$S◆ \rightarrow SS◆$
$S◆ \rightarrow bSa◆$
$S◆ \rightarrow \varepsilon◆$
$a◆a \rightarrow ◆$
$b◆b \rightarrow ◆$

Define *o* as $o(S◆ \rightarrow SS◆) = S \rightarrow SS$, $o(S◆ \rightarrow bSa◆) = S \rightarrow aSb$, $o(S◆ \rightarrow \varepsilon◆) = S \rightarrow \varepsilon$, $o(a◆a \rightarrow ◆) = \varepsilon$, and $o(b◆b \rightarrow ◆) = \varepsilon$. For brevity, by using the rule labels, we express $o(S◆ \rightarrow SS◆) = S \rightarrow SS$, $o(S◆ \rightarrow bSa◆) = S \rightarrow aSb$, and $o(S◆ \rightarrow \varepsilon◆) = S \rightarrow \varepsilon$ as $o(S◆ \rightarrow SS◆) = 1$, $o(S◆ \rightarrow bSa◆) = 2$, and $o(S◆ \rightarrow \varepsilon◆) = 3$, respectively. As a result, we have

$o(S◆ \rightarrow SS◆) = 1$, $o(S◆ \rightarrow bSa◆) = 2$, $o(A◆ \rightarrow \varepsilon◆) = 3$, and
$o(a◆a \rightarrow ◆) = \varepsilon$, $o(b◆b \rightarrow ◆) = \varepsilon$

At this point, we also obtain the *G*-based top-down transducer $\Pi = (\Sigma, R, o)$; for instance, from $S◆ \rightarrow SS◆$ and $o(S◆ \rightarrow SS◆) = 1$, we obtain Π's rule of the form $S◆ \rightarrow SS◆1$ to put it in terms of Conventions 3.11. Figure 3.9 summarizes the definitions of *G*, *M*, *o*, and Π.

G	*M*	*o*	Π
1: $S \rightarrow SS$	$S◆ \rightarrow SS◆$	$o(S◆ \rightarrow SS◆) = 1$	$S◆ \rightarrow SS◆1$
2: $S \rightarrow aSb$	$S◆ \rightarrow bSa◆$	$o(S◆ \rightarrow bSa◆) = 2$	$S◆ \rightarrow bSa◆2$
3: $S \rightarrow \varepsilon$	$S◆ \rightarrow \varepsilon◆$	$o(S◆ \rightarrow \varepsilon◆) = 3$	$S◆ \rightarrow \varepsilon◆3$
	$a◆a \rightarrow ◆$	$o(a◆a \rightarrow ◆) = \varepsilon$	
	$b◆b \rightarrow ◆$	$o(b◆b \rightarrow ◆) = \varepsilon$	

Figure 3.9 *Top-down parsing models.*

Π translates *abaabb*, representing ***begin end begin begin end end***, as

▶S◆$abaabb$◀	$\Rightarrow$	▶SS◆$abaabb$◀1	[$S◆ \rightarrow SS◆1$]
	$\Rightarrow$	▶$SbSa$◆$abaabb$◀12	[$S◆ \rightarrow bSa◆2$]
	$\Rightarrow$	▶SbS◆$baabb$◀12	[$a◆a \rightarrow ◆$]
	$\Rightarrow$	▶Sb◆$baabb$◀123	[$S◆ \rightarrow ◆3$]
	$\Rightarrow$	▶S◆$aabb$◀123	[$b◆b \rightarrow ◆$]
	$\Rightarrow$	▶bSa◆$aabb$◀1232	[$S◆ \rightarrow bSa◆2$]
	$\Rightarrow$	▶bS◆abb◀1232	[$a◆a \rightarrow ◆$]
	$\Rightarrow$	▶$bbSa$◆abb◀12322	[$S◆ \rightarrow bSa◆2$]
	$\Rightarrow$	▶bbS◆bb◀12322	[$a◆a \rightarrow ◆$]
	$\Rightarrow$	▶bb◆bb◀123223	[$S◆ \rightarrow ◆3$]

$$\Rightarrow \blacktriangleright b\blacklozenge b\blacktriangleleft 123223 \qquad [b\blacklozenge b \to \blacklozenge]$$
$$\Rightarrow \blacktriangleright\blacklozenge\blacktriangleleft 123223 \qquad [b\blacklozenge b \to \blacklozenge]$$

That is, $\blacktriangleright S\blacklozenge abaabb\blacktriangleleft \Rightarrow^* \blacktriangleright\blacklozenge\blacktriangleleft 123223$ in Π, so (*abaabb*, 123223) ∈ τ(Π). Notice that *G* makes $S\ {}_{lm}\!\Rightarrow abaabb$ [123223], so 123223 represents the left parse of *abaabb* in *G*.

■

Recursive-Descent Parser

Algorithm 3.12 constructs a *G*-based top-down parser $M = ({}_M\Sigma, {}_MR)$ as a pushdown automaton, which requires, strictly speaking, an implementation of a pushdown list. There exists, however, a top-down parsing method, called *recursive descent*, which frees us from this implementation. Indeed, the pushdown list is invisible in this method because it is actually realized by the pushdown used to support recursion in the programming language in which we write the recursive-descent parser. As this method does not require an explicit manipulation with the pushdown list, it comes as no surprise that it is extremely popular in practice. Therefore, in its next description, we pay a special attention to its implementation.

Goal. Recursive-descent parser based upon a grammar *G*.

Gist. Consider a programming language defined by a grammar *G*. Let $w = t_1\ldots t_jt_{j+1}\ldots t_m$ be an input string or, more pragmatically speaking, the tokenized version of a source program. Like any top-down parser, *G*-based recursive-descent parser, symbolically denoted as *G-rd-parser,* simulates the construction of a parse tree with its frontier equal to *w* by using *G*'s rules so it starts from the root and works down to the leaves, reading *w* in a left-to-right way. In terms of derivations, *G-rd-parser* looks for the leftmost derivation of *w*. To find it, for each nonterminal, *A*, *G-rd-parser* has a Boolean function, *rd-function A*, which simulates rewriting the leftmost nonterminal *A*. More specifically, with the right-hand side of an *A*-rule, $A \to X_1\ldots X_iX_{i+1}\ldots X_n$, *rd-function A* proceeds from X_1 to X_n. Assume that *rd-function A* currently works with X_i and that t_j is the input token. At this point, depending on whether X_i is a terminal or a nonterminal, this function works as follows:

- If X_i is a terminal, *rd-function A* matches X_i against t_j. If $X_i = t_j$, it reads t_j and proceeds to X_{i+1} and t_{j+1}. If $X_i \neq t_j$, a syntactical error occurs, which the parser has to handle.

- If X_i is a nonterminal, *G-rd-parser* calls *rd-function* X_i, which simulates rewriting X_i according to a rule in *G*.

G-rd-parser starts the parsing process from *rd-function S*, which corresponds to *G*'s start symbol, and it ends when it eventually returns to this function after completely reading *w*. If during this entire process no syntactical error occurs, *G-rd-parser* has found the leftmost derivation of *w*, which thus represents a syntactically well-formed program written in the source language; otherwise, *w* is syntactically incorrect.

As this method does not require explicitly manipulating a pushdown list, it is very popular in practice; in particular, it is suitable for parsing declarations and general program flow as the next case study illustrates.

Case Study 8/35 *Recursive-Descent Parser.* Recursive-descent parsing is particularly suitable for the syntax analysis of declarations and general program flow as this case study illustrates in terms of FUN. Consider the FUN declarations generated by the next grammar ${}_{decl}G$, where ⟨declaration part⟩ is its start symbol:

⟨declaration part⟩ → ***declaration*** ⟨declaration list⟩
⟨declaration list⟩ → ⟨declaration⟩; ⟨declaration list⟩
⟨declaration list⟩ → ⟨declaration⟩
⟨declaration⟩ → ***integer*** ⟨variable list⟩
⟨declaration⟩ → ***real*** ⟨variable list⟩
⟨declaration⟩ → ***label*** ⟨label list⟩
⟨variable list⟩ → *i*, ⟨variable list⟩
⟨variable list⟩ → *i*

Next, we construct $_{decl}G$*-rd-parser,* which consists of the Boolean functions corresponding to the nonterminals in $_{decl}G$. Throughout this construction, we obtain the tokens by programming function **INPUT-SYMBOL**, described next.

Definition 3.14. INPUT-SYMBOL is a lexical-analysis programming function that returns the current input symbol or, in other words, the current token when called. After a call of this function, the input string is advanced to the next symbol.
■

First, consider the start symbol ⟨declaration part⟩, and the rule with ⟨declaration part⟩ on its left-hand side:

⟨declaration part⟩ → ***declaration*** ⟨declaration list⟩

This rule says that ⟨declaration part⟩ derives a string that consists of ***declaration*** followed by a string derived from ⟨declaration list⟩. Formally,

$$\langle\text{declaration part}\rangle\ {}_{lm}\Rightarrow^{*}\ \textbf{\textit{declaration}}\ x$$

where ⟨declaration list⟩ $_{lm}\Rightarrow^{*} x$. The next *rd-function* ⟨declaration part⟩ simulates this derivation.

```
function ⟨declaration part⟩ : boolean;
begin
    ⟨declaration part⟩ := false;
    if INPUT-SYMBOL = 'declaration' then
        if ⟨declaration list⟩ then
            ⟨declaration part⟩ := true;
end
```

Consider ⟨declaration list⟩ and the two rules with ⟨declaration list⟩ on its left-hand side:

⟨declaration list⟩ → ⟨declaration⟩; ⟨declaration list⟩
⟨declaration list⟩ → ⟨declaration⟩

These two rules say that ⟨declaration list⟩ derives a string that consists of some substrings separated by semicolons so that each of these substrings is derived from ⟨declaration⟩. That is,

$$\langle\text{declaration list}\rangle\ {}_{lm}\Rightarrow^{*}\ y_1;\ \ldots;\ y_n$$

where ⟨declaration⟩ $_{lm}\Rightarrow^{*} y_i$, $1 \le i \le n$, for some $n \ge 1$ ($n = 1$ means ⟨declaration list⟩ $_{lm}\Rightarrow^{*} d_1$). The following *rd-function* ⟨declaration list⟩ simulates this leftmost derivations.

```
function ⟨declaration list⟩ : boolean;
    begin
        ⟨declaration list⟩ := false;
        if ⟨declaration⟩ then
            if INPUT-SYMBOL = ';' then ⟨declaration list⟩ := ⟨declaration list⟩
            else ⟨declaration list⟩ := true;
    end
end
```

With the left-hand side equal to ⟨declaration⟩, there exist these two rules:

⟨declaration⟩ $\rightarrow$ ***integer*** ⟨variable list⟩
⟨declaration⟩ $\rightarrow$ ***real*** ⟨variable list⟩

According to them, ⟨declaration⟩ derives a string that starts with ***integer*** or ***real*** behind which there is a string derived from ⟨variable list⟩. That is,

$$\langle\text{declaration}\rangle\ {}_{lm}\Rightarrow^{*} ay$$

where $a \in \{$***integer***, ***real***$\}$ and ⟨variable list⟩ ${}_{lm}\Rightarrow^{*} y$. The following *rd-function* ⟨declaration⟩ simulates the above leftmost derivation.

```
function ⟨declaration⟩ : boolean;
begin
    ⟨declaration⟩ := false;
    if INPUT-SYMBOL in [real, integer] then
        if ⟨variable list⟩ then
            ⟨declaration⟩ := true;
end
```

Consider the ⟨variable list⟩-rules:

⟨variable list⟩ $\rightarrow i$, ⟨variable list⟩
⟨variable list⟩ $\rightarrow i$

Thus, ⟨variable list⟩ derives a string, *w*, that consists of *i* followed by zero or more *i*s separated by commas; in other words,

$$\langle\text{variable list}\rangle\ {}_{lm}\Rightarrow^{*} i, \ldots, i$$

The next *rd-function* ⟨variable list⟩ simulates this leftmost derivations.

```
function ⟨variable list⟩ : boolean;
begin
    ⟨variable list⟩ := false;
    if INPUT-SYMBOL = 'i' then
        if INPUT-SYMBOL = ',' then ⟨variable list⟩ := ⟨variable list⟩
        else ⟨variable list⟩ := true;
end.
```

At this point, we have completed the construction of ${}_{decl}G$-*rd-parser*, consisting of the previous four functions corresponding to ${}_{decl}G$'s nonterminals. This parser starts from function ⟨declaration part⟩ because ⟨declaration part⟩ is ${}_{decl}G$'s start symbol, and it ends in this function after completely reading the input string of symbols or, to put in terms of lexical analysis, tokens. If during this entire process no syntactical error occurs, the input string is syntactically correct.

■

Elimination of Left Recursion

Under some undesirable grammatical conditions, the recursive-descent parsing fails to work properly. Specifically, suppose that A is a nonterminal that derives a string that starts again with A. Consequently, *rd-function* A may call itself endlessly and, therefore, the recursive decent parser does not work at this point.

More precisely, consider a grammar $G = ({}_G\Sigma, {}_GR)$. A nonterminal $A \in {}_GN$ is *immediately left-recursive* if there exists a rule of the form $A \rightarrow Aw$, for some $w \in {}_G\Sigma^*$. G is *immediately left-recursive* if ${}_GN$ contains an immediately left recursive nonterminal. Observe that $A \rightarrow Aw \in {}_GR$ implies $A \Rightarrow Aw$ in G. As a result, no recursive descent parser can be based on an immediately left-recursive grammar because for every immediately left-recursive nonterminal, A, its corresponding *rd-function* A would call itself over and over again. Fortunately, there exists a transformation converting any immediately left-recursive grammar to an equivalent grammar that is not immediately left-recursive.

Goal. Convert any immediately left-recursive grammar $G = ({}_G\Sigma, {}_GR)$ to an equivalent grammar $H = ({}_H\Sigma, {}_HR)$ that is not immediately left-recursive.

Gist. Consider an immediately left-recursive grammar $G = ({}_G\Sigma, {}_GR)$. Add every rule that is not immediately left recursive to ${}_HR$. For every immediately left-recursive symbol introduce a new nonterminal A_{new} to ${}_HN$ and for any pair of rules, $A \rightarrow Aw$ and $A \rightarrow u$, where $A \notin$ **prefix**(u), introduce $A \rightarrow uA_{new}$, $A_{new} \rightarrow wA_{new}$ and $A_{new} \rightarrow \varepsilon$ to ${}_HR$. Repeat this transformation until nothing can be added to ${}_HN$ or ${}_HR$ in this way.

Algorithm 3.15 ***Elimination of Immediate Left Recursion.***

Input • an immediately left-recursive grammar $G = ({}_G\Sigma, {}_GR)$.

Output • an equivalent grammar $H = ({}_H\Sigma, {}_HR)$ that is not immediately left-recursive.

Method

```
begin
  set $_HN$ to $_GN$;
  for every nonterminal $A \in {}_GN$ that is not immediately left-recursive do
    add each $A$-rule from $_GR$ to $_HR$;
  repeat
    for every immediately left-recursive nonterminal $A \in {}_GN$ do
    begin
      add a new nonterminal, $A_{new}$, to $_HN$ ;
      extend $_HR$ by $\{A \rightarrow \textbf{\textit{rhs}}(p)A_{new} | \ p \in {}_GR, \textbf{\textit{lhs}}(p) = A, A \neq \textbf{symbol}(\textbf{\textit{rhs}}(p), 1)\}$
      $\cup \{A_{new} \rightarrow \textbf{suffix}(\textbf{\textit{rhs}}(r), |\textbf{\textit{rhs}}(r)| - 1)A_{new} | \ r \in {}_GR, \textbf{\textit{lhs}}(r) = A, A = \textbf{symbol}(\textbf{\textit{rhs}}(r), 1)\}$
      $\cup \{A_{new} \rightarrow \varepsilon\}$
    end
  until no change
```

end.

Case Study 9/35 ***Elimination of Immediate Left Recursion.*** Consider the grammar ${}_{cond}H$ (see Figure 3.7), defined as

$E \rightarrow E \vee T$
$E \rightarrow T$
$T \rightarrow T \wedge F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow i$

Observe that E and T are immediately left-recursive, so ${}_{cond}H$ is an immediately left-recursive grammar. Algorithm 3.15 converts this grammar to the following equivalent grammar, which is not immediately left-recursive:

$E \rightarrow TA$
$A \rightarrow \vee TA$
$A \rightarrow \varepsilon$
$T \rightarrow FB$
$B \rightarrow \wedge FB$
$B \rightarrow \varepsilon$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow i$

where A and B are new nonterminals; the details of this conversion are left as an exercise. As obvious, the resulting grammar is not immediately left-recursive, so the principal goal is achieved. Notice, however, that this grammatical transformation has some side effects. From a quite intuitive viewpoint, compared to the original grammar, the transformed grammar specifies the FUN arithmetic expressions less clearly. Besides, with several new nonterminals and rules, the transformed grammar has increased in size, which complicates the construction of the parser based on this grammar. Finally, as opposed to the original grammar, the transformed grammar contains several ε-rules, which are disallowed by some parsing methods (see Section 5.1).

■

The immediate left recursion discussed above is a special case of general left recursion, which represents a more hidden trap to which the recursive descent parser may face. More rigorously, in a grammar $G = ({}_G\Sigma, {}_GR)$, a nonterminal $A \in {}_GN$ is *left-recursive* if $A \Rightarrow^+ Aw$ in G, for some $w \in {}_G\Sigma^*$. G is *left-recursive* if ${}_GN$ contains a left recursive nonterminal. Even this general left recursion can be always eliminated. This kind of left recursion rarely occurs in reality, however, so we discuss its elimination as an exercise.

Bottom-Up Parsing

For a grammar, $G = ({}_G\Sigma, {}_GR)$, its *G-based bottom-up parser* is formalized by a pushdown automaton, $M = ({}_M\Sigma, {}_MR)$, like the G-based top-down parser. However, starting from the frontiers, M builds up a parse tree so it proceeds towards its root. In terms of derivations, M actually models a rightmost derivation in reverse. That is, ${}_MR$ contains a rule corresponding to each rule in ${}_GR$, and by this rule, M models a rightmost derivation step made by the grammatical rule in G so that on its pushdown top, M reduces the right-hand side of this rule to its left-hand side. In addition, ${}_MR$ has auxiliary rules, most of which only shift the input symbols onto the pushdown. By analogy with the top-down parser, the *G-M parsing correspondence* between G's rightmost derivations and M's computation is expressed by a function o from ${}_MR^*$ to ${}_GR^*$ mapping all the reduction rules in M to their corresponding grammatical rules while erasing the other auxiliary rules. A *right parse* of $w \in {}_G\Sigma^*$ is defined as *reversal*(π), where $\pi \in {}_GR^*$ satisfies ${}_GS\ {}_{rm}{\Rightarrow}^*\ w\ [\pi]$ in G. M models ${}_GS\ {}_{rm}{\Rightarrow}^*\ w\ [\pi]$ in reverse by making a sequence of moves according to $\rho \in {}_MR^*$ so that o maps ρ precisely to *reversal*(π). Observe that the *G-based bottom-up transducer* $\Pi = ({}_M\Sigma, {}_MR, o)$ underlain by $M = ({}_M\Sigma, {}_MR)$ defines

$$\tau(\Pi) = \{(w, \mu) \mid w \in L(G) \text{ and } \mu \text{ is a right parse of } x \text{ in } G\}$$

Goal. Given a grammar, $G = ({}_G\Sigma, {}_GR)$, construct the G-based bottom-up parser, M.

Gist. Consider ${}_GS\ {}_{rm}{\Rightarrow}^*\ w$ in G, where $w \in \Delta^*$. With w as its input, M works so it simulates this derivation in reverse, proceeding from w towards ${}_GS$. More specifically, express this derivation as ${}_GS\ {}_{rm}{\Rightarrow}^*\ zv\ {}_{rm}{\Rightarrow}^*\ tv$ in G, where $t, v \in \Delta^*$, $w = tv$, and $z \in \Sigma^*$. After reading t and making a sequence of moves corresponding to $zv\ {}_{rm}{\Rightarrow}^*\ tv$, M uses its pushdown to record z and contains v as the remaining input to read. In brief, it occurs in the configuration $\blacktriangleright z \blacklozenge v \blacktriangleleft$. To explain M's next move that models G's rightmost derivation step, express ${}_GS\ {}_{rm}{\Rightarrow}^*\ zv\ {}_{rm}{\Rightarrow}^*\ tv$ in greater detail as

$$\begin{array}{lll} {}_GS & {}_{rm}{\Rightarrow}^* & yAv \\ & {}_{rm}{\Rightarrow} & yxv \quad [A \to x] \\ & {}_{rm}{\Rightarrow}^* & tv \end{array}$$

where $yx = z$ and $A \to x \in {}_GR$. From $\blacktriangleright {}_MSyx \blacklozenge v \blacktriangleleft$, which equals $\blacktriangleright {}_MSz \blacklozenge v \blacktriangleleft$, M simulates $yAv\ {}_{rm}{\Rightarrow}\ yxv\ [A \to x]$ in reverse as

$$\blacktriangleright {}_MSyx \blacklozenge v \blacktriangleleft \ \Rightarrow\ \blacktriangleright {}_MSyA \blacklozenge v \blacktriangleleft$$

In addition, whenever needed, M can shift the first symbol of the remaining input onto the pushdown. In this way, step by step, M models ${}_GS\ {}_{rm}{\Rightarrow}^*\ zv\ {}_{rm}{\Rightarrow}^*\ tv$ in G until it reaches the configuration $\blacktriangleright {}_MS{}_GS \blacklozenge \blacktriangleleft$. To reach $\blacktriangleright \blacklozenge \blacktriangleleft$ and complete the acceptance of w, we add ${}_MS{}_GS\blacklozenge \to \blacklozenge$ to ${}_MR$.

Algorithm 3.16 ***Bottom-Up Parser for a Grammar.***

Input • a grammar $G = ({}_G\Sigma, {}_GR)$.

Output • a *G-based bottom-up parser* represented by a pushdown automaton $M = ({}_M\Sigma, {}_MR)$ that accepts $L(G)$.

Method

```
begin
    $_M\Sigma := {}_G\Sigma \cup \{_MS\}$ with $_M\Gamma = {}_GN \cup \{_MS\}$, $_M\Delta = {}_G\Delta$, $_MS \notin {}_G\Sigma$ is a new symbol;
    $_MR := \emptyset$;
    for each $A \to x \in {}_GR$ do
        add $x\blacklozenge \to A\blacklozenge$ to $_MR$;          {reductions}
    for each $a \in {}_G\Delta$ do
```

```
        add $◆a \rightarrow a◆$ to ${}_MR$;          {shifts}
    add ${}_MS_GS◆ \rightarrow ◆$ to ${}_MR$;
end.
```

Observe the G-based bottom-up parser $M = ({}_M\Sigma, {}_MR)$ constructed by Algorithm 3.16 satisfies

$$A \rightarrow x \in {}_GR \text{ if and only if } x◆ \rightarrow A◆ \in {}_MR$$

Introduce the *M-G parsing correspondence* as a mapping from ${}_MR^*$ to ${}_GR^*$ defined as $o(x◆ \rightarrow A◆) = A \rightarrow x$, for every $A \rightarrow x \in {}_GR$, $o(◆a \rightarrow a◆) = \varepsilon$, for each $a \in {}_G\Delta$, and $o({}_MS_GS◆ \rightarrow ◆) = \varepsilon$. Observe that for every $\pi \in {}_GR^*$ such that $u\ {}_{rm}\Rightarrow^* vy\ [\pi]$, $u \in {}_G\Sigma^*{}_GN$, $v \in {}_G\Sigma^*{}_GN \cup \{\varepsilon\}$, $y \in {}_G\Delta^*$, there exists precisely one $\rho \in {}_GR^*$ such that $o(reversal(\rho)) = \pi$ and

$$▶_MSv◆y◀ \Rightarrow^* ▶_MSu◆◀\ [\rho] \text{ in } M \text{ if and only if } u\ {}_{rm}\Rightarrow^* vy\ [\pi] \text{ in } G$$

Notice that for $u = {}_GS$ and $v = \varepsilon$, this equivalence actually says

$$▶_MS◆y◀ \Rightarrow^* ▶_MS_GS◆◀ \text{ in } M \text{ if and only if } {}_GS\ {}_{rm}\Rightarrow^* y \text{ in } G$$

As ${}_MS_GS◆ \rightarrow ◆ \in {}_MR$,

$$▶_MS◆y◀ \Rightarrow^* ▶◆◀ \text{ in } M \text{ if and only if } {}_GS\ {}_{rm}\Rightarrow^* y \text{ in } G,$$

so $L(M) = L(G)$. The *G-based top-down transducer* is defined as the pushdown transducer $\Pi = ({}_M\Sigma, {}_MR, o)$ underlain by M and its translation $\tau(\Pi)$ satisfies

$$\tau(\Pi) = \{(x, \mu) | \ x \in L(G) \text{ and } \mu \text{ is a right parse of } x \text{ in } G\}$$

Lemma 3.17. Let $G = ({}_G\Sigma, {}_GR)$ be a grammar. With G as its input, Algorithm 3.16 correctly constructs the G-based bottom-up parser represented by a pushdown automaton $M = ({}_M\Sigma, {}_MR)$ that accepts $L(G)$.

Proof. To demonstrate that Algorithm 3.16 correctly constructs M, we need to prove that $L(G) = L(M)$. To make this proof, we first establish Claims A and B.

Claim A. Let

$$\begin{aligned} {}_GS\ &{}_{rm}\Rightarrow^*\ xv \\ &{}_{rm}\Rightarrow^*\ uv \quad [\pi] \end{aligned}$$

in G, where $v, u \in {}_G\Delta^*$, $x \in {}_G\Sigma^*$, and $\pi \in {}_GR^*$. Then, $▶_MS◆uv◀ \Rightarrow^* ▶_MSx◆v◀$ in M.

Proof of Claim A by induction on $|\pi| \geq 0$.

Basis. Let $|\pi| = 0$. That is, $S\ {}_{rm}\Rightarrow^* xv\ {}_{rm}\Rightarrow^0 uv$ in G, so $u = x$. Observe that $▶_MS◆uv◀ \Rightarrow^{|u|} ▶_MSu◆v◀$ in M so M shifts u from the input tape onto the pushdown by $|u|$ applications of the rules that have the form $◆a \rightarrow a◆$, where $a \in {}_G\Delta$.

Induction hypothesis. Suppose that the claim holds for each $\pi \in {}_GR^*$ with $|\pi| \leq i$, for some $i \geq 0$.

Induction step. Consider

$$\begin{array}{lll} S & {}_{rm}\Rightarrow^* & xv \\ & {}_{rm}\Rightarrow^* & uv \quad [p\pi] \end{array}$$

in G, where $\pi \in {}_GR^*$, $|\pi| = i$, and $p \in {}_GR$. Express xv ${}_{rm}\Rightarrow^*$ uv $[p\pi]$ as

$$\begin{array}{llll} xv & {}_{rm}\Rightarrow & y\mathbf{rhs}(p)v & [p] \\ & {}_{rm}\Rightarrow^* & uv & [\pi] \end{array}$$

where $x = y\mathbf{lhs}(p)$. By Algorithm 3.16, $p \in {}_GR$ implies $\mathbf{rhs}(p)\blacklozenge \rightarrow \mathbf{lhs}(p)\blacklozenge \in {}_MR$. In the rest of this proof, we distinguish these two cases—(1) $y\mathbf{rhs}(p) \in {}_G\Delta^*$ and (2) $y\mathbf{rhs}(p) \notin {}_G\Delta^*$.

(1) Let $y\mathbf{rhs}(p) \in {}_G\Delta^*$. Then, $y\mathbf{rhs}(p) = u$. Construct $\blacktriangleright_M S\blacklozenge uv\blacktriangleleft \Rightarrow^{|u|} \blacktriangleright_M Sy\mathbf{rhs}(p)\blacklozenge v\blacktriangleleft$ in M by shifting u onto the pushdown, so

$$\begin{array}{llll} \blacktriangleright_M S\blacklozenge uv\blacktriangleleft & \Rightarrow^{|u|} & \blacktriangleright_M Sy\mathbf{rhs}(p)\blacklozenge v\blacktriangleleft & \\ & \Rightarrow & \blacktriangleright_M Sy\mathbf{lhs}(p)\blacklozenge v\blacktriangleleft & [\mathbf{rhs}(p)\blacklozenge \rightarrow \mathbf{lhs}(p)\blacklozenge] \end{array}$$

As $x = y\mathbf{lhs}(p)$, we have just proved $\blacktriangleright_M S\blacklozenge uv\blacktriangleleft \Rightarrow^* \blacktriangleright_M Sx\blacklozenge v\blacktriangleleft$ in M.

(2) Let $y\mathbf{rhs}(p) \notin {}_G\Delta^*$. Express $y\mathbf{rhs}(p)$ as $y\mathbf{rhs}(p) = zBt$, where $t \in {}_G\Delta^*$, $z \in {}_G\Sigma^*$, and $B \in {}_GN$, so B is the rightmost nonterminal appearing in $y\mathbf{rhs}(p)$. Consider

$$\begin{array}{llll} S & {}_{rm}\Rightarrow^* & xv & \\ & {}_{rm}\Rightarrow & zBtv & [p] \\ & {}_{rm}\Rightarrow^* & uv & [\pi] \end{array}$$

Since $|\pi| = i$, $\blacktriangleright_M S\blacklozenge uv\blacktriangleleft \Rightarrow^* \blacktriangleright_M SzB\blacklozenge tv\blacktriangleleft$ by the inductive hypothesis. By shifting t onto the pushdown, we obtain $\blacktriangleright_M SzB\blacklozenge tv\blacktriangleleft \Rightarrow^* \blacktriangleright_M SzBt\blacklozenge v\blacktriangleleft$. As $y\mathbf{rhs}(p) = zBt$, we have

$$\begin{array}{llll} \blacktriangleright_M S\blacklozenge uv\blacktriangleleft & \Rightarrow^* & \blacktriangleright_M Sy\mathbf{rhs}(p)\blacklozenge v\blacktriangleleft & \\ & \Rightarrow & \blacktriangleright_M Sy\mathbf{lhs}(p)\blacklozenge v\blacktriangleleft & [\mathbf{rhs}(p)\blacklozenge \rightarrow \mathbf{lhs}(p)\blacklozenge] \end{array}$$

Therefore, $\blacktriangleright_M S\blacklozenge uv\blacktriangleleft \Rightarrow^* \blacktriangleright_M Sx\blacklozenge v\blacktriangleleft$ in M because $x = y\mathbf{lhs}(p)$, so the proof of Claim A is completed.

Claim B. Let $\blacktriangleright_M S\blacklozenge uv\blacktriangleleft \Rightarrow^* \blacktriangleright_M Sx\blacklozenge v\blacktriangleleft$ $[\rho]$ in M, where $u, v \in {}_M\Delta^*$, $x \in {}_G\Sigma^*$, and $\rho \in ({}_MR - \{{}_MS_GS \blacklozenge \rightarrow \blacklozenge\})^*$. Then, xv ${}_{rm}\Rightarrow^*$ uv in G.

Proof of Claim B by induction on $|\rho| \geq 0$.

Basis. Let $|\rho| = 0$, so $u = x = \varepsilon$ and $\blacktriangleright_M S\blacklozenge v\blacktriangleleft \Rightarrow^0 \blacktriangleright_M S\blacklozenge v\blacktriangleleft$ in M. Clearly, v ${}_{rm}\Rightarrow^0$ v in G.

Induction hypothesis. Suppose that the claim holds for each $\rho \in ({}_MR - \{{}_MS_GS \blacklozenge \rightarrow \blacklozenge\})^*$ with $|\rho| \leq i$, for some $i \geq 0$.

Induction step. Consider any sequence of $i + 1$ moves. Suppose that this sequence is of the form $\blacktriangleright_M S\blacklozenge uv\blacktriangleleft \Rightarrow^* \blacktriangleright_M Sx\blacklozenge v\blacktriangleleft$ $[\rho r]$ in M, where $u, v \in {}_M\Delta^*$, $x \in {}_G\Sigma^*$, $\rho \in ({}_MR - \{{}_MS_GS\blacklozenge \rightarrow \blacklozenge\})^*$, $|\rho| = i$, and $r \in {}_MR$. Express $\blacktriangleright_M S\blacklozenge uv\blacktriangleleft \Rightarrow^* \blacktriangleright_M Sx\blacklozenge v\blacktriangleleft$ $[\rho r]$ as

$$\begin{array}{lll} \blacktriangleright_M S\blacklozenge uv\blacktriangleleft & \Rightarrow^* \ \blacktriangleright_M Sy\blacklozenge t\blacktriangleleft & [\rho] \\ & {}_{rm}\Rightarrow \ \blacktriangleright_M Sx\blacklozenge v\blacktriangleleft & [r] \end{array}$$

where $y \in {}_G\Sigma^*$ and $t \in {}_M\Delta^*$. As $r \neq {}_MS\blacklozenge \to \blacklozenge$, either $\boldsymbol{rhs}(r) = A\blacklozenge$ with $A \in {}_GN$ or $\boldsymbol{rhs}(r) = a\blacklozenge$ with $a \in {}_M\Delta$, and we next distinguish these two cases.

(1) Let $\boldsymbol{rhs}(r) = A\blacklozenge$ with $A \in {}_GN$, so $t = v$, r is of the form $z\blacklozenge \to A\blacklozenge$, $x = hA$, and $y = hz$, for some $A \to z \in {}_GR$ and $h \in {}_G\Sigma^*$. By using $A \to z \in {}_GR$, G makes $hAv \ {}_{rm}\Rightarrow hzv$. By the induction hypothesis, $yv \Rightarrow^* uv$ in G. Thus, $xv \ {}_{rm}\Rightarrow^* uv$ in G.

(2) Let $\boldsymbol{rhs}(r) = a\blacklozenge$ with $a \in {}_M\Delta$, so $t = av$, r is of the form $\blacklozenge a \to a\blacklozenge$, $x = ya$. Thus, $xv = yt$, so $yv \Rightarrow^* uv$ in G by the induction hypothesis, which completes the proof of Claim B.

Consider Claim A for $v = \varepsilon$ and $x = {}_GS$. At this point, for all $u \in {}_G\Delta^*$, ${}_GS \ {}_{rm}\Rightarrow^* u$ in G implies $\blacktriangleright_M S\blacklozenge u\blacktriangleleft \Rightarrow^* \blacktriangleright_M S_G S\blacklozenge\blacktriangleleft$ in M. By using ${}_MS_GS\blacklozenge \to \blacklozenge$, M makes $\blacktriangleright_M S_G S\blacklozenge\blacktriangleleft \Rightarrow^* \blacktriangleright\blacklozenge\blacktriangleleft$ in M. Hence, $L(G) \subseteq L(M)$.

Consider Claim B for $v = \varepsilon$ and $x = {}_GS$. Under this consideration, if $\blacktriangleright_M S\blacklozenge u\blacktriangleleft \Rightarrow^* \blacktriangleright_M S_G S\blacklozenge\blacktriangleleft$ $[\rho]$ in M, then ${}_GS \ {}_{rm}\Rightarrow^* u$ in G, for all $u \in {}_G\Delta^*$. During any acceptance of a string, $u \in {}_G\Delta^*$, M uses ${}_MS_GS\blacklozenge \to \blacklozenge$ precisely once to make the very last step in order to remove ${}_MS_GS$ and reach the configuration $\blacktriangleright\blacklozenge\blacktriangleleft$. Indeed, observe that any earlier application of this rule implies that subsequently M can never completely read u and, simultaneously, empty the pushdown; the details of this observation is left as an exercise. Thus, $\blacktriangleright_M S\blacklozenge u\blacktriangleleft \Rightarrow^* \blacktriangleright\blacklozenge\blacktriangleleft$ $[\rho]$ in M implies ${}_GS \ {}_{rm}\Rightarrow^* u$ in G, so $L(M) \subseteq L(G)$.

Consequently, $L(G) = L(M)$ because $L(G) \subseteq L(M)$ and $L(M) \subseteq L(G)$, so this lemma holds true.

■

Case Study 10/35 *Bottom-Up Parser.* Consider the grammar ${}_{expr}G$ (see Figure 3.16), defined as

1: $E \to E + T$
2: $E \to E - T$
3: $E \to T$
4: $T \to T * F$
5: $T \to T / F$
6: $T \to F$
7: $F \to (E)$
8: $F \to i$

Algorithm 3.16 produces the ${}_{expr}G$-based bottom-up parser $M = ({}_M\Sigma, {}_MR)$ as follows. Its first loop constructs these reduction rules

$E + T\blacklozenge \to E\blacklozenge$
$E - T\blacklozenge \to E\blacklozenge$
$T\blacklozenge \to E\blacklozenge$
$T * F\blacklozenge \to T\blacklozenge$
$T / F\blacklozenge \to T\blacklozenge$
$F\blacklozenge \to T\blacklozenge$
$(E)\blacklozenge \to F\blacklozenge$
$i\blacklozenge \to F\blacklozenge$

For each terminal $a \in \{+, -, *, /, (,), i\}$, the second loop introduces these shift rules

$$\blacklozenge a \rightarrow a\blacklozenge$$

to ${}_MR$. Finally, this algorithm adds $\blacktriangleright SE\blacklozenge\blacktriangleleft \rightarrow \blacklozenge$ to ${}_MR$, where S is M's new start pushdown symbol. The M-G parsing correspondence o maps M's reduction rules to the corresponding grammatical rules, represented by their labels in Figure 3.10, while mapping all the other rules of M to ε.

***Rule** r **in** M*	***Rule** o(r) **in** G*
$E + T\blacklozenge \rightarrow E\blacklozenge$	1
$E - T\blacklozenge \rightarrow E\blacklozenge$	2
$T\blacklozenge \rightarrow E\blacklozenge$	3
$T * F\blacklozenge \rightarrow T\blacklozenge$	4
$T / F\blacklozenge \rightarrow T\blacklozenge$	5
$F\blacklozenge \rightarrow T\blacklozenge$	6
$(E)\blacklozenge \rightarrow F\blacklozenge$	7
$i\blacklozenge \rightarrow F\blacklozenge$	8

Figure 3.10 ***M-G parsing correspondence.***

Consider the ${}_{expr}G$-based bottom-up transducer $\Pi = ({}_M\Sigma, {}_MR, o)$ and $i{+}i{*}i \in L({}_{expr}G)$. Π translates $i{+}i{*}i$ to its right parse 86386841 as

$$\begin{array}{lll}
\blacktriangleright S\blacklozenge i{+}i{*}i\blacktriangleleft & \Rightarrow \blacktriangleright Si\blacklozenge{+}i{*}i\blacktriangleleft & [\blacklozenge i \rightarrow i\blacklozenge] \\
& \Rightarrow \blacktriangleright SF\blacklozenge{+}i{*}i\blacktriangleleft 8 & [i\blacklozenge \rightarrow F\blacklozenge 8] \\
& \Rightarrow \blacktriangleright ST\blacklozenge{+}i{*}i\blacktriangleleft 86 & [F\blacklozenge \rightarrow T\blacklozenge 6] \\
& \Rightarrow \blacktriangleright SE\blacklozenge{+}i{*}i\blacktriangleleft 863 & [T\blacklozenge \rightarrow E\blacklozenge 3] \\
& \Rightarrow \blacktriangleright SE{+}\blacklozenge i{*}i\blacktriangleleft 863 & [\blacklozenge + \rightarrow +\blacklozenge] \\
& \Rightarrow \blacktriangleright SE{+}i\blacklozenge{*}i\blacktriangleleft 863 & [\blacklozenge i \rightarrow i\blacklozenge] \\
& \Rightarrow \blacktriangleright SE{+}F\blacklozenge{*}i\blacktriangleleft 8638 & [i\blacklozenge \rightarrow F\blacklozenge 8] \\
& \Rightarrow \blacktriangleright SE{+}T\blacklozenge{*}i\blacktriangleleft 86386 & [F\blacklozenge \rightarrow T\blacklozenge 6] \\
& \Rightarrow \blacktriangleright SE{+}T{*}\blacklozenge i\blacktriangleleft 86386 & [\blacklozenge * \rightarrow *\blacklozenge] \\
& \Rightarrow \blacktriangleright SE{+}T{*}i\blacklozenge\blacktriangleleft 86386 & [\blacklozenge i \rightarrow i\blacklozenge] \\
& \Rightarrow \blacktriangleright SE{+}T{*}F\blacklozenge\blacktriangleleft 863868 & [i\blacklozenge \rightarrow F\blacklozenge 8] \\
& \Rightarrow \blacktriangleright SE{+}T\blacklozenge\blacktriangleleft 8638684 & [T * F\blacklozenge \rightarrow T\blacklozenge 4] \\
& \Rightarrow \blacktriangleright SE\blacklozenge\blacktriangleleft 86386841 & [E + T\blacklozenge \rightarrow E\blacklozenge 1] \\
& \Rightarrow \blacktriangleright\blacklozenge\blacktriangleleft 86386841 & [SE\blacklozenge\blacktriangleleft \rightarrow \blacklozenge]
\end{array}$$

In short, $\blacktriangleright\blacklozenge i{+}i{*}i\blacktriangleleft \Rightarrow^* \blacktriangleright\blacklozenge\blacktriangleleft 86386841$ in Π. Figure 3.11 describes the relationship between the bottom-up construction of the parse tree for $i{+}i{*}i$ in G and $\blacktriangleright\blacklozenge i{+}i{*}i\blacktriangleleft \Rightarrow^* \blacktriangleright\blacklozenge\blacktriangleleft 86386841$ in Π.

Reverse 86386841 to 14868368 in order to obtain the sequence of rules according to which ${}_{expr}G$ derives $i{+}i{*}i$ in the rightmost way as follows

$$\begin{array}{lll}
\underline{E} & \Rightarrow E + \underline{T} & [1{:}\ E \rightarrow E + T] \\
& \Rightarrow E + T * \underline{F} & [4{:}\ T \rightarrow T * F] \\
& \Rightarrow E + \underline{T} * i & [8{:}\ F \rightarrow i] \\
& \Rightarrow E + \underline{F} * i & [6{:}\ T \rightarrow F] \\
& \Rightarrow \underline{E} + i * i & [8{:}\ F \rightarrow i] \\
& \Rightarrow \underline{T} + i * i & [3{:}\ E \rightarrow T] \\
& \Rightarrow \underline{F} + i * i & [6{:}\ T \rightarrow F] \\
& \Rightarrow i + i * i & [8{:}\ F \rightarrow i]
\end{array}$$

Parse Tree	***Translation in*** Π
$i+i*i$	$\blacktriangleright S\blacklozenge i+i*i\blacktriangleleft$
$F\langle i\rangle+i*i$	$\Rightarrow \blacktriangleright SF\blacklozenge +i*i\blacktriangleleft 8$
$T\langle F\langle i\rangle\rangle+i*i$	$\Rightarrow \blacktriangleright ST\blacklozenge +i*i\blacktriangleleft 86$
$E\langle T\langle F\langle i\rangle\rangle\rangle+i*i$	$\Rightarrow \blacktriangleright SE\blacklozenge +i*i\blacktriangleleft 863$
	$\Rightarrow \blacktriangleright SE+\blacklozenge i*i\blacktriangleleft 863$
	$\Rightarrow \blacktriangleright SE+i\blacklozenge *i\blacktriangleleft 863$
$E\langle T\langle F\langle i\rangle\rangle\rangle+F\langle i\rangle*i$	$\Rightarrow \blacktriangleright SE+F\blacklozenge *i\blacktriangleleft 8638$
$E\langle T\langle F\langle i\rangle\rangle\rangle+T\langle F\langle i\rangle\rangle*i$	$\Rightarrow \blacktriangleright SE+T*\blacklozenge i\blacktriangleleft 86386$
	$\Rightarrow \blacktriangleright SE+T*i\blacklozenge \blacktriangleleft 86386$
$E\langle T\langle F\langle i\rangle\rangle\rangle+T\langle F\langle i\rangle\rangle*F\langle i\rangle$	$\Rightarrow \blacktriangleright SE+T*F\blacklozenge \blacktriangleleft 863868$
$E\langle T\langle F\langle i\rangle\rangle\rangle+T\langle T\langle F\langle i\rangle\rangle*F\langle i\rangle\rangle$	$\Rightarrow \blacktriangleright SE+T\blacklozenge \blacktriangleleft 8638684$
$E\langle E\langle T\langle F\langle i\rangle\rangle\rangle+T\langle T\langle F\langle i\rangle\rangle*F\langle i\rangle\rangle\rangle$	$\Rightarrow \blacktriangleright SE\blacklozenge \blacktriangleleft 86386841$
	$\Rightarrow \blacktriangleright \blacklozenge \blacktriangleleft 86386841$

Figure 3.11 ***Bottom-Up Parsing and the Translation to the Right Parse.***

■

The parsers discussed in this section are of little importance in practice for their non-deterministic selection of applied rules. Nevertheless, they fulfill an important role in this book. Indeed, they theoretically underlie most deterministic parsing algorithms given in Chapters 4 and 5. Furthermore, they are equivalent to grammars and, in this sense, represent their automaton-based counterparts. Simply stated, they are crucial to the theoretical foundations of parsing, and that is what we discuss in the next section.

3.3 Theory

This section discusses the fundamental theoretical results relevant to parsing and their models—grammars and pushdown automata. We start with two crucial theorems regarding their power in Section 3.3.1. Section 3.3.2 demonstrates how to verify that the grammatical syntax specification is correct. Section 3.3.3 simplifies the grammars to make this specification easy and clear. Section 3.3.4 achieves two normal forms of grammars and base parsing upon them. Section 3.3.5 explains how to demonstrate that some syntax constructs cannot be specified by grammars, so neither can they be parsed by pushdown automata. Finally, Section 3.3.6 discusses some decidable problems concerning grammars and pushdown automata.

In this section, we always explain the basic ideas behind the theoretical results. However, as we are primarily interested in the syntax specification and parsing rather than purely mathematical proofs, we sometimes leave the rigorous proofs of these results as an exercise. As a matter of fact, most of the results are easy to grasp immediately from the basic ideas underlying them.

3.3.1 Power of Parsing Models

This section briefly compares the power of grammars and pushdown automata, including their deterministic versions.

Theorem 3.18. The pushdown automata and the grammars are equally powerful.

Proof. In the previous section, we have constructed two parsers for any grammar. As these parsers represent special cases of pushdown automata, from any grammar, we can construct an equivalent pushdown automaton. We can also transform any pushdown automaton to an equivalent grammar. In reality, this transformation is obviously hardly ever needed in parsing, so we leave it as an exercise.

■

The deterministic versions of pushdown automata fulfill obviously a crucial role in parsing because they are easy to implement and apply in reality. Unfortunately, they are not as powerful as their general versions, which are as powerful as grammars by the previous theorem.

Theorem 3.19. The deterministic pushdown automata are less powerful than the pushdown automata.

Proof. By Definition 2.6, every deterministic pushdown automaton is a special case of a pushdown automaton. On the other hand, consider the language $L = \{wv| \ w, v \in \{0, 1\}^*, v = reversal(w)\}$. A pushdown automaton can accept L so it first moves w onto the pushdown. Then, it reads v and, simultaneously, compares it with the pushdown contents. In greater detail, during every move of this comparison, it verifies that the current input symbol and the pushdown top symbol coincide. If they do, the automaton pops the pushdown top symbol, reads the next input symbol, and continues by making the next comparison. If in this way, the pushdown automaton empties the pushdown and, simultaneously, reads the entire input string, it accepts. We thus intuitively see that during this process, the automaton has to non-deterministically choose the move from which it starts the comparison in a non-deterministic way. As an exercise, we leave a rigorous proof that no deterministic pushdown automaton accepts L.

■

3.3.2 Verification of the Grammatical Syntax Specification

Having specified the syntax of a programming language L by a grammar G, we want to be sure that this specification is correct. As G may contain many rules, a verification of this specification requires a rigorous proof rather than an intuitive justification. In other words, we must formally demonstrate that $L(G) = L$. As obvious, this verification always depends on the specific grammar, so we can hardly give a general guide how to make it. Instead, we give a single example of this verification, which demonstrates that even for a simple abstract language and a rather small grammar, this verification may represent a non-trivial mathematical task.

Example 3.5 ***Verification of the Generated Language.*** Consider the grammar G defined as

1: $S \rightarrow aB$
2: $S \rightarrow bA$
3: $A \rightarrow a$
4: $A \rightarrow aS$
5: $A \rightarrow bAA$
6: $B \rightarrow b$
7: $B \rightarrow bS$
8: $B \rightarrow aBB$

We want to verify that G generates the language L that contains all non-empty strings that consist of an equal number of *a*s and *b*s; formally,

$$L(G) = \{w| \ w \in \{a, b\}^+ \text{ and } occur(w, a) = occur(w, b)\}$$

A verification of this kind often turns out easier if we first prove a claim that states something more than we actually need; subsequently, we obtain the desired statement as a straightforward consequence of the stronger claim. In this way, we proceed to verify the equation above.

Before we give the next claim, representing the stronger statement, recall that for a symbol a and a string w, $occur(a, w)$ denotes the number of occurrences of a in w (see Section 1.1).

Claim. For all $w \in \{a, b\}^*$, these three equivalences hold

I.	$S \Rightarrow^* w$	if and only if	$occur(a, w) = occur(b, w)$
II.	$A \Rightarrow^* w$	if and only if	$occur(a, w) = occur(b, w) + 1$
III.	$B \Rightarrow^* w$	if and only if	$occur(b, w) = occur(a, w) + 1$

Proof by induction on $|w| \geq 1$.

Basis. Let $|w| = 1$.

I. From S, G generates no sentence of length one. On the other hand, no sentence of length one satisfies $occur(a, w) = occur(b, w)$. Thus, in this case, the basis holds vacuously.

II. Examine G to see that if $A \Rightarrow^* w$ with $|w| = 1$, then $w = a$. For $w = a$, $A \Rightarrow^* w$ [3]. Therefore, II holds in this case.

III. Prove this by analogy with the proof of II.

Consequently, the basis holds.

Induction Hypothesis. Assume that there exists a positive integer $n \geq 1$ such that the claim holds for every $w \in \{a, b\}^*$ satisfying $1 \leq |w| \leq n$.

Induction Step. Let $w \in \{a, b\}^*$ with $|w| = n + 1$.

I. *Only if*. Consider any derivation of the form $S \Rightarrow^* w$ [ρ], where ρ is a sequence of rules. This derivation starts from S. As only rules 1 and 2 have S on the left-hand side, express $S \Rightarrow^* w$ [ρ] as $S \Rightarrow^* w$ [$r\pi$], where $\rho = r\pi$ and $r \in \{1, 2\}$.

- If $r = 1$, $S \Rightarrow^* w$ [1π], where 1: $S \rightarrow aB$. At this point, $w = av$, and $B \Rightarrow^* v$ [π], where $|v| = n$. By the induction hypothesis, III holds for v, so $occur(b, v) = occur(a, v) + 1$. Therefore, $occur(a, w) = occur(b, w)$.

- If $r = 2$, $S \Rightarrow^* w$ [2π], where 2: $S \rightarrow bA$. Thus, $w = bv$, and $A \Rightarrow^* v$ [π], where $|v| = n$. By the induction hypothesis, II holds for v, so $occur(a, v) = occur(b, v) + 1$. As $w = bv$, $occur(a, w) = occur(b, w)$.

If. Let $occur(a, w) = occur(b, w)$. Clearly, $w = av$ or $w = bv$, for some $v \in \{a, b\}^*$ with $|v| = n$.

- Let $w = av$. Then, $|v| = n$ and $occur(a, v) + 1 = occur(b, v)$. As $|v| = n$, by the induction hypothesis, we have $B \Rightarrow^* v$ if and only if $occur(b, v) = occur(a, v) + 1$ from III. By using 1: $S \rightarrow aB$, we obtain $S \Rightarrow aB$ [1]. Putting $S \Rightarrow aB$ and $B \Rightarrow^* v$ together, we have $S \Rightarrow aB \Rightarrow^* av$, so $S \Rightarrow^* w$ because $w = av$.

- Let $w = bv$. Then, $|v| = n$ and $occur(a, v) = occur(b, v) + 1$. By the induction hypothesis, we have $A \Rightarrow^* v$ if and only if $occur(a, v) = occur(b, v) + 1$ (see II). By 2: $S \rightarrow bA$, G makes $S \Rightarrow bA$. Thus, $S \Rightarrow bA$ and $A \Rightarrow^* v$, so $S \Rightarrow^* w$.

II. *Only if*. Consider any derivation of the form $A \Rightarrow^* w$ [ρ], where ρ is a sequence of G's rules. Express $A \Rightarrow^* w$ [ρ] as $A \Rightarrow^* w$ [$r\pi$], where $\rho = r\pi$ and $r \in \{3, 4, 5\}$ because rules 3: $A \rightarrow a$, 4: $A \rightarrow aS$, and 5: $A \rightarrow bAA$ are all the A-rules in G.

- If $r = 3$, $A \Rightarrow^* w$ [$r\pi$] is a one-step derivation $A \Rightarrow a$ [3], so $w = a$, which satisfies $occur(a, w) = occur(b, w) + 1$.

- If $r = 4$, $A \Rightarrow^* w$ [4π], where 4: $A \to aS$. Thus, $w = av$, and $S \Rightarrow^* v$ [π], where $|v| = n$. By the induction hypothesis, from I, $occur(a, v) = occur(b, v)$, so $occur(a, w) = occur(b, w) + 1$.

- If $r = 5$, $A \Rightarrow^* w$ [5π], where 5: $A \to bAA$. Thus, $w = buv$, $A \Rightarrow^* u$, $A \Rightarrow^* v$, where $|u| \leq n$, $|v| \leq n$. By the induction hypothesis, from II, $occur(a, u) = occur(b, u) + 1$ and $occur(a, v) = occur(b, v) + 1$, so $occur(a, uv) = occur(b, uv) + 2$. Notice that $occur(b, uv) = occur(b, w) - 1$ implies $occur(a, uv) - 2 = occur(b, w) - 1$. Furthermore, from $occur(a, uv) - 2 = occur(b, w) - 1$, it follows that $occur(a, uv) = occur(a, w)$, so $occur(a, w) = occur(b, w) + 1$.

If. Let $occur(a, w) = occur(b, w) + 1$. Obviously, $w = av$ or $w = bv$, for some $v \in \{a, b\}^*$ with $|v| = n$.

- Let $w = av$. At this point, $|v| = n$ and $occur(a, v) = occur(b, v)$. As $|v| = n$, by the induction hypothesis, we have $S \Rightarrow^* v$. By using 4: $A \to aS$, $A \Rightarrow aS$ [4]. Putting $A \Rightarrow aS$ and $S \Rightarrow^* v$ together, we obtain $A \Rightarrow aS \Rightarrow^* av$, so $A \Rightarrow^* w$ because $w = av$.

- Let $w = bv$. At this point, $|v| = n$ and $occur(a, v) = occur(b, v) + 2$. Express v as $v = uz$ so that $occur(a, u) = occur(b, u) + 1$ and $occur(a, z) = occur(b, z) + 1$; as an exercise, we leave a proof that $occur(a, v) = occur(b, v) + 2$ implies that v can always be expressed in this way. Since $|v| = n$, $|u| \leq n \geq |z|$. Thus, by the induction hypothesis (see II), we have $A \Rightarrow^* u$ and $A \Rightarrow^* z$. By using 5: $A \to bAA$, $A \Rightarrow bAA$ [5]. Putting $A \Rightarrow bAA$, $A \Rightarrow^* u$, and $A \Rightarrow^* z$ together, we obtain $A \Rightarrow bAA \Rightarrow^* buz$, so $A \Rightarrow^* w$ because $w = bv = buz$.

III. Prove this inductive step by analogy with the proof of the inductive step of II.

Having established this claim, we easily obtain the desired equation $L(G) = \{w| \ w \in \{a, b\}^+$ and $occur(w, a) = occur(w, b)\}$ as a consequence of Equivalence I. Indeed, this equivalence says that for all $w \in \{a, b\}^*$, $S \Rightarrow^* w$ if and only if $occur(a, w) = occur(b, w)$. Consequently, $w \in L(G)$ if and only if $occur(a, w) = occur(b, w)$. As G has no ε-rules, $\varepsilon \notin L(G)$, so $L(G) = \{w| \ w \in \{a, b\}^+$ and $occur(w, a) = occur(w, b)\}$.

■

3.3.3 Simplification of Grammars

This section simplifies grammars in order to make their specification of the programming language syntax easier and clearer.

Canonical derivations and parse trees

As already explained in Section 3.1, both canonical derivations and parse trees simplify the discussion of parsing because they free us from considering all possible derivations in a grammar. Indeed, assume that a programming language is specified by a grammar. To verify that a string w representing the tokenized version of a source program is syntactically correct, a top-down parser builds a parse tree starting from the root labeled with the grammatical start symbol and proceeding down towards the frontier equal to w. In other words, it constructs a leftmost derivation of w. On the other hand, a bottom-up parser starts from w as the frontier and works up toward the root labeled with the start symbol. In terms of canonical derivations, it builds up a rightmost derivation of w in reverse order. In either approach, we see that canonical derivations and parse trees are more than important to parsing. What we next demonstrate is that without any loss of generality, we can always represent every grammatical derivation of a sentence by a canonical derivation or a parse tree, depending on which of these representations is more appropriate under given discussion.

Theorem 3.20. Let $G = (\Sigma, R)$ be a grammar. Then, $w \in L(G)$ if and only if $S\ {}_{lm}{\Rightarrow}^* w$.

Proof.

If. This part of the proof says that $S\ {}_{lm}{\Rightarrow}^* w$ implies $w \in L(G)$, for every $w \in \Delta^*$. As $S\ {}_{lm}{\Rightarrow}^* w$ is a special case of a derivation from S to w, this implication surely holds.

Only If. To demonstrate that G can generate every $w \in L(G)$ in the leftmost way, we first prove the next claim.

Claim. For every $w \in L(G)$, $S \Rightarrow^n w$ implies $S\ {}_{lm}{\Rightarrow}^n w$, for all $n \geq 0$.

Proof by induction on $n \geq 0$.

Basis. For $n = 0$, this implication is trivial.

Induction Hypothesis. Assume that there exists an integer $n \geq 0$ such that the claim holds for all derivations of length n or less.

Induction Step. Let $S \Rightarrow^{n+1} w$ [ρ], where $w \in L(G)$, $\rho \in R^+$, and $|\rho| = n + 1$. If $S \Rightarrow^{n+1} w$ [ρ] is leftmost, the claim holds, so we assume that this derivation is not leftmost. Express $S \Rightarrow^{n+1} w$ [ρ] as

$$\begin{array}{llll} S & {}_{lm}{\Rightarrow}^* & uAv\underline{B}x & [\sigma] \\ & \Rightarrow & uAvyx & [r\colon B \rightarrow y] \\ & \Rightarrow^* & w & [\theta] \end{array}$$

where $\sigma, \theta \in R^*$, $\rho = \sigma r\theta$, $r\colon B \rightarrow y \in R$, $u \in$ **prefix**(w), $A \in N$, $u \in \Delta^*$, and $v, x, y \in \Sigma^*$. That is, $S\ {}_{lm}{\Rightarrow}^* uAv\underline{B}x$ is the longest beginning of $S \Rightarrow^{n+1} w$ performed in the leftmost way. As $A \in N$ and $w \in L(G)$, $A \notin alph(w)$, so A is surely rewritten during $uAv\underline{y}x \Rightarrow^* w$. Thus, express $S \Rightarrow^{n+1} w$ as

$$\begin{array}{llll} S & {}_{lm}{\Rightarrow}^* & uAv\underline{B}x & [\sigma] \\ & \Rightarrow & uAvyx & [r\colon B \rightarrow y] \\ & \Rightarrow^* & u\underline{A}z & [\pi] \\ & {}_{lm}{\Rightarrow} & utz & [p\colon A \rightarrow t] \\ & \Rightarrow^* & w & [o] \end{array}$$

where $\pi, o \in R^*$, $\theta = \pi po$, $p\colon A \rightarrow t \in R$, $vyx \Rightarrow^* z$, $z \in \Sigma^*$. Rearrange this derivation as

$$\begin{array}{llll} S & {}_{lm}{\Rightarrow}^* & u\underline{A}vBx & [\sigma] \\ & {}_{lm}{\Rightarrow} & utv\underline{B}x & [p\colon A \rightarrow t] \\ & \Rightarrow & utvyx & [r\colon B \rightarrow y] \\ & \Rightarrow^* & utz & [\pi] \\ & \Rightarrow^* & w & [o] \end{array}$$

The resulting derivation $S \Rightarrow^* w$ [$\sigma pr\pi o$] begins with at least $|\sigma p|$ leftmost steps, so its leftmost beginning is definitely longer than the leftmost beginning of the original derivation $S \Rightarrow^{n+1} w$ [ρ].

If $S \Rightarrow^* w$ [$\sigma pr\pi o$] still does not represent a leftmost derivation, we apply the analogical rearrangement to it. After $n - 2$ or fewer repetitions of this derivation rearrangement, we obtain $S \;_{lm}\!\Rightarrow^* w$, which completes the induction step, so the proof of the claim is completed.

From this claim, we see that G can generate every $w \in L(G)$ in the leftmost way, so the theorem holds.

■

We might be tempted to generalize Theorem 3.20 for $w \in \Sigma^*$. That is, we might consider a statement that for every grammar $G = (\Sigma, R)$, $S \Rightarrow^* w$ if and only if $S \;_{lm}\!\Rightarrow^* w$, for all $w \in \Sigma^*$. This statement is false, however. To give a trivial counterexample, consider a grammar with two rules of the form $S \rightarrow AA$ and $A \rightarrow a$. Observe that this grammar makes $S \Rightarrow AA \Rightarrow Aa$; however, there is no leftmost derivation of Aa in G.

Theorem 3.21. Let $G = (\Sigma, R)$ be a grammar. Then, $w \in L(G)$ if and only if $S \;_{rm}\!\Rightarrow^* w$.

From Definition 3.5, we already know how to convert any grammatical derivation to the corresponding parse tree. In the following proof, we describe the opposite conversion.

Theorem 3.22. Let $G = (\Sigma, R)$ be a grammar. Then, $A \Rightarrow^* x$ if and only if there exists a parse tree t such that $root(t) = A$ and $frontier(t) = x$, where $A \in N$, $x \in \Sigma^*$.

Proof.

Only If. This part of the proof says that for every derivation $A \Rightarrow^* x$, there exists a parse tree t such that $root(t) = A$ and $frontier(t) = x$, where $A \in N$, $x \in \Sigma^*$. From Definition 3.5, we know how to construct this tree, denoted by $pt(A \Rightarrow^* x)$.

If. We prove that for every derivation tree t in G with $root(t) = A$ and $frontier(t) = x$, where $A \in N$, $x \in \Sigma^*$, there exists $A \Rightarrow^* x$ in G by induction on $depth(t) \geq 0$.

Basis. Consider any derivation tree t in G such that $depth(t) = 0$. As $depth(t) = 0$, t is a tree consisting of one node, so $root(t) = frontier(t) = A$, where $A \in N$. Observe that $A \Rightarrow^0 A$ in G, so the basis holds.

Induction Hypothesis. Suppose that the claim holds for every derivation tree t with $depth(t) \leq n$, where n is a non-negative integer.

Induction Step. Let t be any derivation trees with $depth(t) = n + 1$. Let $root(t) = A$ and $frontier(t) = x$, where $A \in N$, $x \in \Sigma^*$. Consider the topmost rule tree, $rt(p)$, occurring in t; that is, $rt(p)$ is the rule tree whose root coincides with $root(t)$. Let p: $A \rightarrow u \in R$. If $u = \varepsilon$, t has actually the form $A\langle\rangle$, which means $u = \varepsilon$ and $depth(t) = 1$, and at this point, $A \Rightarrow \varepsilon$ [p], so the induction step is completed. Assume $u \neq \varepsilon$. Let $u = X_1X_2\ldots X_m$, where $m \geq 1$. Thus, t is of the form $A\langle t_1t_2\ldots t_m\rangle$, where each t_i is a parse tree with $root(t_i) = X_i$, $1 \leq i \leq m$, with $depth(t_i) \leq n$. Let $frontier(t_i) = y_i$, where $y_i \in \Sigma^*$, so $x = y_1y_2\ldots y_m$. As $depth(t_i) \leq n$, by the induction hypothesis, we have $X_i \Rightarrow^* y_i$ in G, $1 \leq i \leq m$. Since $A \rightarrow u \in R$ with $u = X_1X_2\ldots X_m$, we have $A \Rightarrow X_1X_2\ldots X_m$. Putting together $A \Rightarrow X_1X_2\ldots X_m$ and $X_i \Rightarrow^* y_i$ for all $1 \leq i \leq m$, we obtain

$$\begin{aligned} A &\Rightarrow X_1X_2\ldots X_m \\ &\Rightarrow^* y_1X_2\ldots X_m \\ &\Rightarrow^* y_1y_2\ldots X_m \end{aligned}$$

$$\begin{array}{ll} \vdots & \\ \Rightarrow^* & y_1y_2\ldots y_m \end{array}$$

Thus, $A \Rightarrow^* x$ in G.

■

Corollary 3.23. Let $G = (\Sigma, R)$ be a grammar. Then, $w \in L(G)$ if and only if there exists a parse tree t such that *root*(t) = S and *frontier*(t) = w.

Proof. Every $w \in L(G)$ satisfies $S \Rightarrow^* w$, so this corollary follows from Theorem 3.22.

■

From the previous statements, we next derive the main result concerning canonical derivations and parse trees. It guarantees that without any loss of generality, we can always restrict our attention to the canonical derivations or parse trees when discussing the language generated by a grammar.

Corollary 3.24. Let $G = (\Sigma, R)$ be a grammar. For every $w \in \Delta^*$, the following statements are equivalent in G:

• $w \in L(G)$;
• $S \Rightarrow^* w$;
• $S\ {}_{lm}\Rightarrow^* w$;
• $S\ {}_{rm}\Rightarrow^* w$;
• there exists a parse tree t such that *root*(t) = S and *frontier*(t) = w.

Proof. From Definition 3.1, for every $w \in \Delta^*$, $w \in L(G)$ if and only if $S \Rightarrow^* w$. The rest of Corollary 3.24 follows from Theorem 3.20, Theorem 3.21, and Corollary 3.23.

■

Proper Grammatical Specification of the Syntax

A grammar G may contain some components that are of no use regarding the generation of its language $L(G)$. These useless components unnecessarily increase the grammatical size, make the programming language syntax specification clumsy and complicate parsing. Therefore, we next explain how to remove them from G. First, we eliminate all useless symbols that are completely superfluous during the specification of any syntax structure. Then, we turn our attention to the removal of some grammatical rules, which might needlessly complicate the syntax specification. Specifically, we explain how to eliminate all the ε-rules and the unit rules that have a single nonterminal on their right-hand side. Finally, we summarize all these grammatical transformations by converting any grammar to an equivalent *proper grammar* without useless components.

Useful symbols. The grammatical symbols that take part in the generation of some sentences from $L(G)$ are useful; otherwise, they are useless. As completely useless, we obviously consider all symbols from which no string of terminals is derivable.

Definition 3.25 ***Terminating Symbols.*** Let $G = (\Sigma, R)$ be a grammar. A symbol $X \in \Sigma$ is *terminating* if $X \Rightarrow^* w$ for some $w \in {}_G\Delta^*$; otherwise, X is *non-terminating*.

■

To eliminate all non-terminating symbols, we first need to know which symbols are terminating and which are not.

Goal. Given a grammar, $G = (\Sigma, R)$, determine the subset $V \subseteq \Sigma$ that contains all terminating symbols.

Gist. Every terminal $a \in \Delta$ is terminating because $a \Rightarrow^* a$, so initialize V with Δ. If a rule $r \in R$ satisfies $\boldsymbol{rhs}(r) \in V^*$, $\boldsymbol{rhs}(r) \Rightarrow^* w$ for some $w \in \Delta^*$; at this point, we add $\boldsymbol{lhs}(r)$ to V because $\boldsymbol{lhs}(r) \Rightarrow \boldsymbol{rhs}(r) \Rightarrow^* w$ and $\boldsymbol{lhs}(r)$ is terminating, too. In this way, we keep extending V until no further terminating symbol can be added to V.

Algorithm 3.26 ***Terminating Symbols.***

Input • a grammar, $G = ({}_G\Sigma, {}_GR)$.

Output • the set $V \subseteq \Sigma$ containing all terminating symbols in G.

Method

```
begin
    $V := {}_G\Delta$;
    repeat
        if $\boldsymbol{rhs}(r) \in V^*$ for some $r \in R$ then
            add $\boldsymbol{lhs}(r)$ to $V$
    until no change;
end.
```

Prove that this simple algorithm works correctly as an exercise.

Example 3.6 ***Terminating Symbols.*** Consider this grammar

$$
\begin{aligned}
&S \rightarrow S\ o\ S\\
&S \rightarrow S\ o\ A\\
&S \rightarrow A\\
&A \rightarrow A\ o\ A\\
&S \rightarrow (S)\\
&S \rightarrow i\\
&B \rightarrow i
\end{aligned}
$$

where o, i, (, and) are terminals, and the other symbols are nonterminals. With this grammar as its input, Algorithm 3.26 first sets V as $V = \{o, i, (,)\}$. Then, it enters the **repeat** loop. As $B \rightarrow i$ with $i \in V$, it adds B to V. For the same reason, $S \rightarrow i$ leads to the inclusion of S to V, so $V = \{o, i, (,), B, S\}$. At this point, the **repeat** loop cannot further increase V, so it exits. As a result, A is non-terminating because $A \notin V$.

■

Besides non-terminating symbols, some symbols play no role in a grammar simply because they do not appear in any sentential form, which are strings derived from the grammatical start symbol. Recall that the set of all sentential forms is denoted by $F(G)$ (see Definition 3.1).

Definition 3.27 ***Accessible Symbol.*** Let $G = (\Sigma, R)$ be a grammar. A symbol $X \in \Sigma$ is *accessible* if $X \in alph(F(G))$; otherwise, X is *inaccessible*.

■

Goal. Given a grammar, $G = (\Sigma, R)$, determine the subset $W \subseteq \Sigma$ that contains all accessible symbols.

Gist. The start symbol S is *accessible* because $S \Rightarrow^0 S$, so initialize W with $\{S\}$. If a rule $r \in R$ satisfies $\textbf{\textit{lhs}}(r) \in W$, we add $alph(\textbf{\textit{rhs}}(r))$ to W because we can always rewrite $\textbf{\textit{lhs}}(r)$ with $\textbf{\textit{rhs}}(r)$ in every sentential form containing $\textbf{\textit{lhs}}(r)$ and obtain a sentential form containing symbols from $alph(\textbf{\textit{rhs}}(r))$, too. We keep extending W in this way until no further symbols can be added to W.

Algorithm 3.28 ***Accessible Symbols.***

Input • a grammar, $G = ({}_G\Sigma, {}_GR)$.

Output • the set $W \subseteq \Sigma$ containing all accessible symbols in G.

Method

```
begin
    $W := \{S\}$;
    repeat
        if $lhs(r) \in W$ for some $r \in R$ then
            add $alph(rhs(r))$ to $W$
    until no change;
end.
```

Prove that Algorithm 3.28 is correct as an exercise.

Example 3.7 ***Accessible Symbols.*** Like in the previous example, consider this grammar

$$S \rightarrow S\ o\ S,\ S \rightarrow S\ o\ A,\ S \rightarrow A,\ A \rightarrow A\ o\ A,\ S \rightarrow (S),\ S \rightarrow i,\ B \rightarrow i$$

With this grammar as its input, Algorithm 3.28 first sets $W = \{S\}$. Then, it enters the **repeat** loop. As $S \rightarrow S\ o\ S$, this loop adds o to W. Furthermore, since $S \rightarrow A$, the **repeat** loop also adds A there. Continuing in this way, this loop exits with W containing all symbols but B, so B is the only inaccessible symbol in this grammar.

■

Definition 3.29 ***Useful Symbols.*** Let $G = (\Sigma, R)$ be a grammar. A symbol $X \in \Sigma$ is *useful* if $S \Rightarrow^* y \Rightarrow^* w$ for some $w \in \Delta^*$ and $y \in \Sigma^*$ with $X \in alph(y)$; otherwise, X is *useless*.

■

Making use of the previous two algorithms, we next turn any grammar to an equivalent grammar that contains only useful symbols.

Algorithm 3.30 ***Useful Symbols.***

Input • a grammar $G = ({}_G\Sigma, {}_GR)$.

Output • a grammar $H = ({}_H\Sigma, {}_HR)$ such that $L(G) = L(H)$ and ${}_H\Sigma$ contains only useful symbols.

Method

```
begin
    (1) by using Algorithm 3.26, determine all terminating symbols in $G$, then eliminate all non-
        terminating symbols together with all the rules that contain them in $G$;
    (2) by using Algorithm 3.28, determine all accessible symbols in $G$, then remove all
        inaccessible symbols together with the rules that contain them from $G$;
    (3) define $H$ as the grammar resulting from these two removals
end.
```

Theorem 3.31. Algorithm 3.30 correctly converts a grammar G to a grammar $H = ({}_H\Sigma, {}_HR)$ so that $L(G) = L(H)$ and ${}_H\Sigma$ contains only useful symbols.

Proof. To prove that ${}_H\Sigma$ contains only useful symbols, we start with

Claim A. Every nonterminal in ${}_HN$ is useful.

Proof of Claim A by contradiction. Assume that $A \in {}_HN$ is useless. Consequently, for every $y \in \Sigma^*$ such that $S \Rightarrow^* y$, $A \notin alph(y)$ or for every $x \in \Sigma^*$ such that $A \Rightarrow^* x$, $x \notin {}_H\Delta^*$. Let $A \notin alph(y)$ for every $y \in \Sigma^*$ satisfying $S \Rightarrow^* y$. At this point, however, A is eliminated by Algorithm 3.26, so $A \notin {}_HN$—a contradiction. Thus, for some $y \in \Sigma^*$ satisfying $S \Rightarrow^* y$, $A \in alph(y)$. Assume that for every $x \in \Sigma^*$ such that $A \Rightarrow^* x$, $x \notin {}_H\Delta^*$. Under this assumption, however, A is eliminated by Algorithm 3.28, so $A \notin {}_HN$, which again contradicts $A \in {}_HN$. As a result, every nonterminal in ${}_H\Sigma$ is useful.

As an exercise, prove that every terminal in ${}_H\Sigma$ is useful and that $L(G) = L(H)$.

■

Observe that the order of the two transformations in Algorithm 3.30 is important. If we reverse them, Algorithm 3.30 does not work correctly. To give a simple counterexample, consider $S \rightarrow a$, $S \rightarrow A$, $A \rightarrow AB$, and $B \rightarrow a$ as a grammar, which generates $\{a\}$. If we apply the transformations in Algorithm 3.30 properly, we obtain an equivalent one-rule grammar $S \rightarrow a$ without the useless symbols A and B. However, if we improperly apply Algorithm 3.28 before Algorithm 3.26, we obtain a two-rule grammar $S \rightarrow a$ and $B \rightarrow a$, in which B is useless.

Example 3.8 *Useful Symbols*. Once again, return to the grammar

$$S \rightarrow S\ o\ S,\ S \rightarrow S\ o\ A,\ S \rightarrow A,\ A \rightarrow A\ o\ A,\ S \rightarrow (S),\ S \rightarrow i,\ B \rightarrow i$$

discussed in the previous two examples. Eliminate the non-terminating symbols from this grammar by Algorithm 3.26. From Example 3.6, we already know that A is the only non-terminating symbol, so the elimination of all rules containing A produces

$S \rightarrow S\ o\ S$
$S \rightarrow (S)$
$S \rightarrow i$
$B \rightarrow i$

Apply Algorithm 3.28 to this grammar to find out that B is the only inaccessible symbol. By removing, we obtain the resulting equivalent grammar with only useful symbols

$S \rightarrow S\ o\ S$
$S \rightarrow (S)$

$S \to i$

■

ε-Rules. It is often convenient to eliminate ε-rules from grammars. Indeed, without ε-rules, grammars can never make any sentential form shorter, and this property obviously simplifies the examination of derivations and, consequently, the parsing process. As a matter of fact, some parsing methods, such as precedence parsers (see Section 5.1), work only with grammars without ε-rules. That is why we next explain how to eliminate these rules. To do that, we first determine the nonterminals from which ε can be derived.

Definition 3.32 ε-*Nonterminals*. Let $G = (\Sigma, R)$ be a grammar. A nonterminal $A \in N$ is *ε-nonterminal* in G if $A \Rightarrow^* \varepsilon$.

■

Goal. Given a grammar, $G = (\Sigma, R)$, determine the set $E \subseteq N$ containing all ε-nonterminals.

Gist. For every ε-rule $A \to \varepsilon$, A is an ε-nonterminal because $A \Rightarrow \varepsilon$; therefore, we initialize E with $\{\boldsymbol{lhs}(p) | \ p \in R, \boldsymbol{rhs}(p) = \varepsilon\}$. If a rule $r \in R$ satisfies $\boldsymbol{rhs}(r) \in E^*$, then $\boldsymbol{rhs}(r) \Rightarrow^* \varepsilon$; therefore, we add $\boldsymbol{lhs}(r)$ to E because $\boldsymbol{lhs}(r) \Rightarrow \boldsymbol{rhs}(r) \Rightarrow^* \varepsilon$, so $\boldsymbol{lhs}(r)$ is thus ε-nonterminal. We keep extending E in this way until no further ε-nonterminals can be added to E.

Algorithm 3.33 ε-*Nonterminals*.

Input • a grammar, $G = ({}_G\Sigma, {}_GR)$.

Output • the set $E \subseteq {}_GN$ containing all ε-nonterminals in G.

Method

```
begin
    $E := \{\boldsymbol{lhs}(p) | \ p \in R, \boldsymbol{rhs}(p) = \varepsilon\}$;
    repeat
        if $\boldsymbol{rhs}(r) \in E^*$ for some $r \in R$ then
            add $\boldsymbol{lhs}(r)$ to $E$
    until no change;
end.
```

Example 3.9 ε-*Nonterminals*. Consider this grammar G

$S \to AB$
$A \to aAb$
$B \to cBd$
$A \to \varepsilon$
$B \to \varepsilon$

that generates $L(G) = \{a^nb^n | \ n \geq 0\}\{c^md^m | \ m \geq 0\}$. With G as its input, Algorithm 3.33 initializes E with A and B because these nonterminals occur as the left-hand sides of the two ε-rules in G. Then, it enters the **repeat** loop. As $S \to AB$ with $AB \in E^*$, it includes S into E. At this point, the **repeat** loop cannot further increase E, so it exits.

■

Having determined the set of all ε-nonterminals in any grammar G, we can now convert G to a grammar H that has no ε-rules and generates $L(G) - \{\varepsilon\}$. In other words, both G and H generate the language consisting of the same non-empty strings; however, H cannot generate ε without ε-rules even if $\varepsilon \in L(G)$, hence $L(H) = L(G) - \{\varepsilon\}$.

Goal. Transform a grammar $G = ({}_G\Sigma, {}_GR)$ to a grammar $H = ({}_H\Sigma, {}_HR)$ so $L(H) = L(G) - \{\varepsilon\}$ and ${}_HR$ contains no ε-rules.

Gist. If we can express the right-hand side of a rule $r \in R$ as $\textbf{\textit{rhs}}(r) = x_0A_1x_1\ldots A_nx_n$ so that $x_0x_1\ldots x_n \neq \varepsilon$ and A_1 through A_n are ε-nonterminals, then add $\textbf{\textit{lhs}}(r) \rightarrow x_0X_1x_1\ldots X_nx_n$ to ${}_HR$ for all $X_i \in \{\varepsilon, A_i\}$, $1 \leq i \leq n$, because each A_i can be erased by $A_i \Rightarrow^* \varepsilon$. We keep expanding ${}_HR$ in this way until no further rules can be added to E.

Algorithm 3.34 *Grammar without ε-Rules.*

Input • a grammar, $G = ({}_G\Sigma, {}_GR)$.

Output • a grammar, $H = ({}_H\Sigma, {}_HR)$, such that $L(H) = L(G) - \{\varepsilon\}$ and ${}_HR$ contains no ε-rules.

Method

```
begin
  Set $_H\Sigma = {}_G\Sigma$ and $_HR = \{r \mid r \in {}_GR; r$ is not an ε-rule$\}$;
  Determine the set $E \subseteq {}_GN$ containing all ε-nonterminals in $G$ by Algorithm 3.33;
  repeat
    if for some $r \in R$, $rhs(r) = x_0A_1x_1\ldots A_nx_n$, where $A_i \in E$, $x_j \in ({}_G\Sigma - E)^*$, $X_i \in \{\varepsilon, A_i\}$,
    $1 \leq i \leq n$, $0 \leq j \leq n$, and $|x_0x_1\ldots x_n| \geq 1$ then
      add $lhs(r) \rightarrow x_0X_1x_1\ldots X_nx_n$ to $_HR$
  until no change;
end.
```

Example 3.10 *Elimination of ε-Rules.* Consider grammar G defined as $S \rightarrow AB$, $A \rightarrow aAb$, $B \rightarrow cBd$, $A \rightarrow \varepsilon$, and $B \rightarrow \varepsilon$ from the previous example. With G as its input, Algorithm 3.34 initializes ${}_HR$ with $\{S \rightarrow AB, A \rightarrow aAb, B \rightarrow cBd\}$, then enters the **repeat** loop. Consider $S \rightarrow AB$. Both A and B are ε-nonterminals, so Algorithm 3.34 adds $S \rightarrow AB$, $S \rightarrow B$, and $S \rightarrow A$ to ${}_HR$. Analogically, from $A \rightarrow aAb$ and $B \rightarrow cBd$, this algorithm constructs $A \rightarrow aAb$, $B \rightarrow cBd$, $A \rightarrow ab$, and $B \rightarrow cd$, respectively. In this way, as the resulting grammar H without ε-rules, Algorithm 3.34 produces grammar H as

$$S \rightarrow AB,\ S \rightarrow A,\ S \rightarrow B,\ A \rightarrow aAb,\ B \rightarrow cBd,\ A \rightarrow ab,\ B \rightarrow cd$$

Observe that H generates $\{a^nb^n \mid n \geq 0\}\{c^nd^n \mid n \geq 0\} - \{\varepsilon\}$, so $L(H) = L(G) - \{\varepsilon\}$.

■

Unit Rules. By using rules with a single nonterminal on the right-hand side, grammars only rename their nonterminals; otherwise, they fulfill no role at all. It is often desirable to remove them from grammars because they usually needlessly increase the grammatical size and, thus, obscure the syntax specification.

Definition 3.35 *Unit Rules.* Let $G = (\Sigma, R)$ be a grammar. A rule $r \in R$ is a *unit rule* if $\textbf{\textit{rhs}}(r) \in N$.

Goal. Transform a grammar $G = ({}_G\Sigma, {}_GR)$ to a grammar $H = ({}_H\Sigma, {}_HR)$ so that $L(H) = L(G)$ and ${}_HR$ contains no unit rules.

Gist. For all possible derivations of the form $A \Rightarrow^* B \Rightarrow x$, where $x \notin {}_GN$ and $A \Rightarrow^* B$ is made according to a sequence of unit rules so that this sequence contains no two identical rules, add $A \rightarrow x$ to ${}_HR$ in order to simulate this derivation. Every derivation of the above form consists of $card({}_GR)$ of fewer steps, so the set of all these derivations is finite. H is the grammar resulting from this addition.

Algorithm 3.36 ***Grammar without Unit Rules.***

Input • a grammar, $G = ({}_G\Sigma, {}_GR)$.

Output • a grammar, $H = ({}_H\Sigma, {}_HR)$, such that $L(H) = L(G) - \{\varepsilon\}$ and ${}_HR$ contains no unit rules.

Method

```
begin
   set $_H\Sigma = {}_G\Sigma$ and $_HR = \emptyset$;
   repeat
      if $A \Rightarrow^n B\ [r_1r_2\ldots r_n] \Rightarrow x$ in $G$, where $A, B \in {}_GN$, $x \in {}_G\Sigma^* - {}_GN$, $1 \le n \le card({}_GR)$, and
      each $r_j$ is a unit rule, $1 \le j \le n$ then
         add $A \rightarrow x$ to $_HR$
   until no change
end.
```

Example 3.11 ***Elimination of Unit Rules.*** In the previous example, the elimination of all ε-rules gives rise to this grammar, containing two unit rules

$$S \rightarrow AB,\ S \rightarrow A,\ S \rightarrow B,\ A \rightarrow aAb,\ B \rightarrow cBd,\ A \rightarrow ab,\ B \rightarrow cd$$

Algorithm 3.36 transforms this grammar to an equivalent grammar without unit rules as follows. From $S \Rightarrow A \Rightarrow aAb$, it produces $S \rightarrow aAb$. Similarly, from $S \Rightarrow A \Rightarrow ab$, it constructs $S \rightarrow ab$. From $S \Rightarrow AB$, it simply obtains $S \rightarrow AB$. As the resulting grammar, it produces

$$S \rightarrow AB,\ S \rightarrow aAb,\ S \rightarrow ab,\ S \rightarrow cBd,\ S \rightarrow cd,\ A \rightarrow aAb,\ B \rightarrow cBd,\ A \rightarrow ab,\ B \rightarrow cd$$

■

To summarize all the previous transformations, we might consider a grammatical syntax specification as *proper* if it is based on a grammar that contains no useless symbols, ε-rules, or unit rules, hence the next definition.

Definition 3.37 ***Proper Grammar.*** Let $G = (\Sigma, R)$ be a grammar such that each symbol in Σ is useful, and let R contain neither ε-rules nor unit rules. Then, G is a *proper grammar*.

Theorem 3.38. For every grammar, G, there exists an equivalent proper grammar, H.

Proof. Let $G = (\Sigma, R)$ be a grammar. Without any loss of generality, assume that each symbol in Σ is useful (see Theorem 3.31). By Algorithm 3.34, convert G to an equivalent grammar containing no ε-rules; then, by Algorithm 3.35, convert the resulting grammar without ε-rules to an equivalent grammar, H, containing no unit rules. Neither of these conversions introduces any useless symbols, and the conversion by Algorithm 3.35 introduces no ε-rules. Thus, H is proper. A detailed version of this proof is left as an exercise.

■

Although the advantages of proper grammars are obvious, we should point out that some parsing methods can work with grammars that are not proper. For instance, Sections 4.2 and 5.2 describe some parsers based upon grammars that are not proper because they contain ε-rules. On the other hand, even the syntax specification by a proper grammar may not be enough for some parsing methods. Indeed, some methods work only with grammars with rules in some strictly prescribed normal forms, which are discussed in the next section.

3.3.4 Grammatical Normal Forms and Parsing Based on Them

In this section, we discuss two normal forms of grammars and describe parsers that work with them. Specifically, we give the Chomsky normal form, in which every grammatical rule has on its right-hand side either a terminal or two nonterminals. We explain how to transform any proper grammar to an equivalent grammar in the Chomsky normal form. Apart from this normal form, we also define the Greibach normal form of grammars, in which every grammatical rule has on its right-hand side a terminal followed by zero or more nonterminals.

Leaving fully rigorous proofs that any proper grammar can be turned to an equivalent grammar in either of the two normal forms, we concentrate our attention on the parsing methods based upon these forms in this section. Let us note that later in this chapter, we also point out their value from a theoretical point of view as well; most importantly, we make use of the Chomsky normal form in some proofs of the next section (see proofs of Lemma 3.46, Corollary 3.47, and Lemma 3.48).

Normal forms

Definition 3.39 ***Chomsky normal form.*** A grammar, $G = (\Sigma, R)$, is in Chomsky normal form if every rule $A \to x \in R$ satisfies $x \in \Delta \cup NN$.

■

Goal. Given a proper grammar, $G = ({}_G\Sigma, {}_GR)$, transform G to an equivalent grammar, $H = ({}_H\Sigma, {}_HR)$, in Chomsky normal form in which every rule $A \to x \in {}_HR$ satisfies $x \in {}_H\Delta \cup {}_HN^2$.

Gist. Move all rules satisfying the Chomsky normal form from ${}_GR$ to ${}_HR$. Introduce a set of new nonterminals, O, such that $card(O) = card({}_G\Delta)$ and a bijection, β, from ${}_G\Sigma$ to $O \cup {}_GN$. Include O into ${}_HN$, and for every $a \in {}_G\Delta$, add $\beta(a) \to a$ to ${}_HR$. For every $A \to X_1X_2 \in {}_GR$, add $A \to \beta(X_1)\beta(X_2)$ to ${}_HR$ (recall that X_i represents any symbol from ${}_G\Sigma$ according to Convention 3.2). For every $A \to X_1X_2X_3\ldots X_{n-1}X_n \in {}_GR$, where $n \geq 3$, include new nonterminals $\langle X_2\ldots X_n\rangle$, $\langle X_3\ldots X_n\rangle$, …, $\langle X_{n-2}X_{n-1}X_n\rangle$ $\langle X_{n-1}X_n\rangle$ into ${}_HN$ and add these rules in Chomsky normal form $A \to \beta(X_1)\langle X_2\ldots X_n\rangle$, $\langle X_2\ldots X_n\rangle \to \beta(X_2)\langle X_3\ldots X_n\rangle$, …, $\langle X_{n-2}X_{n-1}X_n\rangle \to \beta(X_{n-2})\langle X_{n-1}X_n\rangle$, $\langle X_{n-1}X_n\rangle \to \beta(X_{n-1})\,\beta(X_n)$ to ${}_HR$. In this way, $A \Rightarrow X_1X_2X_3\ldots X_{n-1}X_n\,[A \to X_1X_2X_3\ldots X_{n-1}X_n]$ in G is simulated in H as

$$\begin{aligned}
A \quad &\Rightarrow \quad \beta(X_1)\langle X_2\ldots X_n\rangle \\
&\Rightarrow \quad \beta(X_2)\langle X_3\ldots X_n\rangle \\
&\ \vdots \\
&\Rightarrow \quad \beta(X_{n-2})\langle X_{n-1}X_n\rangle \\
&\Rightarrow \quad \beta(X_{n-1})\beta(X_n)
\end{aligned}$$

and $\beta(X_j) \Rightarrow X_j\,[\beta(X_j) \to X_j]$ in H , for every $X_j \in {}_G\Delta$, where $1 \leq j \leq n$.

Algorithm 3.40 ***Chomsky Normal Form.***

Input • a proper grammar, $G = ({}_G\Sigma, {}_GR)$.

Output • a grammar, $H = ({}_H\Sigma, {}_HR)$, in Chomsky normal form such that $L(G) = L(H)$.

Method

```
begin
  set $_H\Sigma = {}_G\Sigma$ and $_HR = \{r \mid r \in {}_GR, \mathbf{rhs}(p) \in {}_G\Delta \cup {}_GN^2\}$;
  introduce a set of new nonterminals, $O$, such that $O \cap {}_G\Sigma = \emptyset$ and $card(O) = card({}_G\Delta)$;
  introduce a bijection, $\beta$, from $_G\Sigma$ to $O \cup {}_GN$;
  include $\{\beta(a) \to a \mid a \in {}_G\Delta\} \cup \{A \to \beta(X_1)\beta(X_2) \mid A \to X_1X_2 \in {}_GR\}$ into $_HR$;
  repeat
    if for some $n \geq 3$, $A \to X_1X_2X_3\ldots X_{n-1}X_n \in {}_GR$ then
      add $A \to \beta(X_1)\langle X_2\ldots X_n\rangle$, $\langle X_2\ldots X_n\rangle \to \beta(X_2)\langle X_3\ldots X_n\rangle$, …,
      $\langle X_{n-2}X_{n-1}X_n\rangle \to \beta(X_{n-2})\langle X_{n-1}X_n\rangle$, $\langle X_{n-1}X_n\rangle \to \beta(X_{n-1})\beta(X_n)$ to $_HR$
  until no change;
end.
```

Example 3.12 ***Chomsky normal form.*** Return to the grammar G obtained in Example 3.8 as $S \to S\ o\ S$, $S \to (S)$, $S \to i$. Recall that $L(G) = \{a^nb^n \mid n \geq 0\}\{c^nd^n \mid n \geq 0\}$. From Example 3.8, we already know that only useful symbols occur in G. Containing neither ε-rules nor unit rules, G represents a proper grammar. Algorithm 3.40 converts G to an equivalent grammar, $H = ({}_H\Sigma, {}_HR)$, in the Chomsky normal form in the following way. As $S \to i$ satisfies Chomsky normal form, it initializes $_HR$ with this rule. Then, it introduces four new nonterminals, A, B, C, and D, and a bijection β from $\{o, (,), i\}$ to $\{A, B, C, D\}$ so that β maps o, (,), and i to A, B, C, and D, respectively. Algorithm 3.40 adds $A \to o$, $B \to ($, $C \to)$, and $D \to i$ to $_HR$. From $S \to S\ o\ S$, this algorithm subsequently constructs $S \to S\langle oS\rangle$, $\langle oS\rangle \to AS$. Analogously, from $S \to (S)$, it constructs $S \to B\langle S)\rangle$, $\langle S)\rangle \to SC$. As the resulting grammar $H = ({}_H\Sigma, {}_HR)$ in the Chomsky normal form, Algorithm 3.40 constructs

$$S \to S\langle oS\rangle,\ \langle oS\rangle \to AS,\ S \to B\langle S)\rangle,\ \langle S)\rangle \to SC,\ A \to o,\ B \to (,\ C \to),\ D \to i,\ S \to i$$

As this example illustrates, the resulting grammar H may contain some useless symbols; specifically, D is inaccessible in H. To detect and remove these symbols as well as rules containing them, use Algorithm 3.30, which removes the superfluous rule $D \to i$ from the above grammar.

■

Apart from the Chomsky normal form, some parsing methods make use of the Greibach normal form of grammars, in which every grammatical rule has on its right-hand side a terminal followed by a string of zero or more nonterminals.

Definition 3.41 ***Greibach normal form.*** A grammar, $G = (\Sigma, R)$, is in Greibach normal form if every rule $A \to x \in R$ satisfies $x \in \Delta N^*$.

■

A transformation that turns any proper grammar to an equivalent grammar satisfying the Greibach normal form is left as an exercise. Next, we turn our attention to a parser based on this form.

Parsers based on normal forms

Consider the general top-down parser described as Algorithm 3.12 in the previous section. This parser reads no input symbol when it replaces a nonterminal with a string on the pushdown top. To create a parser that reads an input symbol during replacements of this kind, we next construct a G-based top-down parser, where G is a grammar in Greibach normal form, in which every rule has a terminal followed by zero or more nonterminals on its right-hand side. Making use of this form of rules, the resulting parser represents a *pushdown automaton without* ε*-rules* that reads an input symbol during every single move it makes.

Definition 3.42 ***Pushdown Automaton without* ε*-Rules*.** Let $M = (\Sigma, R)$ be a pushdown automaton. M is a *pushdown automaton without* ε*-rules* if for every $r \in R$, $lhs(r) \in \Gamma^*Q\Delta$.

■

Goal. Given a grammar, $G = ({}_G\Sigma, {}_GR)$, in Greibach normal form, construct a G-based top-down parser, M, so that M reads an input symbol during every move it makes.

Gist. As G is in Greibach normal form, during any leftmost derivations, G produces sentential forms consisting of a string of terminals followed by a string of nonterminals. Consider $S\ {}_{lm}\Rightarrow^*\ vz\ {}_{lm}\Rightarrow^*\ vt$ in G, where $v, t \in \Delta^*$ and $z \in N^*$. By analogy with Algorithm 3.12, after simulating $S\ {}_{lm}\Rightarrow^*\ vz$, the G-based top-down parser, M, constructed in the next algorithm uses its pushdown to record z in reverse while having t as the remaining input. As opposed to Algorithm 3.12, however, M can simulate a leftmost derivation step made by every rule $A \to aB_1B_2\ldots B_n \in {}_GR$ in Greibach normal form, where a is a terminal and A and Bs are nonterminals, so M replaces A with $B_1B_2\ldots B_n$ on the pushdown top and, at the same time, reads a as the current input symbol. To explain the way M works in greater detail, consider

$$\begin{array}{lll} S & {}_{lm}\Rightarrow^* & vAy \\ & {}_{lm}\Rightarrow & vaB_1B_2\ldots B_ny \quad [r\colon A \to aB_1B_2\ldots B_n] \\ & {}_{lm}\Rightarrow^* & w \end{array}$$

where $va = prefix(w, |va|)$, $y \in {}_GN^*$, and $w = vaz$ with $z \in {}_G\Delta^*$. We construct M so after simulating $S\ {}_{lm}\Rightarrow^*\ vAy\ [\rho]$, M's configuration is $\blacktriangleright reversal(y)A\blacklozenge az\blacktriangleleft$. From this configuration, M simulates $vAy\ {}_{lm}\Rightarrow\ vaB_1B_2\ldots B_ny\ [r\colon A \to aB_1B_2\ldots B_n]$ as

$$\blacktriangleright reversal(y)A\blacklozenge az\blacktriangleleft \Rightarrow \blacktriangleright reversal(y)B_n\ldots B_1\blacklozenge z\blacktriangleleft$$

by using $A\blacklozenge a \to B_n\ldots B_1\blacklozenge$. As a result, M simulates every n-step leftmost derivation in G by making n steps, which is a property not guaranteed by the parser constructed in Algorithm 3.12.

Algorithm 3.43 ***Top-Down Parser for a Grammar in Greibach Normal Form*.**

Input • a grammar, $G = ({}_G\Sigma, {}_GR)$, in Greibach normal form.

Output • a G-based top-down parser, $M = ({}_M\Sigma, {}_MR)$, which reads during every move.

Method

```
begin
    set ${}_M\Sigma = {}_G\Sigma$ with ${}_M\Gamma = {}_GN$, ${}_M\Delta = {}_G\Delta$, ${}_MS = {}_GS$; ${}_MR = \emptyset$;
    for each $A \to ax \in {}_GR$, where $A \in_G N$, $a \in {}_G\Delta$, and $x \in {}_GN^*$ do
        add $A◆a \to reversal(x)◆$ to ${}_MR$;
end.
```

Example 3.13 ***Top-Down Parser Based on Greibach Normal Form.*** Let L be the language consisting of all non-empty even-length palindromes over $\{a, b\}^+$; formally, $L = \{w reversal(w) | w \in \{a, b\}^+\}$. L is generated by the following grammar G in Greibach normal form:

1: $S \to aSA$
2: $S \to bSB$
3: $S \to aA$
4: $S \to bB$
5: $A \to a$
6: $B \to b$

Algorithm 3.43 produces a G-based top-down parser $M = (\Sigma, R)$. From the first rule, 1: $S \to aSA$, this algorithm constructs $S◆a \to reversal(SA)◆ \in {}_MR$. As a result, it produces M given in the second column of Figure 3.12. In the third column, the M-G parsing correspondence o is defined by analogy with Case Study 7/35 in the previous section. From M and o, we obtain the G-based top-down transducer $\Pi = (\Sigma, R, o)$ defined in the fourth column in Figure 3.12.

G	M	o	Π
1: $S \to aSA$	$S◆a \to AS◆$	$o(S◆a \to AS◆) = 1$	$S◆a \to AS◆1$
2: $S \to bSB$	$S◆b \to BS◆$	$o(S◆b \to BS◆) = 2$	$S◆b \to BS◆2$
3: $S \to aA$	$S◆a \to A◆$	$o(S◆a \to A◆) = 3$	$S◆a \to A◆3$
4: $S \to bB$	$S◆b \to B◆$	$o(S◆b \to B◆) = 4$	$S◆b \to B◆4$
5: $A \to a$	$A◆a \to ◆$	$o(A◆a \to ◆) = 5$	$A◆a \to ◆5$
6: $B \to b$	$B◆b \to ◆$	$o(B◆b \to ◆) = 6$	$B◆b \to ◆6$

Figure 3.12 ***Top-down parsing models based on Greibach normal form.***

With *babbab* as its input, Π makes

▶S◆$babbab$◀	$\Rightarrow$	▶BS◆$abbab$◀2	[$S◆b \to BS◆2$]
	$\Rightarrow$	▶BAS◆$bbab$◀21	[$S◆a \to AS◆1$]
	$\Rightarrow$	▶BAB◆bab◀214	[$S◆b \to B◆4$]
	$\Rightarrow$	▶BA◆ab◀2146	[$B◆b \to ◆6$]
	$\Rightarrow$	▶B◆b◀21465	[$A◆a \to ◆5$]
	$\Rightarrow$	▶◆◀214656	[$B◆b \to ◆6$]

***Leftmost Derivation in** G*		***Parse Tree***	***Translation by** Π*
S		S	▶S◆$babbab$◀
$_{lm}\Rightarrow bSB$	[2]	$S\langle bSB\rangle$	$\Rightarrow$ ▶BS◆$abbab$◀2
$_{lm}\Rightarrow baSAB$	[1]	$S\langle bS\langle aSA\rangle B\rangle$	$\Rightarrow$ ▶BAS◆$bbab$◀21
$_{lm}\Rightarrow babBAB$	[4]	$S\langle bS\langle aS\langle bB\rangle A\rangle B\rangle$	$\Rightarrow$ ▶BAB◆bab◀214
$_{lm}\Rightarrow babbAB$	[6]	$S\langle bS\langle aS\langle bB\langle b\rangle\rangle A\rangle B\rangle$	$\Rightarrow$ ▶BA◆ab◀2146
$_{lm}\Rightarrow babbaB$	[5]	$S\langle bS\langle aS\langle bB\langle b\rangle\rangle A\langle a\rangle\rangle B\rangle$	$\Rightarrow$ ▶B◆b◀21465

Figure 3.13 ***Top-down parse of a sentence.***

In short, $\blacktriangleright S\blacklozenge babbab\blacktriangleleft \Rightarrow^* \blacktriangleright\blacklozenge\blacktriangleleft 214656$ in Π, so $(babbab, 214656) \in \tau(\Pi)$. Notice that 214656 represents the left parse of *babbab* in *G*. Figure 3.13 gives the step-by-step correspondence between the leftmost derivation in *G*, the top-down constructed parse tree, and Π's translation of *babbab* to 214656 made by Π.

■

Algorithm 3.43 implies the next corollary because for every grammar, there exists an equivalent grammar in Greibach normal form.

Corollary 3.44. Let $G = ({}_G\Sigma, {}_GR)$ be a grammar. Then, there exists a one-state pushdown automaton without ε-rules that accepts $L(G)$. With $w \in {}_G\Delta^*$ as the input string, *M* thus makes no more than $|w|$ moves.

■

All the previous three parsing algorithms turn any grammar *G* to an equivalent *G*-based parser represented by a pushdown automaton. However, the parsing theory has also developed algorithms that act as parsers, but they are not based upon pushdown automata simulating canonical derivations in *G*. We close this section by giving one of them, called the Cocke-Younger-Kasami parsing algorithm after their authors. This algorithm works in a bottom-up way with grammars in the Chomsky normal form. Recall that in these grammars, every rule has one terminal or two nonterminals on its right-hand side, and Algorithm 3.40 has already demonstrated how to turn any grammar to an equivalent grammar in this normal form.

Goal. Given a grammar, $G = (\Sigma, R)$, in the Chomsky normal form and a string, $x = a_1a_2\ldots a_n$ with $a_i \in {}_G\Delta$, $1 \leq i \leq n$, for some $n \geq 1$, decide whether $x \in L(G)$.

Gist. The Cocke-Younger-Kasami parsing algorithm makes a decision of whether $x \in L(G)$ in a bottom-up way. It works so it constructs sets of nonterminals, $CYK[i, j]$, where $1 \leq i \leq j \leq n$, satisfying $A \in CYK[i, j]$ if and only if $A \Rightarrow^* a_i\ldots a_j$; therefore, as a special case, $L(G)$ contains *x* if and only if $CYK[1, n]$ contains *G*'s start symbol, *S*. To construct the sets of nonterminals, this algorithm initially includes *A* into $CYK[i, i]$ if and only if $A \rightarrow a_i$ is in *R* because $A \Rightarrow a_i$ in *G* by using $A \rightarrow a_i$. Then, whenever $B \in CYK[i, j]$, $C \in CYK[j+1, k]$, and $A \rightarrow BC \in {}_GR$, we add *A* to $CYK[i, k]$ because $B \in CYK[i, j]$ and $C \in CYK[j+1, k]$ imply $B \Rightarrow^* a_i\ldots a_j$, and $C \Rightarrow^* a_{j+1}\ldots a_k$, respectively, so

$$\begin{array}{lll} A & \Rightarrow & BC \qquad [A \rightarrow BC] \\ & \Rightarrow^* & a_i\ldots a_jC \\ & \Rightarrow^* & a_i\ldots a_ja_{j+1}\ldots a_k \end{array}$$

When this construction cannot extend any set, we examine whether $S \in CYK[1, n]$. If so, $S \Rightarrow^* a_1\ldots a_n$ and $x = a_1\ldots a_n \in L(G)$, so this algorithm announces **ACCEPT** (see Convention 1.8); otherwise, $x \notin L(G)$ and the algorithm announces **REJECT**.

Algorithm 3.45 ***Cocke-Younger-Kasami Parsing Algorithm.***

Input • a grammar, $G = ({}_G\Sigma, {}_GR)$, in the Chomsky normal form;
• $w = a_1a_2\ldots a_n$ with $a_i \in {}_G\Delta$, $1 \leq i \leq n$, for some $n \geq 1$.

Output • **ACCEPT** if $w \in L(G)$;
• **REJECT** if $w \notin L(G)$.

Method

```
begin
    introduce sets $CYK[i, j] = \emptyset$ for $1 \le i \le j \le n$;
    for i = 1 to $n$ do
        if $A \to a_i \in R$ then
            add $A$ to $CYK[i, i]$;
    repeat
        if $B \in CYK[i, j]$, $C \in CYK[j+1, k]$, $A \to BC \in R$ for some $A, B, C \in {}_GN$
            then add $A$ to $CYK[i, k]$;
    until no change;
    if $S \in CYK[1, n]$ then
        ACCEPT
    else REJECT;
end.
```

All the parsers given so far work non-deterministically (Algorithms 3.12, 3.16, 3.43, and 3.45), so they are of little importance in practice. In a real world, we are primarily interested in parsing algorithms underlain by easy-to-implement deterministic pushdown automata. These deterministic parsing algorithm work only for some special cases of grammars. Besides, as they are based upon deterministic pushdown automata, they are less powerful than the parsers discussed earlier in this chapter (see Theorem 3.19). Nevertheless, their determinism represents an enormous advantage that makes them by far the most popular parsing algorithms in practice, and that is why we discuss them in detail from a practical viewpoint in Chapters 4 and 5.

3.3.5 Syntax that Grammars cannot Specify

Sometimes, we come across syntactic programming-language constructs that cannot be specified by any grammars. To put it more theoretically, these constructs are out of the family of context-free languages (see the note following Definition 3.1 for this family). However, proving that a language L is out of this family may represent a difficult task because it actually requires to demonstrate that none of all possible grammars generates L. Frequently, we can simplify such a proof by demonstrating that L does not satisfy some conditions that all context-free languages satisfy, so L cannot be context-free. As a result, conditions of this kind are important to the syntax analysis, so we pay a special attention to them in this section. First, we give conditions of the following pumping lemma, then we present some useful closure properties to prove that a language is not context-free.

Pumping Lemma and its Proof

The pumping lemma says that for every context-free language, L, there is a constant $k \ge 1$ such that every $z \in L$ with $|z| \ge k$ can be expressed as $z = uvwxy$ with $vx \neq \varepsilon$ so that L also contains uv^mwx^my, for every non-negative integer m. Consequently, to demonstrate the non-context-freedom of a language, K, by contradiction, assume that K is a context-free language, and k is its pumping lemma constant. Select a string $z \in K$ with $|z| \ge k$, consider all possible decompositions of z into $uvwxy$, and for each of these decompositions, prove that uv^mwx^my is out of K, for some $m \ge 0$, which contradicts the pumping lemma. Thus, K is not context-free.

Without any loss of generality, we prove the context-free pumping lemma based on the Chomsky normal form, discussed in Section 3.3.4. In addition, we make use of some other notions introduced earlier in this chapter, such as the parse tree $pt(A \Rightarrow^* x)$ corresponding to a derivation $A \Rightarrow^* x$ (see Definition 3.5).

Lemma 3.46. Let $G = (\Sigma, R)$ be a grammar in the Chomsky normal form. For every parse tree $pt(A \Rightarrow^* x)$, where $A \in N$ and $x \in \Delta^*$, $|x| \le 2^{depth(pt(A \Rightarrow^* x)) - 1}$.

Proof by induction on $depth(pt(A \Rightarrow^* x)) \geq 1$.

Basis. Let $depth(pt(A \Rightarrow^* x)) = 1$, where $A \in N$ and $x \in \Delta^*$. Because G is in Chomsky normal form, $A \Rightarrow^* x\ [A \rightarrow x]$ in G, where $x \in \Delta$, so $|x| = 1 \leq 2^{depth(pt(A \Rightarrow^* x)) - 1} = 1$.

Induction Hypothesis. Suppose that this lemma holds for all parse trees of depth m or less, for some positive integer n.

Induction Step. Let $depth(pt(A \Rightarrow^* x)) = n + 1$, where $A \in N$ and $x \in \Delta^*$. Let $A \Rightarrow^* x\ [r\rho]$ in G, where $r \in R$ and $\rho \in R^*$. As G is in Chomsky normal form, r: $A \rightarrow BC \in R$, where $B, C \in N$. Let $B \Rightarrow^* u\ [\pi]$, $C \Rightarrow^* v\ [\theta]$, $\pi, \theta \in R^*$, $x = uv$, $\rho = \pi\theta$ so that $A \Rightarrow^* x$ can be expressed in greater detail as $A \Rightarrow \underline{B}C\ [r] \Rightarrow^* u\underline{C}\ [\pi] \Rightarrow^* uv\ [\theta]$. Observe that $depth(pt(B \Rightarrow^* u)) \leq depth(pt(A \Rightarrow^* x)) - 1 = n$, so $|u| \leq 2^{depth(pt(B \Rightarrow^* u)) - 1}$ by the induction hypothesis. Analogously, as $depth(pt(C \Rightarrow^* v)) \leq depth(pt(A \Rightarrow^* x)) - 1 = n$, $|v| \leq 2^{depth(pt(C \Rightarrow^* v)) - 1}$. Thus, $|x| = |u| + |v| \leq 2^{depth(pt(B \Rightarrow^* u)) - 1} + 2^{depth(pt(C \Rightarrow^* v)) - 1} \leq 2^{n-1} + 2^{n-1} = 2^n = 2^{depth(pt(A \Rightarrow^* x)) - 1}$.

■

Corollary 3.47. Let $G = (\Sigma, R)$ be a grammar in the Chomsky normal form. For every parse tree $pt(A \Rightarrow^* x)$, where $A \in N$ and $x \in \Delta^*$, if $|x| \geq 2^m$ for some $m \geq 0$, then $depth(pt(A \Rightarrow^* x)) \geq m + 1$.

Proof. This corollary follows from Lemma 3.46 and the contrapositive law (see Section 1.1).

■

Lemma 3.48 ***Pumping Lemma for Context-Free Languages.*** Let L be an infinite context-free language. Then, there exists a positive integer, $k \geq 1$, such that every string $z \in L$ satisfying $|z| \geq k$ can be expressed as $z = uvwxy$, where $0 < |vx| < |vwx| \leq k$, and $uv^mwx^my \in L$, for all $m \geq 0$.

Proof. Let L be a context-free language, $L = L(G)$ for a grammar, $G = (\Sigma, R)$, in the Chomsky normal form. Set $k = 2^{card(N)}$. Let $z \in L(G)$ satisfying $|z| \geq k$. As $z \in L(G)$, $S \Rightarrow^* z$, and by Corollary 3.47, $depth(pt(S \Rightarrow^* z)) \geq card(N) + 1$, so $pt(S \Rightarrow^* z)$ contains some subtrees in which there is a path with two or more nodes labeled by the same nonterminal. Express $S \Rightarrow^* z$ as $S \Rightarrow^* uAy \Rightarrow^+ uvAxy \Rightarrow^+ uvwxy$ with $uvwxy = z$ so that the parse tree corresponding to $A \Rightarrow^+ vAx \Rightarrow^+ vwy$ contains no proper subtree with a path containing two or more different nodes labeled with the same nonterminal.

Claim A. $0 < |vx| < |vwx| \leq k$

Proof. As G is in the Chomsky normal form, every rule in R has on its right-hand side either a terminal or two nonterminals. Thus, $A \Rightarrow^+ vAx$ implies $0 < |vx|$, and $vAx \Rightarrow^+ vwy$ implies $|vx| < |vwx|$. As the parse tree corresponding to $A \Rightarrow^+ vAx \Rightarrow^+ vwy$ contains no proper subtree with a path containing two different nodes labeled with the same nonterminal, $depth(pt(A \Rightarrow^* vwx)) \leq card(N) + 1$, so by Lemma 3.46, $|vx| < |vwx| \leq 2^{card(N)} = k$.

Claim B. For all $m \geq 0$, $uv^mwx^my \in L$.

Proof. As $S \Rightarrow^* uAy \Rightarrow^+ uvAxy \Rightarrow^+ uvwxy$, $S \Rightarrow^* uAy \Rightarrow^+ uwy$, so $uv^0wx^0y = uwy \in L$. Similarly, since $S \Rightarrow^* uAy \Rightarrow^+ uvAxy \Rightarrow^+ uvwxy$, $S \Rightarrow^* uAy \Rightarrow^+ uvAxy \Rightarrow^+ uvvAxxy \Rightarrow^+ \ldots \Rightarrow^+ uv^mAx^my \Rightarrow^+ uv^mwx^my$, so $uv^mwx^my \in L$, for all $m \geq 1$.

Thus, Lemma 3.48 holds true.

■

Applications of the pumping lemma

We usually use the pumping lemma in a proof by contradiction to demonstrate that a given language L is not context-free. Typically, we make a proof of this kind as follows:

A. Assume that L is context-free.
B. Select a string $z \in L$ whose length depends on the pumping-lemma constant k so that $|z| \geq k$ is necessarily true.
C. For all possible decompositions of z into $uvwxy$ satisfying the pumping lemma conditions, find a non-negative integer m such that $uv^mwx^my \notin L$—a contradiction.
D. Make the conclusion that the assumption in A was incorrect, so L is not context-free.

Example 3.14 ***A Non-Context-Free Language.*** Consider $L = \{a^nb^nc^n | \ n \geq 1\}$. Although this language looks quite simple at a glance, no grammar can specify it because L is not context-free as proved next under the guidance of the recommended proof structure preceding this example.

A. Assume that L is context-free.
B. As L is context-free, there exists a natural number k satisfying Lemma 3.48. Set $z = a^kb^kc^k$ with $|z| = 3k \geq k$.
C. By Lemma 3.48, z can be written as $z = uvwxy$ so that this decomposition satisfies the pumping lemma conditions. As $0 < |vx| < |vwx| \leq k$, $vwx \in \{a\}^*\{b\}^*$ or $vwx \in \{b\}^*\{c\}^*$. If $vwx \in \{a\}^*\{b\}^*$, uv^0wx^0y has k cs but fewer than k as or bs, so $uv^0wx^0y \notin L$, but by the pumping lemma, $uv^0wx^0y \in L$—a contradiction. If $vwx \in \{b\}^*\{c\}^*$, uv^0wx^0y has k as but fewer than k bs or cs, so $uv^0wx^0y \notin L$, but by the pumping lemma, $uv^0wx^0y \in L$—a contradiction.
D. L is not context-free.

■

Omitting some obvious details, we usually proceed in a briefer way than above when proving the non-context-freedom of a language by using the pumping lemma.

Example 3.15 ***A Short Demonstration of Non-Context-Freedom.*** Let $L = \{a^nb^ma^nb^m | \ n, m \geq 1\}$. Assume that L is context-free. Set $z = a^kb^ka^kb^k$ with $|a^kb^ka^kb^k| = 4k \geq k$. By Lemma 3.48, express $z = uvwxy$. Observe that $0 < |vx| < |vwx| \leq k$ implies $uwy \notin L$ in all possible occurrences of vwx in $a^kb^ka^kb^k$; however, from the pumping lemma, $uwy \in L$—a contradiction. Thus, L is not context-free.

■

Even some seemingly trivial unary languages are not context-free as shown next.

Example 3.16 ***A Non-Context-Free Unary Language.*** Consider $K = \{a^i | \ i = n^2 \text{ for some } n \geq 0\}$. To demonstrate the non-context-freedom of K, assume that K is context-free and select $z = a^i \in K$ with $i = k^2$, where k is the pumping lemma constant. As a result, $|z| = k^2 \geq k$, so $z = uvwxy$, which satisfies the pumping-lemma conditions. As $k^2 < |uv^2wx^2y| \leq k^2 + k < k^2 + 2k + 1 = (k + 1)^2$, so $uv^2wx^2y \notin L$, but by Lemma 3.48, $uv^2wx^2y \in L$—a contradiction. Thus, K is not context-free.

■

Closure properties

Combined with the pumping lemma, the closure properties of context-free languages frequently significantly simplify a demonstration of the non-context-freedom of a language, L, in the following way. By contradiction, we first assume that L is context-free, and transform this language to a significantly simpler language, K, by using some operations under which the family of context-free languages is closed. Then, by the pumping lemma, we prove that K is not context-free, so neither is L, and the demonstration is completed.

Next, we discuss whether the family of context-free languages is closed under these operations:

- union
- concatenation
- closure
- intersection
- complement
- homomorphism

Union. To prove that the family of context-free languages is closed under union, we transform any two grammars to a grammar that generates the union of the languages generated by the two grammars.

Goal. Convert any two grammars, H and K, to a grammar G such that $L(G) = L(H) \cup L(K)$.

Gist. Consider any two grammars, $H = ({}_H\Sigma, {}_HR)$ and $K = ({}_K\Sigma, {}_KR)$. Without any loss of generality, suppose that ${}_HN \cap {}_KN = \emptyset$ (if ${}_HN \cap {}_KN \neq \emptyset$, rename the nonterminals in either H or K so that ${}_HN \cap {}_KN = \emptyset$ and the generated language remains unchanged). $G = ({}_G\Sigma, {}_GR)$ contains all rules of H and K. In addition, we include ${}_GS \to {}_HS$ and ${}_GS \to {}_KS$, where ${}_GS \notin {}_HN \cup {}_KN$. If G generates $x \in L(G)$ by a derivation starting with ${}_GS \to {}_HS$, then $x \in L(H)$. Analogically, if G generates $x \in L(G)$ by a derivation starting with ${}_GS \to {}_KS$, then $x \in L(K)$. Thus, $L(G) = L(H) \cup L(K)$.

Algorithm 3.49 ***Grammar for the Union of Context-Free Languages.***

Input • two grammars, $H = ({}_H\Sigma, {}_HR)$ and $K = ({}_K\Sigma, {}_KR)$, such that ${}_HN \cap {}_KN = \emptyset$.

Output • a grammar G such that $L(G) = L(H) \cup L(K)$.

Method

```
begin
    construct $G = ({}_G\Sigma, {}_GR)$ with ${}_G\Sigma = {}_H\Sigma \cup {}_K\Sigma \cup \{{}_GS\}$, where ${}_GS \notin {}_HN \cup {}_KN$, and
    ${}_GR = {}_HR \cup {}_KR \cup \{{}_GS \to {}_HS, {}_GS \to {}_KS\}$
end.
```

Theorem 3.50. The family of context-free languages is closed under union.

Proof. Observe that Algorithm 3.49 correctly converts any two grammars, $H = ({}_H\Sigma, {}_HR)$ and $K = ({}_K\Sigma, {}_KR)$, where ${}_HN \cup {}_KN = \emptyset$, to a grammar, G, such that $L(G) = L(H) \cup L(K)$. Thus, this theorem holds true.

■

Theorem 3.51. The family of context-free languages is closed under concatenation and closure.

Intersection. Recall that the family of regular languages is closed under intersection (see Theorem 2.43). Surprisingly, concerning the family of context-free languages, this result does not hold.

Theorem 3.52. The family of context-free languages is not closed under intersection.

Proof. Consider these two context-free languages, $A = \{a^ib^jc^k \mid i, j, k \geq 1 \text{ and } i = j\}$ and $B = \{a^ib^jc^k \mid i, j, k \geq 1 \text{ and } j = k\}$. From Example 3.14, $A \cap B = \{a^ib^ic^i \mid i \geq 1\}$ is not context-free.

■

However, the intersection of a context-free language and a regular language is always context-free. Recall that every context-free language is accepted by an one-state pushdown automaton without ε-rules (see Corollary 3.44), and every regular language is accepted by a deterministic finite automaton (see Theorems 2.28 and 2.37). We use these two models to demonstrate that the family of context-free languages is closed under intersection with regular languages.

Goal. Convert any deterministic finite automaton X and any one-state pushdown automaton Y without ε-rules to a pushdown automaton, M, such that $L(M) = L(X) \cap L(Y)$.

Gist. Let $X = ({}_X\Sigma, {}_XR)$ and $Y = ({}_Y\Sigma, {}_YR)$ be any deterministic finite automaton and any one-state pushdown automaton without ε-rules, respectively; recall that Y's only state is denoted by ◆ (see Convention 3.8). M's states, final states, and the start state coincide with X's states, final states, and the start state, respectively. For every $qa \to o \in {}_XR$ and every $A◆a \to x◆ \in {}_YR$, add $Aqa \to xo$ to M's rules in order to simulate a move by $qa \to o$ in X and, simultaneously, a move by $A◆a \to x◆$ in Y. In this way, M simultaneously simulates the sequences of moves in X and Y, so they both result into acceptance if and only if M accepts; in other words, $L(M) = L(X) \cap L(Y)$.

Algorithm 3.53 ***Pushdown Automaton for the Union of a Context-Free Language and a Regular Language.***

Input • a deterministic finite automaton, $X = ({}_X\Sigma, {}_XR)$;
• a one-state pushdown automaton $Y = ({}_Y\Sigma, {}_YR)$ without ε-rules.

Output • a pushdown automaton $M = ({}_M\Sigma, {}_MR)$ without ε-rules such that $L(M) = L(X) \cap L(Y)$.

Method

```
begin
    set $_M\Sigma = {}_MQ \cup {}_M\Gamma \cup {}_M\Delta \cup \{▶, ◀\}$ with
    $_MQ = {}_XQ$, $_Ms = {}_Xs$, $_MF = {}_XF$, $_M\Gamma = {}_Y\Gamma$, $_MS = {}_YS$, $_M\Delta = {}_X\Delta \cap {}_Y\Delta$, and
    $_MR = \{Aqa \to xo \mid qa \to o \in {}_XR, q, o \in {}_XQ, a \in {}_X\Delta, A◆a \to x◆ \in {}_YR, A, x \in {}_Y\Gamma^*\}$
end.
```

Theorem 3.54. The family of context-free languages is closed under the intersection with regular languages.

Complement. Making use of Theorem 3.50, we can easily prove that the family of context-free languages is not closed under complement.

Theorem 3.55. The family of context-free languages is not closed under complement.

Proof by contradiction. Assume that the family of context-free languages is closed under complement. Consider these two context-free languages, $A = \{a^ib^jc^k \mid i, j, k \geq 1 \text{ and } i = j\}$ and $B = \{a^ib^jc^k \mid i, j, k \geq 1 \text{ and } j = k\}$. Theorem 3.50, *complement*(*complement*(A) ∪ *complement*(B)), is context-free. However, by DeMorgan's laws (see Section 1.1), *complement*(*complement*(A) ∪ *complement*(B)) $= A \cap B = \{a^ib^ic^i \mid i \geq 1\}$, which is not context-free (see Example 3.14)—a contradiction.

■

Homomorphism. Next, we prove that the family of context-free languages is closed under homomorphism. That is, we demonstrate that for every grammar $G = ({}_G\Sigma, {}_GR)$ in Chomsky normal form, every alphabet Ξ and every homomorphism h from ${}_G\Delta^*$ to Ξ^*, there is a grammar $H = ({}_H\Sigma, {}_HR)$ such that $L(H) = h(L(G))$. As a result, the desired closure property holds true because every context-free language is generated by a grammar in Chomsky normal form (see Theorem 3.38 and Algorithm 3.40).

Goal. Convert any grammar $G = ({}_G\Sigma, {}_GR)$ in Chomsky normal form and any homomorphism h from ${}_G\Delta^*$ to Ξ^*, where Ξ is an alphabet, to a grammar $H = ({}_H\Sigma, {}_HR)$ such that $L(H) = h(L(G))$.

Gist. Without any loss of generality, assume that ${}_GN \cap \Xi = \emptyset$. Set ${}_H\Sigma$ to ${}_GN \cup \Xi$. As G is in Chomsky normal form, ${}_GR$ has on its right-hand side either a terminal or two nonterminals. Move every rule that has two nonterminals on its right-hand side from ${}_GR$ to ${}_HR$. For every rule $A \rightarrow a \in {}_GR$, add $A \rightarrow h(a)$ to ${}_HR$. As a result, $a_1a_2\dots a_n \in L(G)$ if and only if $h(a_1)\, h(a_2)\dots h(a_n) \in L(H)$, so $L(H) = h(L(G))$.

Algorithm 3.56 ***Grammar for Homomorphism*.**

Input
- a grammar $G = ({}_G\Sigma, {}_GR)$ in Chomsky normal form;
- a homomorphism h from ${}_G\Delta^*$ to Ξ^*, where Ξ is an alphabet, ${}_GN \cap \Xi = \emptyset$.

Output
- a grammar, $H = ({}_H\Sigma, {}_HR)$, such that $L(H) = h(L(G))$.

Method

```
begin
    set $_H\Sigma = {}_GN \cup \Xi$ and
    $_HR = \{A \rightarrow h(a) | A \rightarrow a \in {}_GR, A \in {}_GN, a \in {}_G\Delta\}$
    $\cup \{A \rightarrow BC | A \rightarrow BC \in {}_GR, A, B, C \in {}_GN\}$
end.
```

Theorem 3.57. The family of context-free languages is closed under homomorphism.

■

Applications of the closure properties

Combined with the pumping lemma, we also use the closure properties to prove that a language L is not context-free. A proof of this kind is made by contradiction, and its typical structure follows next.

A. Assume that L is context-free.
B. From L and some context-free languages, construct a new language K by using closure operations under which the family of context-free languages is closed, so under the assumption that L is context-free, so is K.
C. By the pumping lemma for context-free languages, prove that K is not context-free, which contradicts B.
D. Conclude that L is not context-free.

Example 3.17 ***A Non-Context-Free Language*.** Let $L = \{ww | w \in \{a, b\}^+\}$. Assume that L is context-free. As $\{a\}^+\{b\}^+\{a\}^+\{b\}^+$ is regular, $L \cap \{a\}^+\{b\}^+\{a\}^+\{b\}^+ = \{a^nb^ma^nb^m | n, m \geq 1\}$

represents a context-free language. However, $\{a^nb^ma^nb^m \mid n, m \geq 1\}$ is not a context-free language (see Example 3.15), so neither is L.

■

For a common high-level programming language, such as Pascal, consider the language consisting of all well-formed source programs rather than their tokenized versions. As a rule, by using the closure properties, we can easily demonstrate that this language is not context-free.

Example 3.18 ***Pascal is not Context-Free.*** Let *Pascal* denote the language consisting of all well-formed Pascal programs. Assume that *Pascal* is context-free. Let L denote this regular language

{**program**}{□}{p}{□}{(}{**input**}{)}{;}{**var**}{□}$\{X,Y\}^+${:**integer**}{;}{**begin**}{}$\{X,Y\}^+${:=0} {□}{**end**}{.}

where □ represents a blank. Let h be the homomorphism defined as $h(X) = X$, $h(Y) = Y$, and $h(Z) = \varepsilon$, where Z denotes any symbol from the Pascal alphabet but X or Y. Pascal requires that all identifiers are declared; on the other hand, this programming language places no limit on their length. As a result, $h(Pascal \cap L) = \{ww \mid w \in \{X, Y\}^+\}$. Introduce the homomorphism g from $\{X, Y\}^*$ to $\{a, b\}^*$ defined as $g(X) = a$ and $g(Y) = b$. By Theorem 3.57, $g(\{ww \mid w \in \{X,Y\}^+\}) = \{ww \mid w \in \{a, b\}^+\}$ is context-free. However, Example 3.17 proves that $\{ww \mid w \in \{a, b\}^+\}$ is not context-free—a contradiction. Thus, *Pascal* is not context-free.

■

The argument used in the previous example can be applied, in principle, to every programming language that requires the declaration of all identifiers whose length is unbounded. As this is true in most programming languages, there arises a question how we can handle their analysis based on models for context-free languages, such as grammars. However, as already pointed out, the above argument is applicable to the untokenized versions of programming languages. As syntax analysis works with their tokenized versions produced by lexical analysis, many non-context-free aspects of programming languages are removed in this way. Unfortunately, even the tokenized versions may contain some language features that grammars fail to capture. For instance, these grammars cannot guarantee that the number of formal parameters coincides with the number of the corresponding actual parameters, or they fail to detect the reference to an undeclared identifier. Fortunately, the semantic analysis can handle features like these by using the symbol tables (see Section 6.3). Nonetheless, we should always avoid introducing any useless non-context-freedom into a newly designed programming language. As the next example illustrates, sometimes this avoidance requires an extreme caution even during the design of seemingly simple syntax constructs.

Example 3.19 ***A Non-Context-Free Syntax Construct.*** Suppose that when designing a new programming language, we introduce a command that simultaneously sets several variables to their values in the way illustrated by this example

$$index\ time\ length = 2\ 9.15\ 8$$

which sets *index*, *time*, and *length* to 2, 9.15, and 8, respectively. To give its general description, let **r**, **i**, ¢, and # be tokens representing the real variables, the integer variables, the real numbers, and the integer numbers, respectively; we define the tokenized version of this command as

$$x_1, x_2, \ldots, x_n = y_1, y_2, \ldots, y_n$$

where $n \geq 1$, $x_j \in \{\mathbf{r}, \mathbf{i}\}$, $y_j \in \{\#, ¢\}$, $x_j = \mathbf{r}$ if and only if $y_j = ¢$, and $x_j = \mathbf{i}$ if and only if $y_j = \#$, $1 \leq$

$j \le n$. Let L denote the set of all these commands. Introduce the homomorphism g from $\{\mathbf{r}, \mathbf{i}, ¢, \#, =\}^*$ to $\{a, b\}^*$ as $g(=) = \varepsilon$, $g(\mathbf{r}) = g(¢) = a$, and $g(\mathbf{i}) = g(\#) = b$. Notice that $g(L) = \{ww | \ w \in \{a, b\}^+\}$. By Theorem 3.57, $g(L)$ is context-free, which is not the case by Example 3.17. Thus, L is non-context-free.

■

Demonstration of Context-Freedom. Before we close this section, let us note that the closure properties are also frequently used in a positive way to prove that a seemingly complicated language is context-free. Suppose that a language L is syntactically complex; nevertheless, we believe that it is context-free. A direct proof of its context-freedom usually consists in a design of a grammar and a rigorous verification that this grammar generates L, which is often a tedious and difficult task. To simplify this task, we introduce several simple context-free languages and combine them by operations under which the family of context-free languages is closed so that the language resulting from this combination is L; as a result, L is context-free.

Example 3.20 ***A Proof that a Language is Context-Free.*** Consider the non-context-free language $L = \{a^nb^nc^n | \ n \ge 1\}$ (see Example 3.14). To demonstrate that its complement is context-free, express *complement*(L) = $A \cup B \cup C$, where $A = \{a, b, c\}^* - \{a\}^+\{b\}^+\{c\}^+$, $B = \{a^ib^jc^k | \ i, j, k \ge 1, i \ne j\}$, and $C = \{a^ib^jc^k | \ i, j, k \ge 1, j \ne k\}$. As A, B, and C are obviously context-free, so is *complement*(L) by Theorem 3.50.

■

3.3.6 Decidable Problems

The present section discusses the following four problems relevant to parsing:

- membership problem
- emptiness problem
- infiniteness problem
- finiteness problem

That is, we give algorithms that decide these problems in terms of these automata. Each algorithm contains Boolean variable *answer*. With one of the allowable instances as its input, the algorithm sets *answer* to **true** to express the affirmative answer to the question in this instance; otherwise, it sets *answer* to **false**. Since all these algorithm are simple, we present them less formally than the algorithms given in the previous parts of this chapter.

Convention 3.58. Throughout the rest of this section, we automatically assume that each grammar is in Chomsky normal form (see Definition 3.39).

■

Membership problem. This problem has already been by four parsing algorithms—Algorithms 3.12, 3.16, 3.43, and 3.45, which decide whether $w \in L(G)$ for any context-free grammar G and any string w. Based on Lemma 3.46, the following algorithm solves this problem for grammars in Chomsky normal form even in a simpler way (as the grammars in this form do not generate ε, we rule out ε as an input word). The algorithm is based upon the property that a grammar G generates w by making $2|w| - 1$ or fewer derivation steps if G is in Chomsky normal form.

Membership problem for grammars.
Instance: A grammar, $G = (\Sigma, R)$, in Chomsky normal form and a string $w \in \Delta^+$.
Question: Is w a member of $L(G)$?

Algorithm 3.59 ***Decision of Membership Problem for Grammars.*** Search the set of all sentences that G generates by making no more than $2|w| - 1$ derivation steps. If this set contains w, set *answer* to **true**; otherwise, set *answer* to **false**.

Emptiness problem. If a grammar generates no sentence, it obviously wrongly designed. As a result, an algorithm that decides whether a grammar generates $\varnothing$ is highly relevant to parsing.

Emptiness problem for grammars.
***Instance*:** A grammar, $G = (\Sigma, R)$.
***Question*:** Is $L(G)$ empty?

Algorithm 3.60 ***Decision of Emptiness Problem for Grammars.*** If G's start symbol S is terminating (see Algorithm 3.26), set *answer* to **false**; otherwise, set *answer* to **true**.

Infiniteness problem. As most programming languages are infinite, an algorithm that decides whether a grammar generates an infinite language is highly relevant to the grammatically based syntax specification.

Infiniteness problem for grammars.
***Instance*:** A grammar, $G = (\Sigma, R)$.
***Question*:** Is $L(M)$ infinite?

Set $k = 2^{card(N)}$. As an exercise, we prove that $L(G)$ is infinite if and only if $L(G)$ contains a sentence x such that $k \le |x| < 2k$. This equivalence underlies the next two algorithms.

Algorithm 3.61 ***Decision of Infiniteness Problem for Grammars.*** If G generates a sentence x such that $k \le |x| < 2k$, set *answer* to **true**; otherwise, set *answer* to **false**.

Finiteness problem for grammars.
***Instance*:** A grammar $G = (\Sigma, R)$.
***Question*:** Is $L(M)$ finite?

Algorithm 3.62 ***Decision of Finiteness Problem for Grammars.*** If G does not generate a sentence x such that $k \le |x| < 2k$, set *answer* to **true**; otherwise, set *answer* to **false**.

Exercises

3.1. Incorporate more common programming language constructs, such as procedures and case statements, into the programming language FUN. Extend *FUN-GRAMMAR* by rules for these newly incorporated constructs (see Case Study 5/35 *Grammar*).

3.2. Design a graphical representation for pushdown automata and transducers.

3.3. Consider grammar ${}_{cond}H$ (see Case Study 6/35). Recall its rules

$$E \to E \vee T,\ E \to T,\ T \to T \wedge F,\ T \to F,\ F \to i,\ F \to (E)$$

Prove that ${}_{cond}H$ is unambiguous. Take $(i \vee i) \wedge (i \wedge i)$. Make the rightmost derivation, the leftmost derivation, and the parse tree of this string in ${}_{cond}H$. Consider $\neg$ as a unary operator that denotes a logical negation. Extend ${}_{cond}H$ so it also generates the Boolean expressions containing $\neg$. Make this extension so the resulting extended grammar is again unambiguous.

3.4. Consider this grammar $S \to aSbS$, $S \to bSaS$, $S \to \varepsilon$. Define the language generated by this grammar. Is this grammar ambiguous? Justify your answer. If it is ambiguous, is there an unambiguous grammar equivalent to this grammar?

3.5. Consider the grammar defined by rules $S \to SS$, $S \to (S)$, $S \to \varepsilon$. Describe the language that this grammar generates. Is this grammar unambiguous?

3.6. Consider each of the four following grammars. Determine the language it generates. Prove or disprove that the grammar is ambiguous. If the grammar is ambiguous, change it to an equivalent unambiguous grammar. If the grammar is unambiguous, change it to an equivalent grammar that may be ambiguous but contains fewer rules or nonterminals.

(a) ⟨start⟩ → ⟨if-statement⟩
⟨if-statement⟩ → **if** ⟨expression⟩ **then** ⟨statement⟩ **else** ⟨if-statement⟩
⟨if-statement⟩ → **if** ⟨expression⟩ **then** ⟨if-statement⟩
⟨if-statement⟩ → ⟨ statement⟩
⟨statement⟩ → *s*
⟨expression⟩ → *e*

(b) ⟨start⟩ → ⟨statement⟩
⟨statement⟩ → **if** ⟨expression⟩ **then** ⟨statement⟩ **else** ⟨statement⟩
⟨statement⟩ → **if** ⟨expression⟩ **then** ⟨statement⟩
⟨statement⟩ → *s*
⟨expression⟩ → *e*

(c) ⟨start⟩ → ⟨if-statement⟩
⟨if-statement⟩ → **if** ⟨expression⟩ **then** ⟨statement⟩ **else** ⟨if-statement⟩
⟨if-statement⟩ → **if** ⟨expression⟩ **then** ⟨statement⟩
⟨if-statement⟩ → *s*
⟨expression⟩ → *e*

(d) ⟨start⟩ → ⟨if-statement⟩
⟨if-statement⟩ → **if** ⟨expression⟩ **then** ⟨if-statement⟩ **else** ⟨statement⟩
⟨if-statement⟩ → **if** ⟨expression⟩ **then** ⟨if-statement⟩
⟨if-statement⟩ → ⟨ statement⟩
⟨statement⟩ → *s*
⟨expression⟩ → *e*

3.7. Consider each of the four grammars discussed in Exercise 3.6. Construct an equivalent pushdown automaton.

3.8. Consider the language of all arithmetic expressions in the Polish prefix notation with an operand *i* and operators + and * (see Exercise 1.16 and its solution). Construct a context-free grammar that generates this language. Construct a deterministic pushdown automaton that accepts this language.

3.9. By analogy with Case Study 8/35 *Recursive-Descent Parser*, design a recursive descent parser based on this grammar

⟨start⟩ → ⟨statements⟩ **finish**
⟨statements⟩ → ⟨statement⟩⟨more statements⟩

$\langle$more statements$\rangle \rightarrow$; $\langle$statements$\rangle$
$\langle$more statements$\rangle \rightarrow \square$
$\langle$statement$\rangle \rightarrow$ **set** *i* = $\langle$expression$\rangle$
$\langle$statement$\rangle \rightarrow$ **read** *i*
$\langle$statement$\rangle \rightarrow$ **write** $\langle$expression$\rangle$
$\langle$expression$\rangle \rightarrow \langle$term$\rangle$ + $\langle$expression$\rangle$
$\langle$expression$\rangle \rightarrow \langle$term$\rangle$ – $\langle$expression$\rangle$
$\langle$expression$\rangle \rightarrow \langle$term$\rangle$
$\langle$term$\rangle \rightarrow \langle$factor$\rangle$ * $\langle$term$\rangle$
$\langle$term$\rangle \rightarrow \langle$factor$\rangle$ / $\langle$term$\rangle$
$\langle$term$\rangle \rightarrow \langle$factor$\rangle$
$\langle$factor$\rangle \rightarrow$ ($\langle$expression$\rangle$)
$\langle$factor$\rangle \rightarrow$ *i*

3.10. Write a program to implement the parser obtained in Exercise 3.9. Test it on several programs, including

read *k*; **read** *l*; **set** *l* = *k* * (*l* + *k*); **write** *l* – *k* **finish**

3.11. Prove that Algorithm 3.12 *Top-Down Parser for a Grammar* is correct.

3.12. Write a program to implement Case Study 8/35 *Recursive-Descent Parser*.

3.13. Write a program to implement Algorithm 3.15 *Elimination of Immediate Left Recursion*. Test it on ${}_{cond}G$ (see Case Study 6/35).

3.14. Design an algorithm to eliminate all left recursive nonterminals from a grammar (see the note that follows Case Study 9/35). Write a program to implement the resulting algorithm.

3.15. Write a program to implement Algorithm 3.16 *Bottom-Up Parser for a Grammar*.

3.16. Reformulate Case Study 10/35 *Bottom-Up Parser* in terms of ${}_{cond}G$ (see Case Study 6/35).

3.17. Prove Theorems 3.18 and 3.19 in a rigorous way.

3.18. Write a program to implement

(a) Algorithm 3.26 *Terminating Symbols*;
(b) Algorithm 3.28 *Accessible Symbols*;
(c) Algorithm 3.30 *Useful Symbols*.

Test the program that implements (c) on the grammar discussed in Example 3.8 *Useful Symbols*. In addition, make several grammars, containing many symbols, some of which are useless. Test the program on these grammars as well.

3.19. Write a program to implement

(a) Algorithm 3.33 ε-*Nonterminals*;
(b) Algorithm 3.34 *Grammar without* ε-*Rules*;
(c) Algorithm 3.36 *Grammar without Unit Rules*.

3.20. In a fully rigorous way, prove Theorem 3.38, which states that for every grammar, *G*, there exists an equivalent proper grammar, *H*.

3.21. Write a program to implement Algorithm 3.40 *Chomsky Normal Form*. Test this program on the grammars discussed in Case Study 6/35.

3.22. Design an algorithm that transforms any proper grammar to an equivalent grammar satisfying the Greibach normal form. Write a program to implement the resulting algorithm. Test this program on the grammars discussed in Case Study 6/35.

3.23. Write a program to implement Algorithm 3.43 *Top-Down Parser for a Grammar in Greibach Normal Form*.

3.24. Write a program to implement Algorithm 3.45 *Cocke-Younger-Kasami Parsing Algorithm*. Test this program on the grammars obtained in Exercise 3.21.

3.25. In the proof of Lemma 3.48, we assume that *G* is in Chomsky normal form. Prove this lemma without this assumption.

3.26. Write a program to implement

(a) Algorithm 3.49 *Grammar for the Union of Context-Free Languages*;
(b) Algorithm 3.53 *Pushdown Automaton for the Union of a Context-Free Language and a Regular Language*;
(c) Algorithm 3.56 *Grammar for Homomorphism*.

3.27. Consider Theorem 3.51, which states that the family of context-free languages is closed under concatenation and closure. Prove this theorem in detail.

3.28. Consider the proof of Theorem 3.52, which states that the family of context-free languages is not closed under intersection. Give an alternative proof of this result.

3.29. Return to the proof of Theorem 3.54, which states that the family of context-free languages is not closed under complement. Prove this result in a different way.

3.30. Return to the algorithms that decide various problems concerning parsing in Section 3.3.6. We have only sketched them and omitted their verification. Describe these algorithms precisely and give rigorous proofs that they work correctly.

3.31. Consider each of the following languages over $\{a, b\}$. Construct a grammar that generates the language. Can you also construct a deterministic pushdown automaton that accepts the language? If so, make the construction. If not, explain why this construction is impossible.

(a) $\{x \mid x \in \Sigma^*, occur(x, a) < occur(x, b)\}$
(b) $\{x \mid x \in \Sigma^*, occur(x, a) = 2occur(x, b)\}$
(c) $\{xy \mid x, y \in \Sigma^*, y = reversal(x)\}$

3.32*Solved*. There exist various definitions of pushdown automata and their languages in the literature. Indeed, in theoretical computer science, a pushdown automaton, $M = ({}_M\Sigma, {}_MR)$, is usually defined without ▶ and ◀, and during every move, *M* rewrites precisely one symbol on the pushdown top. On the other hand, *M* may have several states, not just a single state, ◆. Furthermore, the language that *M* accepts is sometimes defined by emptying its pushdown without necessarily entering a final state. More specifically, *M accepts an input string w by empty pushdown* if it makes a sequence of moves from Ssw to q, where S is M's start symbol, s is the start state, and q is a state. The set of all strings accepted in this way is *the language M accepts by empty pushdown*, denoted by ${}_{\varepsilon}L(M)$. Formalize the definition of M and ${}_{\varepsilon}L(M)$ quite rigorously.

3.33. Let L be any language. Demonstrate that $L = {}_{\varepsilon}L(M)$, where M is defined in the way described in Exercise 3.32, if and only if $L = L(O)$, where O is a pushdown automata according to Definition 3.7.

3.34*Solved*. Design an algorithm that converts any pushdown automaton, $M = ({}_{M}\Sigma, {}_{M}R)$, defined in the way given in Exercise 3.32 to a grammar, $G = ({}_{G}\Sigma, {}_{G}R)$, such that $L(G) = {}_{\varepsilon}L(M)$. Give a rigorous proof that this algorithm works correctly.

3.35*Solved*. Consider the pushdown automaton M defined by these rules $Ssa \rightarrow Sas$, $asa \rightarrow aas$, $asb \rightarrow q$, $aqb \rightarrow q$, $Sq \rightarrow q$. Observe that ${}_{\varepsilon}L(M) = \{a^nb^n \mid n \geq 1\}$. By using the algorithm designed in Exercise 3.34, convert M to a grammar, G, such that $L(G) = {}_{\varepsilon}L(M)$.

3.36. Consider the following languages. Use the pumping lemma and closure properties to demonstrate that none of them is context-free.

(a) $\{a^ib^jc^k \mid i, j, k \geq 0 \text{ and } i < j < k\}$
(b) $\{a^ib^j \mid i, j \geq 0 \text{ and } j = i^2\}$
(c) $\{a^i \mid i \text{ is a prime}\}$
(d) $\{w \mid w \in \{a, b, c\}^* \text{ and } \#_aw = \#_bw = \#_cw\}$
(e) $\{a^ib^ic^j \mid i, j \geq 0 \text{ and } j \neq i\}$
(f) $\{a^ib^ic^j \mid i, j \geq 0 \text{ and } i \neq j \neq 2i\}$
(g) $\{a^ib^jc^k \mid i, j, k \geq 0, i \neq j, k \neq i, \text{ and } j \neq k\}$
(h) $\{a^ib^jc^jd^i \mid i, j \geq 0 \text{ and } j \neq i\}$
(i) $\{a^ib^{2i}c^i \mid i \geq 0\}$
(j) $\{ww \mid w \in \{a, b\}^*\}$
(k) $\{ww^Rw \mid w \in \{a, b\}^*\}$
(l) $\{0^i10^i10^i10^i \mid i \geq 1\}$
(m) $\{wcv \mid v, w \in \{a, b\}^* \text{ and } w = vv\}$

3.37. Consider each of the context-free languages discussed in Exercise 2.23. Select a non-context-free subset of the language. Prove that the selected subset is not context-free by the pumping lemma. For instance, consider $\{a^ib^jc^k \mid i, j, k \geq 0 \text{ and } i = j \text{ or } j = k\}$. Observe that this language contains $\{a^nb^nc^n \mid n \geq 1\}$ as its subset. By using the pumping lemma, Example 6.1.2 has already proved that $\{a^nb^nc^n \mid n \geq 1\}$ represents a non-context-free language.

3.38. Based on the Greibach normal form, give an alternative proof of Lemma 3.48 *Pumping Lemma for Context-Free Languages.* Then, prove this lemma based on the pushdown automata.

3.39. Let $G = (\Sigma, R)$ be a proper context-free grammar. A nonterminal, $A \in N$, is *recursive* if and only if $A \Rightarrow^* uAv$, for some $uv \in \Delta^+$. Give a formal proof of the following theorem.

Theorem 3.63. Let $G = (\Sigma, R)$ be a proper context-free grammar. Then, $L(G)$ is infinite if and only if N contains a recursive nonterminal.

3.40. Prove the next lemma.

Lemma 3.64 ***the Ogden Lemma.*** Let L be a context-free language over an alphabet, Δ. Then, there exists a natural number, k, such that if for some $n \geq k$, $z_0a_1z_1a_2z_2\ldots a_nz_n \in L$, where $z_i \in \Delta^*$ for all $i = 0, \ldots, n$, and $a_j \in \Delta$ for all $j = 0, \ldots, n$, then $z_0a_1z_1a_2z_2\ldots a_nz_n$ can be written as $z_0a_1z_1a_2z_2\ldots a_nz_n = uvwxy$ where (1) for some $t = 1, \ldots, n$, a_t appears in vx; (2) if vwx contains $a_r, \ldots, a_s$, for some $r = 1, \ldots, n$, and $s = r, \ldots, n$, then $s - r \leq k - 1$; (3) $uv^mwx^my \in L$, for all $m \geq 0$.

3.41. Consider the following languages:

(a) $\{a^ib^jc^kd^l \mid i, j, k, l \geq 0$, and either $i = 0$ or $j = k = l\}$
(b) $\{a^ib^jc^k \mid i, j, k \geq 0, i \neq j, k \neq i$, and $j \neq k\}$

Use the Ogden lemma to prove that neither of these languages is context-free (see Lemma 3.64 in Exercise 3.40).

3.42. Consider each of the operations defined in Exercise 2.30. Prove or disprove that the family of languages accepted by pushdown automata is closed with respect to the operation under consideration.

3.43. For a language, L, over an alphabet, Δ, and a symbol, $a \in \Delta$, $_aEraser(L)$ is the language obtained by deleting all as in the strings from L. Formalize $_aEraser(L)$. Prove that if L is a context-free language, then so is $_aEraser(L)$.

3.44. Introduce several language operations under which the family of languages generated by grammars is not closed with respect to the operation under consideration. For each operation, give a specific grammar, G, such that the operation applied to $L(G)$ results into a non-context-free language.

3.45. Consider the language consisting of all well-written programs in your favorite programming language, such as Java. Prove or disprove that the language under consideration is context-free.

3.46. Select a subset of a natural language, such as English or your native language. Prove or disprove that the language under consideration is context-free.

3.47. Consider the following problems and design algorithms that decide these problems.

Recursive nonterminal problem for context-free grammars.
***Instance*:** A context-free grammar, $G = (\Sigma, R)$, and $A \in N$.
***Question*:** Is A recursive?

Emptiness problem for pushdown automata.
***Instance*:** A pushdown automaton, $M = (\Sigma, R)$.
***Question*:** Is $L(M)$ empty?

Derivation problem for context-free grammars.
***Instance*:** A context-free grammar, $G = (\Sigma, R)$, and $v, w \in \Sigma^*$.
***Question*:** Is it true that vw is in $L(G)$?

3.48. Consider

Definition 3.65 ***Linear Grammar.*** Let $G = (\Sigma, R)$ be a grammar (see Definition 3.1). G is a *linear grammar* if each rule $A \rightarrow y \in R$ satisfies $y \in \Delta^*(N \cup \{\varepsilon\})\Delta^*$.

■

Consider the family of languages generated by linear grammars. This family is properly included in the family of context-free languages, but it properly includes the family of regular languages. Prove this result.

3.49. Consider

Definition 3.66 ***General Grammar and its Language.*** A *general grammar* is a rewriting system, $G = (\Sigma, R)$, where

- Σ is divided into two disjoint subalphabets, denoted by N and Δ;
- R is a finite *set of rules* of the form $x \rightarrow y$, where $x \in \Sigma^*N\Sigma^*$ and $y \in \Sigma^*$.

N and Δ are referred to as the *alphabet of nonterminal symbols* and the *alphabet of terminal symbols,* respectively. N contains a special nonterminal called the *start symbol,* denoted by S.

If $S \Rightarrow^* w$, where $w \in \Sigma^*$, *G derives w*. $F(G)$ denotes the set of all sentential forms derived by G. The *language generated by G*, symbolically denoted by $L(G)$, is defined as $L(G) = F(G) \cap \Delta^*$.

■

Explain the fundamental difference between the grammars according to Definition 3.1 and the general grammars according to Definition 3.66. Prove that the grammars according to Definition 3.1 are less powerful than the general grammars.

3.50. Consider each language in Exercise 3.36. Construct a general grammar that generates the language.

3.51. Consider the pushdown automata (see Definition 3.7). Extend them to *two-pushdown automata* by adding another pushdown. Formalize this extension. Prove that the two-pushdown automata are as powerful as the general grammars (see Definition 3.65).

3.52. Consider a pushdown automaton, $M = (\Sigma, R)$. If $card(\Gamma) = 1$, M is said to be a *counter*. Compare the power of pushdown automata and the power of counters.

Solution to Selected Exercises

3.32. Define a pushdown automaton as a rewriting system, $M = (\Sigma, R)$, where

- Σ is divided into subalphabets Q, Γ, Δ, such that $Q \cap (\Gamma \cup \Delta) = \emptyset$;
- R is a finite *set of rules* of the form $x \rightarrow y$, where $x \in \Gamma Q(\Delta \cup \{\varepsilon\})$ and $y \in \Gamma^*Q$.

Q, Γ, and Δ have the same meaning as in Definition 3.7. Like in Definition 3.7, define $u \Rightarrow v$, $u \Rightarrow^n v$ for $n \geq 0$, and $u \Rightarrow^* v$, where $u, v \in \Sigma^*$. Set ${}_{\varepsilon}L(M) = \{w | Ssw \Rightarrow^* q \text{ in } M, \text{ where } q \in Q\}$, where S is the start symbol and s is the start state (notice that q may not be final).

3.34. Recall that this exercise is solved in terms of pushdown automata defined in Exercise 3.32.

Goal. Given a pushdown automaton, $M = ({}_M\Sigma, {}_MR)$, construct a grammar, $G = ({}_G\Sigma, {}_GR)$, such that $L(G) = {}_{\varepsilon}L(M)$.

Gist. Consider a pushdown automaton, $M = ({}_M\Sigma, {}_MR)$, defined in the way given in Exercise 3.32. For all states $p, q \in {}_MQ$, and all pushdown symbols $A \in {}_M\Gamma$, introduce a nonterminal $\langle pAq \rangle$ into

${}_GN$. Construct ${}_GR$'s rules so $\langle pAq\rangle\ {}_{lm}\Rightarrow^* w$ in G if and only if $Apw \Rightarrow^* q$, where $w \in {}_M\Delta^*$. In addition, add ${}_GS \rightarrow \langle {}_Ms_MSo\rangle$ to ${}_GR$, for all $o \in {}_MQ$. Thus, ${}_GS \Rightarrow \langle {}_Ms_MSo\rangle\ {}_{lm}\Rightarrow^* w$ in G if and only if ${}_Ms_Msw \Rightarrow^* o$ in M, so $L(G) = L(M)$.

Algorithm 3.67 ***Grammar for a Pushdown Automaton.***

Input: • a pushdown automaton $M = ({}_M\Sigma, {}_MR)$.

Output: • a grammar $G = ({}_G\Sigma, {}_GR)$ such that $L(G) = {}_\varepsilon L(M)$.

Method

```
begin
   $_G\Sigma = {}_GN \cup {}_G\Delta$  with $_G\Delta = {}_M\Delta$ and $_GN = \{\langle pAq\rangle | A \in {}_M\Gamma, p, q \in {}_MQ\} \cup \{{}_GS\}$;

   repeat
      if $Aq_0a \rightarrow B_nB_{n-1}\ldots B_1q_1 \in {}_MR$,
        where $A \in {}_M\Gamma$, $B_i \in {}_M\Gamma$, $1 \le i \le n$, $a \in {}_M\Delta \cup \{\varepsilon\}$, $q_0, q_1 \in {}_MQ$, for some $n \ge 1$ then
          include $\{\langle q_0Aq_{n+1}\rangle \rightarrow a\langle q_1B_1q_2\rangle\langle q_2B_2q_3\rangle\ldots\langle q_{n-1}B_{n-1}q_n\rangle\langle q_nB_nq_{n+1}\rangle | q_j \in {}_MQ, 2 \le j \le n+1\}$
          into $_GR$
   until no change;

   repeat
      if $Aq_0a \rightarrow q_1 \in {}_MR$, where $q_0, q_1 \in {}_MQ$, $A \in {}_M\Gamma$, $a \in {}_M\Delta \cup \{\varepsilon\}$ then
          add $\langle q_0Aq_1\rangle \rightarrow a$ to $_GR$
   until no change;
end.
```

Lemma 3.68. Let M be a pushdown automaton. With M as its input, Algorithm 3.67 correctly constructs a context-free grammar G such that $L(G) = {}_\varepsilon L(M)$.

Proof. To establish $L(G) = {}_\varepsilon L(M)$ this proof first demonstrates the following claim.

Claim. For all $w \in {}_M\Delta^*$, $A \in {}_M\Gamma$, and $q, q' \in Q$, $\langle qAq'\rangle\ {}_{lm}\Rightarrow^* w$ in G if and only if $Aqw \Rightarrow^* q$ in M.

Only if. For all $i \ge 0$, $\langle qAq'\rangle\ {}_{lm}\Rightarrow^i w$ in G implies $Aqw \Rightarrow^* q'$ in M, where $w \in {}_M\Delta^*$, $A \in {}_M\Gamma$, and q, $q' \in Q$.

Basis. For $i = 0$, G cannot make $\langle qAq'\rangle\ {}_{lm}\Rightarrow^i w$. Thus, this implication holds vacuously, and the basis is true.

Induction hypothesis. Assume that the implication holds for all j-step derivations, where $j = 1, \ldots, i$, for some $i > 0$.

Induction step. Consider $\langle qAq'\rangle\ {}_{lm}\Rightarrow^* w\ [p\pi]$ in G, where π consists of i rules, and $p \in {}_GR$. Thus, $\langle qAq'\rangle\ {}_{lm}\Rightarrow^* w\ [p\pi]$ is a derivation that has $i + 1$ steps. From Algorithm 3.67, p has the form p: $\langle qAq'\rangle \rightarrow a\langle q_1B_1q_2\rangle\langle q_2B_2q_3\rangle\ldots\langle q_nB_nq_{n+1}\rangle$, where $q' = q_{n+1}$. Express $\langle qAq'\rangle\ {}_{lm}\Rightarrow^* w\ [p\pi]$ as

$$\langle qAq'\rangle\ {}_{lm}\Rightarrow a\langle q_1B_1q_2\rangle\langle q_2B_2q_3\rangle\ldots\langle q_nB_nq_{n+1}\rangle\ {}_{lm}\Rightarrow^* w$$

In greater detail, $\langle qAq'\rangle\ {}_{lm}\Rightarrow^* a\langle q_1B_1q_2\rangle\langle q_2B_2q_3\rangle\ldots\langle q_nB_nq_{n+1}\rangle\ {}_{lm}\Rightarrow^* w$, where $w = aw_1w_2\ldots w_n$ and

$\langle q_jB_jq_{j+1}\rangle\ _{lm}\Rightarrow^* w_j$ in G, for all $j = 1, \ldots, n$. As π consists of no more than i rules, the induction hypothesis implies $B_jq_jw_j \Rightarrow^* q_{j+1}$ in M, so $B_n\ldots B_{j+1}B_jq_jw_j \Rightarrow^* B_n\ldots B_{j+1}q_{j+1}$ in M. As p: $\langle qAq'\rangle \rightarrow a\langle q_1B_1q_2\rangle\langle q_2B_2q_3\rangle\ldots\langle q_nB_nq_{n+1}\rangle \in {}_GR$, ${}_MR$ contains r: $Aqa \rightarrow B_n\ldots B_1q_1$. Thus, $Aqw \Rightarrow B_n\ldots B_1q_1w_1w_2\ldots w_n$ $[r]$ in M. Consequently, $Aqw \Rightarrow B_n\ldots B_1q_1w_1w_2\ldots w_n \Rightarrow^* B_n\ldots B_2q_2w_2\ldots w_n \Rightarrow^* B_nq_nw_n \Rightarrow^* q_{n+1}$ in M. Because $q' = q_{n+1}$, $Aqw \Rightarrow^* q'$ in M, and the inductive step is completed.

If. For $i \geq 0$, $Aqw \Rightarrow^i q'$ in M implies $\langle qAq'\rangle\ _{lm}\Rightarrow^* w$ in G, where $w \in {}_M\Delta^*$, $A \in {}_M\Gamma$, and $q, q' \in Q$.

Basis. For $i = 0$, $Aqw \Rightarrow^i q'$ in M is ruled out, so this implication holds vacuously, and the basis is true.

Induction hypothesis. Assume that the above implication holds for all j-step derivations, where $j = 1, ..., i$, for some $i > 0$.

Induction step. Consider $Aqw \Rightarrow^* q'$ $[r\rho]$ in M, where ρ represents a rule word consisting of i rules, and $r \in {}_MR$. Thus, $Aqw \Rightarrow^{i+1} q'$ $[r\rho]$ in M. From Algorithm 3.67, r has the form r: $Aqa \rightarrow B_n\ldots B_1q_1$. Now express $Aqw \Rightarrow^* q'$ $[r\rho]$ as $Aqav_1v_2...v_n \Rightarrow B_n\ldots B_1q_1v_1v_2...v_n$ $[r] \Rightarrow^* B_n\ldots B_2q_2v_2\ldots v_n$ $[\rho_1] \Rightarrow^* \ldots \Rightarrow^* B_nq_nv_n$ $[\rho_{n-1}] \Rightarrow^* q_{n+1}$ $[\rho_n]$, where $q' = q_{n+1}$, $w = av_1v_2\ldots v_n$, and $\rho = \rho_1\ldots\rho_{n-1}\rho_n$. As ρ_j consists of no more than i rules, the induction hypothesis implies $\langle q_jB_jq_{j+1}\rangle\ _{lm}\Rightarrow^* v_j$ $[\pi_j]$ in G, for all $j = 1, \ldots, n$. As r: $Aqa \rightarrow B_n....B_1q_1 \in {}_MR$ and $q_2,..., q_{n+1} \in {}_MQ$, ${}_GR$ contains p: $\langle qAq_{n+1}\rangle \rightarrow a\langle q_1B_1q_2\rangle\langle q_2B_2q_3\rangle\ldots\langle q_nB_nq_{n+1}\rangle$ from the **for** loop of Algorithm 3.67. Consequently, $\langle qAq_{n+1}\rangle\ _{lm}\Rightarrow a\langle q_1B_1q_2\rangle\langle q_2B_2q_3\rangle\ldots\langle q_nB_nq_{n+1}\rangle$ $[p]\ _{lm}\Rightarrow^* av_1v_2\ldots v_n$ $[\pi]$, where $\pi = \pi_1\pi_2\ldots\pi_n$. As $q' = q_{n+1}$ and $w = av_1v_2...v_n$, G makes this derivation $\langle qAq'\rangle\ _{lm}\Rightarrow^* w$. That is, the inductive step is completed. Consequently, the if part of this claim is true as well, so the claim holds.

Consider the above claim for $A = {}_MS$ and $q = {}_Ms$. At this point, for all $w \in {}_M\Delta^*$, $\langle {}_Ms_MSq'\rangle\ _{lm}\Rightarrow^* w$ in G if and only if ${}_MS_Msw \Rightarrow^* q'$ in M. Therefore, ${}_GS\ _{lm}\Rightarrow \langle {}_Ms_MSq'\rangle\ _{lm}\Rightarrow^* w$ if and only if ${}_MS_Msw \Rightarrow^* q'$ in M. In other words, $L(G) = {}_\varepsilon L(M)$. Thus, Lemma 3.68 holds.

■

3.35. Consider Algorithm 3.67 *Grammar for a Pushdown Automaton* (see the solution to Exercise 3.34). Initially, this algorithm sets

$$N = \{\langle sSs\rangle, \langle qSq\rangle, \langle sSq\rangle, \langle qSs\rangle, \langle sas\rangle, \langle qaq\rangle, \langle saq\rangle, \langle qas\rangle, S\}$$

and $\Delta = \{a,b\}$. Then, Algorithm 3.67 enters its **for** loop. From $Ssa \rightarrow Sas$, this loop produces $\langle sSs\rangle \rightarrow a\langle sas\rangle\langle sSs\rangle$, $\langle sSs\rangle \rightarrow a\langle saq\rangle\langle qSs\rangle$, $\langle sSq\rangle \rightarrow a\langle sas\rangle\langle sSq\rangle$, $\langle sSq\rangle \rightarrow a\langle saq\rangle\langle qSq\rangle$ and adds these four rules to ${}_GR$. Analogously, from $asa \rightarrow aas$, the **for** loop constructs $\langle sas\rangle \rightarrow a\langle sas\rangle\langle sas\rangle$, $\langle sas\rangle \rightarrow a\langle saq\rangle\langle qas\rangle$, $\langle saq\rangle \rightarrow a\langle sas\rangle\langle saq\rangle$, $\langle saq\rangle \rightarrow a\langle saq\rangle\langle qaq\rangle$ and adds these four rules to ${}_GR$. Based on $asb \rightarrow q$, this loop adds $\langle saq\rangle \rightarrow b$ to ${}_GR$. From $aqb \rightarrow q$, the **for** loop constructs $\langle qaq\rangle \rightarrow b$ and includes this rule in ${}_GR$. Finally, from $Sq \rightarrow q$, this loop produces $\langle qSq\rangle \rightarrow \varepsilon$ and adds it to ${}_GR$. As a result, ${}_GR$ consists of the following rules

$S \rightarrow \langle sSs\rangle$, $S \rightarrow \langle sSq\rangle$, $\langle sSs\rangle \rightarrow a\langle sas\rangle\langle sSs\rangle$, $\langle sSs\rangle \rightarrow a\langle saq\rangle\langle qSs\rangle$, $\langle sSq\rangle \rightarrow a\langle sas\rangle\langle sSq\rangle$, $\langle sSq\rangle \rightarrow a\langle saq\rangle\langle qSq\rangle$, $\langle sas\rangle \rightarrow a\langle sas\rangle\langle sas\rangle$, $\langle sas\rangle \rightarrow a\langle saq\rangle\langle qas\rangle$, $\langle saq\rangle \rightarrow a\langle sas\rangle\langle saq\rangle$, $\langle saq\rangle \rightarrow a\langle saq\rangle\langle qaq\rangle$, $\langle saq\rangle \rightarrow b$, $\langle qaq\rangle \rightarrow b$, $\langle qSq\rangle \rightarrow \varepsilon$

For simplicity, by using Algorithm 3.30 *Useful Symbols*, turn this grammar to the following equivalent grammar containing only useful symbols

p_0: $S \rightarrow \langle sSq \rangle$
p_1: $\langle sSq \rangle \rightarrow a\langle saq \rangle\langle qSq \rangle$
p_2: $\langle saq \rangle \rightarrow a\langle saq \rangle\langle qaq \rangle$
p_3: $\langle saq \rangle \rightarrow b$
p_4: $\langle qaq \rangle \rightarrow b$
p_5: $\langle qSq \rangle \rightarrow \varepsilon$

Observe that this grammar generates $\{a^n b^n \mid n \geq 1\}$.

CHAPTER 4

Deterministic Top-Down Parsing

From now on, compared to the previous chapters, this book becomes less theoretical and more practical. Regarding parsing, while the previous chapter has explained its basic methodology in general, this chapter and the next chapter give a more realistic insight into parsing because they discuss its deterministic methods, which fulfill a central role in practice.

A deterministic top-down parser verifies that the tokenized version of a source program is syntactically correct by constructing its parse tree. Reading the input string representing the tokenized program from left to the right, the parser starts from its root and proceeds down toward the frontier denoted by the input string. To put it alternatively in terms of derivations, it builds up the leftmost derivation of this tokenized program starting from the start symbol. Frequently, this parser is based upon *LL grammars*, where the first *L* stands for the *l*eft-to-right scan of tokens and the second *L* stands for the *l*eftmost derivations. By making use of *predictive sets* constructed for rules in these grammars, the parser makes a completely deterministic selection of an applied rule during every leftmost derivation.

Based on LL grammars, we concentrate our attention on *predictive parsing*, which is perhaps the most frequently used deterministic top-down parsing method in practice. More specifically, first, we return to the popular recursive descent method (see Section 3.2), which frees us from explicitly implementing a pushdown list, and create its deterministic version. Then, we use the LL grammars and their predictive sets to make a predictive table used by a deterministic predictive table-driven parser, which explicitly implements a pushdown list. In this parser, any grammatical change only leads to a modification of the table while its control procedure remains unchanged, which is its key pragmatic advantage. We also explain how this parser handles the syntax errors to recover from them.

Synopsis. Section 4.1 introduces and discusses predictive sets and LL grammars. Then, based upon the LL grammar, Section 4.2 discusses the predictive recursive-descent and table-driven parsing.

4.1 Predictive Sets and LL Grammars

Consider a grammar, $G = ({}_G\Sigma, {}_GR)$, and a G-based top-down parser working with an input string w (see Section 3.2). Suppose that the parser has already found the beginning of the leftmost derivation for w, $S\ {}_{lm}\Rightarrow^* tAv$, where t is a prefix of w. More precisely, let $w = taz$, where a is the current input symbol, which follows t in w, and z is the suffix of w, which follows a. In tAv, A is the leftmost nonterminal to be rewritten in the next step. Assume that there exist several different A-rules, so the parser has to select one of them to continue the parsing process, and if the parser works deterministically, it cannot revise this selection later on. A *predictive parser* selects the right rule by predicting whether its application gives rise to a leftmost derivation of a string starting with a. To make this prediction, every rule $r \in {}_GR$ is accompanied with its *predictive set* containing all terminals that can begin a string resulting from a derivation whose first step is made by r. If the A-rules have their predictive sets pairwise disjoint, the parser deterministically selects the rule whose predictive set contains a. To construct the predictive sets, we first need the *first* and *follow* sets, described next.

First. The predictive set corresponding to $r \in {}_G R$ obviously contains the terminals that occur as the *first* symbol in a string derived from ***rhs***(r).

Definition 4.1 *first*. Let $G = ({}_G\Sigma, {}_G R)$ be a grammar. For every string $x \in {}_G\Sigma^*$,

$$first(x) = \{a \mid x \Rightarrow^* w, \text{ where either } w \in {}_G\Delta^+ \text{ with } a = symbol(w, 1) \text{ or } w = \varepsilon = a\},$$

where *symbol*$(w, 1)$ denotes the leftmost symbol of w (see Section 1.1).

■

In general, *first* is defined in terms of $\Rightarrow$. However, as for every $w \in {}_G\Delta^*$, $x \Rightarrow^* w$ if and only if $x \;{}_{lm}{\Rightarrow^*}\; w$ (see Theorem 3.20), we could equivalently rephrase this definition in terms of the leftmost derivations, which play a crucial role in top-down parsing, as

$$first(x) = \{a \mid x \;{}_{lm}{\Rightarrow^*}\; w, \text{ where either } w \in {}_G\Delta^+ \text{ with } a = symbol(w, 1) \text{ or } w = \varepsilon = a\}$$

Furthermore, observe that if $x \Rightarrow^* \varepsilon$, where $x \in \Sigma^*$, then ε is in *first*(x); as a special case, for $x = \varepsilon$, *first*$(\varepsilon) = \{\varepsilon\}$.

Next, we will construct the *first* sets for all strings contained in ${}_G\Delta \cup \{$***lhs***$(r) \mid r \in {}_G R\} \cup \{y \mid y \in$ *suffix*es(***rhs***(r)) with $r \in {}_G R\}$. We make use of some subsets of these *first* sets later in this section (see Algorithm 4.4 and Definition 4.5).

Goal. Construct *first*(x) for every $x \in {}_G\Delta \cup \{$***lhs***$(r) \mid r \in {}_G R\} \cup \{y \mid y \in$ *suffix*es(***rhs***(r)) with $r \in {}_G R\}$.

Gist. Initially, set *first*(a) to $\{a\}$ for every $a \in {}_G\Delta \cup \{\varepsilon\}$ because $a \;{}_{lm}{\Rightarrow^0}\; a$ for these as. Furthermore, if $A \rightarrow uw \in {}_G R$ with $u \Rightarrow^* \varepsilon$, then $A \Rightarrow^* w$, so add the symbols of *first*(w) to *first*(A) (notice that $u \;{}_{lm}{\Rightarrow^*}\; \varepsilon$ if and only if u is a string consisting of ε-nonterminals, determined by Algorithm 3.33). Repeat this extension of all the *first* sets in this way until no more symbols can be added to any of the *first* sets.

Algorithm 4.2 *First Sets*.

Input • a grammar $G = ({}_G\Sigma, {}_G R)$.

Output • *first*(u) for every $u \in {}_G\Delta \cup \{$***lhs***$(r) \mid r \in {}_G R\} \cup \{y \mid y \in$ *suffix*es(***rhs***(r)) with $r \in {}_G R\}$.

Method

```
begin
    set first(a) = {a} for every $a \in {}_G\Delta \cup \{\varepsilon\}$, and
    set all the other constructed first sets to $\emptyset$;
    repeat
        if $r \in {}_G R$, $u \in prefixes(\mathbf{rhs}(r))$, and $u \;{}_{lm}{\Rightarrow^*}\; \varepsilon$ then
            extend $first(suffix(\mathbf{rhs}(r), |\mathbf{rhs}(r)| - |u|))$ by $first(symbol(\mathbf{rhs}(r), |u| + 1))$ and
            extend $first(\mathbf{lhs}(r))$ by $first(suffix(\mathbf{rhs}(r), |\mathbf{rhs}(r)| - |u|))$
    until no change
end.
```

Follow. The ε-rules deserve our special attention because they may give rise to derivations that erase substrings of sentential forms, and this possible erasure makes the parser's selection of the next applied rule even more difficult. Indeed, consider a grammar $G = ({}_G\Sigma, {}_GR)$ and $A \to x \in {}_GR$ with $x\ {}_{lm}\!\Rightarrow^* \varepsilon$. At this point, the parser needs to decide whether from A, it should make either $A\ {}_{lm}\!\Rightarrow x\ {}_{lm}\!\Rightarrow^* \varepsilon$ or a derivation that produces a non-empty string. To make this decision, we need to determine the set *follow*(A) containing all terminals that can follow A in any sentential form; if this set contains the current input token that is out of *first*(x), the parser simulates $A\ {}_{lm}\!\Rightarrow x\ {}_{lm}\!\Rightarrow^* \varepsilon$. In addition, we include ◀ into *follow*(A) to express that a sentential form ends with A (we assume that ◀ $\notin {}_G\Sigma$).

Definition 4.3 *follow*. Let $G = ({}_G\Sigma, {}_GR)$ be grammar. For every $A \in {}_GN$,

$$follow(A) = \{a \in {}_G\Delta \cup \{◀\} \mid Aa \in substrings(F(G)\{◀\})\},$$

where $F(G)$ denotes the set of G's sentential forms (see Definition 3.1).

■

Goal. Construct *follow*(A) for every $A \in N$.

Gist. S is a sentential form, so we initialize *follow*(S) with {◀}. Consider any $B \to uAv \in {}_GR$. If a is a terminal in *first*(v), then $Aa \in substrings(F(G))$, so we add a to *follow*(A). In addition, if ε is in *first*(v), then $v\ {}_{lm}\!\Rightarrow^* \varepsilon$ and, consequently, *follow*(B) $\subseteq$ *follow*(A), so we add all symbols from *follow*(B) to *follow*(A). Keep extending all the *follow* sets in this way until no symbol can be added to any of them.

Algorithm 4.4 *Follow Sets*.

Input
- a grammar $G = ({}_G\Sigma, {}_GR)$;
- *first*(u) for every $u \in \{y \mid y \in suffixes(\mathbf{rhs}(r))$ with $r \in {}_GR\}$ (see Algorithm 4.2).

Output
- *follow*(A), for every $A \in {}_GN$.

Method

```
begin
    set $follow(S) = \{◀\}$, and
    set all the other constructed follow sets to $\emptyset$;
    repeat
        if $r \in {}_GR$ and $Au \in suffixes(\mathbf{rhs}(r))$, where $A \in {}_GN$, $u \in {}_G\Sigma^*$ then
        begin
            add the symbols in $(first(u) - \{\varepsilon\})$ to $follow(A)$;
            if $\varepsilon \in first(u)$ then
                add the symbols in $follow(\mathbf{lhs}(r))$ to $follow(A)$
        end
    until no change
end.
```

Predictive sets. Based on the *first* and *follow* sets, we define the predictive sets as follows.

Definition 4.5 *Predictive Set*. The *predictive set* of each $r \in {}_GR$, symbolically denoted by *predictive-set*(r), is defined in the following way

- if $\varepsilon \notin first(\mathbf{rhs}(r))$, $predictive\text{-}set(r) = first(\mathbf{rhs}(r))$;

• if $\varepsilon \in$ *first*(**rhs**(r)), *predictive-set*(r) = (*first*(**rhs**(r)) – {ε}) ∪ *follow*(**lhs**(r)).

■

LL grammars. Reconsider the parsing situation described in the beginning of the present section. That is, a top-down parser based on a grammar $G = ({}_G\Sigma, {}_GR)$ has found the beginning of the leftmost derivation $S\ {}_{lm}\Rightarrow^* tAv$ for an input string taz, where a is the current input token, and it needs to select one of several different A-rules to rewrite A in tAv and, thereby, make another step. If for an A-rule r, $a \in$ *predictive-set*(r) and for any other A-rule p, $a \notin$ *predictive-set*(p), the parser obviously selects r. This idea leads to the next definition of LL grammars.

Definition 4.6 *LL Grammars*. A grammar $G = ({}_G\Sigma, {}_GR)$ is an *LL grammar* if for each $A \in N$, any two different A-rules, $p, q \in {}_GR$ and $p \neq q$, satisfy *predictive-set*(p) ∩ *predictive-set*(q) = ∅.

■

As already noted, in *LL grammars*, the first *L* stands for a *l*eft-to-right scan of tokens and the other *L* stands for a *l*eftmost derivation. Sometimes, in greater detail, the literature refers to the LL grammars as *LL*(1) *grammars* to point out that the top-down parsers based on these grammars always look at one input token during each step of the parsing process. Indeed, these grammars represent a special case of *LL*(k) *grammars*, where $k \geq 1$, which underlie parsers that make a k-tokens lookahead. In this introductory textbook, however, we discuss only LL(1) grammars and simply refer to them as LL grammars for brevity.

Case Study 11/35 *LL Grammar*. Return to the grammar obtained in Case Study 9/35 in Section 3.2. The present section demonstrates that the grammar is an LL grammar; hence, we denote this grammar by ${}_{LL}H = ({}_{LL}{}_H\Sigma, {}_{LL}{}_HR)$. Recall that ${}_{LL}{}_HR$ contains the following rules

$E \rightarrow TA$
$A \rightarrow \vee TA$
$A \rightarrow \varepsilon$
$T \rightarrow FB$
$B \rightarrow \wedge FB$
$B \rightarrow \varepsilon$
$F \rightarrow (E)$
$F \rightarrow i$

First (Algorithm 4.2). For each rule in this grammar, we construct *first*(u) for every $u \in \Delta \cup$ {**lhs**(r)| $r \in {}_{LL}{}_HR$} ∪ {y| $y \in$ *suffix*es(**rhs**(r)) with $r \in {}_{LL}{}_HR$} by Algorithms 4.2. First, we set *first*(a) = {a} for every $a \in$ {i, (,), ∨, ∧} ∪ {ε}. Consider $F \rightarrow i$. By the **repeat** loop of Algorithms 4.2, as *first*(i) = {i}, we include i into *first*(F). As $i \in$ *first*(F) and $T \rightarrow FB \in {}_{LL}{}_HR$, we add i to *first*(FB) as well. Complete this construction as an exercise. However, to construct *predictive-set*(r) for each $r \in {}_{LL}{}_HR$, we only need {*first*(**rhs**(r))| $r \in {}_{LL}{}_HR$}, which represents a subset of all the *first* sets constructed by Algorithm 4.2. The members of {*first*(**rhs**(r))| $r \in {}_{LL}{}_HR$} are listed in the second column of the table given in Figure 4.1.

Follow (Algorithm 4.4). Consider *first*(u) for each $u \in$ {y| $y \in$ *suffix*es(**rhs**(r)) with $r \in {}_GR$}. We construct *follow*(A), for every $A \in {}_{LL}{}_HN$ by Algorithm 4.4 as follow. We initially have *follow*(E) = {◀}. As $F \rightarrow (E) \in {}_{LL}{}_HR$ and) ∈ *first*()), we add) to *follow*(E). As $E \rightarrow TA \in {}_{LL}{}_HR$ and *follow*(E) contains ◀ and), we add ◀ and) to *follow*(A), too. Since ε ∈ *first*(A),) and ◀ are in *follow*(E), and $E \rightarrow TA \in {}_{LL}{}_HR$, we add) and ◀ to *follow*(T). As ∨ ∈ *first*(A), we add ∨ to *follow*(T), too.

Complete this construction as an exercise. The third column of the table in Figure 4.1 contains *follow*(**lhs**(*r*)) for each $r \in {}_{LL}HR$.

Predictive sets (Definition 4.5). The fourth column of the table in Figure 4.1 contains *predictive-set*(*r*) for each $r \in {}_{LL}HR$. Notice that *follow*(**lhs**(*r*)) is needed to determine *predictive-set*(*r*) if $\varepsilon \in$ *first*(**rhs**(*r*)) because at this point *predictive-set*(*r*) = (*first*(**rhs**(*r*)) − {ε}) ∪ *follow*(**lhs**(*r*)); if $\varepsilon \notin$ *first*(**rhs**(*r*)), it is not needed because *predictive-set*(*r*) = *first*(**rhs**(*r*)) (see Definition 4.5). Take, for instance, $E \rightarrow TA \in {}_{LL}HR$ with *first*(*TA*)) = {*i*, (}. As $\varepsilon \notin$ *first*(**rhs**(*r*)), *predictive-set*($E \rightarrow TA$) = *first*(*TA*) = {*i*, (}. Consider $A \rightarrow \varepsilon \in {}_{LL}HR$ with *first*(ε) = {ε} and *follow*(**lhs**(*A*)) = {), ◀}. As $\varepsilon \in$ *first*(**rhs**($A \rightarrow \varepsilon$)), *predictive-set*($A \rightarrow \varepsilon$) = *first*(ε) − {ε} ∪ *follow*(**lhs**(*A*)) = ∅ ∪ {), ◀} = {), ◀}. Complete this construction as an exercise.

LL grammar. Observe that *predictive-set*($A \rightarrow \vee TA$) ∩ *predictive-set*($A \rightarrow \varepsilon$) = {∨} ∩ {), ◀} = ∅. Analogously, *predictive-set*($B \rightarrow \wedge FB$) ∩ *predictive-set*($B \rightarrow \varepsilon$) = ∅ and *predictive-set*($F \rightarrow (E)$) ∩ *predictive-set*($F \rightarrow i$) = ∅. Thus, ${}_{LL}H$ is an LL grammar.

Rule *r* **in** ${}_{LL}H$	*first*(**rhs**(*r*))	*follow*(**lhs**(*r*))	*predictive-set*(*r*)
$E \rightarrow TA$	*i*, (	), ◀	*i*, (
$A \rightarrow \vee TA$	∨	), ◀	∨
$A \rightarrow \varepsilon$	ε	), ◀	), ◀
$T \rightarrow FB$	*i*, (	∨,), ◀	*i*, (
$B \rightarrow \wedge FB$	∧	∨,), ◀	∧
$B \rightarrow \varepsilon$	ε	∨,), ◀	∨,), ◀
$F \rightarrow (E)$	(	∧, ∨,), ◀	(
$F \rightarrow i$	*i*	∧, ∨,), ◀	*i*

Figure 4.1 *first***,** *follow* **and** *predictive* **sets for** ${}_{LL}H$**.**

■

As demonstrated in the following section, the LL grammars underlie the most important deterministic top-down parsers, so their significance is indisputable. Not all grammars represent LL grammars, however. Some non-LL grammars can be converted to equivalent LL grammars by an algorithm that removes an undesirable grammatical phenomenon as illustrated next by a grammar transformation called *left factoring*. Unfortunately, in general, for some grammars, there exist no equivalent LL grammars (see Exercises 4.6 and 4.7).

Left Factoring. Consider a grammar $G = ({}_{G}\Sigma, {}_{G}R)$ such that *R* contains two different *A*-rules of the form $A \rightarrow zx$ and $A \rightarrow zy$ in *R*. If *G* derives a non-empty string of terminals from *z*, the predict sets of these rules have some terminals in common, so *G* is no LL grammar. Therefore, we transform *G* to an equivalent *left-factored* grammar in which any pair of different *A*-rules *r* and *p*, which are not ε-rules, satisfies *symbol*(**rhs**(*r*), 1) ≠ *symbol*(**rhs**(*p*), 1), so the above grammatical phenomenon is a priori ruled out.

Goal. Conversion of any grammar to an equivalent left-factored grammar.

Gist. Consider a grammar $G = ({}_{G}\Sigma, {}_{G}R)$. Let $A \rightarrow zx$ and $A \rightarrow zy$ be two different *A*-rules in ${}_{G}R$, where $z \neq \varepsilon$ and *prefixes*{*x*} ∩ *prefixes*{*y*} = {ε}. Then, replace these two rules with these three rules $A \rightarrow zA_{lf}$, $A_{lf} \rightarrow x$, and $A_{lf} \rightarrow y$, where A_{lf} is a new nonterminal. Repeat this modification until the resulting grammar is left-factored. In essence, the resulting grammar somewhat defers the original derivation process to select the right rule. More specifically, by $A \rightarrow zA_{lf}$, it first changes

A to zA_{lf}. Then, it derives a terminal string from z. Finally, based upon the current input symbol, it decides whether to apply $A_{lf} \rightarrow x$ or $A_{lf} \rightarrow y$ to continue the derivation.

Algorithm 4.7 ***Left-Factored Grammar*****.**

Input • a grammar $G = ({}_G\Sigma, {}_GR)$.

Output • a left-factored grammar $H = ({}_H\Sigma, {}_HR)$ equivalent to G.

Method

```
begin
    $_H\Sigma := {}_G\Sigma$;
    $_HR := {}_GR$;
    repeat
        if $p, r \in {}_GR$, $p \neq r$, $\textbf{\textit{lhs}}(p) = \textbf{\textit{lhs}}(r)$, and $\textit{prefix}\text{es}(\textbf{\textit{rhs}}(p)) \cap \textit{prefix}\text{es}(\textbf{\textit{rhs}}(r)) \neq \{\varepsilon\}$ then
        begin
            introduce a new nonterminal $A_{lf}$ to $_HN$;
            in $_HR$, replace $p$ and $r$ with these three rules
            $\textbf{\textit{lhs}}(p) \rightarrow xA_{lf}$, $A_{lf} \rightarrow \textit{suffix}(\textbf{\textit{rhs}}(p), |\textbf{\textit{rhs}}(p)| - |x|)$, and $A_{lf} \rightarrow \textit{suffix}(\textbf{\textit{rhs}}(r), |\textbf{\textit{rhs}}(r)| - |x|)$
                where $x$ is the longest string in $\textit{prefix}\text{es}(\textbf{\textit{rhs}}(p)) \cap \textit{prefix}\text{es}(\textbf{\textit{rhs}}(r))$
        end
    until no change;
end.
```

Case Study 12/35 ***Left Factoring*****.** Consider the next grammar, which defines the ***integer*** and ***real*** FUN declarations.

⟨declaration part⟩ → ***declaration*** ⟨declaration list⟩
⟨declaration list⟩ → ⟨declaration⟩; ⟨declaration list⟩
⟨declaration list⟩ → ⟨declaration⟩
⟨declaration⟩ → ***integer*** ⟨variable list⟩
⟨declaration⟩ → ***real*** ⟨variable list⟩
⟨variable list⟩ → *i*, ⟨variable list⟩
⟨variable list⟩ → *i*

Because the predictive sets of ⟨declaration list⟩-rules and ⟨variable list⟩-rules are not disjoint, this grammar is not an LL grammar. Apply Algorithm 4.7 to this grammar. This algorithm replaces

⟨declaration list⟩ → ⟨declaration⟩; ⟨declaration list⟩
⟨declaration list⟩ → ⟨declaration⟩

with these three rules

⟨declaration list⟩ → ⟨declaration⟩⟨more declarations⟩
⟨more declarations⟩ → ; ⟨declaration list⟩
⟨more declarations⟩ → ε

where ⟨more declarations⟩ is a new nonterminal. Analogously, it replaces

⟨variable list⟩ → *i*, ⟨variable list⟩
⟨variable list⟩ → *i*

with
⟨variable list⟩ → *i* ⟨more variables⟩
⟨more variables⟩ → , ⟨variable list⟩
⟨more variables⟩ → ε

where ⟨more variables⟩ is a new nonterminal. As a result, we obtain this LL grammar:

⟨declaration part⟩ → ***declaration*** ⟨declaration list⟩
⟨declaration list⟩ → ⟨declaration⟩ ⟨more declarations⟩
⟨more declarations⟩ → ; ⟨declaration list⟩
⟨more declarations⟩ → ε
⟨declaration⟩ → ***integer*** ⟨variable list⟩
⟨declaration⟩ → ***real*** ⟨variable list⟩
⟨variable list⟩ → *i* ⟨more variables⟩
⟨more variables⟩ → , ⟨variable list⟩
⟨more variables⟩ → ε

■

4.2 Predictive Parsing

In this section, we first reconsider the recursive-descent parsing method (see Section 3.2). Specifically, based upon LL grammars, we create its deterministic version. Then, we use the LL grammars and their predictive sets to create predictive tables and deterministic top-down parsers driven by these tables. Finally, we explain how to handle errors in predictive parsing.

Predictive Recursive-Descent Parsing

The present section discusses the deterministic version of recursive-descent parsers, described in Section 3.2 with a detailed illustration given in Case Study 8/35 *Recursive-Descent Parser*. This determinism is achieved by the predictive sets constructed in the previous section, hence *predictive recursive-descent parsing*. That is, for an LL grammar $G = ({}_G\Sigma, {}_GR)$, we construct a *G-based predictive recursive-descent parser*, denoted by *G-rd-parser*. We describe this construction in the following specific case study, which makes use of these programming routines

- **ACCEPT** and **REJECT**
- **INPUT-SYMBOL**
- **RETURN-SYMBOL**

ACCEPT and **REJECT** are announcements described in Convention 1.8. In the next parser as well as in all the other parsers given in the rest of this book, **ACCEPT** announces a successful completion of the parsing process while **REJECT** announces a syntax error in the parsed program.

INPUT-SYMBOL has the meaning described in Definition 3.14—that is, it is a lexical-analysis function that returns the input symbol, representing the current token, after which the input string is advanced to the next symbol.

Definition 4.8 RETURN-SYMBOL. Procedure **RETURN-SYMBOL** returns the current token in **INPUT-SYMBOL** back to the lexical analyzer, which thus sets **INPUT-SYMBOL** to this returned token during its next call.

■

The following parser makes use of **RETURN-SYMBOL** when a rule selection is made by using a token from the *follow* set; at this point, this token is not actually used up, so when the next call of **INPUT-SYMBOL** occurs, the parser wants the lexical analyzer to set **INPUT-SYMBOL** to this very same token, which is accomplished by **RETURN-SYMBOL**.

Case Study 13/35 ***Predictive Recursive Descent Parser.*** Reconsider the LL grammar ${}_{LL}H$ discussed in Case Study 11/35. Figure 4.2 repeats its rules together with the corresponding predictive sets.

***Rule** r **in** ${}_{LL}H$*	*predictive-set*(*r*)
$E \rightarrow TA$	i, (
$A \rightarrow \vee TA$	$\vee$
$A \rightarrow \varepsilon$	), ◀
$T \rightarrow FB$	i, (
$B \rightarrow \wedge FB$	$\wedge$
$B \rightarrow \varepsilon$	$\vee$,), ◀
$T \rightarrow F$	i, (
$F \rightarrow (E)$	(
$F \rightarrow i$	i

Figure 4.2 ***Rules in*** ${}_{LL}H$ ***and Their Predictive Sets.***

Just like in Case Study 8/35, we now construct an ${}_{LL}H$-based recursive-descent parser ${}_{LL}H$*-rd-parser* as a collection of Boolean *rd-functions* corresponding to the nonterminals in ${}_{LL}H$. In addition, however, we make use of the predictive sets of ${}_{LL}H$'s rules so this parser always selects the next applied rule deterministically.

Consider the start symbol E and the only E-rule $E \rightarrow TA$ with *predictive-set*($E \rightarrow TA$) = $\{i$, (}. As a result, *rd-function E* has this form

```
function E: Boolean ;
begin
    E := false;
    if INPUT-SYMBOL in {i, (} then
        begin
            RETURN-SYMBOL;
            if T then
                if A then E := true
        end
end
```

Consider the two A-rules that include $A \rightarrow \vee TA$ with *predictive-set*($A \rightarrow \vee TA$) = $\{\vee\}$ and $A \rightarrow \varepsilon$ with *predictive-set*($A \rightarrow \varepsilon$) = {), ◀}. Therefore, *rd-function A* selects $A \rightarrow \vee TA$ if the input symbol is $\vee$ and this function selects $A \rightarrow \varepsilon$ if this symbol is in {), ◀}. If the input symbol differs from $\vee$,), or ◀, the syntax error occurs. Thus, *rd-function A* has this form

```
function A: Boolean;
begin
    A := false;
    case INPUT-SYMBOL of
    $\vee$:  if $T$ then {$A \to \vee TA$}
            if $A$ then
                $A$ := true;
    ), ◀:
        begin {$A \to \varepsilon$}
            RETURN-SYMBOL; {) and ◀ are in follow($A$)}
            $A$ := true
        end
    end {case};
end
```

There exists a single *T*-rule of the form $T \to FB$ in $_{LL}H$ with *predictive-set*($T \to FB$) = {*i*, (}. Its *rd-function T*, given next, is similar to *rd-function E*.

```
function $T$: Boolean ;
begin
    $T$ := false;
    if INPUT-SYMBOL in {$i$, (} then
        begin
            RETURN-SYMBOL;
            if $F$ then
                if $B$ then
                    $T$ := true
        end
end
```

Consider the two *B*-rules $B \to \wedge FB$ with *predictive-set*($B \to \wedge FB$) = {$\wedge$} and $B \to \varepsilon$ with *predictive-set*($B \to \varepsilon$) = {$\vee$,), ◀}. Therefore, *rd-function B* selects $B \to \wedge FB$ if the input symbol is $\wedge$, and it selects $B \to \varepsilon$ if this symbol is in {$\vee$,), ◀}. The *rd-function B* has thus this form

```
function $B$: Boolean ;
begin
    $B$ := false;
    case INPUT-SYMBOL of
    $\wedge$:  if $F$ then {$B \to \wedge FB$}
            if $B$ then
                $B$ := true;
    $\vee$, ), ◀:
        begin {$B \to \varepsilon$}
            RETURN-SYMBOL; {$\vee$, ), and ◀ are in follow($B$)}
            $B$ := true
        end
    end {case};
end
```

Finally, consider the *F*-rules $F \to (E)$ with *predictive-set*($F \to (E)$) = {(} and $F \to i$ with *predictive-set*($F \to i$) = {*i*}. Therefore, *rd-function F* selects $F \to (E)$ if the input symbol is (and this function selects $F \to i$ if the input symbol is *i*. If the input symbol differs from (or *i*, the syntax error occurs. Thus, *rd-function F* has this form

```
function F: Boolean ;
begin
   F := false;
   case INPUT-SYMBOL of
   (:  if E then {F → (E)}
            if INPUT-SYMBOL = ')' then
               F := true;
   i:  F := true
   end {case};
end
```

Having these functions in *$_{LL}H$-rd-parser,* its main body is based on the following simple **if** statement, which decides whether the source program is syntactically correct by the final Boolean value of *rd-function E*:

```
if E then
   ACCEPT
else
   REJECT
```

As an exercise, reconsider each application of **RETURN-SYMBOL** in the above functions and explain its purpose in detail.

■

Predictive Table-Driven Parsing

In the predictive recursive-descent parsing, for every nonterminal and the corresponding rules, there exists a specific Boolean function, so any grammatical change usually necessitates reprogramming several of these functions. Therefore, unless a change of this kind is ruled out, we often prefer an alternative *predictive table-driven parsing* based on a single general control procedure that is completely based upon a predictive table. At this point, a change of the grammar only implies an adequate modification of the table while the control procedure remains unchanged. As opposed to the predictive recursive-descent parsing, however, this parsing method maintains a pushdown explicitly, not implicitly via recursive calls like in predictive recursive-descent parsing.
Consider an LL grammar $G = ({}_G\Sigma, {}_GR)$. Like a general top-down parser (see Algorithm 3.12), a *G-based predictive table-driven parser*, denoted by *G-td-parser*, is underlain by a pushdown automaton, $M = ({}_M\Sigma, {}_MR)$ (see Definition 3.7). However, strictly mathematical specification of M, including all its rules in ${}_MR$, would be unbearably tedious and difficult-to-follow. Therefore, to make this specification brief, clear, and easy to implement, we describe *G-td-parser* as an algorithm together with a *G-based predictive table*, denoted by *G-predictive-table,* by which *G-td-parser* determines a move from every configuration. *G-predictive-table*'s rows and columns are denoted by the members of ${}_GN$ and ${}_G\Delta \cup \{◀\}$, respectively. Each of its entry contains a member of ${}_GR \cup \{☹\}$. More precisely, for each $A \in {}_GN$ and each $t \in {}_G\Delta$, if there exists $r \in R$ such that ***lhs***$(r) = A$ and $t \in$ *predictive-set(r)*, *G-predictive-table*$[A, t] = r$; otherwise, *G-predictive-table*$[A, t] = ☹$, which means a syntax error. Making use of *G-predictive-table*, *G-td-parser M* works with the input string, representing the tokenized source program, as described next.

Goal. Construction of a *G-td-parser* $M = ({}_M\Sigma, {}_MR)$ for an LL grammar $G = ({}_G\Sigma, {}_GR)$.

Gist. A predictive table-driven parser always performs a computational step based on the pushdown top symbol X and the current input symbol a. This step consists in one of the following actions:

- if $X \in {}_G\Delta$ and $X = a$, the pushdown top symbol is a terminal coinciding with the input token, so *M pops* the pushdown by removing X from its top and, simultaneously, advances to the next input symbol, occurring behind a;
- if $X \in {}_G\Delta$ and $X \neq a$, the pushdown top symbol is a terminal that differs from the input token, so the parser announces an error by **REJECT**;
- if $X \in {}_GN$ and *G-predictive-table*$[X, a] = r$ with $r \in {}_GR$, where ***lhs***$(r) = X$, *M expands* the pushdown by replacing X with *reversal*(***rhs***(r)).
- if $X \in {}_GN$ and *G-predictive-table*$[X, a]$ = ☹, *M* announces an error by **REJECT**;
- if ▶ = X and a = ◀, the pushdown is empty and the input token string is completely read, so *M* halts and announces a successful completion of parsing by **ACCEPT**.

Before going any further in the predictive table-driven parsing, we introduce Convention 4.9 concerning the way we frequently describe the configuration of a parser throughout the rest of this book. Specifically, Algorithm 4.11, which represents *G-td-parser*, makes use of this convention several times.

Convention 4.9. Consider a parser's configuration, ▶u◆v◀, where u and v are the current *pd* and *ins*, respectively (see the notes following Convention 3.8 for *pd* and *ins*). For $i = 1, \dots, |u| + 1$, pd_i denotes the ith topmost symbol stored in the pushdown, so pd_1 refers to the very topmost pushdown symbol, which occurs as the rightmost symbol of *pd*, while $pd_{|u|+1}$ denotes ▶, which marks the *bottom* of the pushdown. Furthermore, for $h = 1, \dots, |v| + 1$, ins_h denotes the hth input symbol in *ins*, so ins_1 refers to the current input symbol, which occurs as the leftmost symbol of *ins*, while $ins_{|v|+1}$ denotes ◀, which marks the *ins* end. Making use of this notation, we can express ▶u◆v◀ as

$$\blacktriangleright pd_{|u|} \dots pd_2\, pd_1 \blacklozenge ins_1\, ins_2 \dots ins_{|v|} \blacktriangleleft$$

■

Throughout the rest of this section, we also often make use of operations **EXPAND** and **POP**, described next.

Definition 4.10 *Operations* EXPAND *and* POP. Operations **EXPAND** and **POP** modify *G-td-parser*'s current *pd* top as follows

EXPAND$(A \to x)$ expands the pushdown according to a rule $A \to x$; in terms of Algorithm 3.12, **EXPAND**$(A \to x)$ thus means an application of A◆ $\to$ *reversal*(x)◆.

POP pops the topmost symbol from the current pushdown.

■

Algorithm 4.11 *Predictive Table-Driven Parsing.*

Input • a *G-predictive-table* of an LL grammar $G = ({}_G\Sigma, {}_GR)$;
• an input string *ins* = w◀ with $w \in {}_G\Delta^*$.

Output • **ACCEPT** if $w \in L(G)$, and **REJECT** if $w \notin L(G)$.

Method

```
begin
    initialize pd with ▶S; {initially pd = ▶S}
    set $i$ to 1; {initially the current input symbol is $ins_1$—the leftmost symbol of $w$◀}

    repeat
        case $pd_1$ of

            in $_G\Delta$: if $pd_1 = ins_i$ then {$pd_1$ is a terminal}
                begin
                    POP; {pop $pd$}
                    $i := i + 1${advance to the next input symbol}
                end
                else
                    REJECT; {$pd_1 \neq ins_i$}
            in $_GN$: if $G$-precedence-table[$pd_1$, $ins_i$] = $r$ with $r \in {_GR}$ then {$pd_1$ is a nonterminal}
                    EXPAND($r$)
                else
                    REJECT {the table entry is ☹};
            ▶: if $ins_i$ = ◀ then
                    ACCEPT
                else
                    REJECT {the pushdown is empty but the input string is not}
        end{case};

    until ACCEPT or REJECT

end.
```

Case Study 14/35 ***Predictive Table-Driven Parser.*** Return to grammar $_{LL}H$ (see Case Study 12/35) defined as:

1: $E \to TA$
2: $A \to \vee TA$
3: $A \to \varepsilon$
4: $T \to FB$
5: $B \to \wedge FB$
6: $B \to \varepsilon$
7: $T \to F$
8: $F \to (E)$
9: $F \to i$

By using the predictive sets corresponding to these rules (see Figure 4.1), we construct the predictive table of this grammar (see Figure 4.3).

	$\vee$	$\wedge$	(	)	i	◀
E	☹	☹	1	☹	1	☹
A	2	☹	☹	3	☹	3
T	☹	☹	4	☹	4	☹
B	6	5	☹	6	☹	6
F	☹	☹	8	☹	9	☹
▶	☹	☹	☹	☹	☹	☺

Figure 4.3 $_{LL}$***H-predictive-table.***

Observe that from Algorithm 4.11 and the ${}_{LL}H$*-predictive-table*, we could obtain a strictly mathematical specification of *G-td-parser* as a pushdown automaton $M = ({}_{M}\Sigma, {}_{M}R)$ according to Definition 3.7. In essence, *M* has the form of the parser constructed in Algorithm 3.12 in Section 3.2. That is, *M* makes **ACCEPT** by $\blacktriangleright\blacklozenge\blacktriangleleft \rightarrow \blacklozenge$. If a pair of the *pd* top *X* and the current input symbol *b* leads to **REJECT**, ${}_{M}R$ has no rule with $X\blacklozenge b$ on its left-hand side. It performs **POP** by rules of the form $a\blacklozenge a \rightarrow \blacklozenge$ for each $a \in {}_{LL}H\Delta$. Finally, if a pair of the *pd* top and the current input symbol leads to **EXPAND**(*r*), where *r* is a rule from ${}_{G}R$, *M* makes this expansion according to *r* by $A\blacklozenge \rightarrow reversal(x)\blacklozenge$. For instance, as ${}_{LL}H$*-predictive-table*$[E, i] = 1$ and $1\colon E \rightarrow TA \in {}_{LL}HR$, ${}_{M}R$ has $E\blacklozenge \rightarrow AT\blacklozenge$ to make an expansion according to 1. A completion of this mathematical specification of *M* is left as an exercise. However, not only is this completion a tedious task, but also the resulting parser *M* specified in this way is difficult to understand what it actually does with its incredibly many rules. That is why, as already pointed out, we always prefer the description of a parser as an algorithm together with a parsing table throughout the rest of this book.

Configuration	***Table Entry and Rule***	***Sentential Form***
		$\underline{E}$
$\blacktriangleright E\blacklozenge i \wedge i \vee i\blacktriangleleft$	$[E, i] = 1\colon E \rightarrow TA$	$\underline{T}A$
$\blacktriangleright AT\blacklozenge i \wedge i \vee i\blacktriangleleft$	$[T, i] = 4\colon T \rightarrow FB$	$\underline{F}BA$
$\blacktriangleright ABF\blacklozenge i \wedge i \vee i\blacktriangleleft$	$[F, i] = 9\colon F \rightarrow i$	$i\,\underline{B}A$
$\blacktriangleright ABi\blacklozenge i \wedge i \vee i\blacktriangleleft$		
$\blacktriangleright AB\blacklozenge \wedge i \vee i\blacktriangleleft$	$[B, \wedge] = 5\colon B \rightarrow \wedge FB$	$i \wedge \underline{F}BA$
$\blacktriangleright ABF\wedge\blacklozenge \wedge i \vee i\blacktriangleleft$		
$\blacktriangleright ABF\blacklozenge i \vee i\blacktriangleleft$	$[F, i] = 9\colon F \rightarrow i$	$i \wedge i\underline{B}A$
$\blacktriangleright ABi\blacklozenge i \vee i\blacktriangleleft$		
$\blacktriangleright AB\blacklozenge \vee i\blacktriangleleft$	$[B, \vee] = 6\colon B \rightarrow \varepsilon$	$i \wedge i\underline{A}$
$\blacktriangleright A\blacklozenge \vee i\blacktriangleleft$	$[A, \vee] = 2\colon A \rightarrow \vee TA$	$i \wedge i \vee \underline{T}A$
$\blacktriangleright AT\vee\blacklozenge \vee i\blacktriangleleft$		
$\blacktriangleright AT\blacklozenge i\blacktriangleleft$	$[T, i] = 7\colon T \rightarrow F$	$i \wedge i \vee \underline{F}A$
$\blacktriangleright AF\blacklozenge i\blacktriangleleft$	$[F, i] = 9\colon F \rightarrow i$	$i \wedge i \vee i\underline{A}$
$\blacktriangleright Ai\blacklozenge i\blacktriangleleft$		
$\blacktriangleright A\blacklozenge\blacktriangleleft$	$[A, \blacktriangleleft] = 3\colon A \rightarrow \varepsilon$	$i \wedge i \vee i$
$\blacktriangleright\blacklozenge\blacktriangleleft$	$[\blacktriangleright, \blacktriangleleft] = ☺$	

Figure 4.4 ***Predictive Table-Driven Parsing in Terms of Leftmost Derivation.***

Configuration	***Rule***	***Parse Tree***
		$\underline{E}$
$\blacktriangleright E\blacklozenge i \wedge i \vee i\blacktriangleleft$	1	$E\langle \underline{T}A\rangle$
$\blacktriangleright AT\blacklozenge i \wedge i \vee i\blacktriangleleft$	4	$E\langle T\langle \underline{F}B\rangle A\rangle$
$\blacktriangleright ABF\blacklozenge i \wedge i \vee i\blacktriangleleft$	9	$E\langle T\langle F\langle i\rangle \underline{B}\rangle A\rangle$
$\blacktriangleright ABi\blacklozenge i \wedge i \vee i\blacktriangleleft$		
$\blacktriangleright AB\blacklozenge \wedge i \vee i\blacktriangleleft$	5	$E\langle T\langle F\langle i\rangle B\langle \wedge \underline{F}B\rangle\rangle A\rangle$
$\blacktriangleright ABT\wedge\blacklozenge \wedge i \vee i\blacktriangleleft$		
$\blacktriangleright ABF\blacklozenge i \vee i\blacktriangleleft$	9	$E\langle T\langle F\langle i\rangle B\langle \wedge F\langle i\rangle \underline{B}\rangle\rangle A\rangle$
$\blacktriangleright ABi\blacklozenge i \vee i\blacktriangleleft$		
$\blacktriangleright AB\blacklozenge \vee i\blacktriangleleft$	6	$E\langle T\langle F\langle i\rangle B\langle \wedge F\langle i\rangle B\langle\varepsilon\rangle\rangle\rangle \underline{A}\rangle$
$\blacktriangleright A\blacklozenge \vee i\blacktriangleleft$	2	$E\langle T\langle F\langle i\rangle B\langle \wedge F\langle i\rangle B\langle\varepsilon\rangle\rangle\rangle A\langle \vee \underline{T}A\rangle\rangle$
$\blacktriangleright AT\vee\blacklozenge \vee i\blacktriangleleft$		
$\blacktriangleright AT\blacklozenge i\blacktriangleleft$	7	$E\langle T\langle F\langle i\rangle B\langle \wedge F\langle i\rangle B\langle\varepsilon\rangle\rangle\rangle A\langle \vee T\langle \underline{F}\rangle A\rangle\rangle$
$\blacktriangleright AF\blacklozenge i\blacktriangleleft$	9	$E\langle T\langle F\langle i\rangle B\langle \wedge F\langle i\rangle B\langle\varepsilon\rangle\rangle\rangle A\langle \vee T\langle F\langle i\rangle\rangle \underline{A}\rangle\rangle$
$\blacktriangleright Ai\blacklozenge i\blacktriangleleft$		
$\blacktriangleright A\blacklozenge\blacktriangleleft$	3	$E\langle T\langle F\langle i\rangle B\langle \wedge F\langle i\rangle B\langle\varepsilon\rangle\rangle\rangle A\langle \vee T\langle F\langle i\rangle\rangle A\langle\varepsilon\rangle\rangle\rangle$
$\blacktriangleright\blacklozenge\blacktriangleleft$	☺	

Figure 4.5 ***Predictive Table-Driven Parsing in Terms of Parse Tree.***

Consider $i \wedge i \vee i$ as the input string, which ${}_{LL}H$ generates in this leftmost way

$$
\begin{array}{lll}
\underline{E} & {}_{lm}\Rightarrow \underline{T}A & [1] \\
 & {}_{lm}\Rightarrow \underline{F}BA & [4] \\
 & {}_{lm}\Rightarrow i\underline{B}A & [9] \\
 & {}_{lm}\Rightarrow i \wedge \underline{F}BA & [5] \\
 & {}_{lm}\Rightarrow i \wedge i\underline{B}A & [9] \\
 & {}_{lm}\Rightarrow i \wedge i\underline{A} & [6] \\
 & {}_{lm}\Rightarrow i \wedge i \vee \underline{T}A & [2] \\
 & {}_{lm}\Rightarrow i \wedge i \vee \underline{F}A & [7] \\
 & {}_{lm}\Rightarrow i \wedge i \vee i\underline{A} & [9] \\
 & {}_{lm}\Rightarrow i \wedge i \vee i & [3]
\end{array}
$$

With $i \wedge i \vee i$, ${}_{LL}$*H-td-parser* works as described in the tables given in Figures 4.4 and 4.5. In the first column of both tables, we give ${}_{LL}$*H-td-parser*'s configurations, each of which has the form ▶x◆y◀, where ▶x and y◀ are the current *pd* and *ins*, respectively. In Figure 4.4, the second and the third columns describe the parsing of $i \wedge i \vee i$ in terms of the leftmost derivation. That is, its second column describes ${}_{LL}$*H-predictive-table*'s entries and the rules they contain. The third column gives the sentential forms derived in the leftmost derivation. Figure 4.5 reformulates this description in terms of the parse tree $pt(E\ {}_{lm}\Rightarrow i \wedge i \vee i)$. In the second column of Figure 4.5, we give the labels of the applied rules. The third column gives a step-by-step construction of $pt(E\ {}_{lm}\Rightarrow i \wedge i \vee i)$ so that we always underline the node to which a rule tree is being attached during the corresponding step of the parsing process.

■

Handling Errors

Of course, a predictive parser struggles to handle each syntax error occurrence as best as it can. It carefully diagnoses the error and issues an appropriate error message. Then, it recovers from the error by slightly modifying *pd* or skipping a short prefix of *ins*. After the recovery, the parser resumes its analysis of the syntactically erroneous program, possibly discovering further syntax errors. Most crucially, no matter what it does during handling errors, the parser has to avoid any endlessly repeated routines so all the tokenized source string is eventually processed. Throughout the rest of this section, we sketch two simple and popular error recovery methods in terms of a *G-td-parser*, where $G = ({}_G\Sigma, {}_GR)$ is an LL grammar while leaving their straightforward adaptation for *G-rd-parser* as an exercise.

Panic-mode error recovery. For every nonterminal $A \in {}_GN$, we define a set of synchronizing input symbols as *synchronizing-set*(A) = *first*$(A) \cup$ *follow*(A). Supposing that *G-td-parser* occurs in a configuration with $pd_1 = X$ and $ins_1 = a$ when a syntax error occurs, this error recovery method handles the error, in essence, as follows.

- If $pd_1 \in {}_GN$ and *G-precedence-table*$[pd_1, ins_1]$ = ☹, skip the input symbols until the first occurrence of t that is in *synchronizing-set*(pd_1). If $t \in$ *first*(pd_1), resume parsing according to *G-precedence-table*$[pd_1, t]$ without any further change. If $t \in$ *follow*(pd_1), pop pd_1 from *pd* and resume parsing.

- If $pd_1 \in {}_G\Delta$ and $pd_1 \neq a$, pop pd_1 and resume parsing.

As an exercise, we discuss further variants of this method, most of which are based on alternative definitions of the synchronizing sets.

Ad-hoc recovery. In this method, we design a specific recovery routine, **RECOVER**[*X*, *t*], for every error-implying pair of a top pushdown symbol *X* and an input symbol *t*. That is, if *X* and *t* are two different input symbols, we make **RECOVER**[*X*, *t*]. We also design **RECOVER**[*X*, *t*] if $X \in N \cup \{$▶$\}$ and *G-predictive-table*[*X*, *t*] = ☹. **RECOVER**[*X*, *t*] figures out the most probable mistake that leads to the given syntax error occurrence, issues an error message, and decides on a recovery procedure that takes a plausible action to resume parsing. Typically, **RECOVER**[*X*, *t*] skips a tiny portion of *ins* or modifies *pd* by changing, inserting, or deleting some symbols; whatever it does, however, the recovery procedure needs to guarantee that this modification surely avoids any infinite loop so that the parser eventually proceeds its normal process. *G-td-parser* with this type of error recovery works just like Algorithm 4.11 except that if the error occurs in a configuration such that either *G-predictive-table*[pd_1, ins_1] = ☹ or $ins_1 \neq pd_1$ with $pd_1 \in {}_G\Delta$, it performs **RECOVER**[pd_1, ins_1]. In this way, *G-td-parser* detects and recovers from the syntax error, after which it resumes the parsing process. Of course, once *G-td-parser* detects a syntax error and recovers from it, it can never proclaim that the program is syntactically correct later during the resumed parsing process. That is, even if *G-td-parser* eventually reaches *G-predictive-table*[pd_1, ins_1] = *G-predictive-table*[▶,◀] = ☺ after a recovery from a syntax error, it performs **REJECT**, not **ACCEPT**.

Case Study 15/35 ***Error Recovery in Predictive Parsing.*** Return to the grammar $_{LL}H$ discussed in Case Study 14/35. Consider)*i* as the input string. As obvious, $)i \notin L(_{LL}H)$. With)*i*, $_{LL}H$*-td-parser* immediately interrupts its parsing because entry $_{LL}H$*-predictive-table*[*E*,)] equals ☹. Reconsidering this interruption in detail, we see that if *E* occurs on the pushdown top and) is the input token, then the following error-recovery routine is appropriate.

name:	**RECOVER**[*E*,)]
diagnostic:	) starts *ins* or no expression occurs between parentheses
recovery:	If) starts *ins*, skip it. If no expression occurs between (and), remove *E* from *pd* and skip) in *ins*; in this way, *pd* and *ins* are changed so that the missing expression problem can be ignored. Resume parsing.

Designing the other error-recovery routines is left as an exercise. With)*i*, the parser works as described in Figure 4.6, in which the error-recovery information is pointed up.

Configuration	***Table Entry***	***Sentential Form***
		E
▶*E*◆) *i*◀	[*E*,)] = ☹, so **RECOVER**[*E*,)]	
▶*E*◆*i*◀	[*E*, *i*] = 1: $E \rightarrow TA$	*TA*
▶*AT*◆*i*◀	[*T*, *i*] = 4: $T \rightarrow FB$	*FBA*
▶*ABF*◆*i*◀	[*F*, *i*] = 9: $F \rightarrow i$	*iBA*
▶*AB i*◆*i*◀		
▶*AB*◆◀	[*B*,◀] = 6: $B \rightarrow \varepsilon$	*iA*
▶*A*◆◀	[*A*,◀] = 3: $A \rightarrow \varepsilon$	*i*
▶◆◀		

Figure 4.6 ***Predictive Table-Driven Parsing with Error Recovery.***

■

Even if a parser handles syntax errors very sophisticatedly, it sometimes repeatedly detects more and more errors within a relatively short, but heavily erroneous program construct, such as a single statement. At this point, it usually skips all the erroneous construct until its end delimited, for instance, by a semicolon and issues a message stating that there are too many errors to analyze this construct. Then, it continues with the syntax analysis right behind this construct.

Exercises

4.1*Solved*. Consider the next three grammars. Determine the languages they generate. Observe that the languages generated by grammars (a) and (b) represent simple programming languages while the language generated by grammar (c) is rather abstract. Construct the predictive tables corresponding to grammars (a) through (c).

(a) 1: ⟨program⟩ → ⟨statement⟩⟨statement list⟩.
2: ⟨statement list⟩ → ; ⟨statement⟩⟨statement list⟩
3: ⟨statement list⟩ → ε
4: ⟨statement⟩ → *i* = ⟨expression⟩
5: ⟨statement⟩ → **read** *i*
6: ⟨statement⟩ → **write** ⟨expression⟩
7: ⟨statement⟩ → **for** *i* = ⟨expression⟩ **to** ⟨expression⟩ **perform** ⟨statement⟩
8: ⟨statement⟩ → **begin** ⟨statement⟩⟨statement list⟩ **end**
9: ⟨expression⟩ → ⟨operand⟩⟨continuation⟩
10: ⟨continuation⟩ → ⟨operator⟩⟨expression⟩
11: ⟨continuation⟩ → ε
12: ⟨operand⟩ → (⟨expression⟩)
13: ⟨operand⟩ → *i*
14: ⟨operator⟩ → +
15: ⟨operator⟩ → –

(b) 1: ⟨program⟩ → **begin** ⟨statement⟩⟨statement list⟩ **end**
2: ⟨statement list⟩ → ; ⟨statement⟩⟨statement list⟩
3: ⟨statement list⟩ → ε
4: ⟨statement⟩ → *i* = ⟨boolean expression⟩
5: ⟨statement⟩ → **read** *i*
6: ⟨statement⟩ → **write** ⟨boolean expression⟩
7: ⟨statement⟩ → **if** ⟨boolean expression⟩ **then** ⟨statement⟩ **else** ⟨statement⟩
8: ⟨statement⟩ → **while** ⟨boolean expression⟩ **do** ⟨statement⟩
9: ⟨statement⟩ → **repeat** ⟨statement⟩⟨statement list⟩ **until** ⟨boolean expression⟩
10: ⟨statement⟩ → **begin** ⟨statement⟩⟨statement list⟩ **end**
11: ⟨boolean expression⟩ → ⟨operand⟩⟨continuation⟩
12: ⟨continuation⟩ → ⟨operator⟩⟨boolean expression⟩
13: ⟨continuation⟩ → ε
14: ⟨operand⟩ → (⟨boolean expression⟩)
15: ⟨operand⟩ → *i*
16: ⟨operand⟩ → **not** ⟨boolean expression⟩
17: ⟨operator⟩ → **and**
18: ⟨operator⟩ → **or**

(c) 1: $S \rightarrow ABb$
2: $A \rightarrow CD$
3: $B \rightarrow dB$
4: $B \rightarrow \varepsilon$
5: $C \rightarrow aCb$
6: $C \rightarrow \varepsilon$
7: $D \rightarrow cDd$
8: $D \rightarrow \varepsilon$

4.2*Solved*. Prove that grammars (a) through (d), given next, are not LL grammars. Determine their languages. Construct proper LL grammars equivalent to them.

(a)
1: ⟨program⟩ → **begin** ⟨statement list⟩ **end**
2: ⟨statement list⟩ → ⟨statement⟩ ; ⟨statement list⟩
3: ⟨statement list⟩ → ⟨statement⟩
4: ⟨statement⟩ → **read** *i*
5: ⟨statement⟩ → **write** *i*
6: ⟨statement⟩ → *i* = **sum** (⟨item list⟩)
7: ⟨item list⟩ → *i* , ⟨item list⟩
8: ⟨item list⟩ → *i*

(b)
1: ⟨program⟩ → **begin** ⟨statement list⟩ **end**
2: ⟨statement list⟩ → ⟨statement⟩ ; ⟨statement list⟩
3: ⟨statement list⟩ → ⟨statement⟩
4: ⟨statement⟩ → **read** ⟨item list⟩
5: ⟨statement⟩ → **write** ⟨item list⟩
6: ⟨statement⟩ → *i* = ⟨expression⟩
7: ⟨item list⟩ → *i* , ⟨item list⟩
8: ⟨item list⟩ → *i*
9: ⟨expression⟩ → *i* + ⟨expression⟩
10: ⟨expression⟩ → *i* – ⟨expression⟩
11: ⟨expression⟩ → *i*

(c)
1: ⟨program⟩ → ⟨function⟩ ; ⟨function list⟩
2: ⟨function list⟩ → ⟨function⟩
3: ⟨function⟩ → ⟨head⟩⟨body⟩
4: ⟨head⟩ → **function** *i* ( ) : ⟨type⟩
5: ⟨head⟩ → **function** *i* (⟨parameter list⟩) : ⟨type⟩
6: ⟨parameter list⟩ → ⟨parameter⟩ , ⟨parameter list⟩
7: ⟨parameter list⟩ → ⟨parameter⟩
8: ⟨parameter⟩ → *i* : ⟨type⟩
9: ⟨type⟩ → **integer**
10: ⟨type⟩ → **real**
11: ⟨type⟩ → **string**
12: ⟨body⟩ → **begin** ⟨statement list⟩ **end**
13: ⟨statement list⟩ → ⟨statement⟩ ; ⟨statement list⟩
14: ⟨statement list⟩ → ⟨statement⟩
15: ⟨statement⟩ → *s*
16: ⟨statement⟩ → ⟨program⟩

(d)
1: ⟨block⟩ → **begin** ⟨declaration list⟩⟨execution part⟩ **end**
2: ⟨declaration list ⟩ → ⟨declaration list⟩ ; ⟨declaration⟩
3: ⟨declaration list⟩ → ⟨declaration⟩
4: ⟨declaration⟩ → **integer** ⟨variable list⟩
5: ⟨declaration⟩ → **real** ⟨variable list⟩
6: ⟨variable list⟩ → *i* , ⟨variable list⟩
7: ⟨variable list⟩ → *i*
8: ⟨execution part⟩ → **compute** ⟨statement list⟩
9: ⟨statement list⟩ → ⟨statement list⟩ ; ⟨statement⟩

10: ⟨statement list⟩ → ⟨statement⟩
11: ⟨statement⟩ → s
12: ⟨statement⟩ → ⟨block⟩

4.3$_{\textit{Solved}}$**.** Demonstrate that the following grammar generates the expressions in Polish postfix notation with operand i and operators + and *. Prove that it is not an LL grammar. Construct an equivalent LL grammar.

1: ⟨expression⟩ → ⟨expression⟩⟨expression⟩ +
2: ⟨expression⟩ → ⟨expression⟩⟨expression⟩ *
3: ⟨expression⟩ → i

4.4. Consider Definitions 4.12 and 4.13, given next. Design algorithms that construct *last*(x) for every $x \in {}_G\Sigma^*$and *precede*(A) for every $A \in {}_GN$. Write programs to implement them.

Definition 4.12 *last*. Let $G = ({}_G\Sigma, {}_GR)$ be a grammar. For every string $x \in {}_G\Sigma^*$,

$$last(x) = \{a \mid x \Rightarrow^* w, \text{ where either } w \in {}_G\Delta^+ \text{ with } a = symbol(w, |w|) \text{ or } w = \varepsilon = a\},$$

where *symbol*(w, $|w|$) denotes the rightmost symbol of w (see Section 1.1).

■

Definition 4.13 *precede*. Let $G = ({}_G\Sigma, {}_GR)$ be a grammar. For every $A \in {}_GN$,

$$precede(A) = \{a \in {}_G\Delta \cup \{\blacktriangleright\} \mid aA \in substrings(\{\blacktriangleright\}F(G))\},$$

where $F(G)$ denotes the set of G's sentential forms (see Definition 3.1).

■

4.5$_{\textit{Solved}}$**.** Consider Definition 4.5 *Predictive Set*. For a proper grammar, $G = ({}_G\Sigma, {}_GR)$, the definition of its rules' predictive sets can be simplified. Explain how and give this simplified definition.

4.6. Prove that no LL grammar generates $\{a^ib^i \mid i \geq 1\} \cup \{a^jb^{2j} \mid j \geq 1\}$.

4.7. Prove that every LL grammar is unambiguous (see Definition 3.6). Based on this result, demonstrate that no LL grammar can generate any inherently ambiguous language, such as $\{a^ib^jc^k \mid i, j, k \geq 1, \text{ and } i = j \text{ or } j = k\}$.

4.8. Consider the transformation of any grammar G to an equivalent proper grammar H described in the proof of Theorem 3.38. Prove or disprove that if G is an LL grammar, then H, produced by this transformation, is an LL grammar as well.

4.9. Consider grammars ${}_{cond}G$, ${}_{cond}H$, ${}_{expr}G$, and ${}_{expr}H$ (see Case Study 6/35). Prove that none of them represents an LL grammar. Turn them to equivalent LL grammars.

4.10. Generalize LL grammars to *LL*(k) *grammars*, where $k \geq 1$, which underlie parsers that make a k-token lookahead. Reformulate the notions introduced in Section 4.1, such as *first* and *follow* sets, in terms of these generalized grammars. Discuss the advantages and disadvantages of this generalization from a practical point of view.

4.11. By analogy with left factoring (see Section 4.1), discuss right factoring. Modify Algorithm

4.7 *Left-Factored Grammar* so it converts any grammar to an equivalent right-factored grammar.

4.12. Reconsider Case Study 12/35 *Left Factoring* in terms of right factoring. Make use of the solution to Exercise 4.4.

4.13. Write a program to implement predictive recursive-descent parser; including **RETURN-SYMBOL**, **EXPAND**, and **POP**; described in Section 4.2. In addition, write a program to implement Algorithm 4.11 *Predictive Table-Driven Parsing*.

4.14. Complete Case Study 15/35 *Error Recovery in Predictive Parsing*.

4.15. Consider the programs obtained in Exercise 4.13. Extend them so they perform the panic-mode error recovery described in Section 4.2. Alternatively, extend them so they perform the ad-hoc recovery described in Section 4.2.

4.16. Generalize a predictive table for any grammar G so for each $A \in {}_GN$ and each $t \in {}_G\Delta$, the corresponding entry contains the list of all rules r satisfying ***lhs***$(r) = A$ and $t \in$ *predictive-set*(r); if this list is empty, this entry contains ☹. Notice that an entry of this generalized table may contain more than one rule. Give a rigorous version of this generalization. Discuss the advantages and disadvantages of this generalization from a practical point of view.

4.17*Solved*. Consider this five-rule grammar

1: ⟨if-statement⟩ → **if** ⟨condition⟩ **then** ⟨if-statement⟩⟨else-part⟩
2: ⟨if-statement⟩ → *a*
3: ⟨condition⟩ → *c*
4: ⟨else-part⟩ → **else** ⟨if-statement⟩
5: ⟨else-part⟩ → ε

Construct *predictive-set*(r) for each rule $r = 1, \ldots, 5$. Demonstrate that both *predictive-set*(4) and *predictive-set*(5) contain **else**, so this grammar is no LL grammar. Construct its generalized predictive table, whose entry [⟨else-part⟩, **else**] contains 4 and 5 (see Exercise 4.16). Remove 5 from this multiple entry. Demonstrate that after this removal, Algorithm 4.11 *Predictive Table-Driven Parsing* works with the resulting table quite deterministically so it associates an **else** with the most recent unmatched **then**, which is exactly what most real high-level programming languages do.

Solution to Selected Exercises

4.1. We only give the predictive tables of grammars (a) and (c).

(a)

	.	;	*i*	=	**read**	**write**	**for**	**to**	**perform**	**begin**	**end**	(	)	+	–	◀
⟨program⟩	☹	☹	1	☹	1	1	1	☹	☹	1	☹	☹	☹	☹	☹	☹
⟨statement⟩	☹	☹	4	☹	5	6	7	☹	☹	8	☹	☹	☹	☹	☹	☹
⟨statement list⟩	3	2	☹	☹	☹	☹	☹	☹	☹	☹	3	☹	☹	☹	☹	☹
⟨expression⟩	☹	☹	9	☹	☹	☹	☹	☹	☹	☹	☹	9	☹	☹	☹	☹
⟨operand⟩	☹	☹	13	☹	☹	☹	☹	☹	☹	☹	☹	12	☹	☹	☹	☹
⟨continuation⟩	11	11	☹	☹	☹	☹	☹	11	11	☹	11	☹	11	10	10	☹
⟨operator⟩	☹	☹	☹	☹	☹	☹	☹	☹	☹	☹	☹	☹	☹	14	15	☹
▶	☹	☹	☹	☹	☹	☹	☹	☹	☹	☹	☹	☹	☹	☹	☹	☺

(c)

	a	b	c	d	◀
S	1	1	1	1	☹
A	2	2	2	2	☹
B	☹	4	☹	3	☹
C	5	6	6	6	☹
D	☹	8	7	8	☹
▶	☹	☹	☹	☹	☺

4.2. We only give the solution in terms of grammar (a). An equivalent proper LL grammar to this grammar is

1: ⟨program⟩ → **begin** ⟨statement⟩ ⟨statement list⟩
2: ⟨statement list⟩ → ; ⟨statement⟩ ⟨statement list⟩
3: ⟨statement list⟩ → **end**
4: ⟨statement⟩ → **read** *i*
5: ⟨statement⟩ → **write** *i*
6: ⟨statement⟩ → *i* = **sum** (*i* ⟨item list⟩
7: ⟨item list⟩ → , *i* ⟨item list⟩
8: ⟨item list⟩ →)

4.3. An equivalent LL grammar and its predictive table follow next.

1: ⟨expression⟩ → *i* ⟨continuous⟩
2: ⟨continuous⟩ → ⟨expression⟩ ⟨operator⟩ ⟨continuous⟩
3: ⟨continuous⟩ → ε
4: ⟨operator⟩ → +
5: ⟨operator⟩ → *

	+	*	i	◀
⟨expression⟩	☹	☹	1	☹
⟨continuous⟩	3	3	2	3
⟨operator⟩	4	5	☹	☹
▶	☹	☹	☹	☺

4.5. No *follow* set is needed.

Definition 4.14 ***Predictive Sets for a Proper Grammar's Rules.*** Let $G = ({}_G\Sigma, {}_GR)$ be a proper grammar. The *predictive set* of each $r \in {}_GR$, *predictive-set*(r), is defined as *predictive-set*(r) = *first*(***rhs***(r)).

■

4.17. We only give the generalized predictive table:

	if	**then**	**else**	c	a	◀
⟨if-statement⟩	1	☹	☹	☹	2	☹
⟨condition⟩	☹	☹	☹	3	☹	☹
⟨else-part⟩	☹	☹	4, 5	☹	☹	5
▶	☹	☹	☹	☹	☹	☺

CHAPTER 5

Deterministic Bottom-Up Parsing

A bottom-up parser verifies the syntax of a tokenized source program by constructing the parse tree for this program in a left-to-right and bottom-up way. That is, it reads the input string representing the tokenized program from left to right and works up towards the root of the parse tree. To put it in terms of derivations, it builds up the rightmost derivation of the input string in reverse so it starts from the string and proceeds towards the start symbol. Each step of this parsing process represents a *shift* or a *reduction*. The former consists in shifting the current input symbol onto the pushdown. During a reduction, the parser selects a *handle*—that is, an occurrence of the right-hand side of a rule in the current sentential form, and after this selection, it reduces the handle to the rule's left-hand side so that this reduction, viewed in reverse, represents a rightmost derivation step. If the parser always precisely determines how to make each step during this bottom-up parsing process, then it works deterministically, and the deterministic bottom-up parsers obviously fulfill a key role in practice.

In this chapter, we discuss two fundamental deterministic bottom-up parsing methods. First, we describe *precedence parsing*, which is a popular deterministic bottom-up parsing method for expressions whose operators and their priorities actually control the parsing process. Then, we describe *LR parsing*, where *L* stands for a *l*eft-to-right scan of the input string and *R* stands for a *r*ightmost derivation constructed by the parser. LR parsers are as powerful as the deterministic pushdown automata, so they represent the strongest possible parsers that work in a deterministic way. That is probably why they are so often implemented in reality, and we discuss them in detail later in this chapter.

Throughout the chapter, we use the same notions and conventions as in Section 4.2, including **ACCEPT** and **REJECT** (see Convention 1.8).

Synopsis. In Section 5.1, we describe the precedence parsing. In Section 5.2, we explain LR parsing. Both sections have the same structure. First, they describe the fundamental parsing algorithm together with the parsing table the algorithm makes use of. Then, they present the construction of the parsing table. Finally, they explain how to handle errors.

5.1 Precedence Parsing

Given a grammar $G = ({}_G\Sigma, {}_GR)$, we first describe a *G-based operator-precedence parser*, denoted by *G-op-parser*, which makes use of its ${}_{cond}$*G-based operator precedence table*, denoted by *G-op-table*, whose construction is explained later in this section.

Operator Precedence Parsing Algorithm

In *G-op-parser*, *pd-top-terminal* denotes the topmost pushdown terminal, and if the pushdown contains no terminal, then *pd-top-terminal* = ▶. *G-op-table*'s rows and columns of *G-op-table* are denoted by the members of ${}_G\Delta \cup \{$▶$\}$ and ${}_G\Delta \cup \{$◀$\}$, respectively. Each ${}_{cond}$*G-op-table* entry is filled with a member of {⌊, ⌋, |, ☹, ☺}. *G-op-parser* determines each parsing step based upon *G-op-table*[*pd-top-terminal*, ins_1]—that is, the *G-op-table* entry in the row denoted by *pd-top-terminal* and the column denoted by ins_1. If *G-op-table*[*pd-top-terminal*, ins_1] contains ⌊ or |, *G-op-parser* makes a shift. If *G-op-table*[*pd-top-terminal*, ins_1] is ⌋, *G-op-parser* makes a reduction in the following way. Suppose that *G-op-parser's* current configuration is ▶*u*◆*v*◀; at this point, *G-op-parser* determines the current *G-op-parser*'s handle, denoted by *G-op-handle*, as the shortest string wby satisfying $\blacktriangleright u = xawby$, where *G-op-table*$[a, b] = \llcorner$, $a \in {}_G\Delta \cup \{\blacktriangleright\}$, $b \in {}_G\Delta$, $w \in {}_GN^*$, $x, y \in {}_G\Sigma^*$. After this determination of *G-op-handle*, it selects a rule with the right-hand side equal to *G-op-handle* and reduces this handle to the left-hand side of the selected rule to complete the reduction. Finally, *G-op-table*[*pd-top-terminal*, ins_1] = ☹ specify a syntax error while *G-op-table*[*pd-top-terminal*, ins_1] = ☺ means a successful completion of the parsing process. *G-op-parser* is described as Algorithm 5.2, which makes operations **REDUCE** and **SHIFT**, defined next.

Definition 5.1 *Operations* REDUCE *and* SHIFT. In a G-based bottom-up parser, where $G = ({}_G\Sigma, {}_GR)$ is a grammar, we use two operations, **REDUCE** and **SHIFT**, which modify the current *pd* top as follows

REDUCE$(A \rightarrow x)$ makes a reduction according to $A \rightarrow x \in {}_GR$ (in terms of Algorithm 3.12, **REDUCE**$(A \rightarrow x)$ thus denotes an application of $x\blacklozenge \rightarrow A\blacklozenge$ in M).

SHIFT pushes ins_1 onto *pd* and advances to the next input symbol.

■

Algorithm 5.2 *Operator Precedence Parser*.

Input
- a grammar $G = (\Sigma, R)$;
- a *G-op-table*;
- $ins = w\blacktriangleleft$ with $w \in {}_G\Delta^*$.

Output
- **ACCEPT** if $w \in L(G)$, and **REJECT** if $w \notin L(G)$.

Method

```
begin
    set pd to $\blacktriangleright$;
    repeat
        case G-op-table[pd-top-terminal, $ins_1$] of
            $|$ : SHIFT;
            $\llcorner$ : SHIFT;
            $\lrcorner$ :  if G contains a rule $A \rightarrow x$ with $x$ = G-op-handle then
                    REDUCE($A \rightarrow x$);
                 else REJECT {no rule to reduce by};
            ☹ : REJECT {G-op-table-detected error};
            ☺ : ACCEPT;
        end; {case}
    until ACCEPT or REJECT
end.
```

Case Study 16/35 ***Operator Precedence Parser.*** In reality, Algorithm 5.2 is usually used to parse expressions. Therefore, in this case study, we reconsider ${}_{cond}G$ for logical expressions (see Case Study 6/35 in Section 3.1) and describe ${}_{cond}G$*-op-parser* with ${}_{cond}G$*-op-table* in Figure 5.1. Recall that ${}_{cond}G$ is defined as

$$
\begin{aligned}
&C \rightarrow C \vee C\\
&C \rightarrow C \wedge C\\
&C \rightarrow (C)\\
&C \rightarrow i
\end{aligned}
$$

	$\wedge$	$\vee$	i	(	)	$\blacktriangleleft$
$\wedge$	⌋	⌋	⌊	⌊	⌋	⌋
$\vee$	⌊	⌋	⌊	⌊	⌋	⌋
i	⌋	⌋	☹	☹	⌋	⌋
(	⌊	⌊	⌊	⌊	|	☹
)	⌋	⌋	☹	☹	⌋	⌋
$\blacktriangleright$	⌊	⌊	⌊	⌊	☹	☺

Figure 5.1 ***Operator Precedence Table.***

Consider, for instance, $i \wedge (i \vee i)\blacktriangleleft$. With this input, ${}_{cond}G$*-op-parser* works as described in Figures 5.2 and 5.3. In the first column of both tables, we give ${}_{cond}G$*-op-parser*'s configurations, which have the form $\blacktriangleright x\blacklozenge y\blacktriangleleft$, where $\blacktriangleright x$ and $y\blacktriangleleft$ are the current *pd* and *ins*, respectively. In *pd*, *pd-top-terminal* is specified by underlining. In Figure 5.2, the second column gives ${}_{cond}G$*-op-table*'s entries. The third column gives the parser's operations. If the ${}_{cond}G$*-op-table* entry is ⌋, this operation is **REDUCE**, and at this point, the third column also contains the rule according to which the reduction is made. In Figure 5.3, we reformulate this parsing process in terms of the constructed parse tree. Its second column gives the rules according to which the reductions are made. Finally, the third column describes how the rule trees are attached to the parser tree, in which the attached rule trees are pointed up and their roots are, in addition, underlined.

Configuration	***Table Entry***	***Parsing Action***
$\underline{\blacktriangleright}\blacklozenge i \wedge (i \vee i)\blacktriangleleft$	$[\blacktriangleright, i] =$ ⌊	**SHIFT**
$\blacktriangleright \underline{i}\blacklozenge \wedge (i \vee i)\blacktriangleleft$	$[i, \wedge] =$ ⌋	**REDUCE**$(C \rightarrow i)$
$\underline{\blacktriangleright} C\blacklozenge \wedge (i \vee i)\blacktriangleleft$	$[\blacktriangleright, \wedge] =$ ⌊	**SHIFT**
$\blacktriangleright C \underline{\wedge}\blacklozenge(i \vee i)\blacktriangleleft$	$[\wedge, (] =$ ⌊	**SHIFT**
$\blacktriangleright C \wedge \underline{(}\blacklozenge i \vee i)\blacktriangleleft$	$[(, i] =$ ⌊	**SHIFT**
$\blacktriangleright C \wedge (\underline{i}\blacklozenge \vee i)\blacktriangleleft$	$[i, \vee] =$ ⌋	**REDUCE**$(C \rightarrow i)$
$\blacktriangleright C \wedge \underline{(}C\blacklozenge \vee i)\blacktriangleleft$	$[(, \vee] =$ ⌊	**SHIFT**
$\blacktriangleright C \wedge (C \underline{\vee}\blacklozenge i)\blacktriangleleft$	$[\vee, i] =$ ⌊	**SHIFT**
$\blacktriangleright C \wedge (C \vee \underline{i}\blacklozenge)\blacktriangleleft$	$[i,)] =$ ⌋	**REDUCE**$(C \rightarrow i)$
$\blacktriangleright C \wedge (C \underline{\vee} C\blacklozenge)\blacktriangleleft$	$[\vee,)] =$ ⌋	**REDUCE**$(C \rightarrow C \vee C)$
$\blacktriangleright C \wedge \underline{(}C\blacklozenge)\blacktriangleleft$	$[(,)] =$ |	**SHIFT**
$\blacktriangleright C \wedge (C\underline{)}\blacklozenge\blacktriangleleft$	$[), \blacktriangleleft] =$ ⌋	**REDUCE**$(C \rightarrow (C))$
$\blacktriangleright C \underline{\wedge} C\blacklozenge\blacktriangleleft$	$[\wedge, \blacktriangleleft] =$ ⌋	**REDUCE**$(C \rightarrow C \wedge C)$
$\underline{\blacktriangleright} C\blacklozenge\blacktriangleleft$	$[\blacktriangleright, \blacktriangleleft] =$ ☺	ACCEPT

Figure 5.2 ***Operator-Precedence Parsing.***

As explained in Section 3.2 in general, if an input string is accepted, a bottom-up parser should allow us to obtain the right parse of an input string—that is, the reverse sequence of rules according to which the rightmost derivation of the input string is made in the grammar the parser is based on. By ${}_{cond}G$*-op-parser*, we can obtain the right parse by writing out the applied reduction rules during the parsing process. Specifically, take the sequence of reduction rules in the second column of Figure 5.3 to obtain the right parse of $i \wedge (i \vee i)$ in ${}_{cond}G$; the details of this straightforward task are left as an exercise.

Configuration	*Rule*	*Parse Tree*
$\underline{\blacktriangleright}\blacklozenge i \wedge (i \vee i)\blacktriangleleft$		
$\blacktriangleright \underline{i}\blacklozenge \wedge (i \vee i)\blacktriangleleft$	$C \rightarrow i$	$\underline{C}\langle i\rangle \wedge (i \vee i)$
$\underline{\blacktriangleright}\, C\blacklozenge \wedge (i \vee i)\blacktriangleleft$		
$\blacktriangleright C \underline{\wedge}\blacklozenge (i \vee i)\blacktriangleleft$		
$\blacktriangleright C \wedge \underline{(}\blacklozenge i \vee i)\blacktriangleleft$		
$\blacktriangleright C \wedge (\underline{i}\blacklozenge \vee i)\blacktriangleleft$	$C \rightarrow i$	$C\langle i\rangle \wedge (\underline{C}\langle i\rangle \vee i)$
$\blacktriangleright C \wedge \underline{(}C\blacklozenge \vee i)\blacktriangleleft$		
$\blacktriangleright C \wedge (C \underline{\vee}\blacklozenge i)\blacktriangleleft$		
$\blacktriangleright C \wedge (C \vee \underline{i}\blacklozenge)\blacktriangleleft$	$C \rightarrow i$	$C\langle i\rangle \wedge (C\langle i\rangle \vee \underline{C}\langle i\rangle)$
$\blacktriangleright C \wedge (C \underline{\vee} C\blacklozenge)\blacktriangleleft$	$C \rightarrow C \vee C$	$C\langle i\rangle \wedge (\underline{C}\langle C\langle i\rangle \vee C\langle i\rangle\rangle)$
$\blacktriangleright C \wedge \underline{(}C \blacklozenge)\blacktriangleleft$		
$\blacktriangleright C \wedge (C\underline{)}\blacklozenge\blacktriangleleft$	$C \rightarrow (C)$	$C\langle i\rangle \wedge \underline{C}\langle (C\langle C\langle i\rangle \vee C\langle i\rangle\rangle)\rangle$
$\blacktriangleright C \underline{\wedge} C\blacklozenge\blacktriangleleft$	$C \rightarrow C \wedge C$	$\underline{C}\langle C\langle i\rangle \wedge C\langle (C\langle C\langle i\rangle \vee C\langle i\rangle\rangle)\rangle\rangle$
$\underline{\blacktriangleright}\, C\blacklozenge\blacktriangleleft$		

Figure 5.3 ***Construction of Parse Tree by Operator-Precedence Parser.***

■

Construction of Operator Precedence Table

We approach to the construction of *G-op-table* in a rather pragmatic way. Indeed, we use common sense and elementary mathematical rules to construct this table by which *G-op-parser* controls the parsing process. Consider $\lfloor$ and $\rfloor$ as relations over *G*'s terminals. Intuitively, for two terminals, *a* and *b*, $a\lfloor b$ means that *a* has a lower precedence than *b*, so a handle containing *a* is reduced after a handle containing *b*. Regarding the other relation, $a\rfloor b$ says that *a* has a precedence before *b*, meaning that a handle containing *a* is reduced before a handle containing *b*. To obtain $\lfloor$ and $\rfloor$ so they fulfill the handle-delimiting role as described above, we define both relations based upon the mathematical rules concerning the precedence and associativity as follows.

I. If *a* is an operator that has a higher mathematical precedence than operator *b*, then $a\rfloor b$ and $b\lfloor a$.

II. If *a* and *b* are left-associative operators of the same precedence, then $a\rfloor b$ and $b\rfloor a$. If *a* and *b* are right-associative operators of the same precedence, then $a\lfloor b$ and $b\lfloor a$.

III. If *a* can legally precede operand *i*, then $a\lfloor i$. If *a* can legally follow *i*, then $i\rfloor a$.

IV. If *a* can legally precede (, then $a\lfloor($. If *a* can legally follow (, then $(\lfloor a$. Similarly, if *a* can legally precede), then $a\rfloor)$, and if *a* can legally follow), then $)\rfloor a$.

In *G-op-table* (see Figure 5.1), we specify $\lfloor$ and $\rfloor$ so that if $a\lfloor b$ with $a \in {}_G\Delta \cup \{\blacktriangleright\}$, $b \in {}_G\Delta \cup \{\blacktriangleleft\}$, then ${}_{cond}G$*-op-table* $[a, b] = \lfloor$, and analogously, if $a\rfloor b$, then ${}_{cond}G$*-op-table* $[a, b] = \rfloor$. In addition, we set *G-op-table*[(,)] to $|$. Furthermore, we set ${}_{cond}G$*-op-table* $[\blacktriangleright, \blacktriangleleft]$ = ☺ to express a successful completion of the parsing process. Finally, to complete *G-op-table*, we fill all the remaining entries with ☹ to specify a syntax error.

Case Study 17/35 ***Construction of the Operator Precedence Parsing Table.*** Specifically, in terms of operators occurring as terminals in ${}_{cond}G$, suppose that $\vee$ and $\wedge$ satisfy the standard mathematical precedence and associative rules. That is, $\wedge$ has a precedence before $\vee$, and both operators are left-associative. From I, as $\wedge$ has a precedence before $\vee$, $\wedge\rfloor\vee$ and $\vee\lfloor\wedge$. From II, since ∧ is left-associative, we have ∧⌋∧. Regarding *i*, as ∨ can legally precede *i*, we have ∨⌊*i* according to III. Considering the parentheses, IV implies ∧⌊(. Complete the definition of ⌊ and ⌋ as an exercise.

In $_{cond}$*G-op-table* (see Figure 5.1), we specify ⌊ and ⌋ so that if *a*⌊*b* with *a* ∈ {∨, ∧, (,), *i*, ▶}, *b* ∈ {∨, ∧, (,), *i*, ◀}, then $_{cond}$*G-op-table* [*a*, *b*] = ⌊, and analogously, if *a*⌋*b*, then $_{cond}$*G-op-table* [*a*, *b*] = ⌋. Furthermore, we set $_{cond}$*G-op-table*[(,)] to |, which forces $_{cond}$*G-op-parser* to apply $C \rightarrow (C)$. We set $_{cond}$*G-op-table* [▶,◀] = ☺ to express a successful completion of the parsing process. Finally, to complete $_{cond}$*G-op-table*, we fill all the remaining entries with ☹ to specify a syntax error.

■

Handling Errors

In operator precedence parsing, we distinguish two basic kinds of errors:

- table-detected errors
- reduction errors

Table-Detected Errors. If *G-op-table*[*pd-top-terminal, ins*$_1$] = ☹, an error is detected by the table. To handle it, the parser modifies its pushdown or the input by changing, inserting, or deleting some symbols. Whatever the parser does, it has to guarantee a recovery from the error. After the recovery, it resumes the parsing process so that all the input string is eventually processed. The precise modification depends on the type of an error that the table detects, and the exact recovery action chosen always requires good intuition and ingeniousness from the compiler's author, who may even consult some other computer science disciplines, such as artificial intelligence or computational psychology, to select the best possible action. When *G-op-parser* reaches $_{cond}$*G-op-table*[*pd-top-terminal, ins*$_1$] = ☹, the error-recovery routine corresponding to this entry is performed.

Reduction Errors. If *G-op-table*[*pd-top-terminal, ins*$_1$] = ⌋, *G-op-parser* is supposed to perform a reduction. If there is no rule by which *G-op-parser* can make this reduction, a reduction error occurs. More precisely, this error occurs when no grammatical rule has the right-hand side equal to *G-op-handle*. After issuing a diagnostic of this error, *G-op-parser* selects a *recovery rule* $A \rightarrow x$ such that *x* can be obtained from *G-op-handle* by a slight modification, such as the deletion of a single symbol. To recover, it changes *G-op-handle* to *A* on the *pd* top, then resumes the parsing process.

G-op-parser can work more sophisticatedly and detect many errors of this kind earlier during the parsing process by some additional checks. Indeed, *G-op-parser* can make the *length-of-handle check* to verify that *G-op-handle* always occurs within the pushdown top consisting of no more than *j* symbols, where *j* is the length of the longest right-hand side of a rule in *G*. At this point, *G-op-parser* can narrow its attention to finitely many illegal strings of length *j* or less and associate a recovery rule with each of them in advance. Furthermore, a reduction error can be frequently detected earlier by a *valid handle-prefix check*. This check verifies that there is a rule *r* whose prefix occurs on the pushdown top; if there is none, a reduction error is inescapable.

Case Study 18/35 *Operator Precedence Parsing with Error Recovery*. Consider *i*(*i* ∨)◀ as the input expression parsed by $_{cond}$*G-op-parser*. With this obviously ill-formed expression, Algorithm 5.2, which represents $_{cond}$*G-op-parser*, interrupts its parsing process after making only the two steps described in Figure 5.4 because it reaches ☹ in the table, and ☹ means a syntax error. Like any other well-constructed parser, when a syntax error occurs, $_{cond}$*G-op-parser* detects and reports this error. Then, it takes an appropriate action to recover from it so that it can proceed, possibly handling some other errors appearing in the rest of the program.

Configuration	*Table Entry*	*Parsing Action*
$\underline{▶}$◆i (i ∨)◀	[▶, i] = ⌊	**SHIFT**
▶ $\underline{i}$◆(i ∨)◀	[i, (] = ☹	

Figure 5.4 ***Operator-Precedence Parsing of Erroneous Expression.***

When a recovery action leads to skipping some input symbols, then it represents a rather straightforward task. Consider, for instance, ${}_{cond}$*G-op-table*[▶,)] = ☹, which detects an error. To recover, ${}_{cond}$*G-op-parser* skips) by advancing to the input symbol behind this), issues an error diagnostic stating that an unbalanced right parenthesis has occurred, and continues with the parsing process. A recovery action that modifies *pd* is usually more difficult. Consider ${}_{cond}$*G-op-table*[i, (] = ☹. To recover, ${}_{cond}$*G-op-parser* first changes the *pd* top symbol, $pd_1 = i$, to C, then pushes ∧ onto the pushdown top. As a result, after this modification of the pushdown, ∧ is the topmost pushdown symbol under which C occurs. To complete this recovery, ${}_{cond}$*G-op-parser* issues an error diagnostic that an operator is missing and resumes the parsing process. Unfortunately, we can hardly determine whether the source program's author left out ∧ or ∨ in front of (. We can only speculate that the input left parenthesis might indicate that the author left out ∧ because the parentheses are normally used to change the common priorities of operators in the expressions; for instance, the parentheses are necessary in $i \wedge (i \vee i)$, but they are superfluous in $i \vee (i \wedge i)$. Therefore, we have chosen ∧ as the operator inserted onto the pushdown in this recovery action although we can never be absolutely sure that this is what the source program's author improperly left out. In Figure 5.5, we give ${}_{cond}$*G-op-table* with the error entries filled with the names of the error-recovery routines, ① through ⑥, schematically described next.

	∧	∨	i	(	)	◀
∧	⌋	⌋	⌊	⌊	⌋	⌋
∨	⌊	⌋	⌊	⌊	⌋	⌋
i	⌋	⌋	①	②	⌋	⌋
(	⌊	⌊	⌊	⌊	|	③
)	⌋	⌋	④	⑤	⌋	⌋
▶	⌊	⌊	⌊	⌊	⑥	☺

Figure 5.5 ***Operator Precedence Table with Error-Recovery Routines.***

① **configuration**: $pd_1 = i$ and $ins_1 = i$
diagnostic: missing operator between two *i*s
recovery: change pd_1 to C, then push ∧ onto the *pd* top

② **configuration**: $pd_1 = i$ and $ins_1 =$ (
diagnostic: missing operator between *i* and (
recovery: change pd_1 to C, then push ∧ onto the *pd* top

③ **configuration**: *pd-top-terminal* = (and ins_1 = ◀
diagnostic: missing right parenthesis
recovery: insert) in front of ins_1

④ **configuration**: $pd_1 =$) and $ins_1 = i$
diagnostic: missing operator between) and *i*
recovery: insert ∧ in front of ins_1, so *ins* starts with ∧ *i* after this insertion

⑤ **configuration**: $pd_1 =$) and $ins_1 =$ (
diagnostic: missing operator between) and (

recovery:	insert ∧ in front of ins_1, so *ins* starts with ∧ (after this insertion

⑥	**configuration**:	pd_1 = ▶ and ins_1 =)
	diagnostic:	unbalanced)
	recovery:	ignore ins_1 and advance to the next input symbol

To illustrate the reduction errors, consider ( )◀. By $_{cond}$*G-op-table*, $_{cond}$*G-op-parser* detects no error in this obviously incorrect expression. However, when ( ) occurs as the two-symbol pushdown top, a reduction error is reported because ( ) is $_{cond}$*G-op-handle* that does not coincide with the right-hand side of any rule. To recover, the parser replaces ( ) with (*C*) on the pushdown top and resumes the parsing process as usual. Observe that this reduction error is detectable earlier provided that $_{cond}$*G-op-parser* makes a valid handle-prefix check. Indeed, when (is on the pushdown top and the parser is about to shift) onto the pushdown, it checks whether there is a rule whose right-hand side begins with ( ). As there is none, the parser detects and reports a reduction error even before shifting) onto the pushdown. To recover, it pushes *C* onto the pushdown, then resumes the parsing process. Next, we describe three kinds of reduction errors, ❶ through ❸, from each of which $_{cond}$*G-op-parser* recovers by pushing *C* onto the pushdown top before it makes a shift.

❶	**configuration**:	pd_1 = (and ins_1 =)
	diagnostic:	no expression between parentheses
	recovery:	push *C* onto the *pd* top
❷	**configuration**:	$pd_1 \in \{\wedge, \vee\}$ and $ins_1 \notin \{i, (\}$
	diagnostic:	missing right operand
	recovery:	push *C* onto the *pd* top
❸	**configuration**:	$pd_1 \neq C$ and $ins_1 \in \{\wedge, \vee\}$
	diagnostic:	missing left operand
	recovery:	push *C* onto the *pd* top

Reconsider $\underline{▶}$◆*i* (*i* ∨)◀. Recall that with this expression, $_{cond}$*G-op-parser* without any error recovery stops after making two steps (see Figure 5.4). $_{cond}$*G-op-parser* that handles syntax errors by the routines described above works with this expression as described in Figure 5.6, which points up information related to handling errors. Observe that the parser recovers from both errors occurring in the expressions. As a result, $_{cond}$*G-op-parser* completes the parsing of this expression. The parsing process obviously ends by **REJECT** for the previously detected errors.

Configuration	*Table Entry*	*Parsing Action*
$\underline{▶}$◆*i* (*i* ∨)◀	[▶, *i*] = ⌊	**SHIFT**
▶ $\underline{i}$◆(*i* ∨)◀	[*i*, (] = ②	**table-detected error and recovery** ②
▶*C* $\underline{\wedge}$◆(*i* ∨)◀	[∧, (] = ⌊	**SHIFT**
▶*C* ∧ $\underline{(}$◆*i* ∨)◀	[(, *i*] = ⌊	**SHIFT**
▶*C* ∧ ($\underline{i}$◆∨)◀	[*i*, ∨] = ⌋	**REDUCE**(*C* → *i*)
▶*C* ∧ $\underline{(}$*C*◆∨)◀	[(, ∨] = ⌋	**SHIFT**
▶*C* ∧ (*C* $\underline{\vee}$◆)◀	[∨,)] = ⌋	**Reduction error and recovery** ❷
▶*C* ∧ (*C* $\underline{\vee}$ *C*◆)◀	[∨,)] = ⌋	**REDUCE**(*C* → *C* ∨ *C*)
▶*C* ∧ $\underline{(}$*C* ◆)◀	[(,)] = |	**SHIFT**
▶*C* ∧ (*C*$\underline{)}$◆◀	[),◀] = ⌋	**REDUCE**(*C* → (*C*))
▶*C* $\underline{\wedge}$ *C*◆◀	[∧,◀] = ⌋	**REDUCE**(*C* → *C* ∧ *C*)
$\underline{▶}$*C*◆◀	[▶,◀] = ⌋	**REJECT** because of errors ② and ❷

Figure 5.6 ***Operator Precedence Parsing with Error-Recovery Routines.***

In the second step, ${}_{cond}G$-*op-table* detects that an operator is missing between *i* and (, and recovery routine ② handles this error by replacing *i* with *C* and, then, pushing $\wedge$ on the *pd* top. In the seventh step, ${}_{cond}G$-*op-parser* detects a reduction error because $\vee$ occur on the pushdown top and the next input symbol equals). Indeed, at this point, $C \vee$ forms ${}_{cond}G$-*op-handle* but there is no rule with $C \vee$ as the right-hand side. To recover, recovery routine ❷ pushes *C* onto the *pd* top. As a result, ${}_{cond}G$-*op-parser* completes the parsing of $i(i \vee)$ rather than gets stuck inside of this expression like in Figure 5.4.

■

Generalization

Of course, apart from logical expressions defined by ${}_{cond}G$, the operator precedence parser can elegantly handle most other expressions that occur in programming languages.

Case Study 19/35 ***Generalization of Operator Precedence Parsing.*** In this case study, we first explain how the operator precedence parser handles unary operators. Then, we modify this parser for arithmetic expressions. Finally, we note that this parser works well with both ambiguous and unambiguous grammars.

Unary Operators. Consider $\neg$ as an unary operator that denotes a logical negation. To incorporate this operator, we extend ${}_{cond}G$ by adding $C \rightarrow \neg C$ to obtain the grammar defined as

$$\begin{aligned}
&C \rightarrow \neg C\\
&C \rightarrow C \vee C\\
&C \rightarrow C \wedge C\\
&C \rightarrow (C)\\
&C \rightarrow i
\end{aligned}$$

Assume that $\neg$ satisfies the standard precedence and associative rules used in logic. That is, $\neg$ is a right-associative operator having a higher precedence than $\wedge$ and $\vee$. Return to rules I through IV, preceding Case Study 17/35 in this section. By using these rules, we easily obtain the table that includes this unary operator (see Figure 5.7).

	$\neg$	$\wedge$	$\vee$	i	(	)	◀
$\neg$	⌊	⌋	⌋	⌊	⌊	⌋	⌋
$\wedge$	⌊	⌋	⌋	⌊	⌊	⌋	⌋
$\vee$	⌊	⌊	⌋	⌊	⌊	⌋	⌋
i	⌋	⌋	⌋	☹	☹	⌋	⌋
(	⌊	⌊	⌊	⌊	⌊	|	☹
)	⌋	⌋	⌋	☹	☹	⌋	⌋
▶	⌊	⌊	⌊	⌊	⌊	☹	☺

Figure 5.7 ***Operator Precedence Table with Unary Operator*** $\neg$.

Arithmetic Expressions with Right-Associative Operators. Consider grammar ${}_{expr}H$ for arithmetic expressions (see Case Study 6/35 in Section 3.1), defined as

$$E \rightarrow E + E,\ E \rightarrow E - E,\ E \rightarrow E * E,\ E \rightarrow E / E,\ E \rightarrow (E),\ E \rightarrow i$$

In addition, add $E \rightarrow E{\uparrow}E$ to these rules, where $\uparrow$ denotes the operator of exponentiation. Assume that these operators satisfy the standard arithmetic precedence and associative rules. That is, $\uparrow$ has a precedence before * and /, which have a precedence before + and –. The exponentiation operator $\uparrow$ is right-associative while the others are left-associative. The precedence table for this grammar is straightforwardly made by construction rules I through IV (see Figure 5.8).

	$\uparrow$	*	/	+	–	*i*	(	)	◀
$\uparrow$	⌊	⌋	⌋	⌋	⌋	⌊	⌊	⌋	⌋
*	⌊	⌋	⌋	⌋	⌋	⌊	⌊	⌋	⌋
/	⌊	⌋	⌋	⌋	⌋	⌊	⌊	⌋	⌋
+	⌊	⌊	⌊	⌋	⌋	⌊	⌊	⌋	⌋
–	⌊	⌊	⌊	⌋	⌋	⌊	⌊	⌋	⌋
i	⌋	⌋	⌋	⌋	⌋	☹	☹	⌋	⌋
(	⌊	⌊	⌊	⌊	⌊	⌊	⌊	|	☹
)	⌋	⌋	⌋	⌋	⌋	☹	☹	⌋	⌋
▶	⌊	⌊	⌊	⌊	⌊	⌊	⌊	☹	☺

Figure 5.8 ***Arithmetic-Operator Precedence Table.***

Expressions involving relational operators are handled analogously, so we leave their discussion as an exercise.

Ambiguity. As opposed to most top-down parsers, such as the predictive parsers (see Section 5.8), the precedence parsers work with ambiguous grammars without any problems. In fact, all the previous precedence parsers discussed in this section are based on ambiguous grammars. As obvious, these parsers can be based upon unambiguous grammars, too. To illustrate, extend the unambiguous grammar ${}_{expr}G$ (see Case Study 6/35 in Section 3.1) so it can also generate the arithmetic expressions that contain the operator of exponentiation $\uparrow$. Let the resulting unambiguous grammar be defined as

$$E \rightarrow E + T, E \rightarrow E - T, E \rightarrow T, T \rightarrow T * F, T \rightarrow T / F, T \rightarrow F, F \rightarrow F \uparrow I, F \rightarrow I, I \rightarrow i, I \rightarrow (E)$$

Suppose that all the operators satisfy the same precedence and associative rules as in the equivalent ambiguous grammar above. By rules I through IV, we easily produce a precedence table for this unambiguous grammar. The resulting table coincides with the table given in Figure 5.8.

■

Restriction

As obvious, the precedence parsing works nicely for any expressions used in most high-level programming languages. However, this simple parsing method is not recommendable for all programming-language constructs because it places several significant restrictions on the grammars it is based on. Perhaps most significantly, this method rules out the ε-rules and the rules having the same right-hand side but different left-hand sides. A grammar satisfying these restrictions can hardly provide an elegant specification of all the high-level programming-language syntax. Therefore, various extended versions of precedence parsing methods, which work with less restricted grammars, were designed. Unfortunately, these extended versions are rather complicated, so they lose the simplicity and elegancy of the operator precedence parser described above.

In practice, the operator precedence parsers are frequently used in a combination with other parsers, including predictive parsers discussed in Section 4.2. At this point, the precedence parsers handle the syntax of expressions while the other parsers work with all the other syntax constructs, such as the general program flow constructs. Alternatively, many real compilers use LR parsers, discussed next.

5.2 LR Parsing

This section discusses the *LR parsers* (*L* stands for the *l*eft-to-right scan of tokens, and *R* is for the *r*ightmost derivation, which the bottom-up parsers construct in reverse as we know from Section 3.2). LR parsers are based on LR tables constructed from grammars. From some grammars, however, these tables cannot be built up; hence, the grammars for which there exist their LR tables are called *LR grammars*, and the languages that these grammars generate are *LR languages*. Throughout this section, we naturally restrict our attention to the LR grammars.

In practice, LR parsers belong to the most popular parsers for their several indisputable advantages. First, they work fast. Furthermore, they easily and elegantly handle syntax errors because they never shift an erroneous input symbol onto the pushdown, and this property obviously simplifies the error recovery process. Most importantly, out of all deterministic parsers, they are ultimately powerful because the family of LR languages equals the family of languages accepted by deterministic pushdown automata.

In this section, we first describe the fundamental LR parsing algorithm. Then, we explain how to construct the LR tables, which the algorithm makes use of. Finally, we discuss how this parsing method handles errors.

LR Parsing Algorithm

Consider an LR grammar, $G = ({}_G\Sigma, {}_GR)$, where ${}_G\Delta = \{t_1, \ldots, t_n\}$ and ${}_GN = \{A_1, \ldots, A_k\}$, for some n, $k \geq 1$. Its *G-based LR-table* consists of the *G-based action part* and the *G-based goto part*, denoted by ${}_G$*action* and ${}_G$*goto*, respectively. Both parts have their rows denoted by members of the set ${}_G\Theta = \{\theta_1, \ldots, \theta_m\}$, where $m \geq 1$, whose construction is described later in this section. The columns of ${}_G$*action* and ${}_G$*goto* are denoted by the symbols of ${}_G\Delta$ and ${}_GN$, respectively; recall that ${}_G\Delta$ and ${}_GN$ denote G's alphabets of terminals and nonterminals, respectively. For each $\theta_j \in {}_G\Theta$ and $t_i \in {}_G\Delta \cup \{\blacktriangleleft\}$, where $1 \leq j \leq m$ and $1 \leq i \leq n$, $action[\theta_j, t_i] \in {}_G\Theta \cup {}_GR \cup \{$☹, ☺$\}$ (see Figure 5.9). Frequently, ${}_GR$'s rules are labeled throughout this section, and instead of the rules themselves, only their labels are written in ${}_G$*action* for brevity (see Convention 3.2). For each $\theta_j \in {}_G\Theta$ and $A_i \in {}_GN$, where $1 \leq j \leq m$ and $1 \leq i \leq k$, $goto[\theta_j, A_i] \in {}_GN$ or $goto[\theta_j, A_i]$ is blank (see Figure 5.10).

Convention 5.3. As usual, whenever there is no danger of confusion, we omit G in the denotation above, so we simplify ${}_G\Theta$, ${}_G$*action*, ${}_G$*goto*, ${}_G\Delta$, and ${}_GN$ to Θ, *action*, *goto*, Δ, and N, respectively.

■

	t_1	…	t_i	…	t_n
θ_1					
⋮					
θ_j			$action[\theta_j, t_i]$		
⋮					
θ_m					

Figure 5.9 ***Action Part of LR Table.***

	A_1	…	A_i	…	A_k
θ_1					
⋮					
θ_j			$goto[\theta_j, A_i]$		
⋮					
θ_m					

Figure 5.10 ***Goto Part of LR-table.***

Goal. A G-based LR parser.

Gist. Like an operator precedence parser (see Algorithm 5.2), a LR parser scans w from left to right, and during this scan, it makes shifts and reductions. If $w \in L(G)$, it eventually accepts w; otherwise, it rejects w. During the parsing process, every configuration of the parser is of the form $\blacktriangleright o_m X_{m\text{-}1} o_{m\text{-}1} \ldots X_2 o_2 X_1 o_1 \blacklozenge v \blacktriangleleft$, where the os and the Xs are in Θ and Σ, respectively, and $v \in$ *suffixes*(w). As a result, a member of Θ always appears as the topmost pushdown symbol. The LR parser makes shifts and reductions in a specific LR way, though. Next, we describe operations **LR-REDUCE** and **LR-SHIFT** that denote the actions by which the LR parser makes its reductions and shifts, respectively (as usual, in the definition of **LR-REDUCE** and **LR-SHIFT**, we make use of the *pd* and *ins* notation introduced in Convention 3.8).

Definition 5.4 *Operations* LR-REDUCE *and* LR-SHIFT. Let $G = ({}_G\Sigma, {}_GR)$, ${}_G\Theta$, ${}_G action$, and ${}_G goto$ have the same meaning as above. In a G-based LR parser, we use operations **LR-REDUCE** and **LR-SHIFT** defined as follows

LR-REDUCE. If p: $A \rightarrow X_1X_2 \ldots X_n \in {}_GR$, $X_j \in {}_G\Sigma$, $1 \leq j \leq n$, for some $n \geq 0$ ($n = 0$ means $X_1X_2 \ldots X_n = \varepsilon$), $o_{n+1}X_no_n \ldots o_2X_2o_2X_1o_1$ occur as the $(2n+1)$-symbol *pd* top, where $o_k \in {}_G\Theta$, $1 \leq k \leq n + 1$, then **LR-REDUCE**(p) replaces $X_no_n \ldots o_2X_2o_2X_1o_1$ with Ah on the pushdown top, where $h \in {}_G\Theta$ is defined as $h = {}_G goto[o_{n+1}, A]$; otherwise, **REJECT**.

LR-SHIFT. Let $ins_1 = t$, where $t \in {}_G\Sigma$, and $action[pd_1, t] = o$, where $o \in {}_G\Theta$. At this point, **LR-SHIFT** extends *pd* by *to* and advances to the next input symbol, so after this action, *to* occurs as the two-symbol *pd* top (o is thus the topmost pushdown symbol at this point) and ins_1 refers to the input symbol occurring right behind t in the input string.

■

Algorithm 5.5 *LR Parser.*

Input • a LR grammar, $G = ({}_G\Sigma, {}_GR)$;
• an input string, w, with $w \in {}_G\Delta^*$;
• a G-based LR table consisting of *action* and *goto*.

Output • **ACCEPT** if $w \in L(G)$;
• **REJECT** if $w \notin L(G)$.

Method
```
begin
    pd := ▶θ_1 {θ_1 denotes the first row of action and goto};
    repeat
        case action[pd_1, ins_1] of
            in _GΘ:  LR-SHIFT;
            in _GR:  LR-REDUCE(p);
            ☹:  REJECT;
            ☺:  ACCEPT;
        end{case};
    until ACCEPT or REJECT
end.
```

To obtain the right parse of w, extend Algorithm 5.5 by writing out p whenever **LR-REDUCE**(p) occurs, where $p \in {}_GR$. This straightforward extension is left as an exercise.

Case Study 20/35 ***LR parsing algorithm.*** Consider the logical-expression grammar ${}_{cond}G$ (see Case Study 6/35 in Section 3.1), whose rules are

1: $E \rightarrow E \vee T$
2: $E \rightarrow T$
3: $T \rightarrow T \wedge F$
4: $T \rightarrow F$
5: $F \rightarrow (E)$
6: $F \rightarrow i$

where E is the start symbol. This grammar has its two-part LR table, consisting of *action* and *goto*, depicted in Figures 5.11 and 5.12, respectively. Both *action* and *goto* have their rows denoted by the members of $\Theta = \{\theta_1, \theta_2, \ldots, \theta_{12}\}$. The columns of *action* are denoted by ${}_{cond}G$'s terminals $\vee$, $\wedge$, (,), i, and ◀. The columns of *goto* are denoted by ${}_{cond}G$'s nonterminals E, T, and F.

	$\wedge$	$\vee$	i	(	)	◀
θ_1	☹	☹	θ_6	θ_5	☹	☹
θ_2	☹	θ_7	☹	☹	☹	☺
θ_3	θ_8	2	☹	☹	2	2
θ_4	4	4	☹	☹	4	4
θ_5	☹	☹	θ_6	θ_5	☹	☹
θ_6	6	6	☹	☹	6	6
θ_7	☹	☹	θ_6	θ_5	☹	☹
θ_8	☹	☹	θ_6	θ_5	☹	☹
θ_9	☹	θ_7	☹	☹	θ_{12}	☹
θ_{10}	θ_7	1	☹	☹	1	1
θ_{11}	3	3	☹	☹	3	3
θ_{12}	5	5	☹	☹	5	5

Figure 5.11 ${}_{cond}$***G-Based LR Table Action Part.***

	E	T	F
θ_1	θ_2	θ_3	θ_4
θ_2			
θ_3			
θ_4			
θ_5	θ_9	θ_3	θ_4
θ_6			
θ_7		θ_{10}	θ_4
θ_8			θ_{11}
θ_9			
θ_{10}			
θ_{11}			
θ_{12}			

Figure 5.12 ${}_{cond}$***G-Based LR Table Goto Part.***

With $i \wedge i \vee i \in L({}_{cond}G)$ as the expression, Algorithm 5.5 works as described in Figure 5.13. The algorithm makes the successful parsing of $i \wedge i \vee i$ by the sequence of configurations given in the first column of this figure. The second column gives the relevant entries of *action* and *goto*, and the third column specifies the actions made by the algorithm. Notice that *goto* is relevant only when a reduction is performed; regarding a shift, it is not needed at all.

Configuration	*Table Entry*	*Parsing Action*
$\blacktriangleright\theta_1\blacklozenge i \wedge i \vee i\blacktriangleleft$	$action[\theta_1, i] = \theta_6$	**LR-SHIFT**(i)
$\blacktriangleright\theta_1\, i\ \theta_6\blacklozenge\wedge i \vee i\blacktriangleleft$	$action[\theta_6, \wedge] = 6$, $goto[\theta_1, F] = \theta_4$	**LR-REDUCE**(6)
$\blacktriangleright\theta_1\, F\ \theta_4\blacklozenge\wedge i \vee i\blacktriangleleft$	$action[\theta_4, \wedge] = 4$, $goto[\theta_1, T] = \theta_3$	**LR-REDUCE**(4)
$\blacktriangleright\theta_1\, T\ \theta_3\blacklozenge\wedge i \vee i\blacktriangleleft$	$action[\theta_3, \wedge] = \theta_8$	**LR-SHIFT**($\wedge$)
$\blacktriangleright\theta_1\, T\ \theta_3 \wedge \theta_8\blacklozenge i \vee i\blacktriangleleft$	$action[\theta_8, i] = \theta_6$	**LR-SHIFT**(i)
$\blacktriangleright\theta_1\, T\ \theta_3 \wedge \theta_8\, i\ \theta_6\blacklozenge\vee i\blacktriangleleft$	$action[\theta_6, \vee] = 6$, $goto[\theta_8, F] = \theta_{11}$	**LR-REDUCE**(6)
$\blacktriangleright\theta_1\, T\ \theta_3 \wedge \theta_8\, F\ \theta_{11}\blacklozenge\vee i\blacktriangleleft$	$action[\theta_{11}, \vee] = 3$, $goto[\theta_1, T] = \theta_3$	**LR-REDUCE**(3)
$\blacktriangleright\theta_1\, T\ \theta_3\blacklozenge\vee i\blacktriangleleft$	$action[\theta_3, \vee] = 2$, $goto[\theta_1, E] = \theta_2$	**LR-REDUCE**(2)
$\blacktriangleright\theta_1\, E\ \theta_2\blacklozenge\vee i\blacktriangleleft$	$action[\theta_2, \vee] = \theta_7$	**LR-SHIFT**($\vee$)
$\blacktriangleright\theta_1\, E\ \theta_2 \vee \theta_7\blacklozenge i\blacktriangleleft$	$action[\theta_7, i] = \theta_6$	**LR-SHIFT**(i)
$\blacktriangleright\theta_1\, E\ \theta_2 \vee \theta_7\, i\ \theta_6\blacklozenge\blacktriangleleft$	$action[\theta_6, \blacktriangleleft] = 6$, $goto[\theta_7, F] = \theta_4$	**LR-REDUCE**(6)
$\blacktriangleright\theta_1\, E\ \theta_2 \vee \theta_7\, F\ \theta_4\blacklozenge\blacktriangleleft$	$action[\theta_4, \blacktriangleleft] = 4$, $goto[\theta_7, T] = \theta_{10}$	**LR-REDUCE**(4)
$\blacktriangleright\theta_1\, E\ \theta_2 \vee \theta_7\, T\ \theta_{10}\blacklozenge\blacktriangleleft$	$action[\theta_{10}, \blacktriangleleft] = 1$, $goto[\theta_1, E] = \theta_2$	**LR-REDUCE**(1)
$\blacktriangleright\theta_1\, E\ \theta_2\blacklozenge\blacktriangleleft$	**ACCEPT**	**ACCEPT**

Figure 5.13 $_{cond}G$***-Based LR Parsing.***

By writing out the rules according to which the LR parser makes each reduction, we obtain 64632641 as the right parse of $i \wedge i \vee i$ in $_{cond}G$.

■

Construction of LR Table

The parsing theory has developed many sophisticated methods of constructing the LR tables. Most of them are too complicated to include them into this introductory text, however. Therefore, we just restrict our explanation of this construction to the fundamental ideas underlying a *simple LR table construction*. As its name indicates, it is simpler than the other constructions of LR tables.

Let $G = ({}_G\Sigma, {}_GR)$ be an LR grammar. By using Algorithm 3.16 *Bottom-Up Parser for a Grammar*, turn G to the pushdown automaton $M = ({}_M\Sigma, {}_MR)$ that represents the general G-based bottom-up parser. Next, we explain how to construct the LR table from M.

Sets of Items. In every configuration, the *pd* top contains a handle prefix, which M tries to extend so a complete handle occurs on the *pd* top. As soon as the *pd* top contains a complete handle, M can make a reduction according to a rule $r \in {}_MR$ with ***rhs***(r) equal to the handle. To express that a handle prefix appears as the *pd* top, we introduce an *item* of the form

$$A \to x\blacklozenge y$$

for each rule $A \to z \in {}_GR$ and any two strings x and y such that $z = xy$. Intuitively, $A \to x\blacklozenge y$ means that if x occurs as the *pd* top and M makes a sequence of moves resulting into producing y right behind x on the *pd* top, then M gets z as the handle on the *pd* top, which can be reduced to A according to $z\blacklozenge \to A\blacklozenge \in {}_MR$.

Case Study 21/35 ***Items.*** Return to the six-rule grammar $_{cond}G$, discussed in the previous part of this case study. By Algorithm 3.16 given in Section 3.1, convert $_{cond}G$ to the general $_{cond}G$-based bottom-up parser, M. This conversion is analogical to the conversion described in Case Study 10/35 in Section 3.1, so we leave its details as an exercise. Figure 5.14 gives only $_{cond}G$'s rules together with the corresponding reduction rules in M.

From $E \to E \vee T$, we obtain these four items

$$E \to \blacklozenge E \vee T,\ E \to E\blacklozenge \vee T,\ E \to E \vee\blacklozenge T, \text{ and } E \to E \vee T\blacklozenge$$

Consider, for instance, $E \rightarrow E{\scriptstyle\blacklozenge} \vee T$. In essence, this item says that if E currently appears as the *pd* top and a prefix of the input is reduced to $\vee T$, the parser obtains $E \vee T$ as the handle, which can be reduced to E according to M's rule $E \vee T\blacklozenge \rightarrow E\blacklozenge$.

Rule in $_{cond}G$	***Reduction rule in*** M
$E \rightarrow E \vee T$	$E \vee T\blacklozenge \rightarrow E\blacklozenge$
$E \rightarrow T$	$T\blacklozenge \rightarrow E\blacklozenge$
$T \rightarrow T \wedge F$	$T \wedge F\blacklozenge \rightarrow E\blacklozenge$
$T \rightarrow F$	$F\blacklozenge \rightarrow T\blacklozenge$
$F \rightarrow (E)$	$(E)\blacklozenge \rightarrow F\blacklozenge$
$F \rightarrow i$	$i\blacklozenge \rightarrow F\blacklozenge$

Figure 5.14 $_{cond}G$'s ***Rules with*** M***'s Reduction Rules.***

Before going any further, notice that several different items may be relevant to the same *pd* top string. To illustrate, consider $E \rightarrow T{\scriptstyle\blacklozenge}$ and $T \rightarrow T{\scriptstyle\blacklozenge}\wedge$ F. They both are relevant to all configurations with T on the pushdown top. For instance, take these two configurations $\blacktriangleright T\blacklozenge\blacktriangleleft$ and $\blacktriangleright T\blacklozenge\wedge i\blacktriangleleft$. From the former, M makes

$$\blacktriangleright T\blacklozenge\blacktriangleleft \quad\Rightarrow\quad \blacktriangleright E\blacklozenge\blacktriangleleft$$

according to $T\blacklozenge \rightarrow E\blacklozenge$ while from the latter, M performs

$$\begin{aligned}
\blacktriangleright T\blacklozenge\wedge i\blacktriangleleft \quad &\Rightarrow \quad \blacktriangleright T \wedge\blacklozenge i\blacktriangleleft \\
&\Rightarrow \quad \blacktriangleright T \wedge i\blacklozenge\blacktriangleleft \\
&\Rightarrow \quad \blacktriangleright T \wedge F\blacklozenge\blacktriangleleft \\
&\Rightarrow \quad \blacktriangleright E\blacklozenge\blacktriangleleft
\end{aligned}$$

To give another example, take items $E \rightarrow E \vee T{\scriptstyle\blacklozenge}$ and $T \rightarrow T{\scriptstyle\blacklozenge}\wedge F$. Notice that both items have to be taken into account whenever M occurs in a configuration with the three-symbol string $E \vee T$ as the *pd* top. Consider, for instance, $\blacktriangleright E \vee T\blacklozenge\blacktriangleleft$ and $\blacktriangleright E \vee T\blacklozenge\wedge i\blacktriangleleft$. From $\blacktriangleright E \vee T\blacklozenge\blacktriangleleft$, M makes a reduction according to $E \rightarrow E \vee T$ and, thereby, successfully completes the parsing process. More formally, by using $E \vee T\blacklozenge \rightarrow E\blacklozenge$, M computes

$$\blacktriangleright E \vee T\blacklozenge\blacktriangleleft \quad\Rightarrow\quad \blacktriangleright E\blacklozenge\blacktriangleleft$$

In $\blacktriangleright E \vee T\blacklozenge\wedge i\blacktriangleleft$, M has actually T as a prefix of the handle $T \wedge F$ on the pushdown. As a result, from $\blacktriangleright E \vee T\blacklozenge\wedge i\blacktriangleleft$, M first makes several shifts and reductions before it obtains $T \wedge F$ as the handle on the pushdown top. Then, it reduces this handle to T according to $T \wedge F\blacklozenge \rightarrow T\blacklozenge$, after which it obtains $E \vee T$ as the handle, makes a reduction according to $E \vee T\blacklozenge \rightarrow E\blacklozenge$ and, thereby, completes the parsing process. To summarize this part of parsing process formally, from $\blacktriangleright E \vee T\blacklozenge\wedge i\blacktriangleleft$, M computes

$$\begin{aligned}
\blacktriangleright E \vee T\blacklozenge\wedge i\blacktriangleleft \quad &\Rightarrow \quad \blacktriangleright E \vee T \wedge\blacklozenge i\blacktriangleleft \\
&\Rightarrow \quad \blacktriangleright E \vee T \wedge\blacklozenge i\blacktriangleleft \\
&\Rightarrow \quad \blacktriangleright E \vee T \wedge i\blacklozenge\blacktriangleleft \\
&\Rightarrow \quad \blacktriangleright E \vee T \wedge F\blacklozenge\blacktriangleleft \\
&\Rightarrow \quad \blacktriangleright E \vee T\blacklozenge\blacktriangleleft \\
&\Rightarrow \quad \blacktriangleright E\blacklozenge\blacktriangleleft
\end{aligned}$$

■

As sketched in the conclusion of the previous case study, several items are frequently related to a prefix of the right-hand side of a rule when this prefix occurs on the *pd* top. Next, we use these prefixes as members of ${}_G\Theta$ and group together all items related to each of them into sets. By using these sets, we then construct the *G*-based LR table.

Set

$${}_G\Theta = \{x \mid x \in prefixes(\boldsymbol{rhs}(r)),\ r \in {}_GR\}$$

In what follows, we have to carefully distinguish between ${}_G\Theta$'s members and the sets of items they represent. Let x be a prefix of the right-hand side of a grammatical rule, so $x \in {}_G\Theta$. Let σ be the function that maps each $x \in {}_G\Theta$ to the set of items corresponding to x; accordingly, $\sigma({}_G\Theta)$ denotes $\{\sigma(x) \mid x \in {}_G\Theta\}$. In essence, $\sigma({}_G\Theta)$ contains all the items related to *M*'s configurations with x occurring on the pushdown top in the sense illustrated in the conclusion of the above case study. Before we give the algorithm that constructs the sets denoted by Θ's members, we make the following two observations I and II that underlie this construction.

I. Suppose that $A \rightarrow u\blacklozenge Bv \in \sigma(x)$, where $x \in {}_G\Theta$, and $B \rightarrow y \in P$. Of course, M may obtain B by a reduction according to $B \rightarrow y$. Consider item $B \rightarrow \blacklozenge y$ or, equivalently, $B \rightarrow \varepsilon\blacklozenge y$, which means that no proper prefix of y appears on the *pd* top. As a result, if x appears on the *pd* top, a reduction by $B \rightarrow y$ does not require any change of *pd* at all. Consequently, if $A \rightarrow u\blacklozenge Bv$ is in $\sigma(x)$, so is $B \rightarrow \blacklozenge y$. We thus obtain the following **if** statement that extends $\sigma(x)$ by $B \rightarrow \blacklozenge y$

```
if $A \rightarrow u\blacklozenge Bv \in \sigma(x)$ and $B \rightarrow y \in P$ then
    $\sigma(x) := \sigma(x) \cup \{B \rightarrow \blacklozenge y\}$
```

II. Suppose that Θ contains x and xX. Furthermore, let $A \rightarrow y\blacklozenge Xv \in \sigma(x)$. If M manages to reduce a part of the input to X, then it actually obtains X on the *pd* top. As a result, we extend $\sigma(xX)$ by $\mathrm{A} \rightarrow yX\blacklozenge v$. We thus obtain the following **if** statement that makes this extension

```
if $x, xX \in \Theta$ and $A \rightarrow y\blacklozenge Xv \in \sigma(x)$ then
    $\sigma(xX) := \sigma(xX) \cup \{\mathrm{A} \rightarrow yX\blacklozenge v\}$
```

Case Study 22/35 *Set* Θ. In terms of ${}_{cond}G$, discussed in the previous part of this case study,

$$\Theta = \{\varepsilon, E, T, F, i, E \vee, T \wedge, (E, E \vee T, T \wedge F, (E)\}$$

To illustrate extension I, suppose that $T \rightarrow T \wedge \blacklozenge F \in \sigma(T \wedge)$. At this point, we include $F \rightarrow \blacklozenge i$ and $F \rightarrow \blacklozenge(E)$ into $\sigma(T \wedge)$. Observe that this inclusion covers all possible cases by which the symbol F, following $\blacklozenge$ in $T \rightarrow T \wedge \blacklozenge F$, can subsequently occur as the pushdown top. To put it in terms of *M*'s reduction rules given in Figure 5.14, on the pushdown top, the extension of the current $T \wedge$ by F can result only from a reduction according to either $i\blacklozenge \rightarrow F\blacklozenge$ or $(E)\blacklozenge \rightarrow F\blacklozenge$.

To illustrate extension II, consider $\sigma(T)$, $\sigma(T \wedge)$ and $T \rightarrow T \blacklozenge\wedge F \in \sigma(T)$. At this point, the second **if** statement includes $T \rightarrow T \wedge \blacklozenge F$ into $\sigma(T \wedge)$. To continue, as $\sigma(T \wedge)$ now contains $T \rightarrow T \wedge \blacklozenge F$, this statement adds $T \rightarrow T \wedge F\blacklozenge$ to $\sigma(T \wedge F)$.

■

We are now ready to construct $\sigma(\Theta) = \{\sigma(x) \mid x \in \Theta\}$.

Goal. $\sigma({}_G\Theta)$ for an LR grammar, *G*.

Gist. We obtain $\sigma({}_G\Theta)$ by iterating the two extensions described above. To start this iteration, we change G's start symbol, S, to a new start symbol, Z, and add a dummy rule of the form $Z \rightarrow S$ to R. At this point, every derivation starts with $Z \rightarrow S$, so M completes every successful parse by reducing S to Z with its pushdown and input empty. Therefore, we initialize $\sigma(\varepsilon)$ equal to $\{Z \rightarrow {\blacklozenge}S\}$. Then, we iterate extensions I and II until no member of $\sigma({}_G\Theta)$ can be changed.

Algorithm 5.6 ***Sets of Items.***

Input • an LR grammar, $G = ({}_G\Sigma, {}_GR)$, extended by the dummy rule $Z \rightarrow S$, where Z is the new start symbol.

Output • $\sigma({}_G\Theta)$.

Method

```
begin
   $\sigma(\varepsilon) := \{Z \rightarrow {\blacklozenge}S\}$;
   repeat
      if $x, xX \in {}_G\Theta$ and $A \rightarrow u{\blacklozenge}Xv \in \sigma(x)$ then {extension II}
        $\sigma(xX) := \sigma(xX) \cup \{A \rightarrow uX{\blacklozenge}v\}$;
      if $x \in {}_G\Theta$ and $A \rightarrow u{\blacklozenge}Bv \in \sigma(x)$ and $B \rightarrow z \in {}_GR$ then {extension I}
          $\sigma(x) := \sigma(x) \cup \{B \rightarrow {\blacklozenge}z\}$
   until no change
end.
```

Case Study 23/35 ***Sets of Items.*** Consider again ${}_{cond}G$. Add a dummy rule of the form $Z \rightarrow E$ to its rules, and define Z as the start symbol to obtain this grammar

$Z \rightarrow E$
$E \rightarrow E \vee T$
$E \rightarrow T$
$T \rightarrow T \wedge F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow i$

Recall that $\Theta = \{\varepsilon, E, T, F, i, E \vee, T \wedge, (E, E \vee T, T \wedge F, (E)\}$ (see Case Study 22/35). Set $\sigma(\varepsilon) = \{Z \rightarrow {\blacklozenge}E\}$. In Algorithm 5.6, consider the first **if** statement of the **repeat** loop

```
if $x, xX \in {}_G\Theta$ and $A \rightarrow u{\blacklozenge}Xv \in \sigma(x)$ then {extension II}
    $\sigma(xX) := \sigma(xX) \cup \{A \rightarrow uX{\blacklozenge}v\}$
```

Since $\sigma(\varepsilon)$ contains $Z \rightarrow {\blacklozenge}E$ and the grammar contains $E \rightarrow E \vee T$, this **if** statement first includes $E \rightarrow E{\blacklozenge}\vee T$ into $\sigma(E)$. As $\sigma(E)$ now contains $E \rightarrow E{\blacklozenge} \vee T$, the same **if** statement subsequently places $E \rightarrow E \vee {\blacklozenge}T$ into $\sigma(E\vee)$. Consider the second **if** statement

```
if $x \in \Theta$ and $A \rightarrow u{\blacklozenge}Bv \in \sigma(x)$ and $B \rightarrow z \in P$ then {extension I}
    $\sigma(x) := \sigma(x) \cup \{B \rightarrow {\blacklozenge}z\}$
```

Since $\sigma(E\vee)$ contains $E \rightarrow E \vee {\blacklozenge}T$ and the grammar contains a rule of the form $T \rightarrow T \wedge F$, this statement places $T \rightarrow {\blacklozenge}T \wedge F$ into $\sigma(E\vee)$. As $\sigma(E\vee)$ contains $T \rightarrow {\blacklozenge}T \wedge F$ now and the grammar has $T \to F$ as one of its rules, this statement places $T \to \blacklozenge F$ into $\sigma(E\vee)$. Because $\sigma(E\vee)$ now contains $T \to \blacklozenge F$ and the set of grammatical rules contains $F \to i$, this statement subsequently includes $F \to \blacklozenge i$ into $\sigma(E\vee)$. Continuing in this way, this algorithm eventually produces $\sigma(\Theta)$. Figure 5.15 gives each $x \in \Theta$ in the first column together with the corresponding $\sigma(x)$ in the other column.

$x \in \Theta$	$\sigma(x)$
ε	$Z \to \blacklozenge E, E \to \blacklozenge E \vee T, E \to \blacklozenge T, T \to \blacklozenge T \wedge F, T \to \blacklozenge F, F \to \blacklozenge (E), F \to \blacklozenge i$
E	$Z \to E\blacklozenge, E \to E\blacklozenge \vee T$
T	$E \to T\blacklozenge, T \to T \blacklozenge\wedge F$
F	$T \to F\blacklozenge$
$($	$F \to (\blacklozenge E), E \to \blacklozenge E \vee T, E \to \blacklozenge T, T \to \blacklozenge T \wedge F, T \to \blacklozenge F, F \to \blacklozenge (E), F \to \blacklozenge i$
i	$F \to i\blacklozenge$
$E \vee$	$E \to E \vee\blacklozenge T, T \to \blacklozenge T \wedge F, T \to \blacklozenge F, F \to \blacklozenge (E), F \to \blacklozenge i$
$T \wedge$	$T \to T \wedge\blacklozenge F, F \to \blacklozenge (E), F \to \blacklozenge i$
$(E$	$F \to (E\blacklozenge), E \to E\blacklozenge \vee T$
$E \vee T$	$E \to E \vee T\blacklozenge, T \to T \blacklozenge\wedge F$
$T \wedge F$	$T \to T \wedge F\blacklozenge$
(E)	$F \to (E)\blacklozenge$

Figure 5.15 ***Sets in*** **$\sigma(\Theta)$.**

■

LR Table. Making use of $\sigma(\Theta)$, we now construct the LR table, consisting of *action* and *goto*, in a relatively simple way. To explain the construction of *goto*, consider item $A \to u\blacklozenge Bv \in \sigma(x)$, where $A, B \in N$ and $u, v \in \Sigma^*$. At this point, after reducing a portion of the input string to B, M actually extends the handle's prefix u by B, so uB occurs on the pushdown top. In terms of $\sigma(\Theta)$, from $A \to u\blacklozenge Bv \in \sigma(x)$, M proceeds to the set of $\sigma(\Theta)$ that contains $A \to uB\blacklozenge v$. That is, if $A \to uB\blacklozenge v \in \sigma(y)$, we define $goto[x, B] = y$. As a result, we have

```
**if** $A \to u\blacklozenge Bv \in \sigma(x)$ **and** $A \to uB\blacklozenge v \in \sigma(y)$ **then**
    $goto[x, B] = y$
```

Regarding *action*, by analogy with the construction of the *goto* entries, we obtain *action*'s shift entries as

```
**if** $A \to u\blacklozenge bv \in \sigma(x)$ **and** $A \to ub\blacklozenge v \in \sigma(y)$ **then**
    $action[x, b] = y$
```

where $b \in \Delta$. To explain the construction of the reduction entries in *action*, consider a rule i: $A \to u \in {}_GR$ and $A \to u\blacklozenge \in \sigma(x)$, which means that a complete handle u occurs on the pushdown top. At this point, the parser reduces u to A according to $u\blacklozenge \to A\blacklozenge$ provided that after this reduction, A is followed by a terminal a that may legally occur after A in a sentential form. As a result, we obtain

```
**if** $A \to u\blacklozenge \in \sigma(x)$ **and** $a \in follow(A)$ **and** $i$: $A \to u \in {}_GR$ **then**
    $action[x, a] = i$
```

(see Definition 4.3 and Algorithm 4.4 for the definition and construction of *follow*, respectively). As already noted, the parsing of the input is successfully completed when reducing S to Z according to 0: $Z \to S$ provided that the pushdown top symbol equals ▶, meaning that the pushdown is empty, and the input symbol equals ◀, meaning that all the input is completely read. Therefore,

```
if 0: $Z \rightarrow S\blacklozenge \in \sigma(x)$ then
    $action[x, ◀] = ☺$
```

Based upon the **if** statements given above, we next build up Algorithm 5.7 that constructs the *G*-based LR table.

Goal. LR table for a LR grammar, *G*.

Gist. Iterate the **if** statements given above until no table entry can be changed to obtain the resulting LR table.

Algorithm 5.7 *LR Table.*

Input • an LR grammar, $G = ({}_G\Sigma, {}_GR)$, in which Z is the new start symbol and 0: $Z \rightarrow S$ is the new dummy rule;
• $\sigma({}_G\Theta)$.

Output • a *G*-based LR-table, consisting of *action* and *goto*.

Note • Suppose that $A, B \in {}_GN$, $b \in {}_G\Delta$, and $u, v \in {}_G\Sigma^*$ in this algorithm.

Method

```
begin

    denote the rows of action and goto with ${}_G\Theta$'s members;
    denote the columns of action and goto with ${}_G\Delta$'s and ${}_GN$'s members, respectively;

    repeat
        if $A \rightarrow u\blacklozenge Bv \in \sigma(x)$ and $A \rightarrow uB\blacklozenge v \in \sigma(y)$ then
            $goto[x, B] = y$;                                    {goto entries}
        if $A \rightarrow u\blacklozenge bv \in \sigma(x)$ and $A \rightarrow ub\blacklozenge v \in \sigma(y)$ then
            $action[x, b] = y$;                                  {action shift entries}
        if $A \rightarrow u\blacklozenge \in \sigma(x)$ and $a \in follow(A)$ and $i: A \rightarrow u \in {}_GR$ then
            $action[x, a] = i$                                   {action reduction entries}
    until no change;

    if $Z \rightarrow S\blacklozenge \in \sigma(x)$ then
        $action[x, ◀] = ☺$;                                      {success}
    fill each blank entry of action with ☹                       {error}

end.
```

Case Study 24/35 *Construction of LR Table.* Consider the grammar from the previous part of this case study. Number its rules as

0: $Z \rightarrow E$
1: $E \rightarrow E \vee T$
2: $E \rightarrow T$

3: $T \to T \wedge F$
4: $T \to F$
5: $F \to (E)$
6: $F \to i$

Consider the members of Θ and the sets in $\sigma(\Theta)$ in Figure 5.15. In the **repeat** loop of Algorithm 5.7, consider the first **if** statement

```
if $A \to u\blacklozenge Bv \in \sigma(x)$ and $A \to uB\blacklozenge v \in \sigma(y)$ then
    $goto[x, B] = y$
```

As $E \to \blacklozenge E \vee T \in \sigma(\varepsilon)$ and $E \to E\blacklozenge \vee T \in \sigma(E)$, this **if** statement defines $goto[\varepsilon, E] = E$. Consider the second **if** statement in the **repeat** loop of Algorithm 5.7

```
if $A \to u\blacklozenge bv \in \sigma(x)$ and $A \to ub\blacklozenge v \in \sigma(y)$ then
    $action[x, b] = y$
```

Since $E \to E\blacklozenge\vee T \in \sigma(E)$ and $E \to E \vee\blacklozenge T \in \sigma(E \vee)$, $action[E, \vee] = E \vee$. Finally, consider the third **if** statement

```
if $A \to u\blacklozenge \in \sigma(x)$ and $a \in follow(A)$ and $i$: $A \to u \in {}_GR$ then
    $action[x, a] = i$
```

Because 2: $E \to T\blacklozenge \in \sigma(T)$ and $\vee \in follow(T)$, this statement defines $action[T, \vee] = E \to T$ or, to express this briefly by using the rule label, $action[T, \vee] = 2$. After leaving the **repeat** loop, the algorithm executes

```
if $Z \to S\blacklozenge \in \sigma(x)$ then
    $action[x, \blacktriangleleft] = ☺$
```

That is, it sets $action[E, \blacktriangleleft]$ = ☺ because $\sigma(E)$ contains $Z \to E\blacklozenge$. Finally, the algorithm fills each *action* entry that has not been filled out yet with ☹ to report a syntactical error. For instance, $action[E, i]$ = ☹.

For brevity, rename Θ's members as described in Figure 5.16. After this renaming, we obtain the LR-table *action* and *goto* parts given in Figures 5.11 and 5.12, respectively, as desired.

Old Name	*New Name*
ε	θ_1
E	θ_2
T	θ_3
F	θ_4
(	θ_5
i	θ_6
$E \vee$	θ_7
$T \wedge$	θ_8
$(E$	θ_9
$E \vee T$	θ_{10}
$T \wedge F$	θ_{11}
(E)	θ_{12}

Figure 5.16 ***Renaming*** **Θ's** ***Members.***

■

As already noted, compared to the other constructions of LR tables, the construction described in this section belongs to the simplest methods of obtaining LR tables. Unfortunately, there exist LR grammars for which this construction does not work. In fact, it breaks down even when some quite common grammatical phenomenon occurs. Specifically, it cannot handle *reduction-shift conflict*, whose decision requires to figure out whether shift or reduce when both actions are possible. Furthermore, if two or more different reductions can be made in the given configuration, it cannot decide which of them it should select. To give an insight into this latter *reduction-reduction conflict*, suppose that the same set in $\sigma(\Theta)$ contains two items, $A \rightarrow u\blacklozenge$ and $B \rightarrow v\blacklozenge$, but $follow(A) \cap follow(B) \neq \emptyset$. At this point, Algorithm 5.7 would illegally place both rules, $A \rightarrow u$ and $B \rightarrow v$, into the same reduction entry in the action part of the LR table. There exists a number of complicated constructions that resolve these conflicts. However, these constructions are far beyond this introductory text.

Handling Errors in LR Parsing

Compared to handling errors in precedence parsing, LR parsing handles syntax errors more exactly and elegantly. Indeed, LR parsing detects an error as soon as there is no valid continuation for the portion of the input thus far scanned. As a result, it detects all possible errors by using only the action part of the table; as a side effect, this property allows us to significantly reduce the size of the *goto* part by removing its unneeded blank entries. Next, we sketch two frequently used methods of LR error recovery. As before, we suppose that $G = ({}_G\Sigma, {}_GR)$ is an LR grammar, and M is a G-based LR parser, which uses LR table, consisting of *action* and *goto*.

A *panic-mode error recovery* represents a method that tries to isolate the shortest possible erroneous substring derivable from a selected nonterminal, skips this substring, and resumes the parsing process. This method has a selected set of nonterminals, O, which usually represents major program pieces, such as expressions or statements. In principle, this method finds the shortest string uv such that $u \in \Sigma^*$ is a string obtained from a current pushdown top $x \in (\Sigma\Theta)^*$ by deleting all symbols from Θ, and v is the shortest input prefix followed by an input symbol a from $follow(A)$, where $A \in O$ and $A \Rightarrow^* uv$. Let x be preceded by $o \in \Theta$ and $goto[o, A] = \theta$. To recover, this method replaces x with $A\theta$ on the pushdown top and skips the input prefix v. In this way, it pretends that the parser has reduced a portion of the input string that ends with v to A. After this, it resumes the parsing process from $action[\theta, a]$. As an exercise, we discuss this method in detail as well as its more sophisticated variants.

Ad-hoc recovery. This method resembles the way the precedence parser handles the table-detected errors. That is, this method consider each *action* entry equal to ☹, which only signalizes an error without any diagnostic specification. Based on a typical language usage, it decides the most probable mistake that led to the particular error in question, and accordingly to this decision, it designs an ingenious recovery procedure that takes an appropriate recovery action. This design always requires some ingeniousness from the compiler's author, who is encouraged to consult some other computer-science areas, ranging from computational psychology through artificial intelligence to the formal language theory, in order to design the best possible error recovery routine. Typically, this routine handles an error of this kind so that it modifies the pushdown or the input by changing, inserting, or deleting some symbols; whatever modification it performs, however, it has to guarantee that this modification surely avoids any infinite loop so that the parser eventually proceeds its normal process. Finally, each ☹ is replaced with the reference to the corresponding recovery routine. In practice, this method is very popular, and we illustrate its application in the next case study.

Case Study 25/35 ***LR Error Recovery.*** To demonstrate how LR parsing handles syntax errors by using the ad-hoc recovery method, consider the grammar ${}_{cond}G$ (see Case Study 6/35 in Section 3.1), defined as

1: $E \rightarrow E \vee T$
2: $E \rightarrow T$
3: $T \rightarrow T \wedge F$
4: $T \rightarrow F$
5: $F \rightarrow (E)$
6: $F \rightarrow i$

and its LR table consisting of *action* and *goto* given in Figures 5.11 and 5.12, respectively. Consider the ill-formed expression $i \vee$). With $i \vee$), the G-based LR parser, M interrupts its parsing process after making the six steps described in Figure 5.17 because it reaches ☹ in *action*, so the error recovery is in order.

Configuration	***Table Entry***	***Parsing Action***
▶θ_1◆$i \vee$)◀	*action*[θ_1, i] = θ_6	**LR-SHIFT**(i)
▶θ_1 i θ_6◆$\vee$)◀	*action*[θ_6, $\vee$] = 6, *goto*[θ_1, F] = θ_4	**LR-REDUCE**(6)
▶θ_1 F θ_4◆$\vee$)◀	*action*[θ_4, $\vee$] = 4, *goto*[θ_1, T] = θ_3	**LR-REDUCE**(4)
▶θ_1 T θ_3◆$\vee$)◀	*action*[θ_3, $\vee$] = 2, *goto*[θ_1, E] = θ_2	**LR-REDUCE**(2)
▶θ_1 E θ_2◆$\vee$)◀	*action*[θ_2, $\vee$] = θ_7	**LR-SHIFT**($\vee$)
▶θ_1 E θ_2 $\vee$ θ_7◆)◀	*action*[θ_8,)] = ☹	**REJECT**

Figure 5.17 ***LR Parse of*** $i \vee$) ***without Error Recovery.***

	$\wedge$	$\vee$	i	(	)	◀
θ_1	①	①	θ_6	θ_5	②	①
θ_2	①	θ_7	③	③	②	☺
θ_3	θ_8	2	③	③	2	2
θ_4	4	4	③	③	4	4
θ_5	①	①	θ_6	θ_5	②	①
θ_6	6	6	③	③	6	6
θ_7	①	①	θ_6	θ_5	②	①
θ_8	①	①	θ_6	θ_5	②	①
θ_9	①	θ_7	③	③	θ_{12}	①
θ_{10}	θ_7	1	③	③	1	1
θ_{11}	3	3	③	③	3	3
θ_{12}	5	5	③	③	5	5

Figure 5.18 ***Action Part of LR Table with Ad-Hoc Error-Recovery Routines.***

Next, we schematically describe four diagnoses together with the corresponding recovery procedures—① through ④, whose straightforward incorporation into Algorithm 5.5 is left as an exercise. Figure 5.18 presents a new version of *action* with the appropriate error-recovery routine instead of each ☹. With $i \vee$) as its input, the G-based LR parser makes the error recovery described in Figure 5.19.

①	**diagnostic**:	missing i or (
	recovery:	insert i θ_6 onto the pushdown
②	**diagnostic:**	unbalanced)
	recovery:	delete the input)
③	**diagnostic**:	missing operator
	recovery:	insert $\wedge$ θ_5 onto the pushdown

④ **diagnostic**: missing)
recovery: insert) θ_6 onto the pushdown

Configuration	*Table Entries*	*Parsing Action*
▶θ_1◆$i \vee$)◀	*action*[θ_1, i] = θ_6	**LR-SHIFT**(i)
▶θ_1 i θ_6◆$\vee$)◀	*action*[θ_6, $\vee$] = 6, *goto*[θ_1, F] = θ_4	**LR-REDUCE**(6)
▶θ_1 F θ_4◆$\vee$)◀	*action*[θ_4, $\vee$] = 4, *goto*[θ_1, T] = θ_3	**LR-REDUCE**(4)
▶θ_1 T θ_3◆$\vee$)◀	*action*[θ_3, $\vee$] = 2, *goto*[θ_1, E] = θ_3	**LR-REDUCE**(2)
▶θ_1 E θ_2◆$\vee$)◀	*action*[θ_2, $\vee$] = θ_7	**LR-SHIFT**($\vee$)
▶θ_1 E θ_2 $\vee$ θ_7◆)◀	*action*[θ_7,)] = ②	②—skip)
▶θ_1 E θ_2 $\vee$ θ_7◆◀	*action*[θ_7, ◀] = ①	①—push $i\theta_6$ onto *pd*
▶θ_1 E θ_2 $\vee$ θ_7 i θ_6◆◀	*action*[θ_6, ◀] = 6, *goto*[θ_7, F] = θ_4	**LR-REDUCE**(6)
▶θ_1 E θ_2 $\vee$ θ_7 F θ_4◆◀	*action*[θ_4, ◀] = 4, *goto*[θ_7, T] = θ_{10}	**LR-REDUCE**(4)
▶θ_1 E θ_2 $\vee$ θ_7 T θ_{10}◆◀	*action*[θ_{10}, ◀] = 1, *goto*[θ_1, E] = θ_2	**LR-REDUCE**(1)
▶θ_1 E θ_2◆◀	*action*[θ_2, ◀] = ☺	end with error reports

Figure 5.19 ***LR Parse of*** $i \vee$) ***with Ad-Hoc Error Recovery.***

■

Exercises

5.1. Write a program to implement Algorithm 5.2 *Operator Precedence Parser*, including operations **REDUCE** and **SHIFT** described in Definition 5.1.

5.2. Consider $_{cond}G$*-op-parser* in Case Study 16/35 *Operator Precedence Parser*. Express this parser quite rigorously as a pushdown transducer (see Definition 3.10) that produces the right parse of every sentence generated by $_{cond}G$.

5.3. In Section 5.1, for a grammar, G, we have constructed *G-op-table* based upon the precedence and associativity of operators occurring in G. Design a method that draws up this table directly from G. State the restrictions that G has to satisfy so this method can be used.

5.4. Write a program to implement

(a) the construction of *G-op-table* given in Section 5.1;
(b) the alternative construction of *G-op-table* made in Exercise 5.3.

5.5. Complete the definition of ⌊ and ⌋ sketched in Case Study 17/35 *Construction of the Operator Precedence Parsing Table*.

5.6. Consider operators o_1, o_2, o_3, o_4 in the following precedence table. Determine the precedence and associativity of these operators from the table.

	o_1	o_2	o_3	o_4
o_1	⌊	⌊	⌊	⌊
o_2	⌋	⌊	⌋	⌊
o_3	⌊	⌊	⌊	⌊
o_4	⌋	⌋	⌋	⌋

5.7. Write a program to implement the parser described in Case Study 18/35 *Operator Precedence Parsing with Error Recovery*, including all the error-recovery routines. Then, incorporate length-of-handle and valid handle-prefix checks, whose description precedes Case Study 18/35, into this parser. Finally, extend this parser by handling unary operators in the way described in Case Study 19/35 *Generalization of Operator Precedence Parsing*.

5.8*Solved*. Consider Figure 5.8 *Arithmetic-Operator Precedence Table* and an operator precedence parser based upon this table. Describe how this parser works on $(i * i) \uparrow i$.

5.9. By analogy with Exercise 5.7, write a program to implement a parser for arithmetic expressions. Design this parser based upon the ambiguous grammar ${}_{expr}H$ and the arithmetic-operator precedence table given in Figure 5.8. Then, modify it so it is based on the unambiguous grammar ${}_{expr}G$ (see Case Study 6/35). Does this modification affect the arithmetic-operator precedence table or the implementation of the parser?

5.10. Extend Algorithm 5.2 *Operator Precedence Parser* so it looks at two or more input symbols and at a longer pushdown top during every parsing step. Make this extension so the resulting operator precedence parser can elegantly handle more programming-language constructs than expressions.

5.11*Solved*. Give a two-rule LR grammar, G, such that (a) $L(G)$ is infinite and (b) G is not an LL grammar.

5.12. Write a program to implement

(a) Algorithm 5.5 *LR Parser*, including **LR-REDUCE** and **LR-SHIFT** (see Definition 5.4);
(b) Algorithm 5.6 *Sets of Items*;
(c) Algorithm 5.7 *LR Table*.

5.13. Return to grammar ${}_{cond}G$ (see Case Study 6/35 in Section 3.1). Consider $i \wedge ((i \vee i) \wedge i) \in L({}_{cond}G)$. Describe how Algorithm 5.5 works with this string by analogy with the description given in Figure 5.13 *${}_{cond}G$-based LR Parsing*.

5.14. Case Study 21/35 *Items* has included only some items corresponding to ${}_{cond}G$'s rules. List all of them.

5.15. Discuss Case Study 24/35 *Construction of LR Table* in detail. Specifically, reconsider its discussion in terms of all grammatical rules.

5.16. Illustrate reduction-shift conflicts and reduction-reduction conflicts by some specific examples of grammars.

5.17*Solved*. Consider the following grammar ${}_{prefix}G$ for expressions in Polish prefix notation (see Section 1.1). Construct an LR-table for this grammar and, subsequently, a LR parser based on this table. Describe how this parser works with $* i + i i$ and $i i * i +$.

1: $E \rightarrow + EE$
2: $E \rightarrow * EE$
3: $E \rightarrow i$

5.18. Consider the following grammar. Construct an LR-table for this grammar. Then, make an LR parser based on this table. Describe how this parser works with $i * i + i$.

1: $E \rightarrow E + T$

2: $E \rightarrow T$
3: $T \rightarrow T * i$
4: $T \rightarrow i$

5.19. Consider the following grammar. Construct an LR-table for this grammar. Then, make an LR parser based on this table. Describe how this parser works on **int** *i*; **real** *i*, *i*.

1: ⟨declaration list⟩ → ⟨declaration list⟩ ; ⟨type⟩⟨identifiers⟩
2: ⟨declaration list⟩ → ⟨type⟩⟨identifiers⟩
3: ⟨type⟩ → **int**
4: ⟨type⟩ → **real**
5: ⟨identifiers⟩ → ⟨identifiers⟩ , *i*
6: ⟨identifiers⟩ → *i*

5.20. Return to the panic-mode error recovery method in Section 5.2. Pay a special attention to the use of *follow* sets in this method. Improve this method by making use of both *first* and *follow* sets (see Definitions 4.1 and 4.3).

5.21. Extend the program that implements Algorithm 5.5 *LR Parser* obtained in Exercise 5.12 (a) by incorporating the improved panic-mode error recovery method obtained in Exercise 5.20.

5.22. Consider the error recovery procedures sketched in Case Study 25/35 *LR Error Recovery*. Incorporate them into Algorithm 5.5 *LR Parser*. Write a program to implement the resulting parser.

5.23*Solved*. Consider the following LR grammar.

1: $E \rightarrow EE +$
2: $E \rightarrow EE *$
3: $E \rightarrow i$

By the construction of the LR table described in Section 5.2, we obtain the following two tables. The first table represents the *action* part of the LR table. The other part resembles the *goto* part; however, some entries contain both θ_2 and θ_4, which is incorrect. Explain the reason of these conflicts. Change this table to a proper *goto* part of the LR table by removing either θ_2 or θ_4 from each of the three entries.

	i	+	*	◀
θ_1	θ_3	☹	☹	☹
θ_2	θ_3	☹	☹	☺
θ_3	3	3	3	3
θ_4	θ_3	θ_5	θ_6	☹
θ_5	1	1	1	1
θ_6	2	2	2	2

	E
θ_1	θ_2, θ_4
θ_2	θ_2, θ_4
θ_3	
θ_4	θ_2, θ_4
θ_5	
θ_6	

Solution to the Selected Exercises

5.8. The operator precedence parser works on $(i * i) \uparrow i$ as described in this table

Configuration	*Table Entry*	*Parsing Action*
$\underline{\blacktriangleright}\blacklozenge(i * i) \uparrow i\blacktriangleleft$	$[\blacktriangleright, (] = \lfloor$	**SHIFT**
$\blacktriangleright\underline{(}\blacklozenge i * i) \uparrow i\blacktriangleleft$	$[(, i] = \lfloor$	**SHIFT**
$\blacktriangleright(\underline{i}\blacklozenge * i) \uparrow i\blacktriangleleft$	$[i, *] = \rfloor$	**REDUCE**($E \rightarrow i$)
$\blacktriangleright\underline{(} E\blacklozenge * i) \uparrow i\blacktriangleleft$	$[(, *] = \lfloor$	**SHIFT**
$\blacktriangleright(E \underline{*}\blacklozenge i) \uparrow i\blacktriangleleft$	$[*, i] = \lfloor$	**SHIFT**
$\blacktriangleright(E * \underline{i}\blacklozenge) \uparrow i\blacktriangleleft$	$[i,)] = \rfloor$	**REDUCE**($E \rightarrow i$)
$\blacktriangleright(E \underline{*} E\blacklozenge) \uparrow i\blacktriangleleft$	$[*,)] = \rfloor$	**REDUCE**($E \rightarrow E * E$)
$\blacktriangleright\underline{(} E\blacklozenge) \uparrow i\blacktriangleleft$	$[(,)] = \mid$	**SHIFT**
$\blacktriangleright(E \underline{)}\blacklozenge \uparrow i\blacktriangleleft$	$[), \uparrow] = \rfloor$	**REDUCE**($E \rightarrow (E)$)
$\underline{\blacktriangleright}E\blacklozenge \uparrow i\blacktriangleleft$	$[\blacktriangleright, \uparrow] = \lfloor$	**SHIFT**
$\blacktriangleright E \underline{\uparrow}\blacklozenge i\blacktriangleleft$	$[\uparrow, i] = \lfloor$	**SHIFT**
$\blacktriangleright E \uparrow \underline{i}\blacklozenge\blacktriangleleft$	$[i, \blacktriangleleft] = \rfloor$	**REDUCE**($E \rightarrow i$)
$\blacktriangleright E \underline{\uparrow} E\blacklozenge\blacktriangleleft$	$[\uparrow, \blacktriangleleft] = \rfloor$	**REDUCE**($E \rightarrow E \uparrow E$)
$\underline{\blacktriangleright}E\blacklozenge\blacktriangleleft$	$[\blacktriangleright, \blacktriangleleft] = ☺$	**ACCEPT**

5.11. Define G as $S \rightarrow Sa$ and $S \rightarrow a$.

5.17. Throughout this solution, we make use of the notions and methods described in the construction of LR table in Section 5.2. First, we extend the grammatical rules by a dummy rule of the form 0: $Z \rightarrow E$ to obtain the grammar having these four rules 0: $Z \rightarrow E$, 1: $E \rightarrow +EE$, 2: $E \rightarrow *EE$, 3: $E \rightarrow i$. We determine Θ as $\Theta = \{\varepsilon, E, +, *, i, +E, *E, +EE, *EE\}$. For all $x \in \Theta$, we construct $\sigma(x)$ as follows. Initially, we set $\sigma(\varepsilon) = \{Z \rightarrow \blacklozenge E\}$. As rules 1 through 3 have E as the left-hand side, we add items $E \rightarrow \blacklozenge +EE$, $E \rightarrow \blacklozenge *EE$, $E \rightarrow \blacklozenge i$ to $\sigma(\varepsilon)$ to obtain $\sigma(\varepsilon) = \{Z \rightarrow \blacklozenge E, E \rightarrow \blacklozenge +EE, E \rightarrow \blacklozenge *EE, E \rightarrow \blacklozenge i\}$. Since $\sigma(\varepsilon)$ contains $Z \rightarrow \blacklozenge E$, we include $Z \rightarrow E\blacklozenge$ into $\sigma(E)$. Because $E \rightarrow \blacklozenge +EE \in \sigma(\varepsilon)$, we add $E \rightarrow +\blacklozenge EE$ to $\sigma(+)$. In addition, we add $E \rightarrow \blacklozenge +EE$, $E \rightarrow \blacklozenge *EE$, $E \rightarrow \blacklozenge i$ to $\sigma(+)$. Continuing in this way, we eventually obtain all the sets in $\sigma(\Theta)$ as given in Figure 5.20.

Member of Θ	*Set in* $\sigma(\Theta)$
ε	$Z \rightarrow \blacklozenge E, E \rightarrow \blacklozenge +EE, E \rightarrow \blacklozenge *EE, E \rightarrow \blacklozenge i$
E	$Z \rightarrow E\blacklozenge$
$+$	$E \rightarrow +\blacklozenge EE, E \rightarrow \blacklozenge +EE, E \rightarrow \blacklozenge *EE, E \rightarrow \blacklozenge i$
$*$	$E \rightarrow *\blacklozenge EE, E \rightarrow \blacklozenge +EE, E \rightarrow \blacklozenge *EE, E \rightarrow \blacklozenge i$
i	$E \rightarrow i\blacklozenge$
$+E$	$E \rightarrow +E\blacklozenge E, E \rightarrow \blacklozenge +EE, E \rightarrow \blacklozenge *EE, E \rightarrow \blacklozenge i$
$*E$	$E \rightarrow *E\blacklozenge E, E \rightarrow \blacklozenge +EE, E \rightarrow \blacklozenge *EE, E \rightarrow \blacklozenge i$
$+EE$	$E \rightarrow +EE\blacklozenge$
$*EE$	$E \rightarrow *EE\blacklozenge$

Figure 5.20 ***Sets in*** $\sigma(\Theta)$ ***for*** $_{prefix}G$**.**

In this way, we eventually obtain all the sets in $\sigma(\Theta)$ as given in Figure 5.20, so we are now ready to construct the LR table. Consider $\varepsilon \in \Theta$ and the corresponding set $\sigma(\varepsilon)$. As $Z \rightarrow \blacklozenge E \in \sigma(\varepsilon)$ and $Z \rightarrow E\blacklozenge \in \sigma(E)$, we set $goto[\varepsilon, E] = E$. Since $E \rightarrow \blacklozenge +EE \in \sigma(\varepsilon)$ and $E \rightarrow +\blacklozenge EE \in \sigma(+)$, we set $action[\varepsilon, +] = +$. Analogically, we obtain $action[\varepsilon, *] = *$ and $action[\varepsilon, i] = i$ from $E \rightarrow \blacklozenge *EE \in \sigma(\varepsilon)$ and $E \rightarrow \blacklozenge i \in \sigma(\varepsilon)$, respectively. Now, we take $Z \rightarrow E\blacklozenge \in \sigma(E)$, from which we derive $action[E, \blacktriangleleft] = ☺$. In the same way, we work with $\sigma(+)$ and $\sigma(*)$. Consider $E \rightarrow i\blacklozenge \in \sigma(i)$. As this item ends with $\blacklozenge$, we examine $Follow(E) = \{+, *, i, \blacktriangleleft\}$ to set $action[E, +]$, $action[E, *]$, $action[E, i]$, and $action[E, \blacktriangleleft]$ to 3 as the label denoting $E \rightarrow i$. The resulting *action* and *goto* parts of the LR table are given in Figure 5.21.

	i	+	*	◀	*E*
ε	*i*	+	*	☹	*E*
E	☹	☹	☹	☺	☹
+	*i*	+	*	☹	+*E*
*	*i*	+	*	☹	**E*
i	3	3	3	3	☹
+*E*	*i*	+	*	☹	+*EE*
**E*	*i*	+	*	☹	**EE*
+*EE*	1	1	1	1	☹
**EE*	2	2	2	2	☹

Figure 5.21 ***Action Part and Goto Part of*** $_{prefix}G$***-based LR Table.***

Rename ε, *E*, +, *, *i*, +*E*, **E*, +*EE*, and **EE* to θ_1, θ_2, θ_3, θ_4, θ_5, θ_6, θ_7, θ_8, and θ_9, respectively (see Figure 5.22).

	i	+	*	◀	*E*
θ_1	θ_5	θ_3	θ_4	☹	θ_2
θ_2	☹	☹	☹	☺	☹
θ_3	θ_5	θ_3	θ_4	☹	θ_6
θ_4	θ_5	θ_3	θ_4	☹	θ_7
θ_5	3	3	3	3	☹
θ_6	θ_5	θ_3	θ_4	☹	θ_8
θ_7	θ_5	θ_3	θ_4	☹	θ_9
θ_8	1	1	1	1	☹
θ_9	2	2	2	2	☹

Figure 5.22 ***Action Part and Goto Part of*** $_{prefix}G$***-based LR Table after Renaming.***

$_{prefix}G$-based LR parser rejects $i\ i * i +$ in the way described in Figure 5.23 and accepts $* i + i\ i$ in the way described in Figure 5.24.

Configuration	***Table Entry***	***Parsing Action***
▶θ_1◆ $i\ i * i +$ ◀	*action*[θ_1, *i*] = θ_5	**LR-SHIFT**(*i*)
▶$\theta_1\ i\ \theta_5$◆ $i * i +$ ◀	*action*[θ_5, *i*] = 3, *goto*[θ_1, *E*] = θ_2	**LR-REDUCE**(3)
▶$\theta_1\ i\ \theta_2$◆ $i * i +$ ◀	*action*[θ_2, *i*] = ☹	**REJECT**

Figure 5.23 $_{prefix}G$***-based LR Parser's Actions on*** $i\ i * i +$**.**

Configuration	***Table Entry***	***Parsing***
▶θ_1◆$* i + i\ i$◀	*action*[θ_1, *] = θ_4	**LR-SHIFT**(*)
▶$\theta_1 * \theta_4$◆$i + i\ i$◀	*action*[θ_4, *i*] = θ_5	**LR-SHIFT**(*i*)
▶$\theta_1 * \theta_4\ i\ \theta_5$◆$+ i\ i$◀	*action*[θ_5, +] = 3, *goto*[θ_4, *E*] = θ_7	**LR-REDUCE**(3)
▶$\theta_1 * \theta_4\ E\ \theta_7$◆$+ i\ i$◀	*action*[θ_7, +] = θ_3	**LR-SHIFT**(+)
▶$\theta_1 * \theta_4\ E\ \theta_7 + \theta_3$◆$i\ i$◀	*action*[θ_3, *i*] = θ_5	**LR-SHIFT**(*i*)
▶$\theta_1 * \theta_4\ E\ \theta_7 + \theta_3\ i\ \theta_5$◆$i$◀	*action*[θ_5, *i*] = 3, *goto*[θ_3, *E*] = θ_6	**LR-REDUCE**(3)
▶$\theta_1 * \theta_4\ E\ \theta_7 + \theta_3\ E\ \theta_6$◆$i$◀	*action*[θ_6, *i*] = θ_5	**LR-SHIFT**(*i*)
▶$\theta_1 * \theta_4\ E\ \theta_7 + \theta_3\ E\ \theta_6\ i\ \theta_5$◆◀	*action*[θ_5, ◀] = 3, *goto*[θ_6, *E*] = θ_8	**LR-REDUCE**(3)
▶$\theta_1 * \theta_4\ E\ \theta_7 + \theta_3\ E\ \theta_6\ E\ \theta_8$◆◀	*action*[θ_8, ◀] = 1, *goto*[θ_7, *E*] = θ_9	**LR-REDUCE**(1)
▶$\theta_1 * \theta_4\ E\ \theta_7\ E\ \theta_9$◆◀	*action*[θ_9, ◀] = 2, *goto*[θ_1, *E*] = θ_2	**LR-REDUCE**(2)
▶$\theta_1\ E\ \theta_2$◆◀	**ACCEPT**	**ACCEPT**

Figure 5.24 $_{prefix}G$***-based LR Parser's Actions on*** $* i + i\ i$**.**

5.23. Remove θ_4 from the entry in the first row and θ_2 from the others.

CHAPTER 6

Syntax Directed Translation and Intermediate Code Generation

If the source program is lexically and syntactically correct, the compiler translates this program into its machine-independent intermediate representation. This translation is usually directed by the syntax analyzer during the parse of the program, hence *syntax-directed translation*. During a parsing step according to a rule, this translation performs an action with *attributes* that are attached to the pushdown symbols the parsing step is performed with. Regarding the intermediate code generation, these attributes usually address fragments of the code produced so far from which the attached action creates a larger piece of the generated code by composing these fragments together. In this way, the action generates the code that determines how to interpret the rule it is attached to. From a broader perspective, these actions bridge the gap between the syntax analysis of a source program and its interpretation in terms of intermediate code. In this sense, they actually define the *meaning* of the rules they are attached to, and that is why these actions are often called *semantic actions*.

As explained in Section 3.2, the syntax analyzer verifies the syntactical structure of a source program by finding its parse tree. However, the syntax-directed translation does not implement this tree as the intermediate representation of the program because it contains several superfluous pieces of information, such as nonterminals, which are irrelevant to the rest of the compilation process. Instead, the syntax-directed translation usually produces an *abstract syntax tree* or, briefly, a *syntax tree* as the actual intermediate representation of the program. Derived from a parse tree, the syntax tree still maintains the essential source-program syntax skeleton needed to generate the resulting target code properly. However, it abstracts from any immaterial information regarding the remaining translation process, so it represents a significantly more succinct and, therefore, efficient representation of the program than its parse-tree original. Besides the syntax trees, we describe the generation of two important non-graphic intermediate representations, such as *three-address code* and *postfix notation*. Besides the generation of the intermediate code, however, the syntax-directed translation also performs various checks of the semantic analysis, such as the type checking.

Synopsis. In this chapter, we discuss the syntax-directed translation of the tokenized source programs to their intermediate representation. Most importantly, we explain how to generate the abstract syntax trees from the source programs in this way. Besides these trees, we describe the generation of an important non-graphic intermediate representation called *three-address code*.

In Section 6.1, we explain how a bottom-up parser directs the generation of various types of intermediate code for conditions and expressions. This variety of the generated intermediate code includes syntax trees, three-address code, and postfix notation. Then, we sketch the syntax directed translation under the guidance of a top-down parser in Section 6.2. Specifically, we describe what actions this translation makes during the top-down parse of declarations. In Section 6.3, we discuss the semantic analysis of a source program. As already noted, by analogy with the generation of the intermediate code, the semantic analysis consists in performing actions directed by a parser. A special attention is paid to the verification of the most common semantic aspects of the source program, such as type checking. As the semantic analysis heavily uses the symbol table, we outline some common symbol-table organizations in Section 6.4. Finally, in Section 6.5, we describe basic software tools that allow us to create a lexical analysis and syntax-directed translation automatically.

6.1 Bottom-Up Syntax-Directed Translation and Intermediate Code Generation

Consider a G-based bottom-up parser $M = ({}_M\Sigma, {}_MR)$ for a grammar $G = ({}_G\Sigma, {}_GR)$ (see Algorithm 3.16). The syntax-directed translation guided by M uses *pd* symbols with attributes. To express that an attribute a_i is attached to pushdown symbol pd_i, we write $pd_i\{a_i\}$, $1 \leq i \leq |pd|$. When M performs a reduction of a handle according to a rule from ${}_GR$, the syntax-directed translation performs an action attached to the rule with attributes of the symbols occurring in the handle. The action attached to r: $X_0 \rightarrow X_1X_2\ldots X_n \in {}_GR$ is described by using *formal attributes* \$\$, \$1, \$2, …, \$*n* attached to the symbols $X_0, X_1, X_2, \ldots, X_n$ in r so that \$\$ is the attribute associated with X_0 and \$*j* is the attribute associated with the *j*th symbol X_j on the right-hand side of $X_0 \rightarrow X_1X_2\ldots X_n$. We denote this action by **ACTION**(\$\$, \$1, \$2, …, \$*n*) and place it behind r to obtain the *attribute rule* of the form

$$X_0 \rightarrow X_1X_2\ldots X_n \ \ \{\textbf{ACTION}(\$\$, \$1, \$2, \ldots, \$n)\}$$

To make the association between X_j and \$*j* quite explicit, we sometimes attach {\$*j*} behind X_j in the attribute rule as $X_i\{\$j\}$ and write, in greater detail,

$$X_0\{\$\$\} \rightarrow X_1\{\$1\}X_2\{\$2\}\ldots X_n\{\$n\} \ \ \{\textbf{ACTION}(\$\$, \$1, \$2, \ldots, \$n)\}.$$

On the other hand, in practice, there is often no need to attach an attribute to a symbol occurring in a rule; at this point, the attribute rule simply omits this attribute. By changing G's rules to attribute rules, we obtain an *attribute G-based grammar*.

Let

$$X_1\{a_1\}\ X_2\{a_2\}\ \ldots X_n\{a_n\}$$

be the n top *pd* symbols with their *actual attributes* a_1 through a_n. Suppose that these n symbols form the handle that M reduces by the above rule. During this reduction, the M-directed translation **ACTION**(\$\$, \$1, \$2, …, \$*n*) with $\$j = a_j$, $1 \leq j \leq n$, and $a_0 = \$\$$, where a_0 is the newly determined actual attribute of X_0 occurring as the *pd* top after this reduction. As a result, $X_1\{a_1\}$ $X_2\{a_2\}$ $\ldots X_n\{a_n\}$ is changed to $X_0\{a_0\}$ on *pd*. Regarding shifts, M always shifts an input symbol a onto *pd* with its attribute \$*a*, obtained during the lexical analysis (see Section 2.2).

Throughout this section, we guide the syntax-directed translation by ${}_{cond}$*G-op-parser*, constructed in Section 5.1, or by its straightforward variants. We concentrate our attention on the translation of logical and arithmetic FUN expressions to their intermediate code, in which we use intermediate-code operators given in Figure 6.1. Its first and second columns give the intermediate-code and the FUN source-program operators, respectively. Notice that while the logical source-program operators differ from their intermediate-code operators, the arithmetic operators coincide.

IC-Operator	*SP-Operator*	*Interpretation*
O	$\vee$	logical *or*
A	$\wedge$	logical *and*
N	$\neg$	logical *not*
+	+	arithmetic addition
–	–	arithmetic subtraction
*	*	arithmetic multiplication
/	/	arithmetic division

Figure 6.1 ***Intermediate-Code and Source-Program Operators.***

6.1.1 Syntax Trees

Like a parse tree, a syntax tree graphically describes the syntax structure of a source program obtained during the syntax analysis. As opposed to the parse tree, however, the syntax tree is stripped of any superfluous information, such as nonterminals, regarding the rest of the translation process.

A *syntax tree* is a binary tree whose leaves and interior nodes are denoted with operands and operators, respectively. If an interior node has two children, it is denoted with a binary operator, and if it has a single child, it is denoted with a unary operator. In greater detail, let Ω be a finite set of *operators*. A syntax subtree rooted at a *unary operator u* has the form $u\langle n\rangle$, where $u \in \Omega$ denotes a unary operator and *n* is a syntax subtree. A syntax subtree rooted at *binary operator* has the form $b\langle n, m\rangle$, where $b \in \Omega$ denotes a binary operator, and *n* and *m* are two syntax subtrees.

The syntax-directed translation that generates syntax trees makes use of the following three actions attached to the grammatical rules. Before we describe these actions, however, let us recall tree-related notation introduced in Section 1.1. For a tree *t*, ☞*root*(*t*) denotes the pointer to *t*'s root. On the other hand, if *a* is a pointer to the root of a tree, then *tree*(*a*) denotes this tree as a whole. In terms of this section, if the root of a syntax tree is addressed by an attribute $*j*, *tree*($*j*) denotes the syntax tree with the root addressed by $*j*. Recall that an actual attribute *a* is substituted for $*j* during the performance of an action containing $*j* (see the beginning of the present section). The three actions follow next:

1. $$:= **LEAF**($*l*) allocates a new leaf, labels it with $*l*, and sets $$ to the pointer to this leaf. Specifically, consider *i*{☞*x*} where *i* is a token that specifies identifiers, *x* is a variable, and, as usual, ☞*x* is a pointer addressing the symbol-table entry in which the information about *x* is kept. Set $*l* to ☞*x*. Then, during the performance of $$:= **LEAF**($*l*), a new leaf is created, ☞*x* is copied into this newly created leaf, and $$ is set to the pointer to this leaf.

2. $$:= **TREE**(*o*, $*j*) creates the syntax tree *o*⟨*tree*($*j*)⟩, where *o* is a unary operator (see Figure 6.2) and sets $$ to the pointer to the root of this newly created tree. In greater detail, this action first allocates a new interior node, which has an operator field and a pointer field. Then, it copies *o* and ☞*tree*($*j*) into the operator field and the pointer field, respectively. Finally, it sets $$ to the pointer to the root of *o*⟨*tree*($*j*)⟩, symbolically denoted by ☞*o*⟨*tree*($*j*)⟩.

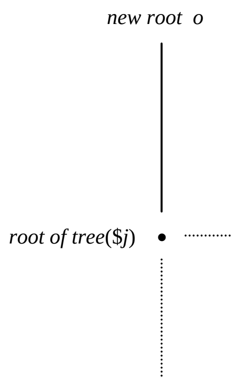


Figure 6.2 ***Syntax tree created by*** **TREE(*o*, $*j*).**

3. \$\$:= **TREE**(o, \$$j$, \$$k$) makes the syntax tree $o\langle tree(\$j), tree(\$k)\rangle$, where o is a binary operator (see Figure 6.3), then this action sets \$\$ to the pointer to the root of this tree. From an implementation point of view, this action allocates a new interior node, which consists of an operator field, a left-pointer field, and a right-pointer field filled with o, ☞$tree(\$j)$, and ☞$tree(\$k)$, respectively. Finally, this action sets \$\$ to the pointer to this newly created node, which becomes the root of the tree $o\langle tree(\$j), tree(\$k)\rangle$.

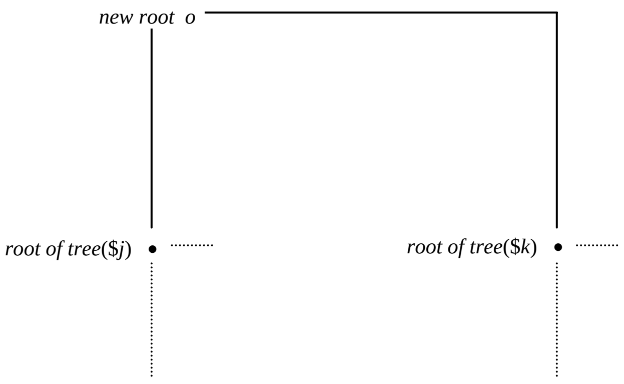


Figure 6.3 ***Syntax tree created by*** **TREE**(o, \$$j$, \$$k$)**.**

Case Study 26/35 ***Syntax Trees.*** Consider the grammar ${}_{cond}G$ (Case Study 6/35 in Section 3.1), defined as

$$C \to C \vee C$$
$$C \to C \wedge C$$
$$C \to (C)$$
$$C \to i$$

and the ${}_{cond}G$-based operator precedence parser, ${}_{cond}G$-*op-parser*, constructed in Section 5.1. As a matter of fact, we could guide the syntax directed translation by any correctly constructed ${}_{cond}G$-based bottom-up parser as demonstrated in the exercises. We have chosen ${}_{cond}G$-*op-parser* because Section 5.1 carefully described its construction for ${}_{cond}G$.

Let $x \wedge (y \vee z)$ be a source program, where x, y, and z are variables. Its tokenized version is $i\{☞x\} \wedge (i\{☞y\} \vee i\{☞z\})$. By omitting the attributes, we obtain $i \wedge (i \vee i)$, which is parsed by ${}_{cond}G$-*op-parser* as described in Figure 5.2. The resulting parse tree $C\langle C\langle i\rangle \wedge C\langle (C\langle C\langle i\rangle \vee C\langle i\rangle\rangle)\rangle\rangle$ and its construction are described in Figure 5.3. After the syntax analysis, however, the compiler does not need all the information, such as Cs, contained in this rather complicated parse tree. Indeed, the compiler only needs the syntax-related information necessary to create the code that executes $x \wedge (y \vee z)$ correctly. Consider the syntax tree $\mathbf{A}\langle ☞x, \mathbf{O}\langle ☞y, ☞z\rangle\rangle$, where **O** and **A** denote the logical *or* and *and* operations, respectively. As obvious, this succinct tree provides the compiler with necessary information that exactly specifies how to execute $x \wedge (y \vee z)$ and, thereby, the compiler can figure out how to generate its code; compared to the parse tree, however, it suppresses any irrelevant information to this generation. Next, we describe how the syntax directed translation guided by ${}_{cond}G$-*op-parser* converts $x \wedge (y \vee z)$ to $\mathbf{A}\langle ☞x, \mathbf{O}\langle ☞y, ☞z\rangle\rangle$.

To generalize ${}_{cond}G$ to a syntax-tree-generating attribute ${}_{cond}G$-based grammar, we attach a formal attribute to each C and assign a semantic action performed with these attributes to each rule. First, consider $C \to C \vee$ C. In terms of the notation introduced in the beginning of this chapter, we attach \$\$ to denote the formal attribute attached to C on the left-hand side of this rule;

on the right-hand side, \$1 and \$3 denote the attributes attached to the first C and the last C, respectively. We do not use \$2 in the attribute rule simply because there is no need to attach an attribute to the second symbol $\vee$ on the right-hand side. Next, we attach a semantic action to this rule to obtain the attribute rule

$$C\{\$\$\} \rightarrow C\{\$1\} \vee C\{\$3\} \;\; \{\$\$:= \mathbf{TREE}(\mathbf{O}, \$1, \$3)\},$$

where **O** represents the logical *or* operation. Analogically, we turn $C \rightarrow C \wedge C$ to

$$C\{\$\$\} \rightarrow C\{\$1\} \wedge C\{\$3\} \;\; \{\$\$:= \mathbf{TREE}(\mathbf{A}, \$1, \$3)\},$$

where **A** represents the logical *and* operation. The use of $C \rightarrow (C)$ does not lead to the construction of any syntax-tree part, so the attribute associated with C on the right-hand side simply becomes the attribute associated with C on the left-hand side as

$$C\{\$\$\} \rightarrow (\, C\{\$2\}\,) \;\; \{\$\$:= \$2\}$$

From $C \rightarrow i$, we make

$$C\{\$\$\} \rightarrow i\{\$1\} \;\; \{\$\$:= \mathbf{LEAF}(\$1)\}$$

where \$1 is attached to i, which is a terminal. As a result, during the translation, an actual attribute that is substituted for \$1 is a pointer to a symbol table entry that stores information about a variable obtained by a lexical analyzer.

Summing up all the rules above, we obtain the following syntax-tree-generating attribute ${}_{cond}G$-based grammar

$$\begin{array}{ll}
C\{\$\$\} \rightarrow C\{\$1\} \vee C\{\$3\} & \{\$\$:= \mathbf{TREE}(\mathbf{O}, \$1, \$3)\} \\
C\{\$\$\} \rightarrow C\{\$1\} \wedge C\{\$3\} & \{\$\$:= \mathbf{TREE}(\mathbf{A}, \$1, \$3)\} \\
C\{\$\$\} \rightarrow (\, C\{\$2\}\,) & \{\$\$:= \$2\} \\
C\{\$\$\} \rightarrow i\{\$1\} & \{\$\$:= \mathbf{LEAF}(\$1)\}
\end{array}$$

Reconsider $x \wedge (y \vee z)$ and its tokenized version $i\{☞x\} \wedge (i\{☞y\} \vee i\{☞z\})$. In Figure 6.4, consisting of two tables, we describe how the syntax-directed translation based on the above attribute grammar and guided by ${}_{cond}G$-*op-parser* translates $i\{☞x\} \wedge (i\{☞y\} \vee i\{☞z\})$ to the syntax tree $\mathbf{A}\langle ☞x, \mathbf{O}\langle ☞y, ☞z\rangle\rangle$. More specifically, the first column of the upper table describes ${}_{cond}G$-*op-parser*'s configurations, including actual attributes. In addition, this column underlines the handles that are reduced. The second column describes the shifts and reductions made by ${}_{cond}G$-*op-parser* (see Case Study 16/35 in Section 5.1 for the details). If the parsing step is reduction, the third column contains the action attached to the reduction rule in the syntax-tree-generating attribute ${}_{cond}G$-based grammar and, in addition, the generated syntax trees whose abbreviations, t_1 through t_5, are to be found in the lower table's first column. In this lower table, the second column gives the syntax trees, including the resulting syntax tree $\mathbf{A}\langle ☞x, \mathbf{O}\langle ☞y, ☞z\rangle\rangle$ in the last row. The third column of the lower table only repeats the actions corresponding to the reductions made by ${}_{cond}G$-*op-parser* except for the last but one reduction, which does not produce any part of the syntax tree.

Let us trace the performance of the actions attached to the six reductions in Figure 6.4 in detail (see the third column of the upper table in Figure 6.4). Consider the first reduction according $C \rightarrow i$, whose attribute version is $C\{\$\$\} \rightarrow i\{\$1\}$ $\{\$\$:= \mathbf{LEAF}(\$1)\}$. Recall that the handle with the actual attribute equals $i\{☞x\}$. During this reduction, the syntax-directed translation performs the attached action $\$\$:= \mathbf{LEAF}(\$1)$ by

• substituting ☞*x* for \$1;
• making **LEAF**(☞*x*), which creates the syntax tree t_1, consisting of a single leaf labeled with ☞*x*;
• setting the actual attribute attached to the left-hand side *C* to ☞t_1.

Configuration with Actual Attributes	*Parsing Action*	*Semantic Action*
$\blacktriangleright\blacklozenge i\{\text{☞}x\} \wedge (i\{\text{☞}y\} \vee i\{\text{☞}z\})\blacktriangleleft$	**SHIFT**	
$\blacktriangleright\underline{i\{\text{☞}x\}}\blacklozenge\wedge (i\{\text{☞}y\} \vee i\{\text{☞}z\})\blacktriangleleft$	**REDUCE**($C \rightarrow i$)	\$\$:= **LEAF**(☞*x*), t_1
$\blacktriangleright C\{\text{☞}t_1\}\blacklozenge\wedge (i\{\text{☞}y\} \vee i\{\text{☞}z\})\blacktriangleleft$	**SHIFT**	
$\blacktriangleright C\{\text{☞}t_1\} \wedge\blacklozenge(i\{\text{☞}y\} \vee i\{\text{☞}z\})\blacktriangleleft$	**SHIFT**	
$\blacktriangleright C\{\text{☞}t_1\} \wedge (\blacklozenge i\{\text{☞}y\} \vee i\{\text{☞}z\})\blacktriangleleft$	**SHIFT**	
$\blacktriangleright C\{\text{☞}t_1\} \wedge (\underline{i\{\text{☞}y\}}\blacklozenge\vee i\{\text{☞}z\})\blacktriangleleft$	**REDUCE**($C \rightarrow i$)	\$\$:= **LEAF**(☞*y*), t_2
$\blacktriangleright C\{\text{☞}t_1\} \wedge (C\{\text{☞}t_2\}\blacklozenge\vee i\{\text{☞}z\})\blacktriangleleft$	**SHIFT**	
$\blacktriangleright C\{\text{☞}t_1\} \wedge (C\{\text{☞}t_2\} \vee\blacklozenge i\{\text{☞}z\})\blacktriangleleft$	**SHIFT**	
$\blacktriangleright C\{\text{☞}t_1\} \wedge (C\{\text{☞}t_2\} \vee \underline{i\{\text{☞}z\}}\blacklozenge)\blacktriangleleft$	**REDUCE**($C \rightarrow i$)	\$\$:= **LEAF**(☞*z*), t_3
$\blacktriangleright C\{\text{☞}t_1\} \wedge (\underline{C\{\text{☞}t_2\} \vee C\{\text{☞}t_3\}}\blacklozenge)\blacktriangleleft$	**REDUCE**($C \rightarrow C \vee C$)	\$\$:= **TREE**(**O**, ☞t_2, ☞t_3), t_4
$\blacktriangleright C\{\text{☞}t_1\} \wedge (C\{\text{☞}t_4\}\blacklozenge)\blacktriangleleft$	**SHIFT**	
$\blacktriangleright C\{\text{☞}t_1\} \wedge \underline{(C\{\text{☞}t_4\})}\blacklozenge\blacktriangleleft$	**REDUCE**($C \rightarrow (C\,)$)	\$\$:= ☞t_4
$\blacktriangleright\underline{C\{\text{☞}t_1\} \wedge C\{\text{☞}t_4\}}\blacklozenge\blacktriangleleft$	**REDUCE**($C \rightarrow C \wedge C$)	\$\$:= **TREE**(**A**, ☞t_1, ☞t_4), t_5
$\blacktriangleright C\{\text{☞}t_5\}\blacklozenge\blacktriangleleft$	**ACCEPT**	

Abbreviation	*Syntax Tree*	*Semantic Action*
t_1	**leaf** ☞*x*	**\$\$:= LEAF(☞*x*)**
t_2	**leaf** ☞*y*	\$\$:= **LEAF**(☞*y*)
t_3	**leaf** ☞*z*	**\$\$:= LEAF(☞*z*)**
t_4	**O**⟨☞*y*, ☞*z*⟩	\$\$:= **TREE**(**O**, ☞t_2, ☞t_3)
t_5	**A**⟨☞*x*, **O**⟨☞*y*, ☞*z*⟩⟩	\$\$:= **TREE**(**A**, ☞t_1, ☞t_4)

Figure 6.4 ***Generation of Syntax Trees.***

Analogously, the syntax-directed translation performs the actions during the second and third reductions. During the second reduction, this translation creates t_2 consisting of a single leaf labeled with ☞*y*, and during the third reduction, it creates t_2 consisting of leaf ☞*z*. Consider the fourth reduction according to $C \rightarrow C \vee C$. Its attribute version is C\{\$\$\} $\rightarrow$ C\{\$1\} $\vee$ C\{\$3\} \{\$\$:= **TREE**(**O**, \$1, \$3)\}. The handle with the actual attributes equals C\{☞t_2\} $\vee$ C\{☞t_3\}. At this point, the syntax-directed translation performs \$\$:= **TREE**(**O**, \$1, \$3) so that it

• substitutes ☞t_2 and ☞t_3 for \$1 and \$3, respectively;
• carries out **TREE**(**O**, ☞t_2, ☞t_3), which creates t_4 of the form **O**⟨☞*y*, ☞*z*⟩;
• sets the actual attribute attached to the left-hand side *C* to ☞t_4 on *pd*.

Consider the fifth reduction according to $C \rightarrow (C)$, whose attribute version is C\{\$\$\} $\rightarrow$ (C\{\$2\}) \{\$\$:= \$2\}. The handle equals (C\{☞t_4\}) at this point. This action only sets the actual attribute attached to the *C* to ☞t_4. Consider the sixth and last reduction according to $C \rightarrow C \wedge C$, whose attribute version is C\{\$\$\} $\rightarrow$ C\{\$1\} $\wedge$ C\{\$3\} \{\$\$:= **TREE**(**A**, \$1, \$3)\}. The handle with the actual attributes equals C\{☞t_1\} $\wedge$ C\{☞t_4\}. The syntax-directed translation makes \$\$:= **TREE**(**A**, \$1, \$3) by

• substituting ☞t_1 and ☞t_4 for \$1 and \$3, respectively;
• performing **TREE**(**A**, ☞t_1, ☞t_4), which creates t_5 of the form **A**⟨☞*x*, **O**⟨☞*y*, ☞*z*⟩⟩;
• setting the actual attribute attached of the *C* to which the handle is reduced to ☞t_5.

Unary-Operator Syntax Trees. To explain the generation of unary-operator syntax-tree portions, we return to the grammar defined in Case Study 19/35 in Section 5.1 as

$$
\begin{aligned}
&C \rightarrow \neg C\\
&C \rightarrow C \vee C\\
&C \rightarrow C \wedge C\\
&C \rightarrow (C)\\
&C \rightarrow i
\end{aligned}
$$

From $C \rightarrow \neg C$, we make the attribute rule

$$C\{\$\$\} \rightarrow \neg C\{\$2\} \quad \{\$\$:= \mathbf{TREE}(\mathbf{N}, \$2)\}$$

where **N** is the intermediate code denoting the logical negation (see Figure 6.1), and the semantic action $\$\$:= \mathbf{TREE}(\mathbf{N}, \$2)$ produces the syntax tree $\mathbf{N}\langle\$2\rangle$, then sets the attribute associated with the left-hand side to the pointer to the root of this tree, ☞$\mathbf{N}\langle\$2\rangle$. As a result, we obtain the following syntax-tree-generating attribute grammar

$$
\begin{array}{ll}
C\{\$\$\} \rightarrow \neg C\{\$2\} & \{\$\$:= \mathbf{TREE}(\mathbf{N}, \$2)\}\\
C\{\$\$\} \rightarrow C\{\$1\} \vee C\{\$3\} & \{\$\$:= \mathbf{TREE}(\mathbf{O}, \$1, \$3)\}\\
C\{\$\$\} \rightarrow C\{\$1\} \wedge C\{\$3\} & \{\$\$:= \mathbf{TREE}(\mathbf{A}, \$1, \$3)\}\\
C\{\$\$\} \rightarrow (\ C\{\$2\}\ ) & \{\$\$:= \$2\}\\
C\{\$\$\} \rightarrow i\{\$1\} & \{\$\$:= \mathbf{LEAF}(\$1)\}
\end{array}
$$

In the way described in Section 5.1, create an operator-precedence parser based on the above grammar, including operator ¬, by using the operator precedence table given in Figure 5.7. Consider the logical expression ¬*y*, where *y* is a variable, and its tokenized version is ¬*i*{☞*y*}. In Figure 6.5, we describe how the syntax-directed translation based on the above attribute grammar and guided by the parser transforms ¬*i*{☞*y*} to the syntax tree **N**⟨☞*y*⟩. In this figure, the first column describes the parser's configurations, including actual attributes. The second column describes the parsing actions. If the parsing step is a reduction, the third column contains its attached semantic action and the first column underlines the handle being reduced.

Configuration	*Parsing Action*	*Semantic Action*
▶ ◆¬*i*{☞*y*}◀	**SHIFT**	
▶¬◆*i*{☞*y*}◀	**SHIFT**	
▶¬ $\underline{i\{\text{☞}y\}}$◀	**REDUCE**($C \rightarrow i$)	\$\$:= **LEAF**(☞*y*)
▶$\underline{\neg C\{\text{☞}\mathbf{leaf}\ \text{☞}y\}}$◆◀	**REDUCE**($C \rightarrow \neg C$)	\$\$:= **TREE**(**N**, ☞**leaf** ☞*y*)
▶*C*{☞**N**⟨☞*y*⟩}◆◀	**ACCEPT**	

Figure 6.5 ***Generation of Unary-Operator Syntax Trees.***

Arithmetic-Operator Syntax Trees. Naturally, the generation of arithmetic-operator syntax trees is analogical to the generation of logical-operator syntax trees. Indeed, consider the grammar ${}_{expr}G$ for arithmetic expressions (see Case Study 6/35 in Section 3.1), defined as

$$
\begin{aligned}
&E \rightarrow E + T\\
&E \rightarrow E - T\\
&E \rightarrow T\\
&T \rightarrow T * F\\
&T \rightarrow T / F\\
&T \rightarrow F
\end{aligned}
$$

$F \to (E)$
$F \to i$

By analogy with ${}_{cond}G$*-op-parser*, create the operator-precedence parser ${}_{expr}G$*-op-parser* as an exercise. By attaching actions to ${}_{expr}G$'s rules, we obtain the following syntax-tree-generating attribute ${}_{expr}G$-based grammar. In these actions, we denote the arithmetic addition, subtraction, multiplication, and division simply by +, –, *, and /, respectively. That is, we denote them by the same symbols as the corresponding terminals in ${}_{expr}G$'s rules.

$E\{\$\$\} \to E\{\$1\} + T\{\$3\}$	{$$:= **TREE**(+, $1, $3)}
$E\{\$\$\} \to E\{\$1\} - T\{\$3\}$	{$$:= **TREE**(–, $1, $3)}
$E\{\$\$\} \to T\{\$1\}$	{$$:= $1}
$T\{\$\$\} \to T\{\$1\} * F\{\$3\}$	{$$:= **TREE**(*, $1, $3)}
$T\{\$\$\} \to T\{\$1\} / F\{\$3\}$	{$$:= **TREE**(/, $1, $3)}
$T\{\$\$\} \to F\{\$1\}$	{$$:= $1}
$F\{\$\$\} \to (E\{\$2\})$	{$$:= $2}
$F\{\$\$\} \to i\{\$1\}$	{$$:= **LEAF**($1)}

Consider $E\{\$\$\} \to E\{\$1\} + T\{\$3\}$ {$$:= **TREE**(+, $1, $3)}. When the parser makes **REDUCE**($E \to E + T$), its action $$:= **TREE**(+, $1, $3) creates the syntax tree +⟨$1, $3⟩ and sets the attribute associated with the left-hand side $$ to the pointer to the root of this newly created tree. Analogously, describe the other actions of the attribute rules as an exercise.

Configuration with Attributes	*Parsing Action*	*Semantic Action*
▶ ◆ i{☞x} + i{☞y} / i{☞z}◀	**SHIFT**	
▶ $\underline{i\{☞x\}}$◆+ i{☞y} / i{☞z}◀	**REDUCE**($F \to i$)	$$:= **LEAF**(☞x), t_1
▶$\underline{F\{☞t_1\}}$◆+ i{☞y} / i{☞z}◀	**REDUCE**($T \to F$)	$$:= ☞t_1
▶$\underline{T\{☞t_1\}}$◆+ i{☞y} / i{☞z}◀	**REDUCE**($E \to T$)	$$:= ☞t_1
▶$\underline{E\{☞t_1\}}$◆+ i{☞y} / i{☞z}◀	**SHIFT**	
▶E{☞t_1} +◆i{☞y} / i{☞z}◀	**SHIFT**	
▶E{☞t_1} + $\underline{i\{☞y\}}$◆/ i{☞z}◀	**REDUCE**($F \to i$)	$$:= **LEAF**(☞y), t_2
▶E{☞t_1} + $\underline{F\{☞t_2\}}$◆/ i{☞z}◀	**REDUCE**($T \to F$)	$$:= ☞t_2
▶E{☞t_1} + T{☞t_2}◆/ i{☞z}◀	**SHIFT**	
▶E{☞t_1} + T{☞t_2}/◆i{☞z}◀	**SHIFT**	
▶E{☞t_1} + T{☞t_2} / $\underline{i\{☞z\}}$◆◀	**REDUCE**($F \to i$)	$$:= **LEAF**(☞z), t_3
▶E{☞t_1} + $\underline{T\{☞t_2\} / F\{☞t_3\}}$◆◀	**REDUCE**($T \to T / F$)	$$:= **TREE**(/, ☞t_2, ☞t_3), t_4
▶E{☞t_1} + $\underline{T\{☞t_4\}}$◆◀	**REDUCE**($E \to E + T$)	$$:= **TREE**(+, ☞t_1, ☞t_4), t_5
▶E{☞t_5}◆◀	**ACCEPT**	

Abbreviation	*Syntax Tree*	*Semantic Action Attached to Reduction*
t_1	**leaf** ☞x	**LEAF**(☞x)
t_2	**leaf** ☞y	**LEAF**(☞y)
t_3	**leaf** ☞z	**LEAF**(☞z)
t_4	/⟨☞y, ☞z⟩	**TREE**(/, ☞t_2, ☞t_3)
t_5	+⟨☞x, /⟨☞y, ☞z⟩⟩	**TREE**(+, ☞t_1, ☞t_4)

Figure 6.6 ***Generation of Arithmetic-Operator Syntax Trees.***

Consider $x + y / z$ as the input arithmetic expression. Its tokenized version is i{☞x} + i{☞y} / i{☞z}. By analogy with Figure 6.4, we describe how the syntax-directed translation creates +⟨☞x, /⟨☞y, ☞z⟩⟩ from i{☞x} + i{☞y} / i{☞z} in Figure 6.6, consisting of two tables. The first column of the upper table describes ${}_{expr}G$*-op-parser*'s configurations with actual attributes. As usual, the handles are underlined in the configurations. The second column describes the parsing actions made by ${}_{expr}G$*-op-parser*. If the parsing action is a reduction, the corresponding third-column entry contains the attached semantic action to this reduction, and if this action actually creates a new part of the syntax tree, the abbreviation of the syntax tree resulting from this action is included there, too. The syntax-tree abbreviations, t_1 through t_5, are in the lower table's first column while the second column of this table gives the corresponding part of the syntax tree. The third column of the lower table repeats the syntax-tree-producing actions attached to the reductions.

■

6.1.2 Three-Address Code

The instructions of three-address code represent simple commands that are easy to convert to an assembly language. In general, a *three-address instruction* is written as

$$[O, X, Y, Z]$$

whose components O, X, Y, and Z denote the instruction's operator, first operand, second operand, and result, respectively. X, Y, and Z are usually represented by the symbol-table addresses to variables, and that is why we refer to this intermediate representation as *three-address code*. If the second, third, or fourth component is a blank, this *nil component* is not used by the instruction. For instance, an *unary-operand three-address instruction* has this form

$$[O, X, \ , Z]$$

where O is an unary operator, X is the only operand of this instruction, the third component is nil, and Z is the result. This instruction sets Z to the result of operation O applied to X. In what follows, however, we most often use *binary-operator three-address instructions* of form

$$[O, X, Y, Z]$$

where O is a binary operator, X and Y are operands, and Z is the result (no component is nil). This instruction sets Z to the result of operation O applied to X and Y.

The three-address code of a single expression written in a high-level programming language usually consists of a long sequence of three-address instructions. Many of their operands are *temporary variables* that the compiler generates in the symbol-table just to temporarily hold semi-results needed in some subsequent instructions. Noteworthy, the syntax trees use no extra temporary variables, so their generation is faster and implementation is more succinct. That is why a compiler often first generates the syntax trees, which are subsequently turned to the corresponding three-address instructions because the three-address code is easy to optimize as demonstrated in Section 7.1.

Generation of Three-Address Code. We can generate the three-address code by semantic actions associated with grammatical rules in a similar way like the generation of the syntax trees. We describe the generation of the syntax trees in the following case study.

Convention 6.1. Throughout the rest of this book, we suppose that the programming language FUN represents logical values *true* and *false* by integers 1 and 0, respectively.

■

Case Study 27/35 *Three-Address Code.* We explain how to make the syntax-directed translation that produces the three-address code for the logical expressions generated by ${}_{cond}G$ (see Case Study 6/35 in Section 3.1). Recall that **O** and **A** denote the logical *or* and *and* operators, respectively (see Figure 6.1). Assume that **O** is applied to the variables x and y, and the result should be stored into variable z. The three-address instruction performing this operation has the form

[**O**, ☞x, ☞y, ☞z]

For instance, if the current value of the logical variable x is 0 (false) and the value of y is 1 (true), then 1 is stored into z. To give another example, consider $a \wedge (b \vee 0)$. Then,

[**O**, ☞b, 0, ☞v]
[**A**, ☞a, ☞v, ☞w]

is the three-address code equivalent to $a \wedge (b \vee c)$. Observe that the second operand in the first instruction is logical value 0 rather than the symbol-table address of a variable. Temporary variable v holds the result of the first operation, which is used as the second operand in the second instruction. Temporary variable w holds the final result after the execution of these two instructions.

To generate the three-address code, we associate actions with $_{cond}G$'s rules to obtain the next three-address-code-generating attribute $_{cond}G$-based grammar

$C\{\$\$\} \rightarrow C\{\$1\} \vee C\{\$3\}$	{**CODE**(**O**, \$1, \$3, \$\$)}
$C\{\$\$\} \rightarrow C\{\$1\} \wedge C\{\$3\}$	{**CODE**(**A**, \$1, \$3, \$\$)}
$C\{\$\$\} \rightarrow (\ C\{\$2\}\ )$	{\$\$:= \$2}
$C\{\$\$\} \rightarrow i\{\$1\}$	{\$\$:= \$1}

Consider the first attribute rule

$C\{\$\$\} \rightarrow C\{\$1\} \vee C\{\$3\}$ {**CODE**(**O**, \$1, \$3, \$\$)}

In terms of the syntax analysis, a parser reduces a handle of the form $C \vee C$ to C. Suppose that during this reduction, \$1 and \$3 are equal to ☞x and ☞y, respectively, where x and y are variables. Action **CODE**(**O**, \$1, \$3, \$\$) creates a new temporary variable z in the symbol table, sets \$\$ to ☞z, and generates [**O**, ☞x, ☞y, ☞z]. Regarding $C\{\$\$\} \rightarrow C\{\$1\} \wedge C\{\$3\}$ {**CODE**(**A**, \$1, \$3, \$\$)}, the action is performed analogically except that **A** is used as an operator, not **O**. Take

$C\{\$\$\} \rightarrow (\ C\{\$2\}\ )$ {\$\$:= \$2}

By $C \rightarrow (\ C\ )$, a parser reduces a handle of the form $(\ C\ )$ to C. Suppose that during this reduction, \$2 equals ☞x, where x is a variable. At this point, the semantic action attached to this rule only sets \$\$ to ☞x. Finally, consider

$C\{\$\$\} \rightarrow i\{\$1\}$ {\$\$:= \$1}

Suppose that during this reduction, \$1 equals ☞a, where a is a source-program variable rather than a temporary variable generated by the compiler. During the reduction of i to C by $C \rightarrow i$, the attached semantic action sets \$\$ to ☞a.

Take $a \wedge (b \wedge c)$ as a logical expression, whose tokenized version equals $i\{$☞$a\} \wedge (i\{$☞$b\} \wedge i\{$☞$c\})$. Figure 6.7 describes how the syntax-directed translation converts this expression to the equivalent three-address code

[**A**, ☞b, ☞c, ☞w]
[**A**, ☞a, ☞w, ☞v]

where ☞*w* and ☞*v* are compiler-generated temporary variables. The first column describes $_{cond}G$-*op-parser*'s configurations, including actual attributes. The second column describes the shifts and reductions made by $_{cond}G$-*op-parser*. If the parsing step is reduction, the third column contains the attached action and the first column underlines the handle that is reduced.

Configuration with Actual attributes	*Parsing Action*	*Semantic Action*
▶◆*i*{☞*a*} ∧ (*i*{☞*b*} ∧ *i*{☞*c*})◀	**SHIFT**	
▶$\underline{i\{☞a\}}$◆∧ (*i*{☞*b*} ∧ *i*{☞*c*})◀	**REDUCE**($C \to i$)	$$:= ☞*a*
▶*C*{☞*a*}◆∧ (*i*{☞*b*} ∧ *i*{☞*c*})◀	**SHIFT**	
▶*C*{☞*a*} ∧◆(*i*{☞*b*} ∧ *i*{☞*c*})◀	**SHIFT**	
▶*C*{☞*a*} ∧ (◆*i*{☞*b*} ∧ *i*{☞*c*})◀	**SHIFT**	
▶*C*{☞*a*} ∧ ($\underline{i\{☞b\}}$◆∧ *i*{☞*c*})◀	**REDUCE**($C \to i$)	$$:= ☞*b*
▶*C*{☞*a*} ∧ (*C*{☞*b*}◆∧ *i*{☞*c*})◀	**SHIFT**	
▶*C*{☞*a*} ∧ (*C*{☞*b*} ∧◆ *i*{☞*c*})◀	**SHIFT**	
▶*C*{☞*a*} ∧ (*C*{☞*b*} ∧ $\underline{i\{☞c\}}$◆)◀	**REDUCE**($C \to i$)	$$:= ☞*c*
▶*C*{☞*a*} ∧ ($\underline{C\{☞b\} \wedge C\{☞c\}}$◆)◀	**REDUCE**($C \to C \wedge C$)	**CODE**(**A**, ☞*b*, ☞*c*, ☞*w*)
▶*C*{☞*a*} ∧ (*C*{☞*w*}◆)◀	**SHIFT**	
▶*C*{☞*a*} ∧ $\underline{(C\{☞w\})}$◆◀	**REDUCE**($C \to (C)$)	$$:= ☞*w*
▶$\underline{C\{☞a\} \wedge C\{☞w\}}$◆◀	**REDUCE**($C \to C \wedge C$)	**CODE**(**A**, ☞*a*, ☞*w*, ☞*v*)
▶*C*{☞*v*}◆◀	**ACCEPT**	

Figure 6.7 ***Generation of three-address code.***

The generation of three-address instructions with unary, arithmetic, and relational operators is left as an exercise.

■

Alternatively, we can generate the three-address code from a syntax tree by traversing this tree according to a postorder of its nodes (see Section 1.1). Indeed, if we go through this postorder in a left-to-right way and simultaneously generate the three-address instructions whenever visiting operator nodes, we obtain the instructions in the order in which they have to be executed. To illustrate, consider the tree **A**⟨☞*x*, **O**⟨☞*y*, ☞*z*⟩⟩ from Figure 6.4 with *postorder*(**A**⟨☞*x*, **O**⟨☞*y*, ☞*z*⟩⟩) = ☞*x* ☞*y* ☞*z* **O** **A**. When we reach **O**, we generate [**O**, ☞*y*, ☞*z*, ☞*u*], where *u* is a new temporary variable. Subsequently, visiting **A**, we generate [**A**, ☞*x*, ☞*u*, ☞*v*], where *v* is another temporary variable. At this point, we complete the three-address-code generation from **A**⟨☞*x*, **O**⟨☞*y*, ☞*z*⟩⟩. A further discussion of this straightforward method is left as an exercise.

6.1.3 Polish Notation

As already noted, in practice, an overwhelming majority of symbols occurring in grammatical rules needs no attributes at all. In fact, the generation of some intermediate codes, such as the postfix Polish notation (see Section 1.1), requires attributes attached only to terminals.

Case Study 28/35 ***Postfix Notation.*** Consider the following postfix-notation-generating attribute $_{cond}G$-based grammar

$C \to C \vee C$	{**POSTFIX**(∨)}
$C \to C \wedge C$	{**POSTFIX**(∧)}
$C \to (C)$	{}
$C \to i$ {$1}	{**POSTFIX**($1)}

The semantic actions attached to the first two rules emit the operators used in these rules. Action **POSTFIX**($1) of the forth rule emits the attribute attached to the identifier.

In the postfix notation, we list an operator behind its operands. This is the sequence followed in right parse. To illustrate, when the syntax directed translation uses $C \to C \vee C$ {**POSTFIX**(∨)}, this translation has generated the operands corresponding to the two *C*s on the right-hand side, so it is the right moment to generate ∨ behind them. By $C \to i$ {\$1}, the translation emits the attribute attached to the identifier. As a result, we obtain exactly the order required by postfix notation. As postfix notation uses no parentheses, a reduction according to $C \to (C)$ makes no semantic action. Leaving a detailed discussion of this postfix-notation-generating syntax-directed translation in general as an exercise, we now reconsider the tokenized expression *i*{☞*x*} ∧ (*i*{☞*y*} ∨ *i*{☞*z*}) from Case Study 26/35. In Figure 6.8, we describe how the syntax-directed translation based upon the above attribute grammar and guided by $_{cond}G$-*op-parser* turns this expression to the postfix notation ☞*x* ☞*y* ☞*z* **O A**. The first three columns have an analogical content to the three columns in Figure 6.7. The fourth column gives the produced postfix notation.

Configuration	*Parsing Action*	*Semantic Action*	*Postfix Notation*
▶◆*i*{☞*x*} ∧ (*i*{☞*y*} ∨ *i*{☞*z*})◀	**SHIFT**		
▶$\underline{i\{☞x\}}$◆∧(*i*{☞*y*} ∨ *i*{☞*z*})◀	**REDUCE**($C \to i$)	**POSTFIX**(☞*x*)	☞*x*
▶*C*◆∧(*i*{☞*y*} ∨ *i*{☞*z*})◀	**SHIFT**		
▶*C* ∧◆(*i*{☞*y*} ∨ *i*{☞*z*})◀	**SHIFT**		
▶*C* ∧ (◆ *i*{☞*y*} ∨ *i*{☞*z*})◀	**SHIFT**		
▶*C* ∧ ($\underline{i\{☞y\}}$◆∨ *i*{☞*z*})◀	**REDUCE**($C \to i$)	**POSTFIX**(☞*y*)	☞*x* ☞*y*
▶*C* ∧ (*C* ◆∨ *i*{☞*z*})◀	**SHIFT**		
▶*C* ∧ (*C* ∨◆*i*{☞*z*})◀	**SHIFT**		
▶*C* ∧ (*C* ∨ $\underline{i\{☞z\}}$◆)◀	**REDUCE**($C \to i$)	**POSTFIX**(☞*z*)	☞*x* ☞*y* ☞*z*
▶*C* ∧ ($\underline{C \vee C}$◆)◀	**REDUCE**($C \to C \vee C$)	**POSTFIX**(∨)	☞*x* ☞*y* ☞*z* **O**
▶*C* ∧ (*C*◆)◀	**SHIFT**		
▶*C* ∧ $\underline{(C)}$◆◀	**REDUCE**($C \to (C)$)	no action	
▶$\underline{C \wedge C}$◆◀	**REDUCE**($C \to C \wedge C$)	**POSTFIX**(∧)	☞*x* ☞*y* ☞*z* **O A**
▶*C*◆◀	**ACCEPT**		

Figure 6.8 ***Generation of Postfix Notation.***

■

We can easily generate the prefix Polish notation from a syntax tree so we delete all occurrences of 〈 and 〉 in the usual one-dimensional representation ℜ of the tree (see Section 1.1). For instance, take **A**〈☞*x*, **O**〈☞*y*, ☞*z*〉〉 from Figure 6.4. Delete all occurrences of 〈 and 〉 to obtain the prefix notation of this expression as **A** ☞*x* **O** ☞*y* ☞*z*.

6.2 Top-Down Syntax-Directed Translation

Strictly speaking, the attributes we have discussed in the previous parts of this chapter are called *synthesized attributes*. During the construction of a parse tree, these attributes are transferred from children to the parent. Apart from this transferring direction, however, we sometimes also need to transfer attributes between siblings or from a parent to a child. In this case, a child actually inherits attributes from the parent, so we refer to the attributes transferred in this way as *inherited attributes*. Apart from the opposite transferring direction, these attributes are used in the same way as the synthesized attributes. Both types of attributes fulfill an important role in the top-down syntax-directed translation, sketched in this section.

Let $G = ({}_G\Sigma, {}_GR)$ be a grammar, $M = ({}_M\Sigma, {}_MR)$ be a *G*-based top-down parser. In an informal way, we next incorporate the inherited attributes into the notation concerning the attribute rules introduced in the beginning of Section 6.1. We use §s to denote the inherited formal attributes by analogy with the use of the synthesized formal attributes denoted by \$s. In a rule

$X_0 \rightarrow X_1X_2\dots X_n \in {}_GR$, we attach a synthesized attribute $\$j$ and an inherited attribute §j behind symbol X_j as $X_j\{\$j\}[\S j]$, and we attach \$\$ and §§ behind X_0 as $X_0\{\$\$\}[\S\S]$. From $X_0 \rightarrow X_1X_2\dots X_n$, we create an attribute rule of the form

$$X_0\{\$\$\}[\S\S] \rightarrow X_1\{\$1\}[\S 1]\dots X_n\{\$n\}[\S n]\ \ \{\textbf{ACTION}(\$\$, \$1, \dots, \$n, \S\S, \S 1, \dots, \S n)\}$$

where **ACTION**(\$\$, \$1, …, \$*n*, §§, §1, …, §*n*) is the attached action in which we use both \$\$, \$1, …, \$*n* and §§, §1, …, §*n*; otherwise, the attribute rule has the same meaning as before. Like before, if some attributes are not needed, they are simply omitted. If *M* makes an expansion according to $X_0 \rightarrow X_1X_2\dots X_n$, the syntax-directed translation guided by *M* performs **ACTION**(\$\$, \$1, …, \$*n*, §§, §1, …, §*n*) with the real synthesized and inherited attributes substituted for their formal counterparts.

The inherited attributes are commonly used during the syntax-directed translation guided by a top-down parser in order to generate the intermediate code for language constructs specifying general program flow, such as loops. Apart from this use, however, the top-down syntax-directed translation often makes use of these attributes in actions for some other purposes, such as obtaining data-type information during the parsing of declaration and recording this information into the symbol table as explained in the next case study.

Case Study 29/35 ***Top-Down Syntax-Directed Translation.*** Consider the next LL grammar that declares a list of FUN **integer** or **real** variables as

⟨declaration⟩ → ⟨type⟩ ⟨variable⟩
⟨type⟩ → ***integer***
⟨type⟩ → ***real***
⟨variable⟩ → *i* ⟨list⟩
⟨list⟩ → , ⟨variable⟩
⟨list⟩ → ;

Consider a FUN declaration list of the form ***real*** *x*, *y*; and its tokenized version ***real*** *i*{☞*x*}, *i*{☞*y*};. Its parse tree is depicted in Figure 6.9, in which the dotted arrows indicate how the data-type information is passed. Observe that besides synthesized attributes, this pass requires inherited attributes because the data type information transfer also moves between siblings and from parents to children. Recall that ☞*x* denotes the symbol-table address of a variable *x* (see Convention 1.4). More specifically, by ☞*x.type*, we hereafter denote the pointer to the symbol-table item that specifies the data type of *x*, and the action ☞*x.type* := *t*, where *t* ∈ {***integer***, ***real***}, sets this item to *t*. Of course, in the following attributed grammar based on the above LL grammar, we describe this action with formal attributes rather than actual attributes. Specifically, its fourth rule's action contains \$1.*type* := §§.

1. ⟨declaration ⟩ → ⟨type⟩{\$1} ⟨variable⟩[§2]	{§2 := \$1}
2. ⟨type⟩{\$\$} → ***integer***	{\$\$:= ***integer***}
3. ⟨type⟩{\$\$} → ***real***	{\$\$:= ***real***}
4. ⟨variable⟩[§§] → *i*{\$1} ⟨list⟩[§2]	{\$1.*type* := §§; §2 := §§}
5. ⟨list⟩[§§] → , ⟨variable⟩[§2]	{§2 := §§}
6. ⟨list⟩[§§] → ;	

Recall that top-down parsing can be expressed as the construction of a leftmost derivation (see Section 3.2). Accordingly, Figure 6.10 describes the top-down syntax-directed translation of ***real*** *i*{☞*x*}, *i*{☞*y*}; based upon the leftmost derivation of ***real*** *i*, *i*; . The first column of the table given in this figure describes the sentential forms of this derivation extended by the actual attributes.

The second column gives the applied rules of the attribute grammar according to which the parser makes the expansions. The third column describes the action attached to the expansion rule. Observe that after the first step, the real synthesized attribute of ⟨type⟩ and the real inherited attributes of ⟨variable⟩ are unknown. After the second derivation step, however, the real synthesized attribute of ⟨type⟩ is determined as ***real***, so the action §2 := $1 is performed with $1 := ***real*** and, thereby, sets the real inherited attributes of ⟨variable⟩ to ***real*** as pointed up in the figure. In the third step, by using rule 4, the syntax directed translation performs ☞*x.type* := ***real***, and ⟨list⟩'s inherited attribute is again equal to ***real***. In the fifth step, by using rule 4, the syntax directed translation performs ☞*y.type* := ***real***. In the last step, the application of rule 6 completes this derivation.

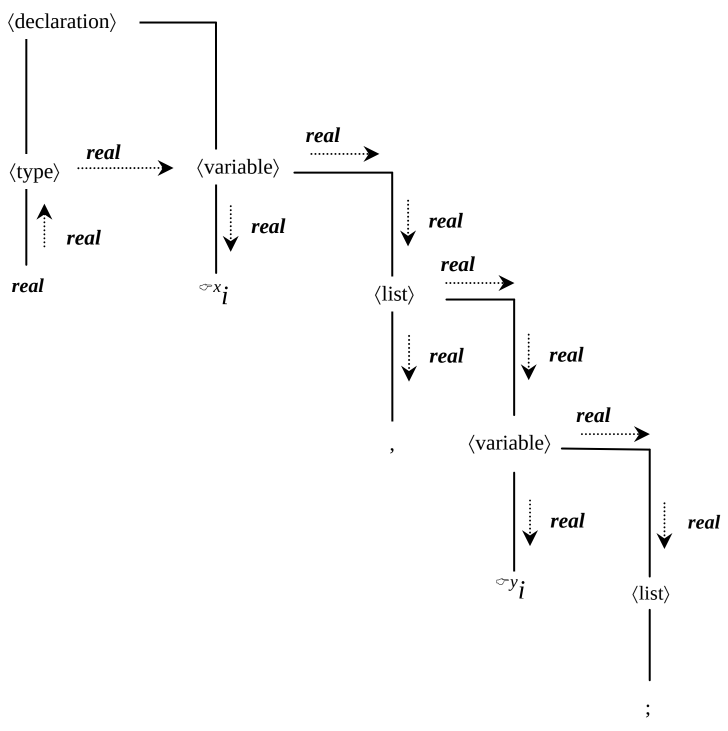


Figure 6.9 ***Data-Type Information Pass.***

Sentential Form with Actual Attributes	*Rule*	*Semantic Action*
⟨declaration ⟩		
$_{lm}\Rightarrow$ ⟨type⟩{?} ⟨variable⟩[?]	1	§2 := $1 with $1 = ?
$_{lm}\Rightarrow$ ***real***{***real***} ⟨variable⟩[***real***]	3	$$:= ***real***, so $1 = ***real*** in the above row
$_{lm}\Rightarrow$ ***real***{***real***} *i*{☞*x*} ⟨list⟩[***real***]	4	☞*x.type* := ***real***; §2 := §§ with §§ = ***real***
$_{lm}\Rightarrow$ ***real***{***real***} *i*{☞*x*}, ⟨variable⟩[***real***]	5	§2 := §§ with §§ = ***real***
$_{lm}\Rightarrow$ ***real***{***real***} *i*{☞*x*}, *i*{☞*y* } ⟨list⟩[***real***]	4	☞*y.type* := ***real***; §2 := §§ with §§ = ***real***
$_{lm}\Rightarrow$ ***real***{***real***} *i*{☞*x* }, *i*{☞*y* };	6	

Figure 6.10 ***Top-Down Syntax-Directed Translation.***

■

As illustrated in the previous case study, during the expansion according to a rule, some of the actual attributes the attached action is performed with may be unknown. In fact, an unlimited number of expansions like this may occur in a row, so their execution has to be postponed until all the involved actual attributes are determined. The syntax-directed translation usually pushes all these unfulfilled actions onto a special *semantic pushdown*, and as soon as all the actual attributed needed in the topmost action are determined, the top action is executed. In many respects, this topic is beyond the scope of this introductory text, so we omit its further discussion here. As a matter of fact, throughout the rest of this book, we only use the synthesized attributes, not the inherited attributes.

6.3 Semantic Analysis

Besides the actions that generate the intermediate code, the parser of a compiler often directs many other actions, such as recording in the symbol table discussed in the conclusion of the previous section. It also directs the actions that verify various semantic aspects of the source program. The collection of these actions is referred to as *semantic analysis*, which is discussed in the present section. This analysis usually includes actions that make the following four checks.

Flow-of-control checks. Statements that cause flow of control to leave a construct require the specification of a place to which to transfer the flow of control. For instance, a jump statement causes control to continue from a statement denoted by a label, so a semantic error occurs if the label denotes no statement. In addition, this continuation often has to occur in the same block as the jump statement, and the semantic analysis verifies that this is the case. As a rule, this verification necessitates the multi-pass design of a compiler (see Section 1.2).

Uniqueness checks. Some objects, such as identifiers or a case statement label, must be defined precisely once within the given source-program context, so the semantic analysis verifies that their definitions are unique.

Name-related checks. Some names necessarily appear two or more times in the source program. For instance, the semantic analysis has to make sure that a block name appears at the beginning and at the end of a program block.

Type-of-operand checks. Perhaps most importantly, the semantic analyzer makes *type checking* by finding out whether the operands are compatible with the given operation. If they are not, it takes an appropriate action to handle this incompatibility. More specifically, it either indicates an error or makes *type coercion*. During this coercion, it converts one of the operands to a compatible type; for instance, it turns a number in an integer format to the same number in a floating-point format.

We describe the semantic analysis rather briefly and informally because its actions are easy to carry out by using the symbol table. To illustrate, the information needed to make type checking is put into the symbol table in the way described in Case Study 29/35. The next case study explains how the syntax directed translation makes use of this information to perform type checking.

Case Study 30/35 ***Type Checking and Coercion.*** Consider the grammar ${}_{expr}G$ for arithmetic expressions (see Case Study 6/35 in Section 3.1), defined as

$$E \to E + T,\ E \to E - T,\ E \to T,\ T \to T * F,\ T \to T / F,\ T \to F,\ F \to (E),\ F \to i$$

Let us note that in this study, we use the same notation as in Case Study 29/35.

From $E \rightarrow E + T$, we make the attribute rule $E\{\$\$\} \rightarrow E\{\$1\} + T\{\$3\}$ with the following attached action described in the following informal way:

- If $1.*type* = $3.*type* = *t*, where $t \in$ {***integer***, ***real***}, generate intermediate-code that makes *t* addition of variables addressed by $1 and $3 and places the result into a variable addressed by $$, whose $$.*type* is set to *t*. That is, if *t* = ***real***, the floating-point addition is generated and $$.*type* = ***real***, and if *t* = ***integer***, the integer addition is produced and $$.*type* = ***integer***.

- If $1.*type* ≠ $3.*type* (one of the two type attributes is an integer variable while the other is a real variable), generate the intermediate-code instructions that (1) convert the integer-form variable to a real-form variable to obtain two real variables, (2) execute actual addition with these two real variables, (3) store the result into a real variable addressed by $$, and (4) set $$.*type* to ***real***.

Analogically, create the other attribute rules as an exercise.

■

6.4 Symbol Table

A symbol table assists almost every phase of the compilation process. Perhaps most importantly, during the analysis of a source program, it provides the semantic analyzer with information to check the source-program semantic correctness as explained in Sections 1.2 and 6.3. Furthermore, during the generation of a target program, it assists the code generator to produce the target code properly and economically. Therefore, the symbol-table mechanism must allow the compiler to add new entries and find existing entries in a speedy and effective way. In this section, we briefly discuss some common symbol-table organizations. Assuming that the reader is familiar with fundamental data structures and their implementation, we concentrate this discussion solely on the symbol table's properties related to compilation.

Organization

A symbol table is usually organized as one of these database structures

- array
- linked list
- binary search tree
- hash table

Array is simple and economical to store, and these properties represent its major advantage. It does not contain any pointers or other overhead, which takes memory but holds no data. Indeed, everything it contains is data needed during compilation. As a fundamental disadvantage, it requires a fixed specification of its size in advance, which implies placing a limit on the number of identifiers. It is searched sequentially, however. As a sequential search on an *n*-item array costs *n*/2 comparisons on average, the use of an array is unbearably slow in practice.

Linked list is expandable as opposed to an array. Searching is sequential; however, a proper self-organizing storage can speed up searching this list significantly. Specifically, a storage of this kind should move frequently used items up to the list head, and this clustering at the beginning of the list shortens the average search time because references to the same identifiers tend to come in bursts during compilation.

Binary search tree offers a quicker access to data items over a linked list. To list the table contents at the end of compilation, it can be displayed alphabetically by a simple in-order traversal. This data structure advantageously combines the size flexibility of a linked data structure with the speed provided by a binary search. Compared to the previous data structures, however, its storage is less economical because every table entry needs two pointers to address its direct decedents in the tree, and this space overhead increases memory consumption significantly.

Hash table uses an array of pointers addressing linear linked lists, called *chains*, whose length are unlimited, and these chains hold the items that hash to that address. Searching is performed in essentially constant time regardless of the number of entries in the table, and this crucial advantage makes this symbol-table organization very popular in practice. Specifically, block-structured tables, discussed later in this section (see Figure 6.14), are often implemented in this way.

Storing Identifier Names

The lengths of some strings stored in the symbol table vary greatly. Perhaps most significantly, strings that represent identifier names are quite different in length because in general these names may contain an unlimited number of alphanumeric characters. Even if we place a limit on their length, such as 31 characters, we have to allocate enough space in this field to accommodate the longest possible name. In practice, however, most identifier names consist of very few characters, so storing these strings directly in the table leads to a waste of memory space. To avoid this waste, we often succinctly store the identifier names out of the table by using any economically data-organized method commonly used in databases. At this point, we also provide the symbol-table management with an efficient method of obtaining this name whenever needed; for instance, obtaining the complete name can be based on recording its length and a pointer to its start in the symbol table.

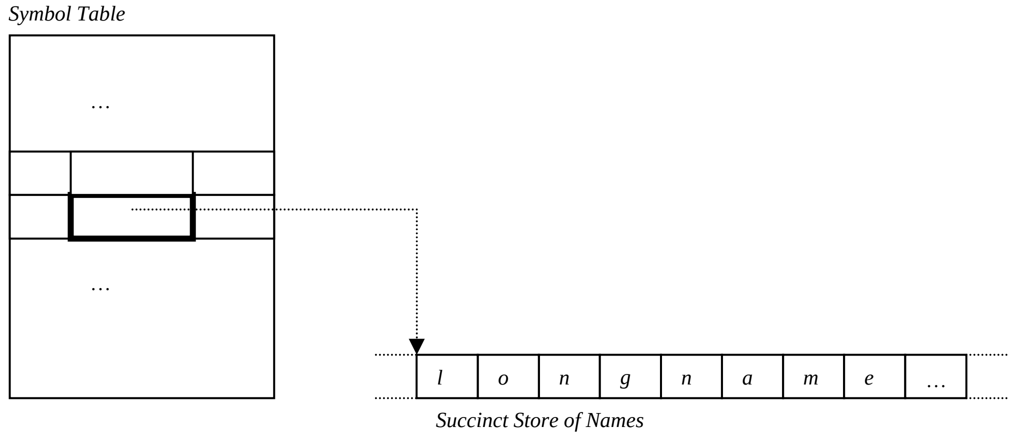


Figure 6.11 ***Storing Identifier Names*.**

Block-Structured Symbol Tables

Block-structured languages, such as C or Pascal, allow identifier *scopes* to be nested. More precisely, an identifier scope is defined as the text enclosed by a program block, such as a subprogram. These blocks can be defined within one another, thus allowing name scopes to be nested in a potentially unlimited way. An identifier is always defined in one or more nested open blocks, whereas the innermost block is the identifier's current scope.

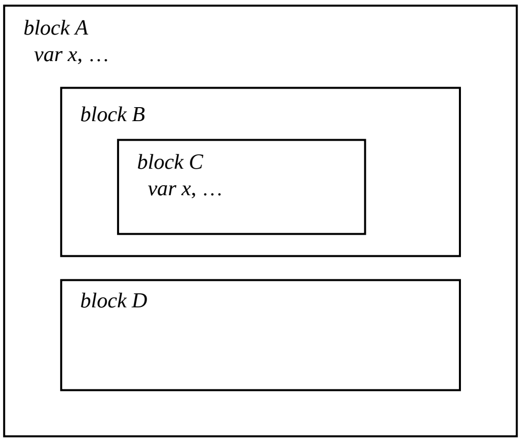


Figure 6.12 ***Block-Structured Program*.**

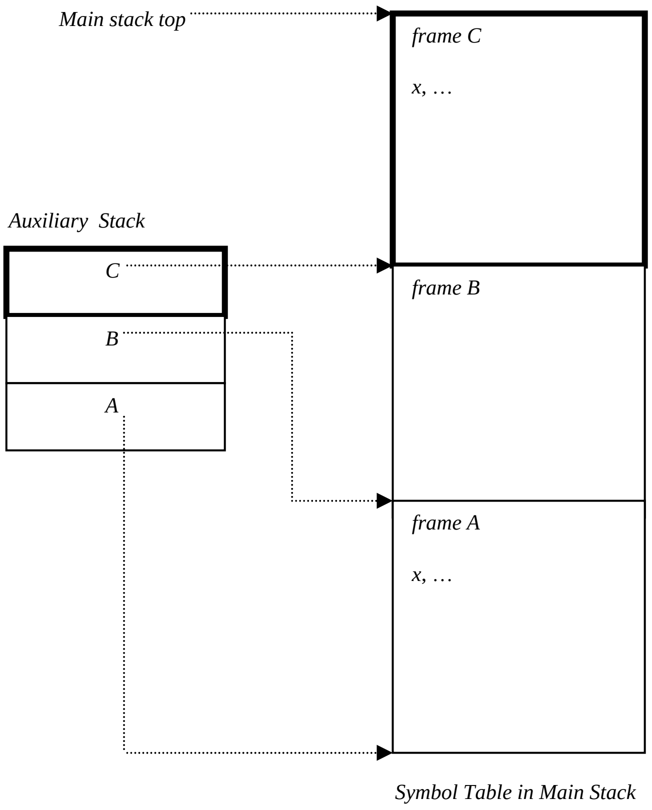


Figure 6.13 ***Multiple symbol table.***

To distinguish between various occurrences of the same identifier in a block-structured program, the symbol table has to reflect the source-program structure, including identifier scopes in nested program blocks. For this purpose, we sketch two popular block-structured symbol-table organizations:

- multiple symbol table
- global symbol table

A *multiple symbol table* has a separate table *frame* for each block, and when the compilation of a program block is finished, the complete frame corresponding to this block is removed. This strategy obviously suggests a stack organization of the table. In fact, besides the main stack that store the table frames corresponding to the nested blocks, we usually employ another stack as an auxiliary table whose entries contain only pointers to the beginnings of the main-stack table frames (see Figure 6.13). When compilation enters a new block, its frame is pushed onto stack, and the frame address is stored onto the auxiliary tabletop. When a compilation leaves a block, its frame, including all the local variables of this block, is popped by setting the main-stack top pointer to the auxiliary table top pointer, which addresses the beginning of the popped frame, and subsequently popping this top pointer off the auxiliary table.

A *global symbol table* contains each identifier marked with the scope to which it belongs. With a hash table, we preserve the scope information by entering each new item at the head of the chain, which is the most common way of inserting items into a chained table anyway (see Figure 6.14). When searching for an identifier, we proceed sequentially through the chain to discover the most recent entry that corresponds to the innermost block, which is precisely what we need to find. Compared to the multiple symbol tables, however, we remove identifiers from a global symbol table in a more lengthy way because we have to locate and remove every single identifier of the block from which the compilation leaves.

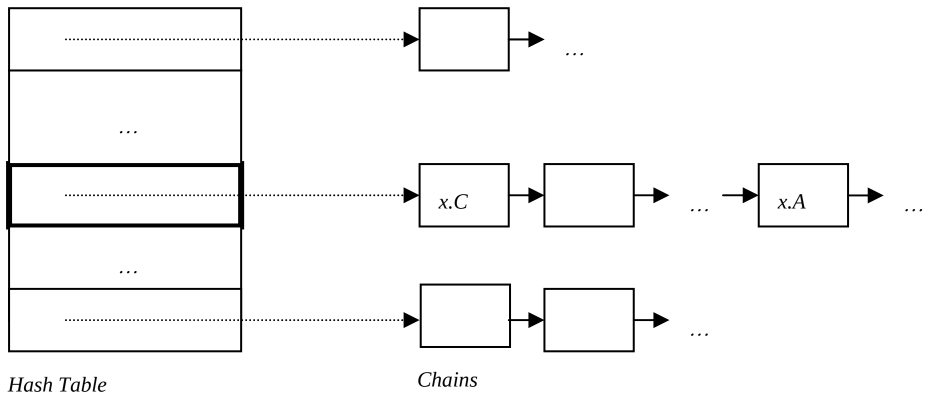


Figure 6.14 ***Global symbol table.***

In either of the above approaches, a removal of some identifiers from the symbol table does not necessarily mean its definite deletion. As a matter of fact, they are deleted only if it is guaranteed that the compilation will never return to them again; otherwise, they are saved, for instance, in an inactive region of memory or in a temporary file. Specifically, in a multi-pass compiler, variables removed from the symbol table may be needed again, so we save them. On the other hand, as soon as a one-pass compiler leaves a block, it deletes all symbol-table identifiers declared in this block because the compiler never returns to them (see Section 1.2 for the difference between one-pass and multi-pass compilers).

Most high-level programming languages allow some other data structures, which are handled by analogy with the block-structured symbol tables during compilation. To illustrate, consider Pascal record structures, which represent a problem similar to specifying identifier scopes in a symbol table because they may also contain duplicates of variables, so we have to carefully distinguish their occurrences. Therefore, we treat their variables by analogy with the treatment of identifier scopes in a symbol table. That is, we either create a subtable for each record or mark each variable with a record number. Both approaches are similar to the two block-structured table organizations above (see Figures 6.13 and 6.14), so their further discussion is left as an exercise.

6.5 Software Tools for Syntax-Directed Translation

Recall that lexical analysis translates the source program into a strings of tokens, and the syntax-directed translation guided by a parser converts this tokenized source program into some intermediate representation, such as syntax trees (see Figure 1.8). These two translation processes are so algorithmically worked out that most operating systems provide their users with tools for an automatic generation of software units that perform these two translations. In this section, we describe the fundamental UNIX tools of this kind—*lex* and *yacc* (several operating systems contain *flex* and *bison* as upwardly compatible analogies to *lex* and *yacc*, respectively).

The specification of translation processes in *lex* and *yacc* always consists of three sections separated by %%. The central section contains the translation rules that specify the translation processes. These rules consist of language constructs and corresponding actions to be taken when the constructs occur. This central section is preceded by the first that declares and defines the translated units that take part in the translation process. On the other hand, the third section contains supporting C-routines that specify the actions associated with the rules, defined in the central section. As a result, in both tools, the specification of translation processes has the following general format:

```
declarations
%%
translation rules
%%
C routines
```

The *lex* and *yacc* UNIX compilers transform this three-section specification into a C program that represents a translator constructed according to the translation rules in the central section. This C program is subsequently compiled to a binary executable form by a C compiler.

Lex

As its name indicates, *lex* is a tool for the construction of a *lex*ical analyzer. Its specification is prepared by writing a program called *lex.l* in the *lex language*. The *lex compiler* translates this program to a C program *lex.yy.c*, which is a lexical analyzer based on a tabular representation of a finite automaton constructed from regular expressions (see Section 2.3.1). Then, *lex.yy.c* is run through the C compiler to produce the desired lexical analyzer that transforms the source-program string into a sequence of tokens.

As already stated, a *lex* program consists of three sections. The first section defines components used in the central section. More specifically, it introduces some manifest constants, which fulfill the role of identifiers representing constants. Most importantly, by *lex patterns*, it describes regular expressions used by the translation rules. The *lex* patterns describe lexemes in a more succinct and elegant way than the ordinary regular expressions (see Definition 2.1) as demonstrated in Figure 6.15. For instance, $x\{m\}$, where x is a *lex* pattern and m is a positive integer, describes m concatenated occurrences of x. As a result, $a\{9\}$ is a succinct *lex* equivalent to the lengthy regular expression $aaaaaaaaa$. Furthermore, $x\{m, n\}$, where m and n are positive integers such that $n \geq m$, describes m through n occurrences of x. For instance, $a\{1, 3\}$ is a *lex* pattern equivalent to $a|aa|aaa$.

Lex pattern	*Regular expression*	*Regular language*
`a(b|c)*`	$a(b\|c)^*$	$\{a\}\{b,c\}^*$
`a+`	a^+	$\{a\}^+$
`(ab)?`	$(ab)\|\varepsilon$	$\{ab, \varepsilon\}$
`[a-h1-3]`	$a\|b\|c\|d\|e\|f\|g\|h\|1\|2\|3$	$\{a, b, c, d, e, f, g, h, 1, 2, 3\}$
`[^b]`	$a\|c\|d\|\ldots\|z$	$\{a, c, d, \ldots, z\}$
`a{9}`	$aaaaaaaaa$	$\{aaaaaaaaa\}$
`a{1, 3}`	$a\|aa\|aaa$	$\{a, aa, aaa\}$

Figure 6.15 ***Samples of lex patterns.***

The second *lex* section contains the translation rules, which have this general form

$$
\begin{array}{l}
p_1\ \{action_1\}\\
p_2\ \{action_2\}\\
\vdots\\
p_i\ \{action_i\}\\
\vdots\\
p_n\ \{action_n\}
\end{array}
$$

In each p_i $\{action_i\}$, p_i is a *lex* pattern and $action_i$ is a program fragment written in C that is performed whenever a lexeme matched by p_i is found. If there exist several *lex* pattern matches, the longest lexeme matched is selected, and if after this selection there are still two or more patterns that match the longest lexeme, the first-listed matching pattern is chosen. Most importantly, $action_i$ returns the token representing the recognized lexeme to the parser. Normally, $action_i$ is specified by using C statements; however, if it is complicated, it is usually described as the call of a function, which is specified in the third *lex* section.

The third section contains C functions called by the actions in Section 2. Typically, one of these functions installs new lexemes into the symbol table and another function tests whether the found alphabetic lexeme represents a keyword. All these functions, specified by the user, are literally copied to the *lex* output file lex.yy.c without any change.

Yacc

Yacc, which stands for *y*et *a*nother *c*ompiler-*c*ompiler, is a UNIX tool for creating a software unit that acts as a parser together with a syntax-directed translation guided by the parser. It is specified as a *yacc* program called *parser.y* in the *yacc language*. The *yacc compiler* translates this program to a C function called *yyparse*, stored in a file called *y.tab.c.*; this function is an LR parser represented by an LR table. Finally, *y.tab.c* is run through the C compiler to produce an executable program, which makes the desired translation of a string of tokens into its intermediate form.

The first *yacc* section defines ordinary C declarations delimited by %{ and %}. In addition, it declares all temporaries, tokens, precedence, and associativity used in the other two sections (see Figure 6.16).

Statement	*Definition*	*Example*
`%token` $X_1\ X_2 \ldots X_i$	tokens	`%token NUMBER ID IF`
`%left` $'o_1'\ 'o_2'\ \ldots\ 'o_j'$	left associativity	`%left '+' '-'`
`%right` $'o_1'\ 'o_2'\ \ldots\ 'o_k'$	right associativity	`%right UMINUS`
`%start` s	start symbol	`%start expr`

Figure 6.16 ***Important statements used in yacc's first section.***

Specifically, statement `%token` declares tokens, which belong to the terminals in the grammar specified in the central section. For instance, by `%token NUMBER ID IF`, we declare tokens NUMBER, ID and IF. Single characters used as terminals are put in single quotes; for example, operators + and – are written as '+' and '–', respectively. To declare that operators + and – have the left associativity, we write `%left '+' '-'`. Observe that operators are enclosed in single quotes and separated by white space. Regarding the description of operators in several lines, it holds that operators in later lines have higher precedence than operators in earlier lines; naturally, operators in the same line have the same precedence. To illustrate, consider

```
%left '+' '-'
%right UMINUS
```

In this example, operators + and – have the same precedence, which is lower than the precedence of `UMINUS`. Typically, we declare the associativity and precedence of operators if they occur in an ambiguous grammar to make their use unambiguous (see Case Study 17/35 in Section 5.1). If statement `%start` is omitted, the left-hand side of the first rule in the second section is taken as the grammatical start symbol. Noteworthy, nonterminals are not declared at all because this declaration automatically follows from their appearance on the left-hand sides of grammatical rules listed in Section 2, discussed next.

The second *yacc* section contains an attribute grammar underlying a syntax-directed translation. More precisely, *yacc* specifies all the attribute *A*-rules

$$A \rightarrow x_1 \{\mathbf{ACTION}_1\}, A \rightarrow x_2 \{\mathbf{ACTION}_2\}, \ldots, A \rightarrow x_n \{\mathbf{ACTION}_n\}$$

as

$$\begin{array}{ll} A & : x_1 \ \{\mathbf{ACTION}_1\} \\ & \mid x_2 \ \{\mathbf{ACTION}_2\} \\ & \mid \ldots \\ & \ \ \vdots \\ & \mid x_n \ \{\mathbf{ACTION}_n\} \\ & ; \end{array}$$

That is, all alternative right-hand sides of *A*-rules are separated by a vertical line. In front of them, we write the left-hand side *A*, followed by a colon, and behind them, we put a semicolon to mark the end of this list. Enclosed in curly braces, semantic actions are written in C behind the corresponding right-hand sides. In these actions, by analogy with the notation we use in this book, the attribute associated with the left-hand side is denoted by $$ and the attribute associated with *i*th symbol on the right-hand side is denoted by $i; for instance, $3 is the attribute associated with the third symbol on the right-hand side.

The third *yacc* section, written in C, always contains the main program that invokes the parser *yyparser*. In addition, it includes an error handler, *yyerror*, called whenever the parser discovers a syntax error, and the code needed to perform the semantic actions specified in the second section.

Yacc-Lex Communication. To describe the communication of a *yacc*-constructed parser with a scanner, recall that *yacc* generates a C function named *yyparse*, stored in a file called *y.tab.c*. Whenever *yyparse* needs the next token, it calls a function called *yylex* to perform an action that returns the next token, represented by a numeric code supplied by the lexical analyzer, possibly with an attribute with information about the recognized lexeme in a global variable *yylval*.

Although *yylex* may be written by the user, it is usually generated by *lex*, which significantly simplifies its construction, and included to *y.tab.c* by #include *lex.yy.c*.

Case Study 31/35 ***Lex and Yacc*.** To keep this case study compact and easy-to-follow, we omit the description of some program parts, and leave their inclusion as an exercise.

Consider FUN's identifiers, defined as alphanumeric strings started with a letter. By the next *lex* program, we recognize these lexemes, install them into a symbol table by a function install_id, and represent them by numeric code 5, which is returned to the parser.

```
%{
#define IDEN 5
%}

letter [A-Za-z]
digit [0-9]
id {letter}({letter}|{digit})*

%%
{id} {yylval = install_id (); return IDEN;}

%%
...
install_id() { /* Procedure to install the lexeme into the symbol table. The identifier's first
character is pointed to by yytext and its length is in yyleng. A pointer to the symbol table entry is
returned. */
}
```

Reconsider the syntax-tree-generating attribute ${}_{expr}G$-based grammar from Case Study 26/35, defined as

$$
\begin{array}{ll}
E\{\$\$\} \rightarrow E\{\$1\} + T\{\$3\} & \{\$\$:= \textbf{TREE}(+, \$1, \$3)\} \\
E\{\$\$\} \rightarrow E\{\$1\} - T\{\$3\} & \{\$\$:= \textbf{TREE}(-, \$1, \$3)\} \\
E\{\$\$\} \rightarrow T\{\$1\} & \{\$\$:= \$1\} \\
T\{\$\$\} \rightarrow T\{\$1\} * F\{\$3\} & \{\$\$:= \textbf{TREE}(*, \$1, \$3)\} \\
T\{\$\$\} \rightarrow T\{\$1\} / F\{\$3\} & \{\$\$:= \textbf{TREE}(/, \$1, \$3)\} \\
T\{\$\$\} \rightarrow F\{\$1\} & \{\$\$:= \$1\} \\
F\{\$\$\} \rightarrow (E\{\$2\}) & \{\$\$:= \$2\} \\
F\{\$\$\} \rightarrow i\{\$1\} & \{\$\$:= \textbf{LEAF}(\$1)\}
\end{array}
$$

Recall that its semantic actions create a syntax tree corresponding to the parsed arithmetic expression. We next code this attributed grammar in *yacc* as

```
⋮
%token IDEN
⋮

%%
expr : expr '+' term      {$$ = makenode($1, '+', $3)}
     | expr '−' term      {$$ = makenode($1, '−', $3)}
     | term        {$$ = $1}
     ;

term : term '*' fact  {$$ = makenode($1, '∗', $3)}
     | term '/' fact  {$$ = makenode($1, '/', $3)}
     | fact           {$$ = $1}
     ;

fact : '(' fact ')'  {$$ = $2}
     | IDEN          {$$ = yylval}
     ;


%%
#include "lex.yy.c"

main() { …
yyparser(); …
}

yyerror() { …
}
```

`makenode() { /*` makenode($1, *'o'*, $3) allocates a new node, labels it with *o*, connects the two sons pointed by $1 and $3 to this node, and returns the pointer to this new node. `*/`
… }
⋮

Let us next reconsider the $_{cond}G$-based ambiguous attributed grammar from Cases Study 27/35, defined as

$$
\begin{array}{ll}
C\{\$\$\} \to C\{\$1\} \vee C\{\$3\} & \{\mathbf{CODE}(\mathbf{O}, \$1, \$3, \$\$)\} \\
C\{\$\$\} \to C\{\$1\} \wedge C\{\$3\} & \{\mathbf{CODE}(\mathbf{A}, \$1, \$3, \$\$)\} \\
C\{\$\$\} \to (\ C\{\$2\}\ ) & \{\$\$:= \$2\} \\
C\{\$\$\} \to i\{\$1\} & \{\$\$:= \$1\}
\end{array}
$$

whose semantic actions create the three-address code for the logical expression. The ambiguity of this grammar necessitates the use of *yacc* statements that define the precedence of operators.

```
⋮
%token IDEN
%left 'O'
%left 'A'
⋮

%%
cond : cond '+' cond  {$$ = makenode($1, 'O', $3)}
     | cond '*' cond  {$$ = makenode($1, 'A', $3)}
     | '(' cond ')'   {$$ = $2}
     | IDEN           {$$ = yylval}
     ;
%%
```

/* Analogical section to the third section of the previous *yacc* program */

■

Exercises

6.1. Consider the tokenized logical expression $\neg(\neg i\{☞x\} \vee i\{☞y\}) \wedge (\neg i\{☞y\} \wedge \neg i\{☞z\})$ in FUN. Translate this expression into a syntax tree, three-address code, and postfix notation.

6.2. Construct the parse tree and the syntax tree for the following FUN ***if*** statement (see Case Study 5/35 in Section 3.1):

if *a* + *b* **gr** 1 ***then if*** *b* ***le*** 9 ***then*** *b* = *a* ***else*** *b* = *a* + *b*

6.3. Consider the general grammatical definition of FUN ***if*** statements (see Case Study 5/35 in Section 3.1). Design their representation in Polish postfix notation.

6.4. Consider the definition of Pascal **case** statements in general. Design its three-address-code intermediate representation.

6.5. Design a three-address code for procedure calls whose parameters can be constants, variable names, or arithmetic expressions.

6.6. Consider the following codes. Translate them to a syntax tree, three-address code, and postfix notation.

(a) in C:

```
main()
{
   int i;
   int j;
   i = 1;
   while (i <= 9) {
      j = i;
   }
}
```

(b) in Smalltalk:

```
result := a < 0
   ifTrue:[ 'negative integer' ]
   ifFalse:[ 'non-negative integer' ]
```

6.7. Design a syntax-directed translation that generates the prefix notation (see Section 1.1).

6.8*Solved*. The *directed acyclic graph* corresponding to a syntax tree is obtained so the syntax tree's nodes for repeated subexpressions are merged into a single node entered by an edge from all the direct ancestors of the original nodes. Thus, the in-degree of the new merged node may be greater than one. For instance, from the syntax tree **A**⟨**O**⟨☞*x*, ☞*y*⟩, **O**⟨☞*x*, ☞*y*⟩⟩, we obtain the corresponding directed acyclic graph by making node **A** from which two edges enter the root of **O**⟨☞*x*, ☞*y*⟩. Notice that **O**⟨☞*x*, ☞*y*⟩ occurs in the resulting directed acyclic graph only once, not twice.

Modify the generation of syntax trees (see Section 6.1.1) to generate the directed acyclic graphs. What is the fundamental advantage of generating the optimized target code from the directed acyclic graphs when compared to this generation from ordinary syntax trees?

6.9. Convert the syntax trees constructed in Exercise 6.6 to the corresponding directed acyclic graph.

6.10. Construct the directed acyclic graph corresponding to this three-address code

O	*X*	*Y*	*Z*
mul	☞*b*	☞*b*	☞*a*
mul	☞*a*	☞*a*	☞*d*
mul	☞*b*	☞*b*	☞*c*
add	☞*a*	☞*d*	☞*a*
mul	☞*a*	☞*d*	☞*b*

6.11*Solved*. Extend Case Study 27/35 so it generates the three-address code for the logical *not* operator, too. Describe the translation of $\neg a$ to the corresponding three-address code in detail.

6.12*Solved*. Consider the following FUN grammatical rule and its interpretation (see Case Study 5/35 in Section 3.1). What semantic-analysis action should be attached to this rule?

Rule: ⟨statement⟩ → ***for*** *i* = ⟨expression⟩ ***through*** ⟨expression⟩ ***iterate*** ⟨statement⟩
Example: ***for*** *q* = 1 ***to*** *o**2 ***do*** *j* = *q* * *j*
Interpretation: The type of variable *i* is declared as ***integer***. Let *k* and *h* be the integer values of the expressions preceding and following ***through***, respectively. The statement after ***iterate*** is repeated for $i = k, k + 1, \ldots, h$ provided that $h \geq k$; if $k > h$, the statement is repeated zero times.

6.13. Write a program to implement the syntax-directed translation that generates

(a) syntax-trees described in Case Study 26/35 *Syntax Trees*;
(b) three-address code described in Case Study 27/35 *Three-Address Code*;
(c) postfix notation described in Case Study 28/35 *Postfix Notation*.

6.14. Write a program to implement the top-down syntax-directed translation described in Case Study 29/35 *Top-Down Syntax-Directed Translation*.

6.15. Design semantic-analysis routines that make flow-of-control, uniqueness, and name-related checks sketched in Section 6.3.

6.16. Write a program to implement the type checking and coercion described in Case Study 30/35 *Type Checking and Coercion*.

6.17. Compare all the symbol-table organizations described in Section 6.4. Discuss their advantages and disadvantages. Write a program to implement the block-structured symbol tables described in Section 6.4.

6.18. Apart from the block-structured symbol table organization given in Section 6.4, design some alternative organizations of these tables and their implementation.

6.19. Consider the following grammar. By incorporating appropriate attributes and semantic actions, extend this grammar to an attribute grammar that specifies the Pascal-like scope rules for identifiers. In addition, design a suitable block-structured symbol table to store the identifiers together with the information about their scopes. Carefully describe the cooperation between the syntax-directed translation based upon this attribute grammar and the symbol table handler to store the identifiers and their scope information properly.

⟨block⟩ → **start** ⟨declaration list⟩ ⟨execution part⟩ **end**
⟨declaration list ⟩ → ⟨declaration list⟩ ; ⟨declaration⟩
⟨declaration list⟩ → ⟨declaration⟩
⟨declaration⟩ → **integer** ⟨variable list⟩

⟨declaration⟩ → **real** ⟨variable list⟩
⟨variable list⟩ → *i*, ⟨variable list⟩
⟨variable list⟩ → *i*
⟨execution part⟩ → **compute** ⟨statement list⟩
⟨statement list⟩ → ⟨statement list⟩ ; ⟨statement⟩
⟨statement list⟩ → ⟨statement⟩
⟨statement⟩ → *s*
⟨statement⟩ → ⟨block⟩

6.20. Complete the lex and yacc program sketched in Case Study 31/35 *Lex and Yacc*.

6.21. Handling declarations, a compiler isolates each declared identifier and enters its name in the symbol table; in addition, it fills in the corresponding symbol table entry with more information, such as the identifier type. There exist several reasonable ways by which a compiler can perform this task. To illustrate, storing the declared identifiers may be performed at a lexical-analysis level (see Section 2.2) or a syntax-analysis level, and either approach has their advantages and disadvantages. Design some reasonable strategies how to handle declarations during compilation and discuss their pros and cons.

6.22. Design a proper data structure by which a compiler organizes source program arrays in an effective way. Specifically, discuss how to organize arrays that may contain other subarrays as their elements.

6.23. Discuss how a compiler handles procedure calls and returns. Specifically, explain how to guarantee that

(a) a procedure has been defined and used properly regarding its type;
(b) the number and type of arguments agree with the formal parameter specification.

Furthermore, explain how to load the arguments and the return address and how to transfer them to the procedure body.

Solutions to the Selected Exercises

6.8. A directed acyclic graph is constructed in much the same way as a syntax tree, but before constructing a node for anything, the procedure first sees whether such a node is already in existence. If it is, that node is reused.

As a directed acyclic graph avoids the redundant subexpressions, it does not duplicate any code when the target code is produced, so a partially optimized target code is directly produced from this graph.

6.11. Let **N** have the same meaning as in Figure 6.1—that is, it denotes the logical *not* operator. Consider $\neg a$, where a is a variable. We execute $\neg a$ by this unary-operator three-address instruction

[**N**, ☞a, , ☞x]

where x is a temporary variable. Extend grammar ${}_{cond}G$ by $C \rightarrow \neg C$; as a result, we obtain this grammar

$C \rightarrow \neg C$
$C \rightarrow C \vee C$

$C \rightarrow C \wedge C$
$C \rightarrow (C)$
$C \rightarrow i$

Change $C \rightarrow \neg C$ to the attribute rule

$$C\{\$\$\} \rightarrow \neg C\{\$2\} \quad \{\textbf{CODE}(\textbf{N}, \$2, \ , \$\$)\}$$

where semantic action **CODE**(**N**, $2, , $$) generates [**N**, $2, , $$] and sets the attribute associated with the left-hand side to $2. The other four rules are changed to the attribute rules like in Case Study 27/35. As a result, we obtain this attribute grammar

$C\{\$\$\} \rightarrow \neg C\{\$2\}$	$\{\textbf{CODE}(\textbf{N}, \$2, \ , \$\$)\}$
$C\{\$\$\} \rightarrow C\{\$1\} \vee C\{\$3\}$	$\{\textbf{CODE}(\textbf{O}, \$1, \$3, \$\$)\}$
$C\{\$\$\} \rightarrow C\{\$1\} \wedge C\{\$3\}$	$\{\textbf{CODE}(\textbf{A}, \$1, \$3, \$\$)\}$
$C\{\$\$\} \rightarrow (C\{\$2\})$	$\{\$\$:= \$2\}$
$C\{\$\$\} \rightarrow i\{\$1\}$	$\{\$\$:= \$1\}$

Consider the logical expression $\neg a$, whose tokenized version is $\neg i\{$☞$a\}$. In Figure 6.17, we describe how the syntax-directed translation translates $\neg i\{$☞$a\}$ to [**N**, ☞a, , ☞x].

Configuration	*Parsing Action*	Semantic Action
▶◆$\neg i\{$☞$a\}$◀	**SHIFT**	
▶$\neg$◆$i\{$☞$a\}$◀	**SHIFT**	
▶$\neg$ $\underline{i\{☞a\}}$◀	**REDUCE**($C \rightarrow i$)	$$:= ☞a
▶$\underline{\neg C\{☞a\}}$◆◀	**REDUCE**($C \rightarrow \neg C$)	**CODE**(**N**, ☞a, , ☞x)
▶$C\{$☞$a\}$◆◀	**ACCEPT**	

Figure 6.17 ***Generation of Unary-Operator Three Address Code.***

6.12. A semantic-analysis routine verifying that the values of the two expressions surrounding ***through*** are always integers should be attached to this rule.

CHAPTER 7

Optimization and Target Code Generation

As a rule, the syntax-directed translation produces three-address code that can be made more efficient so it runs faster or takes less space. Therefore, the optimization, which follows the syntax-directed translation (see Figure 1.8), improves this code by various transformations, such as the replacement or removal of some instructions. In essence, any change of the code is permitted provided that the resulting optimized code does the same job as the code originally produced by the syntax-directed translation. To guarantee that the meaning of the program remains unchanged, the optimization first breaks the program into basic blocks that represent sequences of three-address instructions that are always executed sequentially from their beginning to the end. Then, it analyzes the use of variables within these blocks and between them. Finally, based on this analysis, the optimization selects and carries out an appropriate optimization transformation without changing the meaning of the program.

The target code generator of a compiler concludes the translation of a source program by generating the target program from the optimized tree-address code. As a matter of fact, during this generation, some optimization also takes place because we naturally want to produce the target code so it works as efficiently and economically as possible. For instance, this optimization tries to make the register allocation so it results into a reduction of the number of moves between the registers and the memory. As obvious, optimization of this kind always heavily depends on the target machine, so it is usually called the *machine-dependent optimization* to clearly distinguish it from the previous *machine-independent optimization*, which transforms the three-address code quite independently of the target machine.

Synopsis. Section 7.1 introduces the basic blocks of three-address code and explains how to track the use of a variable between these blocks and within them. Section 7.2 discusses the machine-independent optimization of the three-address code resulting from syntax-directed translation. It distinguishes between *local optimization* that improves the three-address code within a single block and *global optimization* that spans several basic blocks; at the same time, this section demonstrates that both approaches to optimization overlap and support each other. Section 7.3 describes how to translate the three-address code to the target code so this translation includes the machine-dependent optimization of the resulting target code.

Optimization represents a difficult topic of compiler writing, so we only sketch its fundamental principles in this introductory text. Still, as demonstrated shortly, even this basic sketch belongs to the most complicated passages of this book. Therefore, it is highly advisable to pay a special attention to all the notions introduced in Section 7.1 (see Exercise 7.1) in order to grasp the rest of the chapter.

7.1 Tracking the Use of Variables

On the one hand, we naturally want to optimize larger program units than individual statements. On the other hand, we can hardly optimize a program as a whole. Therefore, we first break the program into several basic blocks that are always executed strictly sequentially without interruptions by any branches into or out of them, and this simple way of execution makes them easy to analyze. Then, we track how the computed values of variables are changed between the blocks and within every single block to optimize the program globally and locally, respectively.

Basic blocks

Suppose that the syntax-directed translation produces a sequence of *n* three-address instructions

$$i_1, i_2, \ldots, i_{n-1}, i_n,$$

for a positive integer *n*, so that this sequence is always executed sequentially from the beginning to the end while any longer instruction sequence that contains $i_1, i_2, \ldots, i_{n-1}, i_n$ as its part is not necessarily executed in this way. At this point, $i_1, i_2, \ldots, i_{n-1}, i_n$ represents a *basic block*, *B*, of the three-address code. As a result, i_1 through i_{n-1} contain no branches in or out; only i_n may represent a branch. Instruction i_1 is called *B*'s *leader*, and it is represented by any of these three types of instructions

• start of a program or a procedure
• instruction that is the target of a branch
• instruction immediately after a branch

To divide a program, *p*, into its basic blocks, find all leaders occurring in *p*, then divide *p* into basic blocks so that each block begins at a leader and ends before the next leader or at the end of *p*. Let Π denote the set of all basic blocks of *p*. We capture the flow of control between the basic blocks by *flow relation*, ϕ, over Π so $(A, B) \in \phi$ if and only if the last instruction of *A* may result into a jump at the leader of *B*.

Convention 7.1. Throughout this chapter, a three-address instruction's operator, first operand, second operand, and result are symbolically denoted by ***O***, ***X***, ***Y***, and ***Z***, respectively.

■

Case Study 32/35 *Basic Blocks*. Let the set of three-address instructions used by the FUN compiler contain the instructions with operators ***add***, ***get***, ***jgr***, ***jum***, ***mul***, ***mov***, and ***put*** as described in Figure 7.1. All the values the instructions work with are integer values; an inclusion of real-value instructions is left as an exercise (see Exercise 7.5). In Figure 7.1, all the operands are variables; instead of them, however, integer numbers may be used as the operands as well.

O	***X***	***Y***	***Z***
add	☞*x*	☞*y*	☞*z*
get			☞*z*
jgr	☞*x*	☞*i*	*J*
jum			*J*
mul	☞*x*	☞*y*	☞*z*
mov	☞*x*		☞*z*
put			☞*z*

Figure 7.1 *Three-address instructions used by the* FUN *compiler*.

The interpretation of the seven instructions follows next:

• the ***add*** instruction sets the value of *z* to the addition of values stored in *x* and *y*;
• the ***get*** instruction moves the input value into variable *z*;
• the ***jgr*** instruction transfers the control flow to the *J*th instruction if the integer value of variable *x* is greater than integer *i*;
• the ***jum*** instruction transfers the control flow to the *J*th instruction;
• the ***mul*** instruction sets the value of *z* to the multiplication of values stored in *x* and *y*;

• the ***mov*** instruction moves the value of *x* into *z*;
• the ***put*** instruction prints the value of *z*.

Consider the following simplified FUN program fragment that reads *m*, computes its factorial *f*, and prints the result.

```
read(m);
f = 1;
for i = 2 through m iterate
    f = i * f;
write(f);
```

The three-address code of this FUN program is given in Figure 7.2. In this code, the leaders are instructions **1**, **4**, **5**, and **8**. Indeed, instruction **1** starts the code. Instruction **4** represents the target of instruction **7**. Instruction **5** follows a branch. Instruction **8** is a target of instruction **4**. Break the code into four basic blocks *A*, *B*, *C*, and *D* at the leader. Block *A* consists of instructions **1** through **3** of the original code. *B* consists of the fourth instruction. *C* consists of instructions **5** through **7**. *D* consists of the last instruction **8**. Figure 7.3 describes these blocks and denotes their instructions by $\langle U, i\rangle$, meaning the *i*th instruction within block $U \in \{A, B, C, D\}$. In this way, the figure also specifies the targets of branch instructions in the last column. Specifically, in the last column, we have changed 8 and 4 to $\langle D, 1\rangle$ and $\langle B, 1\rangle$, respectively. We can elegantly describe the flow of control between *A*, *B*, *C*, and *D* by its flow relation ϕ defined as $\phi = \{(A, B), (B, C), (C, B), (B, D)\}$.

#	*O*	*X*	*Y*	*Z*
1	***get***			☞*m*
2	***mov***	1		☞*f*
3	***mov***	2		☞*i*
4	***jgr***	☞*i*	☞*m*	**8**
5	***mul***	☞*i*	☞*f*	☞*f*
6	***add***	☞*i*	1	☞*i*
7	***jum***			**4**
8	***put***			☞*f*

Figure 7.2 ***Three-address* FUN *program.***

Block	*O*	*X*	*Y*	*Z*
$\langle A, 1\rangle$	***get***			☞*m*
$\langle A, 2\rangle$	***mov***	1		☞*f*
$\langle A, 3\rangle$	***mov***	2		☞*i*
$\langle B, 1\rangle$	***jgr***	☞*i*	☞*m*	$\langle D, 1\rangle$
$\langle C, 1\rangle$	***mul***	☞*i*	☞*f*	☞*f*
$\langle C, 2\rangle$	***add***	☞*i*	1	☞*i*
$\langle C, 3\rangle$	***jum***			$\langle B, 1\rangle$
$\langle D, 1\rangle$	***put***			☞*f*

Figure 7.3 ***Basic blocks of the three-address code.***

■

Use of Variables within a Block

Next, we track how the computed values of each variable are changed within a single block, $B \in \Pi$. To do that, we have to carefully distinguish between each variable *x* in *B* and its references by pointers to *x* occurring as components of *B*'s three-address instructions, and that is why we introduce the following notions. Let ${}_{var}B$ denote all variables used in B. Let $|B|$ denote the number of instructions in B, so if B consists of k instructions, for some $k \geq 1$, then $|B| = k$. By $\langle B, i\rangle$, we denote the ith instruction in B, where $i \in \{1, \ldots, |B|\}$. By $\langle B, i, 0\rangle$, $\langle B, i, 1\rangle$, $\langle B, i, 2\rangle$, and $\langle B, i, 3\rangle$, we denote $\langle B, i\rangle$'s operator, first operand, second operand, and result, respectively. Set ${}_B\vartheta = \{\langle B, i, k\rangle |\ 1 \leq i \leq |B|, 1 \leq k \leq 3\}$. In addition, we define the function ${}_Bvar$ from ${}_B\vartheta$ to ${}_{var}B \cup \{0\}$ so that if $\langle B, i, k\rangle$ is the pointer to a variable, x, then ${}_Bvar(\langle B, i, k\rangle) = x$; otherwise, ${}_Bvar(\langle B, i, k\rangle) = 0$. For brevity, we write $var\langle B, i, k\rangle$ instead of ${}_Bvar(\langle B, i, k\rangle)$ whenever there is no danger of confusion. Set ${}_B\varsigma = \{\langle B, i, k\rangle |\ var\langle B, i, k\rangle \in {}_{var}B, 1 \leq i \leq |B|, 1 \leq k \leq 3\}$, so ${}_B\varsigma \subseteq {}_B\vartheta$. For instance, if $\langle B, i\rangle$ equals [***add***, ☞*x*, ☞*y*, ☞*x*], $\langle B, i, 0\rangle$, $\langle B, i, 1\rangle$, $\langle B, i, 2\rangle$, and $\langle B, i, 3\rangle$ denote ***add***, ☞*x*, ☞*y*, and ☞*x*, respectively, $var\langle B, i, 1\rangle = var\langle B, i, 3\rangle = x$, and $var\langle B, i, 2\rangle = y$. To give another example, take [***mov***, 1, , ☞*f*] as $\langle B, i\rangle$. In this instruction, $\langle B, i, 0\rangle$, $\langle B, i, 1\rangle$, and $\langle B, i, 3\rangle$ denote ***mov***, 1, and ☞*f*, respectively. Notice that this instruction has no second operand, so $\langle B, i, 2\rangle$ refers to this unused component of the instruction. Furthermore, $var\langle B, i, 3\rangle = f$, so $\langle B, i, 3\rangle \in {}_B\varsigma$, but $\langle B, i, 1\rangle$ and $\langle B, i, 2\rangle$ are in ${}_B\vartheta - {}_B\varsigma$ because neither of them denotes the pointer to a variable.

Formally, we express how the computed values of the variables are changed within B by the *next-use function* ↴ from ${}_B\varsigma$ to $\{0, 1, \ldots, |B|\}$ defined for every $\langle B, i, k\rangle \in {}_B\varsigma$, where $1 \leq i \leq |B|$ and $1 \leq k \leq 3$, as

- $\langle B, i, k\rangle$↴j where j is the smallest integer in $\{i + 1, \ldots, |B|\}$ such that $var\langle B, i, k\rangle \in \{var\langle B, j, k\rangle |\ 1 \leq k \leq 2\}$ and $var\langle B, i, k\rangle \notin \{var\langle B, h, 3\rangle |\ i \leq h \leq j - 1\}$, and
- $\langle B, i, k\rangle$↴0 otherwise.

(According to Convention 1.2, for a function f, we often write xfy instead of $f(x) = y$ in this book; specifically, in this chapter, we always write $\langle B, i, k\rangle$↴j instead of ↴$(\langle B, i, k\rangle) = j$.) Informally, if $var\langle B, i, k\rangle = u$, $\langle B, i, k\rangle$↴j with j in $\{i + 1, \ldots, |B|\}$ means that j is the next instruction that uses the current contents of u in B, and if this instruction does not exist, $\langle B, i, k\rangle$↴0.

We could compute ↴ so we proceed forward through B searching for the next use of each variable. This forward approach is simple; however, it leads to a repeated scan of B as we go through it. Therefore, more efficiently, we compute ↴ for B by the next algorithm that works backward throughout B, starting from $\langle B, |B|\rangle$ and proceeding towards $\langle B, 1\rangle$. To compute ↴ in this way, we introduce a *next-use column* in the symbol table, and for each variable x, we refer to this column's item by *x.next-use*.

Goal. The definition of ↴ for ${}_B\varsigma$.

Gist. As already pointed out, the backward algorithm proceeds from $\langle B, |B|\rangle$ towards $\langle B, 1\rangle$. Suppose that the algorithm currently defines ↴ for $\langle B, i, 1\rangle$, $\langle B, i, 2\rangle$, and $\langle B, i, 3\rangle$, for some $i \in \{1, \ldots, |B|\}$. At this point, it makes this definition by information currently contained in the symbol-table next-use column previously obtained from the instructions $\langle B, |B|\rangle$ through $\langle B, i + 1\rangle$. After this, the algorithm appropriately changes this symbol-table next-use column so it can subsequently define ↴ for the components of $\langle B, i - 1\rangle$. In this way, it works until it reaches $\langle B, 1\rangle$. More precisely, let the ith instruction have the form

[***O***, ☞*u*, ☞*v*, ☞*w*]

The algorithm obtains ↴ for $\langle B, i, k\rangle$, $1 \leq k \leq 3$, so it defines $\langle B, i, 1\rangle$↴*u.next-use*, $\langle B, i, 2\rangle$↴*v.next-use*, and $\langle B, i, 3\rangle$↴*w.next-use*. Then, it sets *w.next-use* to 0 because the ith instruction modifies the current value of w; therefore, $\langle B, i, 3\rangle$↴0 properly reflects the situation earlier in B. Finally, the algorithm sets both *u.next-use* and *v.next-use* to i because this instruction uses these variables and, therefore, this use is reported to the earlier part of B.

Algorithm 7.2 ***Next-Use Function.***

Input • *a basic block, B*;
• the symbol-table next-use column.

Output • ↴ from $\{\langle B, i, k\rangle \mid 1 \le i \le |B|, 1 \le k \le 3\}$ to $\{0, 1, \dots, |B|\}$.

Method

```
begin
    initialize each item of the symbol-table next-use column with 0;

    for $j = |B|$ downto 1 do
    begin
        if $\langle B, j, 3\rangle \in {}_B\varsigma$ then
        begin
            define $\langle B, j, 3\rangle$↴$var\langle B, j, 3\rangle.next\text{-}use$;
            $var\langle B, j, 3\rangle.next\text{-}use := 0$
        end
        for $k = 1$ to 2 do
            if $\langle B, j, k\rangle \in {}_B\varsigma$ then
            begin
                define $\langle B, j, k\rangle$↴$var\langle B, j, k\rangle.next\text{-}use$;
                $var\langle B, j, k\rangle.next\text{-}use := j$
            end
    end
end.
```

Convention 7.3. For simplicity, we usually specify ↴ directly into *B* so that if ⟨*B*, *i*, *k*⟩↴*j*, for some $j \in \{i + 1, \dots, |B|\} \cup \{0\}$, and ⟨*B*, *i*, *k*⟩ equals ☞*u*, we directly write ☞*u*↴*j* instead of ⟨*B*, *i*, *k*⟩↴*j*. For instance, suppose that ⟨*B*, 1⟩ = [***sub***, ☞*a*, ☞*b*, ☞*c*] with ⟨*B*, 1, 1⟩↴0, ⟨*B*, 1, 2⟩↴2, and ⟨*B*, 1, 3⟩↴3; then, we succinctly write ⟨*B*, *i*⟩ = [***sub***, ☞*a*↴0, ☞*b*↴2, ☞*c*↴3].
■

Case Study 33/35 ***Next-Use Function.*** We apply Algorithm 7.2 to the basic block and the symbol-table next-use column given in Figure 7.4. Notice that all the next-use column's items are set to 0 initially. The contents of the table and the column resulting from the first, second, third, and fourth iteration of the main loop of the algorithm are given in Figures 7.5, 7.6, 7.7, and 7.8, respectively. In the next-use column, we always underline the items that the iteration changes.

Instruction	*O*	*X*	*Y*	*Z*
1	***sub***	☞*a*	☞*b*	☞*c*
2	***mul***	☞*b*	☞*b*	☞*a*
3	***mul***	☞*c*	☞*a*	☞*c*
4	***add***	☞*a*	☞*c*	☞*d*

Variable	*Next-Use Column*
a	0
b	0
c	0
d	0

Figure 7.4 ***Basic Block*** **and** ***Next-Use Column: Initial Contents.***

Instruction	*O*	*X*	*Y*	*Z*
1	***sub***	☞*a*	☞*b*	☞*c*
2	***mul***	☞*b*	☞*b*	☞*a*
3	***mul***	☞*c*	☞*a*	☞*c*
4	***add***	☞*a*↓0	☞*c*↓0	☞*d*↓0

Variable	*Next-Use Column*
a	$\underline{4}$
b	0
c	$\underline{4}$
d	$\underline{0}$

Figure 7.5 ***Basic Block and Next-Use Symbol-Table Column after Iteration*** **1.**

Instruction	*O*	*X*	*Y*	*Z*
1	***sub***	☞*a*	☞*b*	☞*c*
2	***mul***	☞*b*	☞*b*	☞*a*
3	***mul***	☞*c*↓0	☞*a*↓4	☞*c*↓4
4	***add***	☞*a*↓0	☞*c*↓0	☞*d*↓0

Variable	*Next-Use Column*
a	$\underline{3}$
b	0
c	$\underline{3}$
d	0

Figure 7.6 ***Basic Block and Next-Use Symbol-Table Column after Iteration*** **2.**

Instruction	*O*	*X*	*Y*	*Z*
1	***sub***	☞*a*	☞*b*	☞*c*
2	***mul***	☞*b*↓0	☞*b*↓0	☞*a*↓3
3	***mul***	☞*c*↓0	☞*a*↓4	☞*c*↓4
4	***add***	☞*a*↓0	☞*c*↓0	☞*d*↓0

Variable	*Next-Use Column*
a	$\underline{0}$
b	$\underline{2}$
c	3
d	0

Figure 7.7 ***Basic Block and Next-Use Column after Iteration*** **3.**

Instruction	*O*	*X*	*Y*	*Z*
1	***sub***	☞*a*↓0	☞*b*↓2	☞*c*↓3
2	***mul***	☞*b*↓0	☞*b*↓0	☞*a*↓3
3	***mul***	☞*c*↓0	☞*a*↓4	☞*c*↓4
4	***add***	☞*a*↓0	☞*c*↓0	☞*d*↓0

Variable	*Next-Use Column*
a	$\underline{1}$
b	$\underline{1}$
c	$\underline{0}$
d	0

Figure 7.8 ***Basic Block and Next-Use Column after Iteration*** **4:** ***Final Contents.***

Take, for instance, *a* in Figure 7.8. This variable is used as the first operand in $\langle B, 1\rangle$, but the second instruction rewrites its current value. Therefore, we have $\langle B, 1, 1\rangle$↴0, meaning that the current value of *a* used as the first operand in $\langle B, 1\rangle$ has no next use in the rest of *B*. To give another example, consider *c* in $\langle B, 3\rangle$, in which *c* represents both the first operand and the result (see Figure 7.8). According to the algorithm, in the symbol table, we first set *c.next-use* to 0 because the result is *c*, so it is rewritten by this instruction; then, however, we reset *c.next-use* to 3 as desired because the first operand is *c*, too.

■

Use of Variables between Blocks

Next, we track the movement of information between blocks. Specifically, we want to find out how the interconnections of the blocks affect the computed values of variables to determine where they are needed and where they are not. This determination requires introducing some more notions. If the performance of $\langle B, i\rangle$, where $B \in \Pi$, assigns a value to its result, $\langle B, i, 3\rangle$, we say that $\langle B, i, 3\rangle$ represents a *definition of v*, symbolically denoted by ${}_{B}def(\langle B, i\rangle)$. For brevity, we write $def\langle B, i\rangle$ instead of ${}_{B}def(\langle B, i\rangle)$ whenever there is no danger of confusion. By ${}_{B}var(def\langle B, i\rangle)$, we denote the variable whose occurrence is denoted by $def\langle B, i\rangle$. For instance, assume that $\langle B, i\rangle$ is a three-address instruction of the form

[***add***, ☞x, ☞y, ☞x]

that computes the arithmetic addition of the current values of *x* and *y* and assigns the resulting value to *x*. At this point, $def\langle B, i\rangle$ denotes the occurrence of *x* that represents the result of this instruction, not the first operand. As obvious, ${}_{B}var(def\langle B, i\rangle) = x$ in this example. As obvious, in some instructions, $\langle B, i, 3\rangle$ does not represent any definition of a variable. To illustrate, consider, for instance, a three-address instruction that has the form

[***jum***, , , $\langle B, i\rangle$],

which represents an unconditional jump to instruction $\langle B, i\rangle$. This instruction obviously does not define any variable.

As we now discuss the use of variables across several blocks, we also generalize some notions previously introduced for a single block. Specifically, for every $B \in \Pi$, ${}_{var}B$ denotes the variables used in *B*, and for every $\Xi \subseteq \Pi$, ${}_{var}\Xi = \{x|\ x \in {}_{var}B \text{ with } B \in \Xi\}$; in other words, ${}_{var}\Xi$ denotes all variables used in Ξ. Furthermore, $def(\Xi)$ denotes the set of all definitions occurring in the instructions of blocks in Ξ, and $var(def(\Xi))$ denotes all the variables defined by an instruction of a block in Ξ. We can compute $def(\Xi)$ and $var(def(\Xi))$ simply by an instruction-by-instruction trace across Ξ. We leave an algorithm that performs this trivial trace as an exercise.

Let $A, B \in \Pi$. If there exists a computation starting from $\langle A, i\rangle$ and ending at $\langle B, j\rangle$ so that $\langle A, i\rangle$ is the only instruction that defines ${}_{B}var(def\langle A, i\rangle)$, then $def\langle A, i\rangle$ *reaches* $\langle B, j\rangle$. In other words, $def\langle A, i\rangle$ reaches $\langle B, j\rangle$ if the execution of no instruction between $\langle A, i\rangle$ and $\langle B, j\rangle$ redefines $var\langle A, i, 3\rangle$ during this computation; noteworthy, between $\langle A, i\rangle$ and $\langle B, j\rangle$ this computation may pass through several other blocks. If $def\langle A, i\rangle$ reaches $\langle B, j\rangle$ with $j = |B|$, then this definition is an *out-definition of B*. Denote the set of all out-definitions of *B* by $out(B)$. Let $var(out(B))$ denote the set of variables whose definition is in $out(B)$. If a definition is in $out(A)$ and (A, B) is in the flow relation ϕ over Π, then this definition is an *in-definition of B*. Denote the set of all in-definitions of *B* by $in(B)$. Simply stated, $in(B)$ and $out(B)$ denote the definitions that enter and leave *B*, respectively. The computation of $in(B)$ and $out(B)$, for every $B \in \Pi$, can be accomplished by the next simple algorithm.

Algorithm 7.4 ***in* and *out* Sets.**

Input • a flow relation ϕ over Π.

Output • $in(B)$ and $out(B)$ for every $B \in \Pi$.

Method

```
begin

   {initialization of the $in$ and $out$ sets}
   for every $B \in \Pi$ do begin
      $in(B) := \emptyset$;
      $out(B) := \emptyset$;
      for $i = |B|$ downto 1 do
         if $\langle B, i, 3\rangle$ represents $def\langle B, i\rangle$ and ${}_Bvar(def\langle B, i\rangle) \notin var(out(B))$ then
            $out(B) := out(B) \cup \{def\langle B, i\rangle\}$;

   {computation of the $in$ and $out$ sets}
   repeat
      for every $B \in \Pi$ do begin
         $in(B) := in(B) \cup \{d | d \in out(A)$ with $(A, B) \in \phi, A \in \Pi\}$;
         $out(B) := out(B) \cup \{d | d \in in(B), {}_Bvar(d) \notin var(def(\{B\}))\}$
      end
   until no change

end.
```

Case Study 34/35 ***in* and *out* Sets.** Let Π denote the basic blocks given in Figure 7.3, where these blocks compose a single three-address-code program fragment. In Figure 7.9, we separate them.

Block A	***O***	***X***	***Y***	***Z***
$\langle A, 1\rangle$	***get***			☞*m*
$\langle A, 2\rangle$	***mov***	1		☞*f*
$\langle A, 3\rangle$	***mov***	2		☞*i*

Block B	***O***	***X***	***Y***	***Z***
$\langle B, 1\rangle$	***jgr***	☞*i*	☞*m*	$\langle D, 1\rangle$

Block C	***O***	***X***	***Y***	***Z***
$\langle C, 1\rangle$	***mul***	☞*i*	☞*f*	☞*f*
$\langle C, 2\rangle$	***add***	☞*i*	1	☞*i*
$\langle C, 3\rangle$	***jum***			$\langle B, 1\rangle$

Block D	***O***	***X***	***Y***	***Z***
$\langle D, 1\rangle$	***put***			☞*f*

Figure 7.9 ***Separate Basic Blocks.***

Recall the flow relation $\phi = \{(A, B), (B, C), (C, B), (B, D)\}$. Initialize the *in* and *out* sets as

$in(A) = in(B) = in(C) = in(D) = \emptyset$

$\mathit{out}(A) = \{\mathit{def}\langle A, 1\rangle, \mathit{def}\langle A, 2\rangle, \mathit{def}\langle A, 3\rangle\}$
$\mathit{out}(B) = \emptyset$
$\mathit{out}(C) = \{\mathit{def}\langle C, 1\rangle, \mathit{def}\langle C, 2\rangle\}$
$\mathit{out}(D) = \emptyset$

As an exercise, perform the **repeat** loop in Algorithm 7.4 to obtain the final *in* and *out* sets as

$\mathit{in}(A) = \emptyset$
$\mathit{in}(B) = \{\mathit{def}\langle A, 1\rangle, \mathit{def}\langle A, 2\rangle, \mathit{def}\langle A, 3\rangle, \mathit{def}\langle C, 1\rangle, \mathit{def}\langle C, 2\rangle\}$
$\mathit{in}(C) = \{\mathit{def}\langle A, 1\rangle, \mathit{def}\langle A, 2\rangle, \mathit{def}\langle A, 3\rangle, \mathit{def}\langle C, 1\rangle, \mathit{def}\langle C, 2\rangle\}$
$\mathit{in}(D) = \{\mathit{def}\langle A, 1\rangle, \mathit{def}\langle A, 2\rangle, \mathit{def}\langle A, 3\rangle, \mathit{def}\langle C, 1\rangle, \mathit{def}\langle C, 2\rangle\}$
$\mathit{out}(A) = \{\mathit{def}\langle A, 1\rangle, \mathit{def}\langle A, 2\rangle, \mathit{def}\langle A, 3\rangle\}$
$\mathit{out}(B) = \{\mathit{def}\langle A, 1\rangle, \mathit{def}\langle A, 2\rangle, \mathit{def}\langle A, 3\rangle, \mathit{def}\langle C, 1\rangle, \mathit{def}\langle C, 2\rangle\}$
$\mathit{out}(C) = \{\mathit{def}\langle C, 1\rangle, \mathit{def}\langle C, 2\rangle, \mathit{def}\langle A, 1\rangle\}$
$\mathit{out}(D) = \{\mathit{def}\langle A, 1\rangle, \mathit{def}\langle A, 2\rangle, \mathit{def}\langle A, 3\rangle, \mathit{def}\langle C, 1\rangle, \mathit{def}\langle C, 2\rangle\}$

Observe that $\mathit{def}\langle A, 2\rangle$ and $\mathit{def}\langle C, 1\rangle$ are both definitions of f, and $\mathit{def}\langle A, 3\rangle$ and $\mathit{def}\langle C, 2\rangle$ are definitions of i. Noteworthy, $\mathit{out}(C)$ does not contain $\mathit{def}\langle A, 2\rangle$ and $\mathit{def}\langle A, 3\rangle$ because this block redefines f and i; on the other hand, $\mathit{out}(C)$ contains $\mathit{def}\langle A, 1\rangle$ because C does not redefine m. Finally, notice that $\mathit{def}\langle A, 1\rangle$, $\mathit{def}\langle A, 2\rangle$, $\mathit{def}\langle A, 3\rangle$, $\mathit{def}\langle C, 1\rangle$, and $\mathit{def}\langle C, 2\rangle$ reaches D so it passes through B.

■

7.2 Optimization of Intermediate Code

In the previous section, we have divided the program into its basic block and tracked how the computed values of variables are changed within each block and between the blocks. In the same block, we find out how the value of a variable is changed by ↴. Between blocks, we track the values of variables by using *in* and *out* sets. Making use of this information, we now describe some optimization techniques that improve the program written in intermediate code, such as three-address code. A rigorous and detailed description of these optimization techniques is beyond the scope of this introductory text, however. Therefore, we just sketch their ideas and illustrate them by examples.

As already stated, we can optimize the code locally and globally. Local optimization improves code within a single block while global optimization spans several blocks. In reality, however, both approaches are literally inseparable because they overlap and support each other. Consider a loop, whose repetition obviously tends to consume time tremendously, so the compiler pays a special attention to its optimization. Typically, the loop body is interpreted as a single basic block, so its optimization is considered as local. However, this local optimization frequently requires some assistance from global optimization, too. Indeed, within a single block representing the loop body, a local optimization technique may detect a code portion that can be moved to a newly created tiny block placed in front of the loop as a whole, so this improvement actually spans a single block. On the other hand, optimization may compose several blocks into a single block to make the computation faster. Therefore, in what follows, we always describe every optimization technique as a whole, but we explicitly distinguish its local and global parts.

Motion of loop invariants. Consider the following FUN loop.

for *i* := 1 ***through*** 100 ***iterate***
    j := *a* * *b* * *i* + *j*;

Figure 7.10 gives the body of this loop written as a single three-address-code basic block, *B*, which also contains the next-use function ↴, defined and studied in the previous section. When *B* starts, *i* equals 1.

Instruction	*O*	*X*	*Y*	*Z*
⟨*B*, 1⟩	***mul***	☞*a*↴0	☞*b*↴0	☞*x*↴2
⟨*B*, 2⟩	***mul***	☞*x*↴0	☞*i*↴4	☞*y*↴3
⟨*B*, 3⟩	***add***	☞*y*↴0	☞*j*↴0	☞*j*↴0
⟨*B*, 4⟩	***add***	☞*i*↴5	1	☞*i*↴5
⟨*B*, 5⟩	***jle***	☞*i*↴0	100	⟨*B*, 1⟩

Figure 7.10 ***Basic Block with the Next-Use Function.***

As *B* is performed a hundred times, so its optimization surely deserves the compiler's attention. Specifically, ☞*a*↴0 and ☞*b*↴0 means that *a* and *b* represent the *loop invariants* that never change its values during the execution of the loop. As a result, we compute *a* * *b* a hundred times with the same result. To eliminate this needless repetition, we move the code of *a* * *b* in front of the loop to compute *a* * *b* just once. In greater detail, regarding *B*, we change ⟨*B*, 1⟩ to block *A* consisting of a single instruction, ⟨*A*, 1⟩, and rename instructions ⟨*B*, *i*⟩ to ⟨*B*, *i* – 1⟩ for all 2 ≤ *i* ≤ 5 as given in Figure 7.11 (we no longer include ↴ into this figure).

Instruction	*O*	*X*	*Y*	*Z*
⟨*A*, 1⟩	***mul***	☞*a*	☞*b*	☞*x*

Instruction	*O*	*X*	*Y*	*Z*
⟨*B*, 1⟩	***mul***	☞*x*	☞*i*	☞*y*
⟨*B*, 2⟩	***add***	☞*y*	☞*j*	☞*j*
⟨*B*, 3⟩	***add***	☞*i*	1	☞*i*
⟨*B*, 4⟩	***jle***	☞*i*	100	⟨*B*, 1⟩

Figure 7.11 ***Loop Optimized by the Motion of*** ***a* * *b*.**

To put it in general, this *code motion* consists in creating a new block that contains the statements that compute the loop invariants and placing this newly creating block, called the *preheader*, in front of the code of the loop as a whole. A variable *x* is a loop invariant if it is defined either out of the loop or by some other invariants in the same loop. In the latter case, however, any code motion concerning the computation of *x* requires a careful investigation of the block. Specifically, there must be only one definition of *x* in the loop. Assume, on the contrary, the loop contains these two definitions

[***mov***, 1, , ☞*x*], …, [***mov***, 0, , ☞*x*]

If we move both before the loop, *x*'s value remains improperly identical throughout the loop although it should change. Furthermore, we have to verify that always only one definition of *x* enters and leaves the loop. This verification may necessitate a study of some other blocks and, thereby, involve global optimization to some extent. Indeed, consider Figure 7.12 that gives three-address code, where block *C* corresponds to a loop. Suppose that ⟨*C*, *i*⟩ is the only instruction that defines *x*. At this point, the definition of *x* in ⟨*C*, *i*⟩ obviously represents an invariant, so we plan to move this instruction out of *C*. However, the value of *x* at ⟨*D*, 1⟩, which follows *C*, depends on whether the jump at ⟨*B*, 1⟩ is made. If we prove that this jump is never performed because the value of *x* is always a positive integer, we remove ⟨*B*, 1⟩ and move ⟨*C*, *i*⟩ before *C* as a preheader; otherwise, we do not make any code motion.

Instruction	*O*	*X*	*Y*	*Z*
…				
⟨*B*, 1⟩	***jle***	☞*x*	0	⟨*D*, 1⟩
⟨*C*, 1⟩	…			
…				
⟨*C*, *i*⟩	***mov***	0		☞*x*
…				
⟨*C*, *n*⟩	***jle***	…	…	⟨*C*, 1⟩
⟨*D*, 1⟩	…			

Figure 7.12 ***Loop with a Definition of*** *x***.**

Loop unrolling. In motion of loop invariants, we have seen how optimization divides a single block to two blocks. Now, in sharp contrast, we are about to see how a single block is substituted for several basic blocks to make the performance of a loop faster. More precisely, apart from the block that contains instructions that perform the body of a loop, the syntax-directed translation usually produces some other instructions or even whole blocks that perform *loop housekeeping*, which is time consuming, too. Indeed, this housekeeping makes initial setup by setting the lower limit of the loop. Then, on each repetition of the loop, it increments the counter and decide whether another iteration of the loop should take place. If the total number of loop repetitions is small, we avoid this loop housekeeping by *unrolling* the loop so its instructions are executed one by one. To illustrate, consider this FUN loop

```
for i = 1 through 2 iterate
    read(x[i]);
```

Figure 7.13 gives four three-address instructions divided into two basic blocks, *A* and *B*, in which ⟨*B*, 1⟩ is the only non-loop-housekeeping instruction of this loop, which is repeated twice. Under these conditions, optimization *unrolls* this loop by replacing blocks *A* and *B* with a single two-instruction block, *C*, given in Figure 7.14.

Instruction	*O*	*X*	*Y*	*Z*
⟨*A*, 1⟩	***mov***	1		☞ *i*
⟨*B*, 1⟩	***get***			☞*x*[*i*]
⟨*B*, 2⟩	***add***	☞*i*	1	☞*i*
⟨*B*, 3⟩	***jle***	☞*i*	2	⟨*B*, 1⟩

Figure 7.13 ***Two Basic Blocks with Three Loop Housekeeping Instructions.***

Instruction	*O*	*X*	*Y*	*Z*
⟨*C*, 1⟩	***get***			☞*x*[1]
⟨*C*, 2⟩	***get***			☞*x*[2]

Figure 7.14 ***Short Basic Block Resulting from Loop Unrolling.***

Computation during Compilation. The optimizer of a compiler obviously shortens the run time of the generated target program by obtaining some results of the target program during the compilation time. Consider a three-address instruction, [***O***, ***X***, ***Y***, ☞*x*], where ***O*** is a binary operation, ***X*** and ***Y*** are operands, and ☞*x* is the symbol-table address of the result. If ***X*** and ***Y*** are numbers, the optimizer can make this improvement without any further analysis of the code. For instance, [***add***, 2, 5, ☞*x*] always sets *x* to 7, so the optimizer substitutes this setting for the ***add*** instruction and, thereby, makes the code faster. If an operand is a variable, the compiler needs to consult the next-use function ↴, *in* sets and *out* sets (see Algorithms 7.2, 7.4) to find out whether this kind of optimization is possible. For instance, consider these three three-address instructions

[***mov***, 2, , ☞*x*], …, [***add***, ☞*x*, 5, ☞*y*], [***add***, ☞*z*, ☞*y*, ☞*z*]

in a block *B*. Suppose that by consulting ↴, the optimizer finds out that the value of *x* is never modified between [***mov***, 2, , ☞*x*] and [***add***, ☞*x*, 5, ☞*y*], so after the execution of these two instructions, the value of *y* is always 7. At this point, the optimizer can eliminate [***mov***, 2, , ☞*x*] and [***add***, ☞*x*, 5, ☞*y*] and replace [***add***, ☞*z*, ☞*y*, ☞*z*] with [***add***, ☞*z*, 7, ☞*z*] to obtain this functionally equivalent but faster running block

~~[***mov***, 2, , ☞*x*]~~, …, ~~[**add**, ☞*x*, 5, ☞*y*]~~, [**add**, ☞*z*, 7, ☞*z*]

To illustrate this technique in global optimization, assume that a block contains [***mov***, 5, , ☞*hourpay*] and another block contains [***mul***, ☞*hourpay*, 8, ☞*daypay*]. Suppose that by using *in* and *out* sets, the optimizer finds out that between these two instructions the value of *hourpay* is never changed. At this point, during the compilation time, the optimizer can determine that the value of *daypay* is always 40, and this determination subsequently allows us to shorten the code analogously as above.

Copy propagation. As an undesired side effect, the syntax-directed translation or some optimization techniques sometimes produce copy instructions of the form [***mov***, ☞*x*, ,☞*a*].

Instruction	*O*	*X*	*Y*	*R*
…				
⟨*A*, *i*⟩	***mov***	☞*a*		☞*x*
…				
⟨*B*, *j*⟩	***add***	☞*b*	☞*x*	☞*b*
…				

Figure 7.15 ***Three-Address Code with a Copy Instruction.***

Consider the code in Figure 7.15. As obvious, ⟨*A*, *i*⟩ is a copy instruction that just moves the current value of *a* to *x*, which is then used as the second operand of ⟨*B*, *j*⟩. Unless *x* or *a* changes between ⟨*A*, *i*⟩ and ⟨*B*, *j*⟩, the optimizer can eliminate ⟨*A*, *i*⟩ and change ⟨*B*, *j*⟩ as described in Figure 7.16.

Instruction	*O*	*X*	*Y*	*Z*
…				
⟨*B*, *j*⟩	***add***	☞*b*	☞*a*	☞*b*
…				

Figure 7.16 ***Three-Address Code without the Copy Instruction.***

Useless code elimination. Consider the above removal of instruction ⟨*A*, *i*⟩ (see Figures 7.15 and 7.16). Suppose that after this removal, *x* is never used in the program. At this point, the optimizer either eliminates this completely useless variable *x* or uses *x* for other purposes. From a global point of view, a typical example of useless code is *dead code*, which often consists of several never-executed blocks scattered throughout the program. As a matter of fact, this dead code is sometimes introduced into the program by its author. For instance, suppose that an inexperienced programmer introduces a Boolean constant, *check*, and includes several program passages whose execution depends solely on the value of *check*; for instance, in Pascal, a passage of this kind might begin like **if** *check* **then writeln** … . During debugging, the programmer sets *check* to true to trace the program. As soon as the program starts to work correctly, the lazy programmer just sets *check* to false without removing the checking passages from the program; as a result, all these passages become dead code.

Code hoisting. Consider the following part of a Pascal program

```
case a of
    1: b := c * d;
    2: b := c * d + x;
    3: b := c * d – x;
    4: b := c * d * x;
    5: b := c * d / x;
end
```

As every branch contains *c* * *d*, the intermediate code generation produces the identical three-address code for this multiplication in each of these five branches. To save space, the optimizer can move this identical three-address code into a block preceding the code corresponding to the branches so it is computed there and its result is placed into a temporary variable, *y*. At this point, the optimizer substitutes *y* for all the five occurrences of the identical three-address code of *c* * *d*. Notice that this optimization saves space but no time. Indeed, it makes the program size smaller, but it does not speed up its execution at all.

7.3 Optimization and Generation of Target Code

The target code generation translates the intermediate representation of a source program to a functionally equivalent target program and, thereby, completes the compilation process. This final compilation phase always heavily depends on the target language, which is usually machine code or assembly language. In this section, we consider the latter as the target language. First, we describe the hypothetical target assembly language, called TARGET, used throughout the rest of this chapter. Then, we explain techniques that most assembly-language generators make use of.

Convetion 7.5. For each variable, *x*, its memory location is denoted by ☛*x* in what follows. The way by which ☛*x* is determined is sketched in the Conclusion of this introductory book, which does not cover the memory management in full detail.

■

TARGET *description*. TARGET has two-operand instructions of the form

$$\rho o\ I, J$$

where *I* is always a register whose contents represents the first operand. By $\rho \in \{r, m\}$, we specify whether the other operand is in a register or in a memory. That is, *r* specifies a register-to-*r*egister instruction in which *J* is a register containing the other operand while *m* specifies a register-to-*m*emory instruction, where *J* is a memory address where the other operand is stored. By *o*, we denote an operator. After performing *o* with the two operands, the instruction places the result into *I*. More specifically, the instruction *ro I*, *J*

is a register-to-register instruction that performs operation *o* with the contents of registers *I* and *J*, then places the result into *I*. The instruction

mo I, ☛*x*

is a register-to-memory instruction that performs operation *o* with the contents of register *I* and the current value of variable *x* stored at the memory address ☛*x*. More specifically, throughout this section, we use the TARGET instructions with operators

ADD, *SUB*, *MUL*, *DIV*, *LOA*, and *STO*.

ADD, *SUB*, *MUL*, and *DIV* denote arithmetic *add*ition, *sub*traction, *mul*tiplication, and *div*ision, respectively. For instance, *rADD I*, *J* is a register-to-register instruction that performs the arithmetic addition with the values placed in registers *I* and *J* and places the result into *I*. Instruction *mMUL I*, ☛*y* is a register-to-memory instruction that performs the arithmetic multiplication with the contents of *I* and the current value of variable *y* addressed by ☛*y*, then places the result into *I*. Operators *LOA* and *STO* are used in two important register-to-memory instructions, *mLOA I*, ☛*x* and *mSTO I*, ☛*x*. The former fills register *I* with the current value of variable *x* while the latter stores the current contents of *I* into variable *x*, addressed by ☛*x*.

TARGET *Generation*. Now, we explain how a target code generator turns a basic block, *B*, to a functionally equivalent TARGET code. Suppose that TARGET uses a finite set of registers, Φ. As the movement between registers and memory is always time consuming, we design the generator of TARGET instructions so we reduce this movement as much as we can. As a result, we produce the resulting code so it works with registers economically, prefers register-to-register to register-to-memory instructions, and contains as few *mLOA* and *mSTO* instructions as possible.

To generate TARGET instructions in this succinct way, the generator makes use of the flow relation ϕ, the next-use function ↴, *in* sets, and *out* sets constructed in Section 7.2. Specifically, suppose that ⟨*B*, *j*⟩'s first operand is the symbol table address of a variable. As an exercise, we discuss how the generator easily determines whether this variable is used again in the program after ⟨*B*, *j*⟩. If not, which means *var*⟨*B*, *j*, 1⟩↴0, the generator makes use of this information to select an optimal register during the generation of TARGET instructions for ⟨*B*, *j*⟩. To keep track about the current contents of registers, it also maintains the *register-contents function* χ defined from Φ to $_{var}B \cup \{0\}$ so that for every register $I \in \Phi$, $\chi(I) = 0$ if and only if *I* is free to use, and $\chi(I) = x$ with $x \in {}_{var}B$ if and only if *x*'s current value occurs in *I*, so *I* is not available for other purposes. We split up the description of the TARGET generation into three algorithms. First, Algorithm 7.6 chooses an optimal register for the first operand of ⟨*B*, *j*⟩'s first operand. Then, Algorithm 7.7 generates a TARGET code for ⟨*B*, *j*⟩. Finally, Algorithm 7.8 generates a TARGET code for the complete block *B*.

Goal. Given a three-address instruction ⟨*B*, *j*⟩ in a block *B*, select a suitable register to accommodate the first operand of this instruction.

Gist. Consider an instruction ⟨*B*, *j*⟩ in *B*, where $1 \leq j \leq |B|$, such that $\langle B, j, 1\rangle \in {}_B\varsigma$; in other words, ⟨*B*, *j*, 1⟩ denotes the pointer to a variable, *x* (see Section 7.1). The next algorithm selects an appropriate register to hold the current value of *x* as follows. If *x*'s current value occurs in a register and, in addition, this value is not needed in the program after ⟨*B*, *j*⟩, the algorithm obviously picks up this register. If this value occurs in no register, it selects an unused register. If there is no free register, it stores the contents of an occupied register and selects this register.

Algorithm 7.6 ***Selection of a Register.***

Input • $\langle B, j\rangle$ in B with $\langle B, j, 1\rangle \in {}_B\varsigma$;
• $\langle B, j, 1\rangle$↴k with $k \in \{j + 1, \ldots, |B|\} \cup \{0\}$.

Output • a selected register, $H \in \Phi$.

Method

```
begin
    if $I \in \Phi$ satisfies $\chi(I) = var\langle B, j, 1\rangle$ and $var\langle B, j, 1\rangle$↴0 then
        select $I$ as $H$
    else if a register, $J \in \Phi$, satisfies $\chi(J) = 0$ then
            select $J$ as $H$
        else
        begin
            select any register, $K \in \Phi$, containing a variable, $x$;
            generate mSTO $K$, ☚$x$;
            set $\chi(K) = 0$;
            select $K$ as $H$
        end
end.
```

Notice that Algorithm 7.6 may work in a more sophisticated way. Specifically, consider register *K* that Algorithm 7.6 frees by moving its current contents to the memory. We could make some statistics regarding the frequency of moving variables between the memory and registers in this way. Based on this statistic information, we then select *K* containing the value of a variable that is likely to remain in memory longer than the variables stored in the other registers. Improvements of this kind are discussed as an exercise.

Goal. For an instruction $\langle B, j\rangle$ in a block *B*, generate the TARGET code.

As Algorithm 7.7, given next, is simple, we omit its gist and discuss this algorithm as an exercise.

Algorithm 7.7 ***Target Code Generation for a Three-Address Instruction.***

Input • $\langle B, j\rangle$ in a block, *B*, with $\langle B, j, k\rangle \in {}_B\varsigma$, $1 \le k \le 3$;
• next-use function ↴ for *B*;
• a TARGET operator, $o_{\langle B, j, 0\rangle}$, equivalent to three-address operator $\langle B, j, 0\rangle$.

Output • a TARGET code for $\langle B, j\rangle$.

Method

```
begin
    use Algorithm 7.6 to select $H \in \Phi$ for $\langle B, j, 1\rangle$;
    if $\chi(H) = 0$ {the current value of $var\langle B, j, 1\rangle$ is not in $H$} then
    begin
        generate the code that moves the current value of $var\langle B, j, 1\rangle$ to $H$;
        set $\chi(H) = var\langle B, j, 1\rangle$
    end;
    if $\chi(K) = \langle B, j, 2\rangle$ for some $K \in \Phi$ {$var\langle B, j, 2\rangle$ is in $K$} then
    begin
        generate $ro_{\langle B, j, 0\rangle}$ $H, K$;
        if $\langle B, j, 2\rangle$↴0 then
            set $\chi(K) = 0$
    end
    else
        generate $mo_{\langle B, j, 0\rangle}$ $H$, ☛$var\langle B, j, 2\rangle$;
    set $\chi(H) = var\langle B, j, 3\rangle$
end.
```

Algorithm 7.7 deserves a few comments. First, if $\chi(H) = 0$, this algorithm produces the code that moves the current value of $var\langle B, j, 1\rangle$ to H. If the current value of $var\langle B, j, 1\rangle$ is in memory, it makes this move simply by generating *mLOA H*, ☛$var\langle B, j, 1\rangle$. Notice, however, that the current value of $var\langle B, j, 1\rangle$ may occur in another register, $I \in \Phi$; formally, $\chi(I) = var\langle B, j, 1\rangle$. At this point, the generated code loads the contents of I into H. Second, in Algorithm 7.7, we suppose that $\langle B, j\rangle$ in B satisfies $\langle B, j, k\rangle \in {}_B\varsigma$, for all $1 \leq k \leq 3$. Of course, in reality, this requirement may not be satisfied because some three-address instructions' components, such as integers or labels, are different from the pointers to variables. A generalization of Algorithm 7.7 that includes these cases represents a straightforward task, however. As a matter of fact, handling the other kinds of components is usually much simpler. Perhaps most significantly, the variables necessitate keeping track of the locations of their current values while the others do not. Therefore, we leave the discussion of Algorithm 7.7 modified and generalized in this way as an exercise.

As demonstrated next, the generation of TARGET code for B as a whole represents a trivial task consisting in repeatedly calling Algorithm 7.7.

Algorithm 7.8 ***Target Code Generation for a Basic Block.***

Input • a basic block, B.

Output • a TARGET code for B.

Method

```
begin
    for $j = 1$ to $|B|$ do
        generate the TARGET code for $\langle B, j\rangle$;
end.
```

Case Study 35/35 TARGET ***Code Generation.*** Suppose that during the translation of a FUN program, the optimized intermediate code generator produced the block in Figure 7.17. Next, we apply Algorithm 7.8 to this block.

Instruction	*O*	*X*	*Y*	*Z*
1	***sub***	☞a↴0	☞b↴2	☞c↴3
2	***mul***	☞b↴0	☞b↴0	☞a↴3
3	***mul***	☞c↴0	☞a↴4	☞c↴4
4	***add***	☞a↴0	☞c↴0	☞d↴0

Figure 7.17 ***Basic Block to be Translated to*** **TARGET.**

As before, we consider only the TARGET instructions with operators *ADD*, *SUB*, *MUL*, *DIV*, *LOA*, and *STO* described earlier in this section. Assume there are only two registers available—*I*, *J*, initialized to 0. For instruction 1, we generate

mLOA I, ☛a
mSUB I, ☛b

and set $\chi(I) = c$. For instruction 2, we generate

mLOA J, ☛b
rMUL J, J

As the resulting value of $\chi(J)$, we obtain $\chi(J) = a$ while $\chi(I)$ remains unchanged. For instruction 3, we generate a single instruction

rMUL I, J

while $\chi(I)$ and $\chi(J)$ remain the same after the generation of this instruction. For the last instruction, we generate

rADD J, I

and set $\chi(J) = d$ and $\chi(I) = 0$. To summarize this generation of the TARGET code, we obtain

```
mLOA    I, ☛a
mSUB    I, ☛b
mLOA    J, ☛b
rMUL    J, J
rMUL    I, J
rADD    J, I
```

■

Having produced the target code for each basic block of an intermediate-code program, we put these blocks together by branch instructions according to the flow relation ϕ (see Section 7.1), which represents a relatively simple task left as an exercise. At this point, we obtain the target code for the complete program. In other words, the compiler achieves its goal because it completes the translation of a source program to a functionally equivalent target program.

Exercises

7.1. Section 7.1 defines many notions concerning tracking the use of variables between the basic blocks and within them. Make a complete list of all these definitions. Explain each of these definitions informally. Create a three-address code and illustrate each definition in terms of this code.

7.2. Section 7.1 explains how to divide a a three-address-code program in a rather informal way. Describe this division formally as an algorithm.

7.3. Consider the following three-address code. Specify its basic blocks and give the flow relation over the set of these blocks.

Instruction	*O*	*X*	*Y*	*Z*
1	***add***	☞*i*	1	☞*i*
2	***jgr***	☞*i*	☞*m*	**5**
3	***mul***	☞*i*	☞*j*	☞*j*
4	***jum***			**1**
5	***put***			☞*j*
6	***mul***	☞*i*	☞*i*	☞*i*
7	***put***			☞*i*

7.4*Solved*. Translate the next FUN program fragment to a functionally equivalent three-address code (see Section 6.1.2). Break the code into its basic blocks. Describe the flow of control between these blocks.

```
read(k);
j = 1;
for i = 1 through k iterate
    j = j * j;
write(k, j);
```

7.5. Extend the discussion of Case Study 32/35 *Basic Blocks* by including some instructions that work with real numbers. Specifically, consider the following FUN program fragment, where *f* is a **real** variable and *m* has the same meaning like in Case Study 32/35. What does it compute? Reformulate the discussion of Case Study 32/35 in terms of this fragment.

```
read(m, f);
for i = 1 through m iterate
    f = f * f;
write(f);
```

7.6. By Algorithm 7.2 *Next-Use Function*, determine the next-use function ↴ for this basic block

Instruction	*O*	*X*	*Y*	*Z*
1	***sub***	☞*a*	☞*b*	☞*a*
2	***mul***	☞*a*	☞*a*	☞*a*
3	***mul***	☞*b*	☞*b*	☞*c*
4	***add***	☞*a*	☞*c*	☞*c*
5	***add***	☞*a*	☞*c*	☞*a*

7.7. In a fully rigorous way, prove that Algorithm 7.4 *in and out Sets* works correctly. Specifically, carefully explain why it is sufficient that this algorithm records only the latest redefinition of every variable, not all the preceding redefinitions of the variable.

7.8. Consider the basic blocks obtained in Exercise 7.4. Determine their *in* and *out* sets by Algorithm 7.4 *in and out Sets*.

7.9. Write a program to implement

(a) Algorithm 7.2 *Next-Use Function*;
(b) Algorithm 7.4 *in and out Sets*.

7.10*Solved*. Consider the following FUN loop. Write the body of this loop as a three-address-code basic block. If this basic block contains a loop invariant, make its motion in front of the block. That is, remove the loop invariant from the basic block, create a preheader that contains the loop invariant, and place this preheader in front of the code of the loop.

```
for i = 1 through 999 iterate
begin
   j = b * b + j;
   k = i + j *b * b
end
```

7.11. Consider the following Pascal program. Express this program in a three-address code. Then, translate the three-address-code program to a functionally equivalent TARGET assembly-language program by using the methods described in Section 7.3.

```
program(input, output);
var i, j, k: integer;
begin
   read(k);
   begin
      j := 0;
      for i := 1 to k do
         j := j + i * i;
      write(j)
   end
end.
```

7.12. With the following C program, perform the same task as in Exercise 7.11.

```
main()
{
   int i = 5;
   int j = 0;
   while(i-- == 1)
      printf("%d", j);
}
```

7.13. Consider the discussion of useless code elimination in Section 7.2. Design an algorithm that detects and eliminates any useless three-address code. Describe this algorithm in a rigorous way.

7.14. Consider the next Pascal **if** statement. Write this statement as a three-address code, *x*. Explain how to optimize *x* by code hoisting discussed in Section 7.2. That is, detect a fragment that occurs in *x* several times, move this fragment into a block preceding *x* so it is computed there and its result is placed into a temporary variable, substituted for the removed fragment in *x*.

```
if a > b then
   a := a * b * a
else if b > a then
        a := b + a * a * b
     else if b = a then
             b := a * a
```

7.15. Write a program to implement

(a) Algorithm 7.6 *Selection of a Register*;
(b) Algorithm 7.7 *Target Code Generation for a Three-Address Instruction*.

7.16. Consider the cooperation between Algorithm 7.6 *Selection of a Register* and Algorithm 7.7 *Target Code Generation for a Three-Address Instruction*. Improve this cooperation so Algorithm 7.6 returns the best possible pair of registers when Algorithm 7.7 generates a register-register assembly instruction. Suggest some more improvements.

7.17. Apply Algorithm 7.8 *Target Code Generation for a Basic Block* to the following block. Describe this application in a step-by-step way.

Instruction	*O*	*X*	*Y*	*Z*
1	***add***	☞*d*↴2	☞*b*↴2	☞*b*↴2
2	***mul***	☞*b*↴0	☞*d*↴3	☞*a*↴4
3	***mul***	☞*c*↴0	☞*d*↴4	☞*c*↴4
4	***add***	☞*a*↴0	☞*c*↴0	☞*d*↴5
5	***add***	☞*d*↴0	☞*d*↴0	☞*d*↴0

7.18. Translate the following Pascal statement to a functionally equivalent three-address code (see Section 6.1.2). Introduce TARGET instructions with logical operators by analogy with the instructions with arithmetic operators *ADD*, *SUB*, *MUL*, and *DIV* (see Section 7.3); then, translate the three-address-code program to a functionally equivalent sequence to the newly introduced logical TARGET instructions.

a **and not** *b* **or** ((*a* **or** *b* **and** *c*) **or** (**not** *a* **or** *c* **and** *b*))

Solutions to Selected Exercises

7.4. The three-address code of the program is given next.

Instruction	*O*	*X*	*Y*	*Z*
1	***get***			☞*k*
2	***mov***	1		☞*j*
3	***mov***	0		☞*i*
4	***add***	☞*i*	1	☞*i*
5	***jgr***	☞*i*	☞*k*	**8**
6	***mul***	☞*j*	☞*j*	☞*j*
7	***jum***			**4**
8	***put***			☞*k*
9	***put***			☞*j*

In this code, the leaders are instructions **1**, **4**, **6**, and **8** for these reasons. First, instruction **1** starts the code. Instruction **4** represents the target of instruction **7**. Instruction **6** follows a branch. Finally, instruction **8** is the target of instruction **5**. Break the code into the four basic blocks *A*, *B*, *C*, and *D* at the leader as described next.

Instruction	*O*	*X*	*Y*	*Z*
⟨*A*, 1⟩	***get***			☞*k*
⟨*A*, 2⟩	***mov***	1		☞*j*
⟨*A*, 3⟩	***mov***	0		☞*i*
⟨*B*, 1⟩	***add***	☞*i*	1	☞*i*
⟨*B*, 2⟩	***jgr***	☞*i*	☞*k*	**8**
⟨*C*, 1⟩	***mul***	☞*j*	☞*j*	☞*j*
⟨*C*, 2⟩	***jum***			**4**
⟨*D*, 1⟩	***put***			☞*k*
⟨*D*, 2⟩	***put***			☞*j*

The flow of control between *A*, *B*, *C*, and *D* is defined by its flow relation ϕ defined as $\phi = \{(A, B), (B, C), (B, D), (C, B)\}$.

7.10. In the FUN loop, *b* * *b* represents the loop invariant. Express this loop as a three-address-code basic block. By analogy with the explanation of the motion of loop invariants in the beginning of Section 7.2, create a preheader that contains the code that executes *b* * *b* and place this preheader in front of the code of the loop.

Conclusion

The purpose of this concluding chapter is threefold. First, we want to make some historical and bibliographical remarks. Second, we point out some graduate-level topics, which were omitted in this book. Finally, we give an overview of selected current trends in compiler writing. By no means is this chapter intended to be exhaustive in any way. Rather, it sketches only selected general ideas concerning the three topics, including suggestions of further reading about them.

Bibliographical Notes

From a historical point of view, the earliest works about compilers appeared in the 1960's when the first high-level programming languages, such as FORTRAN and ALGOL 60, were developed. The crucial studies published at that time include Aho, A. V. and Ullman, J. D. [1969a], Aho, A. V. and Ullman, J. D. [1969b], Barnett, M. P. and Futrelle, R. P. [1962], Conway, M. E. [1963], de Bakker, J. W. [1969], Evey, J. [1963], Ginsburg, S. and Rice, H. G. [1962], Hartmanis, J., Lewis, P. M., II, and Stearns, R. E. [1965], Irons, E. T [1961], Johnson, W. L., Porter, J. H., Ackley, S. I., and Ross, D. T. [1968], Kasami, T. [1965], Knuth, D. E. [1967a], Knuth, D. E. [1967b], Korenjak, A. J. and Hopcroft. J. E. [1966], Kurki-Suonio, R. [1964], Landin, P. J. [1965], Lewis, P. M., II and Stearns, R. E. [1968], McCarthy, J. [1960], McCarthy, J. and Painter, J. [1967], Naur, P. (ed.) [1960], Oettinger, A. G. [1961], and van Wijngaarden, A. (ed.) [1969]. During the last three decades of the twentieth century, the basic knowledge concerning the construction of compilers was summarized in the books Aho, A. V. and Ullman, J. D. [1972], Aho, A. V. and Ullman, J. D. [1973], Aho, A. V. and Ullman, J. D. [1977], Aho, A. V., Lam, M. S., Sethi, R. and Ullman, J. D. [2007], Alblas, H. [1996], Appel, A. W. [1998], Bergmann, S. [1994], Elder, J. [1994], Fischer, C. N. [1991], Fischer, C. [1999], Fraser, C. W. [1995], Gries, D. [1971], Haghighat, M. R. [1995], Hendrix, J. E. [1990], Holmes, J. [1995], Holub, A. I. [1990], Hunter, R. [1999], Kiong, D. B. K. [1997], Lemone, K. A. [1992a], Lemone, K. A. [1992b], Lewis, P. M., II, Rosenkrantz, D. J. and Stearns, R. E. [1976], Louden, K. C. [1997], Mak, R. [1996], Morgan, R. C. [1998], Muchnick, S. S. [1997], Parsons, T. W. [1992], Pittman, T. [1992], Sampaio, A. [1997], Sorenson, P. G. and Tremblay, J. P. [1985], Waite, W. [1993], Wilhelm, R. [1995], and Wirth, N. [1996]. Most of them are still useful and readable today. Recently, compiler writing has been well explained in Allen, R. [2002], Cooper, K. D. [2004], Grune, D., Bal, H., Jacobs, C., and Langendoen, K. [2000], Lee G. and Yew P. [2001], and Srikant, Y. N. [2003].

As demonstrated in this book, any treatment of compilers is literally inseparable from the theory of formal languages, whose models underlie some phases of compilation. This book has presented only the minimum of this theory needed to explain the elements of compiler writing. For a throughout coverage of this theory, consult Bar-Hillel, Y. [1964], Beigel, R. and Floyd, R. W. [1994], Berstel, J. [1979], Book, R. V. (ed.) [1980], Bucher, W. and Maurer, H. A. [1984], Cleaveland, J. C. and Uzgalis, R. [1977], Dassow, J. and Paun, Gh. [1989], Davis, M. D. and Weyuker, E. J. [1983], Eilenberg, S. [1974], Eilenberg, S. [1976], Ginsburg, S. [1966], Harrison, M. A. [1978], Hopcroft, J. E. and Ullman, J. D. [1979], Kelley, D. [1995], Linz, P. [1990], Martin, J. C. [1991], McNaughton, R. [1982], Meduna, A. [2000], Revesz, G. E. [1983], Rozenberg, G. and Salomaa, A. (eds.) [1997], Salomaa, A. [1969], Salomaa, A. [1973], Salomaa, A. [1985], Shyr, H. J. [1991], Sudkamp, T. A. [1988], and Wood, D. [1987].

Graduate Level Topics

This book was designed so all the material it presents can be comfortably and completely covered during a one-term undergraduate term. As a result, more complicated topics were partially or even entirely omitted in the text. Next, once again, we list the chapters of this book one by one and, simultaneously, suggest how to extend their contents by topics that a graduate level class should definitely cover in detail. These graduate level topics are comprehensively covered in several textbooks, including Aho, A. V., Lam, M. S., Sethi, R. and Ullman, J. D. [2007], Cooper, K. D. [2004], Fischer, C. N. [1991], Fischer, C. [1999], Fraser, C. W. [1995], Grune, D., Bal, H., Jacobs, C., and Langendoen, K. [2000], Holub, A. I. [1990], Morgan, R. C. [1998], Muchnick, S. S. [1997], Sorenson, P. G. and Tremblay, J. P. [1985], Wilhelm, R. [1995], and Wirth, N. [1996].

Chapter 1 *Introduction*. In Section 1.2, this book has described the standard compilation of imperative high-level programming languages, such as Pascal or C. In the computer science, however, there exist many other classes of high-level programming languages, such as functional, logic, and object-oriented languages. As these languages fundamentally differ from the imperative programming languages, so do their compilers. The graduate class should explain these differences in detail. Maurer, D. and Wilhelm, R. [1995] represents a good reference concerning various classes of high-level programming languages and their languages. Regarding functional languages and their translation, consult Henderson, P. [1980].

Chapter 2 *Lexical Analysis*. In reality, to accelerate the scanning process described in Section 2.2, a lexical analyzer often scans ahead on the input to recognize and buffer several next lexemes. The class should describe how to buffer these lexemes by using various economically data-organized methods, such as using pairs of cyclic buffers, commonly used for this purpose in compilers. From a more theoretical viewpoint, regarding the coverage of Section 2.3, the class should rigorously describe how to minimize the number of states in any deterministic finite automata; after this theoretically oriented explanation, point out the practical significance of this minimization in terms of the size of lexical analyzer and its implementation. Lexical analysis represents a classical topic of compiler writing discussed in many papers and books, including Aho, A. V., Lam, M. S., Sethi, R. and Ullman, J. D. [2007], Carroll, J. and Long, D. [1989], Choffrut, C. and Culik II, K. [1983], Fischer, C. N. [1991], Fischer, C. [1999], Johnson, W. L., Porter, J. H., Ackley, S. I. and Ross, D. T. [1968], Chomsky, N. and Miller, G. A. [1958], Elgot, C. C. and Mezei, J. E. [1965], Elgot, C. C. and Mezei, J. E. [1965], Gouda, M. G. and Rosier, L. E. [1985], Hopcroft, J. E. [1971], Kleene, S. C. [1956], Minsky, M. L. [1967], Myhill, J. [1957], Rabin, M. O. and Scott, D. [1959], and Sorenson, P. G. and Tremblay, J. P. [1985].

Chapter 3 *Parsing*. The coverage of general parsing can be significantly extended. Consider a grammar $G = ({}_G\Sigma, {}_GR)$, a G-based parser represented by a pushdown automaton $M = ({}_M\Sigma, {}_MR)$ that accepts $L(G)$, and an input string w. If each sequence of moves that M makes on w is of bounded length, the total number of all these sequences is finite. As a result, a crude way of deterministically simulating M working on w is to linearly order all move sequences made by M in some manner and examine. As obvious, however, the move sequence can be smartly arranged so it is possible to simulate the next move sequence by retracing the last moves made until a configuration is reached in which an untried alternative move is possible, and this alternative is then taken. In addition, this *backtracking parsing* is often improved by local criteria that determine that a move sequence cannot lead to the acceptance of w. The graduate level class should formalize and discuss this parsing in detail. In addition, it should discuss the time and space complexity of various parsing algorithms. Furthermore, it should select some general parsers and express them based upon tables. Specifically, it should reformulate Algorithm 3.45 *Cocke-Younger-Kasami Parsing Algorithm* in a tabular way, then it might cover other common tabular parsing methods, such as the *Earley parsing method*. Aho, A. V. and Ullman, J. D. [1972], Aho, A. V. and Ullman, J. D. [1973], Aho, A. V. and Ullman, J. D. [1977], Aho, A. V., Lam, M. S., Sethi, R. and Ullman, J. D. [2007], Allen, R. [2002], Cooper, K. D. [2004], Fischer, C. N. [1991], Fischer, C. [1999], Fraser, C. W. [1995], Holmes, J. [1995], Holub, A. I. [1990], Hunter, R. [1999], Kiong, D. B. K. [1997], Lemone, K. A. [1992a], Lemone, K. A. [1992b], Louden, K. C. [1997], Mak, R. [1996], Morgan, R. C. [1998], Muchnick, S. S. [1997], Pittman, T. [1992], Sampaio, A. [1997], Sorenson, P. G. and Tremblay, J. P. [1985], Srikant, Y. N. [2003], Waite, W. [1993], Wilhelm, R. [1995], and Wirth, N. [1996] explain general parsing in an easy-to-follow way. A graduate level class that concentrates its attention solely on parsing may be based on Sippu, S. and Soisalon-Soininen, E. [1987].

Chapter 4 *Deterministic Top-Down Parsing*. As obvious, the single-symbol lookahead used in LL grammars is a special case of a *k*-symbol lookahead, where $k \geq 1$. The class should demonstrate that for every $k \geq 1$, the language family generated by LL($k + 1$) is a proper superfamily of the language family generated by LL(k) grammars. Then, from a more practical point of view, it should describe LL(k) parsers based upon LL(k) grammars. Finally, the class should select an automatic top-down parser generator, such as *LLGen* (see Grune, D. and Jacobs, C. [1988] and Grune, D., Bal, H., Jacobs, C., and Langendoen, K. [2000]), and practice its use. From a historical perspective, Knuth, D. E. [1971], Korenjak, A. J. and Hopcroft. J. E. [1966], Kurki-Suonio, R. [1964], Rosenkrantz, D. J. and Stearns, R. E. [1970] represent early crucial studies that established the fundamentals of deterministic top-down parsing. Holub, A. I. [1990] gives a useful overview of automatic top-down parser generators. Aho, A. V., Lam, M. S., Sethi, R. and Ullman, J. D. [2007], Alblas, H. [1996], Allen, R. [2002], Appel, A. W. [1998], Bergmann, S. [1994], Elder, J. [1994], Fischer, C. N. [1991], Fischer, C. [1999], Fraser, C. W. [1995], Grune, D., Bal, H., Jacobs, C., and Langendoen, K. [2000], Haghighat, M. R. [1995], Hendrix, J. E. [1990], Holmes, J. [1995], Holub, A. I. [1990], Hunter, R. [1999], Kiong, D. B. K. [1997], Lee G., Yew P. [2001], and Srikant, Y. N. [2003], Wilhelm, R. [1995], and Wirth, N. [1996] contain an advanced explanation of deterministic top-down parsing methods.

Chapter 5 *Deterministic Bottom-Up Parsing*. The graduate class should generalize the precedence parser so they can handle other language constructs than conditions and expressions (see Gries, D. [1971]). In Section 5.2, we have described only simple construction of the LR tables and the corresponding LR parsers, which are based upon these tables. More complicated constructions of the LR tables and the corresponding LR parsers should be covered in the graduate class. The class should extremely carefully explain how these constructions handle the shift-reduce and reduce-reduce problems, mentioned in the conclusion of Section 5.2. This coverage should definitely include the canonical LR parsers, the lookahead LR parsers, and the Brute-Force lookahead LR parsers; all of them are elegantly explained, for instance, in Parsons, T. W. [1992]. A detailed exposition of deterministic bottom-up parsing and its techniques can be found in Aho, A. V. and Ullman, J. D. [1972], Aho, A. V. and Ullman, J. D. [1973], Aho, A. V., Lam, M. S., Sethi, R. and Ullman, J. D. [2007], Allen, R. [2002], Cooper, K. D. [2004], Fischer, C. N. [1991], Fischer, C. [1999], Fraser, C. W. [1995], Haghighat, M. R. [1995], Hendrix, J. E. [1990], Holmes, J. [1995], Holub, A. I. [1990], Hunter, R. [1999], Kiong, D. B. K. [1997], Lee G. and Yew P. [2001], Lemone, K. A. [1992a], Lemone, K. A. [1992b], Louden, K. C. [1997], Mak, R. [1996], Morgan, R. C. [1998], Muchnick, S. S. [1997], Pittman, T. [1992], Sampaio, A. [1997], Srikant, Y. N. [2003], Waite, W. [1993], and Wilhelm, R. [1995].

Chapter 6 *Syntax-Directed Translation and Intermediate Code Generation*. Apart from a parser pushdown, the implementation of a top-down syntax-directed translation usually employs a semantic pushdown. By analogy with a bottom-up syntax-directed translation, the semantic actions are placed onto the parser pushdown. However, since the attributes are pushed in a different sequence from the right-hand sides of rules, they are pushed onto the other pushdown, which does not work synchronically with the parser pushdown during the top-down syntax-directed translation process. The class should explain this translation in detail. For advanced topics concerning the syntax-directed translation and related topics, consult Aho, A. V. and Ullman, J. D. [1969a], Aho, A. V. and Ullman, J. D. [1973], Aho, A. V. and Ullman, J. D. [1977], Aho, A. V., Lam, M. S., Sethi, R. and Ullman, J. D. [2007], Kiong, D. B. K. [1997], Knuth, D. E. [1967a], Lemone, K. A. [1992a], Lemone, K. A. [1992b], Lewis, P. M., II and Stearns, R. E. [1968], Lewis, P. M., II, Rosenkrantz, D. J. and Stearns, R. E. [1976], Levine, J. R. [1992], Sorenson, P. G. and Tremblay, J. P. [1985], and Vere, S. [1970].

Regarding the symbol tables, Chapter 6 only sketched the most common symbol table organizations and techniques. As a part of a programming project, the stack-implemented tree-structured and hash-structured symbol tables should be implemented. Furthermore, the graduate class should describe how to store arrays. Specifically, for a quick access, the arrays of arrays are usually flattened so they can be treated as a one-dimensional array of individual elements and the one-dimensional array elements are usually stored in consecutive locations in order to achieve a quick access. The class should carefully explain and practice this way of storing arrays. Cooper, K. D. [2004], Fischer, C. N. [1991], Fischer, C. [1999], and Sorenson, P. G. and Tremblay, J. P. [1985] belong to the best sources for the symbol tables.

Chapter 7 *Optimization and Target Code Generation*. There exist many more code optimization methods than the methods covered in Chapter 7. The graduate class should select some advanced optimization methods and discuss their time and space complexity. If a compiler performs many of them, it is customarily called an *optimizing compiler* because it spends a significant amount of compilation time on this phase. From a practical point of view, it is important to demonstrate that if a compiler smartly selects a small number of rather simple optimization techniques and performs them, it significantly improves the running time of the target program, yet it works much faster than an optimizing compiler. Allen, R. [2002], Knoop, J. [1998], Morgan, R. C. [1998], and Srikant, Y. N. [2003] contain a detailed exposition of the matters discussed in Chapter 7.

As a matter of fact, a compiler not only generates the target code but also predetermines its execution in many respects. Specifically, before the compilation is over, a compiler has to specify the way the running target program uses the memory. Perhaps most importantly, it determines how the target program stores and accesses data. We usually classify this *run-time memory management* as static and dynamic. In *static memory management*, storage of all variables is allocated at compile time while in *dynamic memory management*, the storage is found for the variables at run time as needed. As obvious, the static management is simple and requires hardly any overhead; on the other hand, the storage allocation made during compilation cannot be altered during execution, and this significant restriction rules out, for instance, any recursive subprograms. The dynamic management is more complex; however, the memory addresses are allocated during the run time, so this management supports recursive subprograms. In essence, there exist two fundamental strategies of dynamic storage allocation—*stack storage* and *heap storage*. When a procedure is called, the stack storage strategy pushes space for its local variables onto a stack, and when the procedure terminates, the space is popped off the stack. As its name indicates, the heap storage makes use of a heap of reusable storage—that is, an area of virtual memory that allows data elements to obtain storage when they are created and returns that storage when they are invalidated. The graduate class should pay a special attention to the run-time memory management because no systematic coverage of this topic is given in the present book. For many details about the run-time memory management, consult Aho, A. V., Lam, M. S., Sethi, R. and Ullman, J. D. [2007], Allen, R. [2002], Appel, A. W. [1998], Bergmann, S. [1994], Elder, J. [1994], Fischer, C. [1999], Fraser, C. W. [1995], Grune, D., Bal, H., Jacobs, C., and Langendoen, K. [2000], Holub, A. I. [1990], Hunter, R. [1999], Kiong, D. B. K. [1997], Lee G. and Yew P. [2001], Srikant, Y. N. [2003], and Wirth, N. [1996].

Modern Trends

In essence, the modern directions of compiler writing can be classified into the trends dealing with its theory, design, and application. Next, we select only one trend in each of these three categories and sketch its fundamentals.

Theory. In essence, the theoretical foundation of parsing is underlain by the context-free grammars. In the formal language theory, however, there exist many modified versions of context-free grammars that are stronger than classical context-free grammars. In fact, many of these modified grammars can work in a deterministic way if they are guided by parsing tables. As a result, based upon these modified grammars, we may build up *deterministic parsers of non-context-free languages*. Next, we sketch two variants of these modified grammars, based on restrictions of their derivations.

Conditional grammars restrict their derivations by context conditions, which can be classified into these three categories—context conditions placed on derivation domains, context conditions placed on the use of productions, and context conditions placed on the neighborhood of the rewritten symbols. These grammars are significantly more powerful than ordinary context-free grammars even if the number of their components, including nonterminals and rule, is significantly reduced. By their context conditions, they can elegantly and flexibly control their derivation process and, thereby, perform this process deterministically. Regarding conditional grammars, consult Meduna, A. and Švec, M. [2005] and its references.

Regulated grammars restrict their derivations by various additional regulating mechanisms, such as prescribed sequences of applied rules. Based upon these grammars, parsers might consult their parsing tables only during some selected steps, after which they could make several steps under the guidance of the regulating mechanism. Parsers based upon these regulated grammars would be stronger and faster than the standard parsers, which consult their tables during every single step of the parsing process. Compiler writing has made hardly any use of the regulated grammars so far although there exists a huge literature concerning them, including Abraham, S. [1972], Aho, A. V. [1968], Bar-Hillel, Y., Perles, M., and Shamir, E. [1961], Csuhaj-Varju, E. [1992], Dassow, J., Paun, Gh., and Salomaa, A. [1993], Ehrenfeucht, A., Kleijn J., and Rozenberg, G. [1985], Ehrenfeucht, A., Pas ten P., and Rozenberg, G. [1994], Fris, I. [1968], Greibach, S. and Hopcroft, J. [1969], Habel, A. [1992], Kelemen, J. [1984], Kelemen, J. [1989], Kral, J. [1973], Meduna, A. [1986], Meduna, A. [1987a], Meduna, A. [1987b], Meduna, A. [1990a], [1990b], Meduna, A. [1991], Meduna, A. [1992], Meduna, A. [1993a], Meduna, A. [1993b], Meduna, A. [1994], Meduna, A. [1995a], Meduna, A. [1995b], Meduna, A. [1996], Meduna, A. [1997a], Meduna, A. [1997b], Meduna, A. [1997c], Meduna, A. [1998a], Meduna, A. [1998b], Meduna, A. [1998c], Meduna, A. [1999a], Meduna, A. [1999b], Meduna, A. [2000a], Meduna, A. [2000b], Meduna, A. [2000c], Meduna, A. [2000d], Meduna, A. [2001], Meduna, A. [2002], Meduna, A. [2003a], Meduna, A. [2003b], Meduna, A. [2004], Meduna, A. and Csuhaj-Varju, E. [1993], Meduna, A. and Fernau, H. [2003a], Meduna, A. and Fernau, H. [2003b], Meduna, A. and Gopalaratnam, M. [1994], Meduna, A. and Horvath, G. [1988], Meduna, A. and Kolář, D. [2000a], Meduna, A. and Kolář, D. [2000b], Meduna, A. and Kolář, D. [2002a], Meduna, A. and Kolář, D. [2002b], Meduna, A. and Švec, M. [2002], Meduna, A. and Švec, M. [2003a], Meduna, A. and Švec, M. [2003b], Meduna, A. and Vurm, P. [2001], Meduna, A., Crooks, C., and Sarek, M. [1994], Navratil, E. [1970], Paun, Gh. [1979], Paun, Gh. [1985], Rosenkrantz, D. J. [1967], Rosenkrantz, D. J. [1969], Rosenkrantz, D. J. and Stearns, R. E. [1970], Rozenberg, G. [1977], Thatcher, J. W. [1967], Urbanek, F. J. [1983], and Vaszil, G. [2004].

Design. As computational cooperation, distribution, concurrence, and parallelism have recently fulfilled a crucial role in computer science, the current compiler design tends to make use of these computational modes as much as possible. As a matter of fact, these modes can be recognized even in the standard way of compilation, described in Section 1.2. Specifically, some of the six fundamental compilation phases, such as the lexical and syntax analysis, run concurrently within the compilation as a whole. Furthermore, through the symbol table, a compiler distributes various pieces of information to its components. It thus comes as no surprise that today's compilers integrate the highly effective modes of computation so the compilation process works in an optimal way. As a result, some parts of modern compilers are further divided into various subparts, which are then run in parallel. To illustrate, the syntax-directed translation is frequently divided into two parts, which work concurrently. One part is guided by a precedence parser that works with expressions and conditions. The other part is guided by a predictive parser that processes the general program flow. In fact, both parts are sometimes further divided into several subprocesses; for instance, several expressions may be parsed in parallel rather than parse them one by one. Of course, this modern design of a syntax-directed translation requires a modification of compilation as a whole. Indeed, prior to this syntax-directed translation, a pre-parsing decomposition of the tokenized source program separates the syntax constructs for both parsers. On the other hand, after the syntax-directed translation based upon the two parsers is successfully completed, the compiler has to carefully compose all the produced fragments of the intermediate code together so the resulting intermediate code is functionally equivalent to the source program. To give another example of a modern compilation design, various optimization methods are frequently applied to the intermediate code in parallel provided that the compiler controls this multi-optimization process. Apart from their design, these modern compilers usually produce the resulting target code to fully use the highly effective computational modes of today's computers. More specifically, it produces the target code that is run in a parallel and distributed way. This way of production usually results from an explicit requirement in the source program. Sometimes, however, a modern compiler itself recognizes that this production is appropriate under the given computer framework and performs it to speed up the subsequent execution of the target code. Whatever it does, it always has to guarantee that the execution of the target code is functionally equivalent to the source program. As obvious, this massively parallel and cooperating design of today's compilers necessitates a careful control over all the subprocesses running in parallel, and this complicated control obviously has to be based upon a well developed theory of computational cooperation and concurrency. It thus comes as no surprise that the formal language theory, which provides compiler writing with the theoretical fundamentals and models, has paid a special attention to the systems that formalize the computational cooperation, distribution, and parallelism. As a result, this theory has supplied compiler writing with *systems of formal models*, such as *grammar systems*, which compose various cooperating and distributing grammars and automata into uniform units. Since in practice, modern compilers have not fully made use of these systems so far, most likely, they will do that in the near future. Meduna, A. [2004] and its references may be consulted about grammar systems. Allen, R. [2002], Almasi, G. S. [1989], Cooper, K. D. [2004], Grune, D., Bal, H., Jacobs, C., and Langendoen, K. [2000], Haghighat, M. R. [1995], Langendoen, K. [2000], Lee G. and Yew P. [2001], Srikant, Y. N. [2003], Tseng, P. [1990], and Zima, H. [1991] discuss modern trends in compiler design.

Application. As computer science plays an important role in today's science as a whole, in the future, several scientific areas out of computer science will probably apply the systematically developed knowledge concerning the language translation, whose basics are described in this book. In some scientific areas, this application is already happening. For instance, as the linguistics naturally wants to translate natural languages as rigorously as possible, it has already made use of this knowledge significantly. However, even some scientific fields seeming remote from computer science apply this knowledge as well. Consider, for instance, the molecular genetics. The language translation is central to this scientific field although it approaches to the translation in a different way. Indeed, as opposed to the standard translation of a string representing a source program to another string representing a target program, the genetics usually studies a repeated translation of strings, representing some organisms. To be more specific, consider a typical molecular organism consisting of several groups of molecules; for instance, take any organism consisting of several parts that slightly differ in behavior of DNA molecules made by specific sets of enzymes. For a short period of time, each of these groups makes the development in its own specific way, ranging from a sequential through semi-parallel to purely parallel way, so during this period, their development is relatively independent. At times, however, these groups communicate with each other, and this communication usually influences the future behavior of the organism as a whole. The simulation of such an organism leads to several translations of strings under various translations modes. A translation of this kind is thus underlain by different translation principles than the principles explained in this book, which has covered the very basics of the standard translation of programming language by compilers. On the other hand, in the future, these standard principles may be somewhat revised by the new approaches to the language translation in such attractive scientific fields as genetics. Floyd, R. W. [1964a] and Hays, D. G. [1967] belong to important pioneer papers dealing with computational linguistics. Paun, Gh. (ed.) [1995] gives a broad spectrum of various topics related to computational linguistics and compilers. Regarding biologically related applications of the knowledge concerning the language specification and translation, consult Meduna, A. and Švec, M. [2005] and its references. Specifically, Chapter 7 of Meduna, A. and Švec, M. [2005] presents a programming language for applications of this kind and, in addition, sketches the design of its compiler.

Appendix: Implementation

Throughout this book, we have designed various parts of a compiler for the programming language FUN by using an easy-to-understand Pascal-like notation. In practice, however, we implement compilers in real high-level programming languages. Therefore, in this appendix, we demonstrate how to implement an important compiler part in this completely realistic way. Specifically, we implement a precedence parser described in Section 5.1 so the parser guides a syntax-directed translation of infix logical expressions to its postfix equivalents. This translation represents a slightly extended version of the translation given in Case Study 28/35 *Postfix Notation*. We program this implementation in C++, which definitely belongs to the most popular programming languages today.

More specifically, we implement the syntax-directed translation given in Case Study 28/35 *Postfix Notation* extended in the following way. Apart from the FUN identifiers of type **integer**, the source infix expressions may include integer numbers as operands. Every positive integer represents the logical value *true* while the zero means *false*. Lexemes and tokens specifying logical operators *and*, *or*, and *not* are denoted by &, |, and !, respectively. The syntax directed translation is defined by this attribute grammar

$C \to\ !C$	{**POSTFIX**(!)}
$C \to C \mid C$	{**POSTFIX**(|)}
$C \to C\ \&\ C$	{**POSTFIX**(&)}
$C \to (C)$	{}
$C \to i$ {\$1}	{**POSTFIX**(\$1)}
$C \to$ # {\$1}	{**POSTFIX**(\$1)}

where #'s attributes are 0 or 1, representing *true* or *false*, respectively. Otherwise, the description of this translation coincides with that given in Case Study 28/35 *Postfix Notation*. For instance, this syntax directed translation converts 0 | ! (*day* & *night*) to #{0} *i*{☞*day*} *i*{☞*night*} & ! |.

Synopsis. First, we conceptualize the implementation in Section A.1. Then, in Section A.2, we give a C++ code of this implementation. As opposed to the previous parts of this book, we present both sections in a somewhat schematic way because computer program code is usually commented in this way.

A.1 Concept

The implementation is conceptualized as an object oriented system based upon these classes

`Lex` recognizes the next lexeme and produces its token that specifies the lexeme. Each identifier is recorded in the symbol table and represented by a token with an attribute, which is a symbol table address to the symbol table record of the identifier.

`PushdownAutomaton` performs the fundamental pushdown operations.

`SymbolTable` stores identifiers.

`PrecedenceParser` makes the syntax analysis of the tokenized infix expression, obtained by `Lex`, by using the precedence parsing method (see Section 4.1). In addition, it controls the semantic actions to produce the postfix equivalent to the infix expression. `PrecedenceParser` holds the instances of the `Lex`, `PushdownAutomaton`, and `SymbolTable` classes.

The following definitions represent the tokens together with the corresponding lexemes. They are specified in the `Defines.h` file as follows

```
typedef enum PD_SYMBOLS
{
    UNDEF         = -1, // used as end of the rule marker
    TOKEN_IDENT   =  0, // 'i' identificator
    TOKEN_INTEGER =  1, // '#' integer
    TOKEN_LOG_AND =  2, // '&' disjunction
    TOKEN_LOG_OR  =  3, // '|' conjunction
    TOKEN_LOG_NEG =  4, // '!' negation
    TOKEN_B_LEFT  =  5, // '(' left parenthesis
    TOKEN_B_RIGHT =  6, // ')' right parenthesis
    BOTTOM        =  7, // '$' used as the pushdown bottom
    N_COND        =  8, // 'C' nonterminal CONDITION
    LSHARP        =  9, // '<' <
    RSHARP        =  10 // '>' >
};
```

UNDEF symbol is used as an endmarker, which marks, for instance, the end of a rule. LSHARP and RSHARP are pushdown symbols. The other symbols correspond to the input tokens.

Class interface

`Lex` performs

- `T_Token * Lex::getNextToken():` getting the next token;
- `void Lex::setToken(int type, string name, int value):` setting `token`'s `type`, `name`, and `value`;
- definition of instance variable `token` representing input symbol

```
typedef struct T_Token
{
    int    type;
    string name;
    int    value;
};

T_token token;
```

`PushdownAutomaton` is implemented as

```
// Definition of the pushdown symbol
typedef struct T_PdSymbol
{
    PD_SYMBOLS symbol; // type of the symbol
    T_Token *  token;  // pointer to the corresponding token if any
};

// definitions of the pushdown structure
typedef vector<T_PdSymbol *> T_PdString;
typedef T_PdString T_PdAutomaton;
typedef T_PdString::iterator T_PdAutomatonIter;
typedef T_PdString::reverse_iterator T_PdAutomatonIterRev;

T_PdAutomaton pdAutomaton;
```

`PushdownAutomaton` performs

- `void PushdownAutomaton::push(T_PdSymbol * symbol)`: pushing a symbol onto the pushdown top;
- `void pop()`: popping the topmost pushdown symbol;
- `T_PdSymbol * PushdownAutomaton::getTopmostTerm()`: getting the topmost pushdown terminal;
- `bool PushdownAutomaton::switchTop(T_PdString oldTop, T_PdString newTop)`: switching a string for another string on the pushdown top;
- `bool PushdownAutomaton::switchSymbol(PD_SYMBOLS symbol, T_PdString newTop)`: replacing the topmost pushdown symbol with a string;
- `bool PushdownAutomaton::compareTop(T_PdString top)`: comparing a topmost pushdown string with another string; if they coincide, it returns **true**, and if they do not, it returns **false**;
- `T_PdString PushdownAutomaton::getSymbolsToReduce()`: getting the pushdown string that starts at the topmost occurrence of symbol < and continues up to the very top of the pushdown;
- `void PushdownAutomaton::setSymbol(T_PdSymbol * pdSymbol, PD_SYMBOLS symbol, T_Token * token)`: setting `pdSymbol` with `symbol` and its corresponding `token`.

`SymbolTable` is implemented as

```
struct T_SymbolInfo
{
    int length;
};

typedef map<string, T_SymbolInfo> T_SymbolTable;
typedef map<string, T_SymbolInfo>::iterator T_SymbolTableIter;

T_SymbolTable symbolTable;
```

`SymbolTable` performs

- `void SymbolTable::installID(string lexeme)`: storing new lexeme into the symbol table;
- `T_SymbolTableIter SymbolTable::getID(string lexeme)`: return of the symbol-table index addressing the given lexeme.

`PrecedenceParser` class stores the grammatical rules in this way

```
typedef vector<PD_SYMBOLS>                T_Rule;
typedef vector<PD_SYMBOLS>::iterator      T_RuleIter;
typedef map<int, T_Rule>                  T_RuleTable;
typedef map<int, T_Rule>::iterator        T_RuleTableIter;
```

In `PrecedenceParser`, an operator precedence table is implemented as

```
char precedenceTable[PTABLEROW][PTABLECOL];
```

The sematic actions corresponding to the rule used during reductions are implemented as

```
char postfixAction[PTABLEROW];
```

Finally, `PrecedenceParser` contains

- `int PrecedenceParser::findRightSide(T_PdString rightSide)`: finding the rule according to its right-hand side;

- `T_Rule PrecedenceParser::getRule(int index)`: getting the rule according to its table index.

A.2 Code

In this section, we give the C++ code of the implementation, divided into the following parts

I. `main`
II. `PrecedenceParser`: key methods of the implementation
III. `PrecedenceParser`: auxilary methods
IV. `SymbolTable`
V. `PushdownAutomaton`
VI. `Lex`

Parts I and II are central to the implementation while the others represent common C++ routines that complete the implementation as a whole, so an experienced C++ programmer can omit them. Most methods are headed by comments of the form

```
/* Class:  ClassName
 * Method: MethodName
 * Description: a short description of the computation performed in ClassName:: MethodName
 * Parameters:
   * parameter 1: ...
   * parameter 2: ...
   ...
 */
… ClassName:: MethodName …
```

I. `main`

The main() function follows next.

```
#include <iostream>
#include "PrecedenceParser.h"

/* Class:
 * Method: main()
 * Description: the main loop of the program
 */
int main(int argc, char *argv[])
{
    PrecedenceParser * precedenceParser = new PrecedenceParser();

    // Class PrecedenceParser implements the syntax-directed translation guided by
    // the precedence parser. An instance of this class is made to perform the
    // syntax-directed translation.

    precedenceParser->runParsing();
    return 0;
}
```

II. `PrecedenceParser`

```
/* Class:  PrecedenceParser
 * Method: PrecedenceParser()
 * Description:  constructor of the PrecedenceParser class, which initializes the
                 instance variables and data structures.  PrecedenceParser is
                 based upon Algorithm 5.2 Operator Precedence Parser.  However, it
                 uses <, >, and = instead of ⌊,⌋, and |, respectively, and always
                 stores < directly on the pushdown to mark the beginning of a
                 handle, which obviously makes the delimitation of a handle
                 simpler and faster.  Symbol _ denotes a blank entry in the
                 precedence table.
*/

PrecedenceParser::PrecedenceParser()
{
    // initialization of the precedence table
    char precedenceTableInit[PTABLEROW][PTABLECOL] =
        { /*         i   #   &   |   ~   (   )   $  */
          /* i */  {'_','_','>','>','_','_','>','>'},
          /* # */  {'_','_','>','>','_','_','>','>'},
          /* & */  {'<','<','>','>','<','<','>','>'},
          /* | */  {'<','<','<','>','<','<','>','>'},
          /* ~ */  {'<','<','>','>','<','<','>','>'},
          /* ( */  {'<','<','<','<','<','<','=','_'},
          /* ) */  {'_','_','>','>','_','_','>','>'},
          /* $ */  {'<','<','<','<','<','<','_','_'}
        };

    // filling PrecedenceTable with precedenceTableInit
    int size = PTABLEROW * PTABLECOL * sizeof(char);
    memcpy(precedenceTable, precedenceTableInit, size);

// Actions that generate the postfix notation. Index to the postfixAction array
// corresponds to the index of a rule in ruleTable. In this way, we associate each
// rule with its semantic action that generates the postfix notation.
    postfixAction[0] = '!';
    postfixAction[1] = '|';
    postfixAction[2] = '&';
    postfixAction[3] = '_'; // no postfix action defined for brackets
    postfixAction[4] = 'i';
    postfixAction[5] = '#';


    // Initialization of the grammatical rules follows next. The first symbol of
    // each rule detones the left-hand side of the rule.
    PD_SYMBOLS rules[RULECOUNT][RULELENGTH]=  {
          { N_COND, TOKEN_LOG_NEG, N_COND       , UNDEF        , UNDEF },
          { N_COND, N_COND       , TOKEN_LOG_OR , N_COND       , UNDEF },
          { N_COND, N_COND       , TOKEN_LOG_AND, N_COND       , UNDEF },
          { N_COND, TOKEN_B_LEFT , N_COND       , TOKEN_B_RIGHT, UNDEF },
          { N_COND, TOKEN_IDENT  , UNDEF        , UNDEF        , UNDEF },
          { N_COND, TOKEN_INTEGER, UNDEF        , UNDEF        , UNDEF }
    };

    // Filling ruleTable with the rules
    initRules(rules, RULECOUNT);

    // Initialization of symbolTable, lex, and pdAutomaton.
    symbolTable = new SymbolTable;
    lex = new Lex(symbolTable);
    pdAutomaton = new PushdownAutomaton();
}

/* Class: PrecedenceParser
 * Method: runParsing()
 * Description: implementation of the syntax-directed translation
 *              based upon the precedence parser
*/
int PrecedenceParser::runParsing()
{
```

```
T_Token    *token;
T_PdSymbol *pdSymbol;
T_PdSymbol *pdSymbolIns;

PD_SYMBOLS inSymbol;
T_PdString rightSide;
T_PdString newTop;
T_PdString oldTop;
T_PdString stringToReduce;
T_Rule     reduceRule;
T_PdString leftSide;
T_PdString::iterator iter2;
T_PdSymbol * pdSymbolTmp;

// pdSymbol represents a pushdown symbol.
pdSymbol = new T_PdSymbol;
setSymbol(pdSymbol, BOTTOM, NULL);

// pushing symbol $, which marks the pushdown bottom
pdAutomaton->push(pdSymbol);

// getting the first token
token = lex->getNextToken();

// Token is stored into inSymbol.
inSymbol = (PD_SYMBOLS)token->type;

do{
    // pdSymbol records the topmost terminal.
    pdSymbol = pdAutomaton->getTopmostTerm();

    // inSymbol and pdSymbol determines the corresponding precedenceTable
    // entry, which then this entry specifies the parsing action to perform.
    switch(precedenceTable[pdSymbol->symbol][inSymbol])
    {
        case '=': /* two computational actions are performed:
                        (1):  push(inSymbol) &
                        (2):  get the next token
                  */

            /* (1):  push(inSymbol) */
            pdSymbol = new T_PdSymbol;
            setSymbol(pdSymbol, inSymbol, token);
            pdAutomaton->push(pdSymbol);

            /* (2): read the next token */
            token = lex->getNextToken();
            inSymbol = (PD_SYMBOLS)token->type;
            break;

        case '<': /* three computational actions are performed:
                  (1): replacement of pdSymbol with string consisting of
                       pdSymbol followed by symbol <
                  (2): push(inSymbol)
                  (3): reading the next token
                  */

            /* (1): replacement of pdSymbol with a string consisting of
                       pdSymbol followed by symbol <  */

            // preparing the string consisting of
            // pdSymbol followed by symbol <
            newTop.clear();
            newTop.push_back(pdSymbol);
            pdSymbolIns = new T_PdSymbol;
            setSymbol(pdSymbolIns, LSHARP, NULL);
            newTop.push_back(pdSymbolIns);
            // replacing pdSymbol with the string prepared above
            pdAutomaton->switchSymbol(pdSymbol->symbol, newTop)

            /* (2): push(inSymbol) */
            pdSymbolIns = new T_PdSymbol;
            setSymbol(pdSymbolIns, inSymbol, token);
            pdAutomaton->push(pdSymbolIns);

            /* (3): read the next token */
            token = lex->getNextToken();
            inSymbol = (PD_SYMBOLS)token->type;

            break;

        case '>': /* two computational actions are performed:
                  (1):  if the topmost occurrence of symbol < is followed by
                            string y, make sure there exists a rule, r, with
                            the right-hand  side equal to y, and if this is the
                            case, reduce <y to the the left-hand side of r on
                            the pushdown top;
                  (2):  write out r
                  */

            /* (1): reduction */
            // finding <y on the pushdown top
            stringToReduce.clear();
            stringToReduce = pdAutomaton->getSymbolsToReduce();

            // testing that a string starting with symbol <
            // occurs on the pushdown top
            if(!stringToReduce.empty())
            {
                // placing y into handle
                T_PdString handle = stringToReduce;
                handle.erase(handle.begin());

                // finding rule r with y on the right-hand side
                int res = findRightSide(handle);

                // r with y on the right-hand side is found
                if(res != -1)
                {
                    // getting rule r of the form C -> y
                    reduceRule = getRule(res);

                    pdSymbolIns = new T_PdSymbol;
                    setSymbol(pdSymbolIns, *(reduceRule.begin()), NULL);
                    leftSide.clear();
                    leftSide.push_back(pdSymbolIns);

                    //reducing y to C on the pushdown top
                    pdAutomaton->switchTop(stringToReduce, leftSide);

                    // postfixAction[] specifies the semantic action
                    // associated with the rule according to which a reduction
                    // is performed

                    char c = postfixAction[res];
                    if(c != '_')
                    {
                        // some action prescribed
                        if(c == '#') // integer value
                        {
                        // As described in Lex, 0 represents the zero value
                        // while 1 represents any non-zero value in this
                        // implementation.  The next line converts the value
```

```
                            // to the corresponding character that is concatenated
                            // with the current postfix expression string.
                            char cVal =
                                    (*(stringToReduce.back())).token->value+'0';
                                postfixExpr.push_back(cVal);
                            }
                            else if(c == 'i')
                            {
                                postfixExpr.push_back(postfixAction[res]);
                            }
                            else
                            {
                                postfixExpr.push_back(postfixAction[res]);
                            }
                        }
                    }
                    else
                    // no rule has y on its right-hand side
                    {
                        cerr << "SYNTAX ERROR: the right-hand side of the
                                 rule is missing" << endl;
                        return 0;
                    }
                }
                else
                // no symbol < occurs in the pushdown
                {
                    cerr << "SYNTAX ERROR: no symbol < occurs in the pushdown,
                             so no handle is delimited"
                         << endl;
                    return 0;
                }
                break;
            default:
                cerr << "SYNTAX ERROR: table-detected error by a blank entry"
                     << endl;
                return 0;
                break;
        }
        pdSymbolTmp = pdAutomaton->getTopmostTerm();
    }while( !((pdSymbolTmp->symbol == BOTTOM) && (inSymbol == BOTTOM)) );

    return 0;
}
```

Parts I and II, given above, represent the crucial portion of the implementation of the syntax-directed translation. Parts III through VI, given in the rest of this appendix, contain several common routines just to complete the implementation as a whole. Any C++ programmer can only skim these parts.

III. `PrecedenceParser`: auxilary methods

```
/* Class:  PrecedenceParser
 * Method: initRules()
 * Description: storing the grammatical rules into ruleTable
 * Parameters:
   * rules: table containing definitions of the grammatical rules
   * rulecount: the number of rules
 */
void PrecedenceParser::initRules(PD_SYMBOLS rules[RULECOUNT][RULELENGTH], int
rulecount)
{
    T_Rule rule;
    PD_SYMBOLS undefSymbol;
    int j;
    undefSymbol = UNDEF;
    for(int i = 0; i < rulecount; i++)
    {
        rule.clear();
        j = 0;
        while(rules[i][j] != undefSymbol)
        {
            rule.push_back(rules[i][j]);
            j++;
        }
        ruleTable.insert(T_RuleTable::value_type(i, rule));
    }
}


/* Class:  PrecedenceParser
 * Method: getRule
 * Description: getting the rule stored under the given index
 *              in ruleTable
 * Parameters:
   * index: index to ruleTable
 */
T_Rule PrecedenceParser::getRule(int index)
{
    T_Rule rule;
    T_RuleTableIter ruleTableIter;

    ruleTableIter = ruleTable.find(index);

    if(ruleTableIter != ruleTable.end())
    {
        rule = (*ruleTableIter).second;
    }
    return rule;
}


/* Class:  PrecedenceParser
 * Method: findRightSide()
 * Description: looking for the rule with the right-hand side equal
 *              to rightSide
 * Parameters:
   * rightSide: the rule’s right-hand side
 */
int PrecedenceParser::findRightSide(T_PdString rightSide)
{
    T_RuleIter  ruleIter;
    T_PdString::iterator  rightIter;
    T_RuleTableIter ruleTableIter;
    ruleTableIter = ruleTable.begin();
    bool compared;

    while(ruleTableIter != ruleTable.end())
    {
        rightIter = rightSide.begin();

        ruleIter = ((*ruleTableIter).second.begin());

        ruleIter++;
        compared = true;
        while( (ruleIter != (*ruleTableIter).second.end()) &&
               ( rightIter != rightSide.end())
              )
        {
            if((*ruleIter) != (*rightIter)->symbol)
            {
                compared = false;
            }
            ruleIter++;
            rightIter++;
        }
        if( compared &&
            (ruleIter == (*ruleTableIter).second.end()) &&
            (rightIter == rightSide.end())
          )
        {
            return (*ruleTableIter).first;
        }

        ruleTableIter++;
    }
    return -1;
}
```

IV. SymbolTable

```
/* Class:  SymbolTable
 * Method: installID()
 * Description: storing a new lexeme into the symbol table
 * Parameters:
   * lexeme: installed lexeme name
 */
void SymbolTable::installID(string lexeme)
{
    T_SymbolInfo symbolInfo;

    symbolInfo.length = lexeme.length();

    if(symbolTable.count(lexeme) == 0)
    {
        symbolTable.insert(T_SymbolTable::value_type(lexeme, symbolInfo));
    }
}

/* Class:  SymbolTable
 * Method: getID()
 * Description: getting a symbol table address pointing to the
 *              lexeme record
 * Parameters:
   * lexeme: name of the searched lexeme

 */
T_SymbolTableIter SymbolTable::getID(string lexeme)
{
    T_SymbolTableIter symbolTableIter;

    symbolTableIter = symbolTable.find(lexeme);
    return symbolTableIter;
}
```

V. PushdownAutomaton

```
/* Class:  PushdownAutomaton
 * Method: push()
 * Description: pushing a symbol onto the pushdown top
 * Parameters:
   * pdSymbol: a pushdown symbol to store
 */
```

```
void PushdownAutomaton::push(T_PdSymbol * pdSymbol)
{
    pdAutomaton.push_back(pdSymbol);
}


/* Class:  PushdownAutomaton
 * Method: pop()
 * Description: removing the topmost pushdown symbol
 */

void PushdownAutomaton::pop()
{
    if(!pdAutomaton.empty())
    {
        pdAutomaton.pop_back();
    }
}


/* Class:  PushdownAutomaton
 * Method: getTopmostTerm()
 * Description: getting the topmost terminal in the pushdown
 */
T_PdSymbol * PushdownAutomaton::getTopmostTerm()
{
    T_PdAutomatonIterRev pdIterRev;
    pdIterRev = pdAutomaton.rbegin();
    while(pdIterRev != pdAutomaton.rend())
    {
        if( ((*pdIterRev)->symbol >= TOKEN_IDENT) &&
            ((*pdIterRev)->symbol <= BOTTOM)
          )
        {
            return (*pdIterRev);
        }
        else
        {
            pdIterRev++;
        }
    }
    return NULL;
}

/* Class:  PushdownAutomaton
 * Method: switchTop()
 * Description: switching a string for another string on the pushdown top
 * Parameters:
   * oldTop: see parameter newTop
   * newTop: newTop is switched for oldTop
 */
bool PushdownAutomaton::switchTop(T_PdString oldTop, T_PdString newTop)
{
    if(compareTop(oldTop))
    {
        pdAutomaton.erase(pdAutomaton.end() - oldTop.size(), pdAutomaton.end());
        pdAutomaton.insert(pdAutomaton.end(), newTop.begin(), newTop.end());
        return true;
    }
    else
    {
        return false;
    }
}


/* Class:  PushdownAutomaton
```

```
 * Method: switchSymbol()
 * Description: switching a single symbol for a string on the pushdown top
 * Parameters:
   * symbol: see parameter newTop
   * newTop: newTop is switched for symbol
 */
bool **PushdownAutomaton::switchSymbol**(PD_SYMBOLS symbol, T_PdString newTop)
{
    T_PdString::iterator iter;
    T_PdString::iterator iterErased;

    iter = pdAutomaton.end();
    iter--;
    while(iter >= pdAutomaton.begin())
    {
        if((*iter)->symbol == symbol)
        {
            iter = pdAutomaton.erase(iter);
            pdAutomaton.insert(iter, newTop.begin(), newTop.end());
            return true;
        }
        iter--;
    }
    return false;
}


/* Class:  PushdownAutomaton
 * Method: compareTop()
 * Description: comparing a pushdown top string with another string
 * Parameters:
   * cmpTop: compared string
*/
bool **PushdownAutomaton::compareTop**(T_PdString cmpTop)
{
    T_PdAutomatonIter    pdIter;
    T_PdString::iterator cmpTopIter;
    pdIter = pdAutomaton.end();

    if(cmpTop.size() > pdAutomaton.size())
    {
        return false;
    }
    pdIter--;
    cmpTopIter = cmpTop.end();
    cmpTopIter--;
    while(cmpTopIter >= cmpTop.begin())
    {
        if((*cmpTopIter)->symbol != (*pdIter)->symbol)
        {
            return false;
        }
        pdIter--;
        cmpTopIter--;
    }
    return true;
}


/* Class:  PushdownAutomaton
 * Method: getSymbolsToReduce()
 * Description: getting a handle
*/
T_PdString **PushdownAutomaton::getSymbolsToReduce**()
{
    T_PdString symbols;
    T_PdAutomatonIter pdIter;
    pdIter = pdAutomaton.end();
    pdIter--;

    while(pdIter >= pdAutomaton.begin())
    {
        symbols.insert(symbols.begin(),*pdIter);
        if((*pdIter)->symbol == LSHARP)
        {
            return symbols;
        }
        pdIter--;
    }
    return symbols;
}


/* Class:  PushdownAutomaton
 * Method: setSymbol()
 * Description: setting pdSymbol with symbol and its corresponding token
*/

void PushdownAutomaton::setSymbol(T_PdSymbol * pdSymbol, PD_SYMBOLS symbol,
T_Token * token)
{
    pdSymbol->symbol = symbol;
    pdSymbol->token = token;
}
```

VI. Lex

```
/* Class:  Lex
 * Method: Lex()
 * Description: constructor of the Lex class
 * Parameters:
   * pSymbolTable: pointer to a symbolTable entry
*/
Lex::Lex(SymbolTable * pSymbolTable)
{
    symbolTable = pSymbolTable;
    line = 0;
    pos = 0;
}


/* Class:  Lex
 * Method: ~Lex()
 * Description: destructor of the Lex class
 */

Lex::~Lex()
{
    if(token)
        delete token;
}


/* Class:  Lex
 * Method: setToken()
 * Description: setting token’s type, name, and value
 */

void Lex::setToken(int type, string name, int value)
{
    token->type = type;
    token->name = name;
    token->value = value;
}


/* Class:  Lex
 * Method: getNextToken()
 * Description: getting the next token
 */

T_Token * Lex::getNextToken()
{
    STATE state = STATE_START;
    token = new T_Token;
    bool run = true;
    char c;
    string lexeme("");
    T_SymbolTableIter symbolIter;

    while(run)
    {
        // to be able to process last read character
        if(!cin.get(c))
        {
            run = false;
            c = ' ';
        };

        // line counter
        if(c == '\n')
        {
            line++;
            pos = 0;
        }
        else
        {
            // position counter
            pos++;
        }

        //
        // the implementation of a standard finite automaton to recognize a lexeme
        //
        switch(state)
        {
            case STATE_START:
                if(isdigit(c))
                {
                    lexeme += c;
                    state = STATE_INT;
                }
                else if(isalpha(c))
                {
                    lexeme += c;
                    state = STATE_IDENT;
                }
                else if(c == '&')
                {
                    // conjunction symbol recognized
                    setToken(TOKEN_LOG_AND, "&", 0);
                    return token;
                }
                else if(c == '|')
                {
                    // disjunction symbol recognized
                    setToken(TOKEN_LOG_OR, "|", 0);
                    return token;
                }
                else if(c == '!')
                {
                    // logical negation recognized
                    setToken(TOKEN_LOG_NEG, "!", 0);
                    return token;
                }
                else if(c == '(')
                {
                    // left parenthesis recognized
                    setToken(TOKEN_B_LEFT, "(", 0);
                    return token;
                }
                else if(c == ')')
                {
                    // right parenthesis recognized
                    setToken(TOKEN_B_RIGHT, ")", 0);
                    return token;
                }
                else if(isspace(c))
                {
                    continue;
                }
                else
                {
                    cerr << "Undefined symbol on line: " << line << " position:"
                         << pos;
                    return NULL;
                }
                break;
            case STATE_IDENT:
                if(isdigit(c))
                {
                    lexeme += c;
                }
                else if(isalpha(c))
                {
                    lexeme +=c;
                }
                else if(isspace(c))
                {
                    // an identifier is recognized
                    setToken(TOKEN_IDENT, lexeme.c_str(), 0);
                    state = STATE_START;
                    // storing the identifier into the symbol table
                    symbolTable->installID(lexeme);
                    return token;
                }
                else if( c == '&' ||
                         c == '|' ||
                         c == '!' ||
                         c == '(' ||
                         c == ')'
                       )
                {
                    // an identifier is recognized
                    // we make RETURN-CHARACTER operation
                    cin.putback(c);
                    pos--;
                    setToken(TOKEN_IDENT, lexeme.c_str(), 0);
                    state = STATE_START;
                    // we have to store identificators into the symbol table
                    symbolTable->installID(lexeme);
                    return token;
                }
                else
                {
                    cout << "Illegal symbol at line: " << line << " position:"
```

```
                        << pos;
                    return NULL;
                }
                break;
            case STATE_INT:
                if(isdigit(c))
                {
                    lexeme += c;
                }
                else if(isspace(c))
                {
                    state = STATE_START;

                    // we define 0 for zero input, and 1 for
                    // nonzero input
                    int val = ( atoi(lexeme.c_str()) == 0) ? 0 : 1;
                    setToken(TOKEN_INTEGER, "", val);

                    return  token;
                }
                else if( c == '&' ||
                         c == '|' ||
                         c == '!' ||
                         c == '(' ||
                         c == ')'
                       )
                {
                    cin.putback(c);
                    pos--;
                    state = STATE_START;

                    // integer recognized
                    int val = ( atoi(lexeme.c_str()) == 0) ? 0 : 1;
                    setToken(TOKEN_INTEGER, "", val);

                    return token;
                }
                else
                {
                    cerr << "LEXICAL ERROR:” << line << " position: "  << pos;
                    return NULL;
                }
                break;
            default:
                cerr << "Lexical error at line: " << line << " position:" << pos;
                return NULL;
                break;
        } // end of switch
    } // end of while
    setToken(BOTTOM, "$", 0);
    return token;
}
```

In this appendix, we have demonstrated how to implement the syntax-directed translation described in Case Study 28/35 *Postfix Notation*. As obvious, however, a complete compiler represents a more complicated piece of work in reality. As stated in the preface, this book is supported by a web page, which includes several substantial portions of various real compilers.

Bibliography

This bibliographical list contains all the publications referred to in Conclusion and, in addition, some more useful references closely related to compiler writing.

Abraham, S. [1972]. "Compound and Serial Grammars," *Information and. Control* 20, 432-438.

Aho, A. V. (ed.) [1973]. *Currents in the Theory of Computing*, Prentice Hall, Englewood Cliffs, New Jersey.

Aho, A. V. [1968]. "Indexed Grammars - An Extension of Context-free Grammars," *Journal of the ACM* 15, 647-671.

Aho, A. V. [1980]. "Pattern Matching in Strings," in Book, R.V. (ed.), *Formal Language Theory: Perspectives and Open Problems*, Academic Press, New York, 325-247.

Aho, A. V. and Ullman, J. D. [1969a]. "Syntax Directed Translations and the Pushdown Assembler," *Journal of Computer and System Sciences* 3, 37-56.

Aho, A. V. and Ullman, J. D. [1969b]. "Properties of Syntax Directed Relations," *Journal of Computer and System Sciences* 3, 319-334.

Aho, A. V. and Ullman, J. D. [1972]. *The Theory of Parsing, Translation and Compiling, Volume I: Parsing*, Prentice Hall, Englewood Cliffs, New Jersey.

Aho, A. V. and Ullman, J. D. [1973]. *The Theory of Parsing, Translation and Compiling, Volume II: Compiling*, Prentice Hall, Englewood Cliffs, New Jersey.

Aho, A. V. and Ullman, J. D. [1977]. *Principles of Compiler Design*, Addison-Wesley, Reading, Massachusetts.

Aho, A. V., Lam, M. S., Sethi, R. and Ullman, J. D. [2007]. *Compilers*: *Principles, Techniques, and Tools*, Second Edition, Pearson Education, Boston, Massachusetts.

Alblas, H. [1996]. *Practice and Principles of Compiler Building with C*, Prentice Hall, London.

Allen, R. [2002]. *Optimizing Compilers for Modern Architectures: a Dependence-based Approach*, Morgan Kaufmann, London.

Almasi, G. S. [1989]. *Highly Parallel Computing*, Benjamin/Cummings, Redwood City.

Appel, A. W. [1998]. *Modern Compiler Implementation in ML*, Cambridge University Press, Cambridge.

Appel, A. W. [2002]. *Modern Compiler Implementation in Java*, Cambridge University Press, Cambridge.

Arbib, M. A., Kfoury, A. J., and Moll, R. N. [1981]. *A Basis for Theoretical Computer Science*, Springer-Verlag, New York.

Ashcroft, E. A. and Wadge, W. W., LUCID [1976]. “A Formal System for Writing and Proving Programs,” *SIAM Journal on Computing* 5, 336-354.

Backus, J. W. [1959]. “The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference,” *Proceedings of the International Conference on Information Processing*, UNESCO, 125-132.

Banerji, R. B. [1963]. “Phrase Structure Languages, Finite Machines, and Channel Capacity,” *Information and Control* 6, 153-162.

Bar-Hillel, Y. [1964]. *Language and Information*. Addison-Wesley, Reading, Massachusetts.

Bar-Hillel, Y., Perles, M., and Shamir, E. [1961]. “On Formal Properties of Simple Phrase Structure Grammars,” *Zeitschrift fur Phonetik Sprachwissenschaft und Kommunikations-Forschung* 14, 143-172.

Barnes, B. H. [1970]. “A Programmer's View of Automata,” *Computing Surveys* 4, 221-239.

Barnett, M. P. and Futrelle, R. P. [1962]. “Syntactic Analysis by Digital Computer,” *Communications of the ACM* 5, 515-526.

Becker, C. B. [1983]. *Software Testing Techniques*, Van Nastrand Reinhold, New York.

Beckmann, F. S. [1980]. *Mathematical Foundations of Programming*, Addison-Wesley, Reading, Massachusetts.

Beigel, R. and Floyd, R. W. [1994]. *The Language of Machines*, Freeman, New York.

Bellmann, R. E. and Dreyfus, S. E. [1962]. *Applied Dynamic Programming*, Princeton University Press, Princeton, New Jersey.

Bennett, J. P. [1990]. *Introduction to Compiling Techniques: a First Course Using ANSI C, LEX and YACC*, McGraw-Hill, London.

Bentley, J. L. and Ottmann, Th. [1981]. “The Complexity of Manipulating Hierarchically Defined Sets of Rectangles,” Mathematical Foundations of Computer Science 1981, *Springer-Verlag Lecture Notes in Computer Science* 118, 42005.

Bentley, J. L., Ottmann, Th. and Widmayer, P. [1983]. “The Complexity of Manipulating Hierarchically Defined Sets of Rectangles,” in Preparata, F.P.(ed.), *Advances in Computing Research* 1, JAI Press, Greenwich, Connecticut, 127-158.

Berger, R. [1966]. “The Undecidability of the Domino Problem,” *Memoirs of the American Mathematical Society* 66.

Bergmann, S. [1994]. *Compiler Design: Theory, Tools, and Examples*, W.C. Brown, Oxford.

Berlekamp, E. R., Conway, J. H., and Guy, R. K. [1982]. *Winning Ways, Volume 2: Games in Particular*, Academic Press, New York.

Berstel, J. [1979]. *Transductions and Context-Free Languages*, Teubner, Stuttgart, West Germany.

Berstel, J. and Boasson, L. [1974]. “Une Suite Decroissante de Cones Rationnels,” *Springer-Verlag Lecture Notes in Computer Science* 14, 383-397.

Bobrow, L. S. and Arbib, M. A. [1974]. *Discrete Mathematics: Applied Algebra for Computer and Information Science,* W. B. Saunders, Philadelphia, Pennsylvania.

Book, R. V. (ed.) [1980]. *Formal Language Theory: Perspectives and Open Problems,* Academic Press, New York.

Bourne, S. R. [1983]. *The UNIX System*, Addison-Wesley, Reading, Massacussetts.

Braffort, P. and Hirschberg, D. (ed.) [1963]. *Computer Programming and Formal Systems,* North-Holland, Amsterdam.

Brainerd, W. S. and Landweber, L. H. [1974]. *Theory of Computation*, John Wiley & Sons, New York.

Brookshear, J. G. [1989]. *Theory of Computation*, Benjamin/Cummings, Redwood City, California.

Brzozowski, J. A. [1962]. “A Survey of Regular Expressions and Their Applications,” *IEEE Transactions on Electronic Computers* 11, 324-335.

Brzozowski, J. A. [1964]. “Derivates of Regular Expressions,” *Journal of the ACM* 11, 481-494.

Brzozowski, J. A. [1980]. “Open Problems about Regular Languages,” in Book, R.V. (ed.). *Formal Language Theory: Perspectives and Open Problems,* Academic Press, New York, 23-47.

Brzozowski, J. A. and McCluskey, Jr., E. J. [1963]. “Signal Flow Graph Techniques for Sequential Circuit State Diagrams,” *IEEE Transactions on Electronic Computers* EC-12, 67-76.

Brzozowski, J. A. and Yoeli, M. [1976]. *Digital Networks*, Prentice Hall, Englewood Cliffs, New Jersey.

Bucher, W. and Maurer, H. A. [1984]. *Teoretische Grundlagen der Programmiersprachen: Automatem und Sprachen,* Bibliographisches Institut, Zurich.

Buchi, J. R. [1962]. “Turing-Machines and the Entscheidungsproblem,” *Mathematische Annalen* 148, 201-213.

Burge, W. H. [1975]. *Recursive Programming Techniques*, Addison-Wesley, Reading, Massachusetts.

Burks, A. W. (ed.) [1970]. *Essays in Cellular Automata*, University of Illinois Press.

Burks, A. W., Warren, D. W., and Wright, J. B. [1954]. “An Analysis of a Logical Machine Using Parenthesis-Free Notation,” *Mathematical Tables and Other Aids to Computation* 8, 55-57.

Cantor, D. C. [1962]. “On the Ambiguity Problem of Backus Systems,” *Journal of the ACM* 9, 477-479.

Carroll, J. and Long, D. [1989]. *Theory of Finite Automata*, Prentice Hall, New Jersey.

Choffrut, C. and Culik II, K. [1983]. “Properties of Finite and Pushdown Transducers,” *SIAM Journal on Computing* 12, 300-315.

Chomsky, N. [1956]. “Three Models for the Description of Language,” *IRE Transactions on Information Theory* 2, 113-124.

Chomsky, N. [1957]. *Syntactic Structures*, The Hague, Netherlands.

Chomsky, N. [1959]. “On Certain Formal Properties of Grammars,” *Information and Control* 2, 137-167.

Chomsky, N. [1962]. “Context-Free Grammars and Pushdown Storage,” *Quarterly Progress Report* No. 65, MIT Research Laboratory of Electronics, Cambridge, Massachusetts, 187-194.

Chomsky, N. [1963]. “Formal Properties of Grammars,” *Handbook of Mathematical Psychology*, Vol. 2, John Wiley & Sons, New York, 323-418.

Chomsky, N. and Miller, G. A. [1958]. “Finite-State Languages,” *Information and Control* l, 91-112.

Chomsky, N. and Schutzenberger, M. P. [1963]. “The Algebraic Theory of Context Free Languages,” in Braffort, P. and Hirschberg, D.(eds.), *Computer Programming and Formal Systems*, North-Holland, Amsterdam, 118-161.

Christofides, N. [1976]. *Worst-Case Analysis of a New Heuristic for the Traveling Salesman Problem*, Technical Report, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, Pennsylvania.

Church, A. [1936]. “An Unsolvable Problem of Elementary Number Theory,” *American Journal of Mathematics* 58, 345-363.

Church, A. [1941]. “The Calculi of Lambda-Conversion,” *Annals of Mathematics Studies* 6, Princeton University Press, Princeton, New Jersey.

Cleaveland, J. C. and Uzgalis, R. [1977]. *Grammars for Programming Languages*, Elsevier North-Holland, New York.

Clocksin, W. F. and Mullish, C. S. [1981]. *Programming in PROLOG*, Springer-Verlag, Heidelberg.

Cobham, A. [1964]. “The Intrinsic Computational Difficulty of Functions,” Proceedings 1964 *Congress for Logic, Mathematics, and Philosophy of Science*, North-Holland, Amsterdam, 24-30.

Cohen, D. J. and Gotlieb, C. C. [1970]. “A List Structure Form of Grammars for Syntactic Analysis,” *Computing Surveys* 2.

Cohen, J. [1979]. “Nondeterministic Algorithms,” *Computing Surveys* 11, 79-94.

Comer, D. [1979]. “Heuristic for Trie Index Minimization,” *ACM Transactions on Data Base Systems* 4, 383-395.

Conway, M. E. [1963]. “Design of a Separable Transition-Diagram Compiler,” *Communications of the ACM* 6, 396-408.

Cook, S. A. [1971a]. “The Complexity of Theorem-Proving Procedures,” *Proceedings Third Annual ACM Symposium on the Theory of Computing*, 151-158.

Cook, S. A. [1971b]. “Linear-Time Simulation of Deterministic Two-Way Pushdown Automata,” *Proceeding of the 1971 IFIP Congress*, North-Holland, Amsterdam, 75-80.

Cooper, K. D. [2004]. *Engineering a Compiler*, Morgan Kaufmann, London.

Csuhaj-Varju, E. [1992]. “On Grammars with Local and Global Context Conditions,” *Int. Journal of Computer Mathematics* 47, 17-27.

Dantzig, G. B. [1960]. “On the Significance of Solving Linear Programming Problems with Integer Variables,” *Econometrica* 28, 30-44.

Dassow, J. and Paun, Gh. [1989]. *Regulated Rewriting in Formal Language Theory,* Springer, Berlin.

Dassow, J., Paun, Gh., and Salomaa, A. [1993]. “Grammars Based on Patterns,” *Int. Journal of Foundations of Computer Science* 4(1), 41640.

Davis, M. (ed.) [1965]. *The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems, and Computable Functions,* Raven Press, Hewlett, New York.

Davis, M. [1958]. *Computability and Unsolvability*, McGraw-Hill, New York.

Davis, M. [1973]. “Hilbert's Tenth Problem is Unsolvable,” *American Mathematical Monthly* 80, 233-269.

Davis, M. D. and Weyuker, E. J. [1983]. *Computability, Complexity, and Languages*, Academic Press, New York.

Dawes, J., Pickett, M. J., and Wearing, A. [1990]. *Selecting an Ada Compilation System*, Cambridge University Press, Cambridge.

de Bakker, J. W. [1969]. “Semantics of Programming Languages,” in Tou, J.(ed.), *Advances in Information Systems and Sciences*, Vol. 2, Plenum Press, New York, 173-227.

Dekker, J. C. E. (ed.) [1962]. “Recursive Function Theory,” *Proceedings of Symposia in Pure Mathematics* 5, American Mathematical Society, Providence, Rhode Island.

DeMillo, R. A., Dobkin, D. P., Jones, A. K., and Lipton, R. J.(eds.) [1978]. *Foundations of Secure Computation*, Academic Press, New York.

DeMillo, R. A., Lipton, R. J., and Perlis, A. J. [1979]. “Social Processes and Proofs of Theorems and Programs,” *Communications of the ACM* 22, 271-280.

Denning, P. J., Dennis, J. B., and Qualitz, J. E. [1978]. *Machines, Languages, and Computation*, Prentice Hall, Englewood Cliffs, New Jersey.

Dewdney, A., K. [August 1984]. "Computer Recreations: A Computer Trap for the Busy Beaver, the Hardest-Working Turing Machine," *Scientific American* 251, 19-23.

Dijkstra, E. W. [1976]. *A Discipline of Programming*, Prentice Hall, Englewood Cliffs, New Jersey.

Edmonds, J. [1962]. "Covers and Packings in a Family of Sets," *Bulletin of the American Mathematical Society* 68, 494-499.

Edmonds, J. [1965]. "Paths, Trees and Flowers," *Canadian Journal of Mathematics* 17, 499-467.

Ehrenfeucht, A., Karhumaki, J., and Rozenberg, G. [1982]. "The (Generalized) Post Correspondence Problem with Lists Consisting of Two Words is Decidable," *Theoretical Computer Science* 21, 119-144.

Ehrenfeucht, A., Kleijn, J., and Rozenberg, G. [1985]. "Adding Global Forbidding Context to Context-Free Grammars," *Theoretical Computer Science* 37, 337-360.

Ehrenfeucht, A., Parikh, R., and Rozenberg, G. [1981]. "Pumping Lemmas for Regular Sets," *SIAM Journal on Computing* 10, 536-541.

Ehrenfeucht, A., Pas ten, P., and Rozenberg, G. [1994]. "Context-Free Text Grammars," *Acta Informatica* 31, 161-206.

Eilenberg, S. [1974]. *Automata, Languages, and Machines, Volume A*, Academic Press, New York.

Eilenberg, S. [1976]. *Automata, Languages, and Machines, Volume B*, Academic Press, New York.

Elder, J. [1994]. *Compiler Construction: a Recursive Descent Model*, Prentice Hall, London.

Elgot, C. C. and Mezei, J. E. [1965]. "On Relations Defined by Generalized Finite Automata," *IBM Journal of Research and Development* 9, 47-68.

Elspas, B., Levitt, K., Waldinger, R., and Waksman, A. [1972]. "An Assessment of Techniques for Proving Program Correctness," *Computing Surveys* 4, 97-147.

Engelfriet, J. [1980]. "Some Open Questions and Recent Results on Tree Transducers and Tree Languages," in Book, R.V. (ed.), *Formal Language Theory: Perspectives and Open Problems*, Academic Press, New York, 241-286.

Engelfriet, J., Schmidt, E. M., and van Leeuwen, J. [1980]. "Stack Machines and Classes of Nonnested Macro Languages," *Journal of the ACM* 27, 6-17.

Evey, J. [1963]. "Application of Pushdown Store Machines," *Proceedings 1963 Fall Joint Computer Conference*, AFIPS Press, Montvale, New Jersey, 215-227.

Fischer, C. N. [1991]. *Crafting a Compiler with C*, Benjamin/Cummings Pub. Co., Redwood City.

Fischer, C. [1999]. *Crafting a Compiler Featuring Java*, Addison-Wesley, Harlow.

Fischer, M. J. [1968]. “Grammars with Macro-like Productions,” *Proceedings of the Ninth Annual IEEE Symposium on Switching and Automata Theory,* 131-142.

Fisher, J. A. [2005]. *Embedded Computing: a VLIW Approach to Architecture, Compilers and Tools*, Elsevier, London.

Floyd, R. W. [1962]. “On Ambiguity in Phrase Structure Languages,” *Communications of the ACM* 5, 526-534.

Floyd, R. W. [1964a]. *New Proofs and Old Theorems in Logic and Formal Linguistics,* Computer Associates, Wakefield, Massachusetts.

Floyd, R. W. [1964]. “The Syntax of Programming Languages,” A Survey, *IEEE Transactions on Electronic Computers*, Vol. EC-13, 346-353. Reprinted in Rosen, S.(ed.), *Programming Systems and Languages*, McGraw-Hill, New York, 1967; and Pollack, B.W., *Compiler Techniques,* Auerbach Press, Philadelphia, Pennsylvania, 1972.

Floyd, R. W. [1967a]. “Assigning Meaning to Programs,” in Schwartz, J. T.(ed.), *Mathematical Aspects of Computer Science,* American Mathematical Society, Providence, Rhode Island, 19-32.

Floyd, R. W. [1967b]. “Nondeterministic Algorithms,” *Journal of the ACM* 14, 636-644.

Floyd, R. W. and Ullman, J. D. [1984]. “The Compilation of Regular Expressions into Integrated Circuits,” *Journal of the ACM* 29, 603-622.

Fosdick, L. D. and Osterweil, L. J. [1976]. “Data Flow Analysis in Software Reliability,” *Computing Surveys* 8, 305-330.

Foster, J. M. [1968]. “A Syntax-Improving Program,” *Computer Journal* 11, 31-34.

Foster, J. M. [1970]. *Automatic Syntactic Analysis,* American Elsevier, New York.

Fraser, C. W. [1995]. *A Retargetable C Compiler: Design and Implementation,* Addison-Wesley, Wokingham.

Fris, I. [1968]. “Grammars with Partial Ordering of the Rules,” *Information and Control* 12, 415-425.

Galler, B. A. and Perlis, A. J. [1970]. *A View of Programming Languages*, Addison-Wesley, Reading, Massachusetts.

Gardner, M. [1983]. Wheels, *Life and Other Mathematical Amusements*, W.H. Freeman, San Francisco.

Gardner, M. [April 1985]. “The Traveling Saleman's Travail,” *Discover* 6, 87-90.

Garey, M. R. and Johnson, D. S. [1979]. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, San Francisco.

Gesceg, F. and Steinby, M. [1984]. *Tree Automata*, Akademia Kiado, Budapest.

Ginsburg, S. [1966]. *The Mathematical Theory of Context-Free Languages*, McGraw-Hill, New York.

Ginsburg, S. and Greibach, S. A. [1966]. “Deterministic Context-Free Languages,” *Information and Control* 9, 563-582.

Ginsburg, S. and Greibach, S. A. [1969]. “Abstract Families of Languages,” in *Ginsburg, Greibach, and Hopcroft*, 1-32.

Ginsburg, S., Greibach, S. A., and Hopcroft, J. E. [1969]. “Studies in Abstract Families of Languages,” *Memoirs of the American Mathematical Society* 87, Providence, Rhode Island.

Ginsburg, S. and Rice, H. G. [1962]. “Two Families of Languages Related to ALGOL,” *Journal of the ACM* 9, 350-371.

Ginsburg, S. and Rose, G. F. [1963b]. “Operations Which Preserve Definability in Languages,” *Journal of the ACM* 10, 175-195.

Ginsburg, S. and Spanier, E. H. [1963]. “Quotients of Context-Free Languages,” *Journal of the ACM* 10, 487-492.

Goldstine, J. [1980]. “A Simplified Proof of Parikh's Theorem,” unpublished manuscript.

Gonnet, G. H. [1984]. *Handbook of Algorithms and Data Structures*, Addison-Wesley, Reading, Massachusetts.

Gonnet, G. H. and Tompa, F. W. [1983]. “A Constructive Approach to the Design of Algorithms and Data Structures,” *Communications of the ACM* 26, 912-920.

Gouda, M. G. and Rosier, L. E. [1985]. “Priority Networks of Communicating Finite State Machines,” *SIAM Journal on Computing* 14, 569-584.

Gough, B. [2004]. *An Introduction to GCC: for the GNU Compilers gcc and g++*, Network Theory, Bristol.

Graham, R. L. [1978]. “The Combinatorial Mathematics of Scheduling,” *Scientific American* 238, 3, 124-132.

Gray, J. N. and Harison, M. A. [1966]. “The Theory of Sequential Relations,” *Information and Control* 9, 435-468.

Greibach, S. A. [1963]. “The Undecidability of the Ambiguity Problem for Minimal Linear Grammars,” *Information and Control* 6, 117-125.

Greibach, S. A. [1965]. “A New Normal Form Theorem for Context-Free Phrase Structure Grammars,” *Journal of the ACM* 12, 42-52.

Greibach, S. A. [1970]. “Chains of Full AFLs,” *Mathematical Systems Theory* 4, 231-242.

Greibach, S. A. [1972]. “A Generalization of Parikh's Theorem,” *Discrete Mathematics* 2, 347-355.

Greibach, S. A. [1973]. “The Hardest Context-Free Language,” *SIAM Journal on Computing* 2, 304-310.

Greibach, S. and Hopcroft, J. [1969]. “Scattered Context Grammars,” *Journal of Computer and Systems Sciences* 3, 233-247.

Gries, D. [1971]. *Compiler Construction for Digital Computers*, John Wiley & Sons, New York.

Gries, D. [1981]. *The Science of Programming*, Springer-Verlag, New York.

Grune, D. and Jacobs, C. [1988]. “A Programmer-friendly LL(1) Parser Generator,” *Software - Practice and Experience* 18, 29-38.

Grune, D., Bal, H., Jacobs, C., and Langendoen, K. [2000]. *Modern Compiler Design*, John Wiley & Sons, New York.

Gruska, J. [1971]. “A Characterization of Context-Free Languages,” *Journal of Computer and System Sciences* 5, 353-364.

Habel, A. [1992]. *Hyperedge Replacement: Grammars and Languages*, LNCS 643, Springer, Berlin.

Haghighat, M. R. [1995]. *Symbolic Analysis for Parallelizing Compilers*, Kluwer Academic, Boston.

Hantler, S. L. and King, J. C. [1976]. “An Introduction to Proving the Correctness of Programs,” *Computing Surveys* 8, 331-353.

Harrison, M. A. [1965]. *Introduction to Switching and Automata Theory*, McGraw-Hill, New York.

Harrison, M. A. [1978]. *Introduction to Formal Language Theory*, Addison-Wesley, Reading, Massachusetts.

Harrison, M. A., Ruzzo, W. L., and Ullman, J. D. [1976]. “Protection in Operating Systems,” *Communications of the ACM* 19, 461-471.

Hartmanis, J. [1967]. “Context-Free Languages and Turing Machine Computations,” in Schwartz, J.T.(ed.), *Mathematical Aspects of Computer Science*, American Mathematical Society, Providence, Rhode Island, 42-51.

Hartmanis, J. and Hopcroft, J. E. [1971]. “An Overview of the Theory of Computational Complexity,” *Journal of the ACM* 18, 444-475.

Hartmanis, J. and Stearns, R. E. [1965]. “On the Computational Complexity of Algorithms,” *Transactions of the AMS* 117, 285-306.

Hartmanis, J., Lewis, P. M., II, and Stearns, R. E. [1965]. “Hierarchies of Memory Limited Computations,” *Proceedings of the Sixth Annual Symposium on Switching Circuit Theory and Logical Design*, 179-190.

Hayes, B. [December 1983]. “Computer Recreations,” *Scientific American* 249, 19-28.

Hayes, B. [March 1984]. “Computer Recreations,” *Scientific American* 250, 10-16.

Hays, D. G. [1967]. *Introduction to Computational Linguistics*, American Elsevier, New York.

Hein, J. L. [1995]. *Discrete Structures, Logic, and Computability*, Jones and Bartlett, London.

Heman, G. T. [1973]. “A Biologically Motivated Extension of ALGOL-like Languages,” *Information and Control* 22, 487-502.

Henderson, P. [1980]. *Functional Programming: Application and Implementation*, Prentice Hall, Englewood Cliffs, New Jersey.

Hendrix, J. E. [1990]. *A Small C Compiler*, Prentice Hall International, London.

Hennie, F. C. [1977]. *Introduction to Computability*, Addison-Wesley, Reading, Massachusetts.

Hennie, F. C. and Stearns, R. E. [1966]. “Two-Tape Simulation of Multitape Turing Machines,” *Journal of the ACM* 13, 533-546.

Herman, G. T. and Rozenberger, G. [1975]. *Developmental Systems and Languages*, American Elsevier, New York.

Hermes, H. [1969]. *Enumerability, Decidability, Computability*, Springer-Verlag, New York.

Hoare, C. A. R. and Allison, D. C. S. [1972]. “Incomputability,” *Computing Surveys* 4, 169-178.

Hoare, C. A. R. and Lauer, P. [1974]. “Consistent and Complementary Formal Theories of the Semantics of Programming Languages,” *Acta Informatica* 3, 135-153.

Hoare, C. A. R. and Wirth, N. [1973]. “An Axiomatic Definition of the Programming Language PASCAL,” *Acta Informatica* 2, 335-355.

Holmes, J. [1995]. *Building Your Own Compiler with C++*, Prentice Hall International, London.

Holub, A. I. [1990]. *Compiler Design in C*, Prentice Hall International, London.

Hopcroft, J. E. [1971]. “An n log n Algorithm for Minimizing the States in a Finite Automaton,” in Kohavi, Z. and Paz, A. (eds.), *Theory of Machines and Computations*, Academic Press, New York, 189-196.

Hopcroft, J. E. and Ullman, J. D. [1979]. *Introduction to Automata Theory, Languages, and Computation*, Second Edition, Addison-Wesley, Reading, Massachusetts.

Horowitz, E. and Sahni, S. [1978]. *Fundamentals of Computer Algorithms*, Computer Science Press, Potomac, Maryland.

Hunter, R. [1999]. *The Essence of Compilers*, Prentice Hall, London.

Irons, E. T [1961]. “A Syntax Directed Compiler for ALGOL 60,” *Communications of the ACM* 4, 51-55.

Ito, M. (ed.) [1992]. *Words, Languages, and Combinatorics*, World Scientific, Singapore.

Johnson, J. H. [1983]. *Formal Models for String Similarity,* Ph.D. Dissertation, Department of Computer Science, University of Waterloo.

Johnson, S. C. [1974]. “YACC—Yet Another Compiler Compiler,” *Computer Science Technical Report* 32, Bell Laboratories, Murray Hill, New Jersey.

Johnson, W. L., Porter, J. H., Ackley, S. I., and Ross, D. T. [1968]. “Automatic Generation of Efficient Lexical Analyzers Using Finite-State Techniques,” *Communications of the ACM* 11, 805-813.

Kaplan, R. M. [1955]. *Constructing Language Processors for Little Languages*, John Wiley & Sons, New York.

Kasami, T. [1965]. “An Efficient Recognition and Syntax Algorithm for Context-Free Languages,” *Scientific Report* AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, Massachusetts.

Kelemen, J. [1984]. “Conditional Grammars: Motivations, Definition, and some Properties,” in: *Proc. Automata, Languages and Math. Systems* (Peak, I. and Szep, J., Eds.), K. Marx University of Economics, Budapest, 110-123.

Kelemen, J. [1989]. “Measuring Cognitive Resources Use (a Grammatical Approach),” *Computers and Artificial Intelligence* 8(1), 29-42.

Kelley, D. [1995]. *Automata and Formal Languages*, Prentice Hall, Englewood Cliffs.

Kernighan, B. W. and Plauger, P. J. [1976]. *Software Tools*, Addison-Wesley, Reading, Massachusetts.

Kfoury, A. J., Moll, R. N., and Arbib, M. A. [1982]. *A Programming Approach to Computability*, Springer-Verlag, New York.

Kiong, D. B. K. [1997]. *Compiler Technology: Tools, Translators, and Language Implementation*, Kluwer Academic Publishers, London.

Kleene, S. C. [1936]. “General Recursive Functions of Natural Numbers,” *Mathematische Annalen* 112, 727-742.

Kleene, S. C. [1943]. “Recursive Predicates and Quantifiers,” *Transactions of the American Mathematical Society* 53, 41-73.

Kleene, S. C. [1952]. *Introduction to Metamathematics*, D. Van Nostrand, Princeton, New Jersey.

Kleene, S. C. [1956]. “Representation of Events in Nerve Nets and Finite Automata,” in Shannon, C. E. and McCarthy, J. (eds.), *Automata Studies*. Princeton University Press, Princeton, New Jersey, 15401.

Kleijn, H. C. M. and Rozenberg, G. [1981]. “Context-Free-Like Restrictions on Selective Rewriting,” *Theoretical Computer Science* 16, 237-239.

Knoop, J. [1998]. *Optimal Interprocedural Program Optimization: a New Framework and Its Application*, Springer, London.

Knuth, D. E. [1967a]. “On the Translation of Languages from Left to Right,” *Information and Control* 8, 611-618.

Knuth, D. E. [1967b]. “The Remaining Trouble Spots in ALGOL 60,” *Communications of the ACM* 10, 611-618.

Knuth, D. E. [1971]. “Top-Down Syntax Analysis,” *Acta Informatica* 1, 79-110.

Knuth, D. E. [1973]. *The Art of Computer Programming, Vol.3: Sorting and Searching*, Addison-Wesley Publishing Co., Reading, Massachusetts.

Knuth, D. E., Morris, J. H., Jr., and Pratt, V. R. [1977]. “Fast Pattern Matching in Strings,” *SIAM Journal on Computing* 6, 323-350.

Korenjak, A. J. and Hopcroft, J. E. [1966]. “Simple Deterministic Languages,” *Proceedings of the Seventh Annual IEEE Symposium on Switching and Automata Theory*, 36-46.

Kral, J. [1973]. “A Note on Grammars with Regular Restrictions,” *Kybernetika* 9(3), 159-161.

Kuich, W. and Salomaa, A. [1985]. *Semirings, Automata, Languages*, Springer-Verlag, New York.

Kurki-Suonio, R. [1964]. “Note on Top-Down Languages,” *Information and Control* 7, 207-223.

Kuroda, S. Y. [1969]. “Classes of Languages and Linear Bounded Automata,” *BIT* 9, 225-238.

Landin, P. J. [1965]. “A Correspondence between Algol 60 and Church's Lambda-Notation,” *Communications of the ACM* 8, 89-101 and 158-165.

Larson, L. C. [1983]. *Problem-Solving through Problems*, Springer-Verlag, New York.

Lauer, P. E., Torrigiani, P. R., and Shields, M. W. [1979]. “COSY: A System Specification Language Based on Paths and Processes,” *Acta Informatica* 12, 109-158.

Lee, G. and Yew, P. [2001]. *Interaction between Compilers and Computer Architectures*, Kluwer Academic Publishers, London.

Lemone, K. A. [1992a]. *Fundamentals of Compilers: an Introduction to Computer Language Translation*, CRC Press, Boca Raton.

Lemone, K. A. [1992b]. *Design of Compilers: Techniques of Programming Language Translation*, CRC Press, Boca Raton.

Lewis, H. R. and Papadimitriou, C. [1981]. *Elements of the Theory of Computation*, Prentice Hall, Englewood Cliffs, New Jersey.

Leupers, L. and Marwedel, P. [2001]. *Retargetable Compiler Technology for Embedded Systems: Tools and Applications*, Kluwer, London.

Levine, J. R. [1992]. *Lex & yacc*, O'Reilly & Associates, Sebastopol.

Lewis, P. M., II and Stearns, R. E. [1968]. “Syntax-Directed Transduction,” *Journal of the ACM* 15, 465-488.

Lewis, P. M., II, Rosenkrantz, D. J., and Stearns, R. E. [1976]. *Compiler Design Theory*, Addison-Wesley, Reading, Massachusetts.

Lewis, P. M., II, Stearns, R. E., and Hartmanis, J. [1965]. “Memory Bounds for Recognition of Context-Free and Context-Sensitive Languages,” *Proceedings of the Sixth Annual IEEE Symposium on Switching Circuit Theory and Logical Design*, 191-202.

Lindenmayer, A. [1971]. “Mathematical Models for Cellular Interactions in Development,” Parts I and II, *Journal of Theoretical Biology* 30, 455-484.

Linger, R. C., Mills, H. D., and Witt, B. I. [1979]. *Structured Programming: Theory and Practice*, Addison-Wesley, Reading, Massachusetts.

Linz, P. [1990]. *An Introduction to Formal Languages and Automata*, D.C. Heath and Co., Lexington, Massachusetts.

Louden, K. C. [1997]. *Compiler Construction: Principles and Practice*, PWS Publishing, London.

Ito, M. (ed.) [1992]. *Words, Languages, and Combinatorics*, World Scientific, Singapore.

Mak, R. [1996]. *Writing Compilers and Interpreters*, John Wiley & Sons, New York.

Mallozi, J. S. and De Lillo, N. J. [1984]. *Computability with PASCAL*, Prentice Hall, Englewood Cliffs, New Jersey.

Marcotty, M., Ledgard, H.F., and Bochmann, G. V. [1976]. “A Sampler of Formal Definitions,” *Computing Surveys* 8, 191-276.

Markov, A. A. [1960]. *The Theory of Algorithms*, (translated from the Russian by J.J. Schorrkon), U.S. Dept. of Commerce, Office of Technical Services, No. OTS 60-5108.

Martin, J. C. [1991]. *Introduction to Languages and the Theory of Computation*, McGraw-Hill, New York.

Maurer, D. and Wilhelm, R. [1995]. *Compiler Design*, Addison-Wesley, Reading, Massachusetts.

McCarthy, J. [1960]. “Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I,” *Communications of the ACM* 3, 184-195.

McCarthy, J. [1963]. “A Basis for a Mathematical Theory of Computation,” in Braffort, P. and Hirschberg, D. (eds.), *Programming and Formal Systems*, North-Holland, Amsterdam, 33-70.

McCarthy, J. and Painter, J. [1967]. “Correctness of a Compiler for Arithmetic Expressions,” in Schwartz, J. T. (ed.), *Mathematical Aspects of Computer Science*, American Mathematical Society, Providence, Rhode Island, 33-41.

McCulloch, W. S. and Pitts, W. [1943]. “A Logical Calculus of the Ideas Immanent in Nervous Activity,” *Bulletin of Mathematical Biophysics* 5, 115-133.

McGuire, T. M. [2005]. *The Austin Protocol Compiler*, Springer, New York.

McNaughton, R. [1982]. *Elementary Computability, Formal Languages, and Automata*, Prentice Hall, Englewood Cliffs, New Jersey.

McNaughton, R. and Papert, S. [1971]. *Counter-Free Automata*, The M.I.T. Press, Cambridge, Massachusetts.

McNaughton, R. and Yamada, H. [1960]. "Regular Expressions and State Graphs for Automata," *IEEE Transactions on Electronic Computers* 9, 39-47.

McWhirter, I. P. [1971]. "Substitution Expressions," *Journal of Computer and System Sciences* 5, 629-637.

Mead, C. A. and Conway, L. A. [1980]. *Introduction to VLSI Systems*, Addison-Wesley, Reading, Massachusetts.

Meduna, A. [1986]. "A Note on Exponential Density of ETOL Languages," *Kybernetika* 22, 514-518.

Meduna, A. [1987a]. "Characterization of the Chomsky Hierarchy through Sequential-Parallel Grammars," *Rostock. Math. Kolloq*. 32, 4-14.

Meduna, A. [1987b]. "Evaluated Grammars," *Acta Cybernetika* 8, 169-176.

Meduna, A. [1990a]. "Context Free Derivations on Word Monoids," *Acta Informatica* 27, 781-786.

Meduna, A. [1990b]. "Generalized Forbidding Grammars," *International Journal of Computer Mathematics* 36, 31-38.

Meduna, A. [1991]. "Global Context Conditional Grammars," *J. Inform. Process. Cybern*. 27, 159-165.

Meduna, A. [1992]. "Symbiotic E0L Systems," *Acta Cybernetica* 12, 164-172.

Meduna, A. [1993a]. "A Formalization of Sequential, Parallel, and Continuous Rewriting," *International Journal of Computer Mathematics* 39, 24-32.

Meduna, A. [1993b]. "Canonical Scattered Rewriting," *International Journal of Computer Mathematics* 51, 122-129.

Meduna, A. [1994]. "Matrix Grammars under Leftmost and Rightmost Restrictions," in vol. *Mathematical Linguistics and Related Topics* (Gh. Paun, ed.), The Publ. House of the Romanian Academy, Bucharest, 243-257.

Meduna, A. [1995a]. "A Trivial Method of Characterizing the Family of Recursively Enumerable Languages by Scattered Context Grammars," *EATCS Bulletin* 56, 104-106.

Meduna, A. [1995b]. "Syntactic Complexity of Scattered Context Grammars," *Acta Informatica* 32, 285-298.

Meduna, A. [1996]. "Syntactic Complexity of Context-Free Grammars over Word Monoids," *Acta Informatica* 33, 457-462.

Meduna, A. [1997a]. “Four-Nonterminal Scattered Context Grammars Characterize the Family of Recursively Enumerable Languages,” *International Journal of Computer Mathematics* 63, 67-83.

Meduna, A. [1997b]. “On the Number of Nonterminals in Matrix Grammars with Leftmost Derivations,” *LNCS* 1217, 27-38.

Meduna, A. [1997c]. “Six-Nonterminal Multi-Sequential Grammars Characterize the Family of Recursively Enumerable Languages,” *International Journal of Computer Mathematics* 65, 179-189.

Meduna, A. [1998a]. “Descriptional Complexity of Multi-Continues Grammars,” *Acta Cybernetica* 13, 375-384.

Meduna, A. [1998b]. “Economical Transformation of Phrase-Structure Grammars to Scattered Context Grammars,” *Acta Cybernetica* 13, 225-242.

Meduna, A. [1998c]. “Uniform Rewriting Based on Permutations,” *International Journal of Computer Mathematics* 69, 57-74.

Meduna, A. [1999a]. “Prefix Pushdown Automata,” *International Journal of Computer Mathematics* 71, 215-228.

Meduna, A. [1999b]. “Terminating Left-Hand Sides of Scattered Context Productions,” *Theoretical Computer Science* 237, 567-601.

Meduna, A. [2000a]. “Generative Power of Three-Nonterminal Scattered Context Grammars,” *Theoretical Computer Science* 246, 276-284.

Meduna, A. [2000b]. “Generative Power of Three-Nonterminal Scattered Context Grammars,” *Theoretical Computer Science*, 625-631.

Meduna, A. [2000c]. “Terminating Left-Hand Sides of Scattered Context Grammars,” *Theoretical Computer Science*, 423-427.

Meduna, A. [2000d]. *Automata and Languages: Theory and Applications*, Springer, London.

Meduna, A. [2001]. “Uniform Generation of Languages by Scattered Context Grammars,” *Fundamenta Informaticae* 44, 231-235.

Meduna, A. [2002]. “Descriptional Complexity of Scattered Rewriting and Multirewriting: An Overview,” *Journal of Automata, Languages and Combinatorics* 7, 571-577.

Meduna, A. [2003a]. “Coincidental Extention of Scattered Context Languages,” *Acta Informatica* 39, 307-314.

Meduna, A. [2003b]. “Simultaneously One-Turn Two-Pushdown Automata,” *International Journal of Computer Mathematics* 80, 679-687.

Meduna, A. [2004]. “Two-Way Metalinear PC Grammar Systems and Their Descriptional Complexity,” *Acta Cybernetica* 92, 126-137.

Meduna, A. and Csuhaj-Varju, E. [1993]. “Grammars with Context Conditions,” *EATCS Bulletin* 32, 112-124.

Meduna, A. and Fernau, H. [2003a]. “A Simultaneous Reduction of Several Measures of Descriptional Complexity in Scattered Context Grammars,” *Information Processing Letters* 86, 235-240.

Meduna, A. and Fernau, H. [2003b]. “On the Degree of Scattered Context-Sensitivity,” *Theoretical Computer Science* 290, 2121-2124.

Meduna, A. and Gopalaratnam, M. [1994]. “On Semi-Conditional Grammars with Productions Having either Forbidding or Permitting Conditions,” *Acta Cybernetica* 11, 309-323.

Meduna, A. and Horvath, G. [1988]. “On State Grammars,” *Acta Cybernetica* 8, 237-245.

Meduna, A. and Kolář, D. [2000a]. “Descriptional complexity of multi-parallel grammars with respect to the number of nonterminals,” *Grammars and Automata for String Processing from Mathematics and Computer Science to Biology, and Back*, Francis and Taylor, 724-732.

Meduna, A. and Kolář, D. [2000b]. “Regulated Pushdown Automata,” *Acta Cybernetica* 18, 653-664.

Meduna, A. and Kolář, D. [2002a]. “Homogenous Grammars with a Reduced Number of Non-Context-Free Productions,” *Information Processing Letters* 81, 253-257.

Meduna, A. and Kolář, D. [2002b]. “One-Turn Regulated Pushdown Automata and Their Reduction,” *Fundamenta Informaticae*, Vol. 2002, No. 16, Amsterdam, NL, p. 399-405.

Meduna, A. and Švec, M. [2002]. “Reduction of Simple Semi-Conditional Grammars with Respect to the Number of Conditional Productions,” *Acta Cybernetica* 15, 353-360.

Meduna, A. and Švec, M. [2003a]. “Descriptional Complexity of Generalized Forbidding Grammars,” *International Journal of Computer Mathematics* 80, 11-17.

Meduna, A. and Švec, M. [2003b]. “Forbidding E0L Systems,” *Theoretical Computer Science* 54, 256-276.

Meduna, A. and Švec, M. [2005]. *Grammars with Context Conditions and Their Applications*, Wiley, Hoboken, New Jersey.

Meduna, A. and Vurm, P. [2001]. “Multisequential Grammars with Homogeneous Selectors,” *Fundamenta Informaticae* 34, 1-7.

Meduna, A., Crooks, C., and Sarek, M. [1994]. “Syntactic Complexity of Regulated Rewriting,” *Kybernetika* 30, 177-186.

Meyer, A. R. and Stockmeyer, L. J. [1972]. “The Equivalence Problem for Regular Expressions with Squaring Requires Exponential Time,” *Proceedings of the Thirteenth Annual IEEE Symposium on Switching and Automata Theory,* 125-129.

Minsky, M. L. [1962]. “Size and Structure of Universal Turing Machines Using Tag Systems,” Marcel Dekker, 229-238.

Minsky, M. L. [1967]. *Computation: Finite and Infinite Machines*, Prentice Hall, Englewood Cliffs, New Jersey.

Minsky, M. L. [August 1960]. “A 6-Symbol, 7-State Universal Turing Machine,” *MIT Laboratory Group Report* 54G-OO27.

Moore, E. F. [1956]. “Gedanken Experiments on Sequential Machines,” in Shannon, C. E. and McCarthy, J. (eds.), *Automata Studies*. Princeton University Press, Princeton, New Jersey, 129-153.

Moore, G. B., Kuhns, J. L., Trefftzs, J. L., and Montgomery, C. A. [1977]. “Accessing Individual Records from Personal Data Files Using Non-Unique Identifiers”. *NBS Special Publication* 500-2, U.S. Department of Commerce, National Bureau of Standards.

Morgan, R. C. [1998]. *Building an Optimizing Compiler*, Butterworth-Heinemann, Oxford.

Muchnick, S. S. [1997]. *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers, London.

Myhill, J. [1957]. “Finite Automata and the Representation of Events,” *WADD* TR-57-624, Wright Patterson AFB, Ohio, 112-137.

Naur, P. (ed.) [1960]. “Report on the Algorithmic Language ALGOL 60,” *Communications of the ACM* 3, 299-314, revised in *Communications of the ACM* 6, 1963, 1-17.

Navratil, E. [1970]. “Context-Free Grammars with Regular Conditions,” *Kybernetika* 6(2), 118-125.

Newman, W. and Sproul, R. [1979]. *Principles of Interactive Computer Graphics*, Second Edition, McGraw-Hill, New York.

Oettinger, A. G. [1961]. “Automatic Syntactic Analysis and Pushdown Store,” *Proceedings of the Symposia in Applied Mathematics* 12, American Mathematical Society, Providence, Rhode Island, 104-109.

Ogden, W. [1968]. “A Helpful Result for Proving Inherent Ambiguity,” *Mathematical Systems Theory* 2, 191-194.

Pagan, F. G. [1981]. *Formal Specification of Programming Languages L: A Panoramic Primer*, Prentice Hall, Englewood Cliffs, New Jersey.

Pansiot, J. J. [1981]. “A Note on Post's Correspondence Problem,” *Information Processing Letters* 12, 233.

Papadimitriou, C. H. and Steiglitz, K. [1982]. *Combinatorial Optimatization: Algorithms and Complexity*, Prentice Hall, Englewood Cliffs, New Jersey.

Parikh, R. J. [1966]. “On Context-Free Languages,” *Journal of the ACM* 13, 570-581.

Parsons, T. W. [1992]. *Introduction to Compiler Construction*, Computer Science, Oxford.

Paun, Gh. [1979]. “On the Genarative Capacity of Conditional Grammars,” *Information and Control* 43, 178-186.

Paun, Gh. [1985]. “A Variant of Random Context Grammars: Semi-Conditional Grammars,” *Theoretical Computer Science* 41, 42736.

Paun, Gh. (ed.) [1995]. *Artificial Life: Grammatical Models,* Black Sea University Press, Bucharest, Romania.

Paun, Gh. (ed.) [1995]. *Mathematical Linguistics and Related Topics*, The Publ. House of the Romanian Academy, Bucharest, Romania.

Pavlenko, V. A. [1981]. “Post Combinatorial Problem with Two Pairs of Words,” *Dokladi AN Ukr. SSR* 33, 9-11.

Pilling, D. L. [1973]. “Commutative Regular Equations and Parikh's Theorem,” *Journal of the London Mathematical Society* II, 6, 663-666.

Pippenger, N. [1978]. “Complexity Theory,” *Scientific American* 238, 6, 114-124.

Pittman, T. [1992]. *The Art of Compiler Design: Theory and Practice*, Prentice Hall, New York.

Post, E. L. [1936]. “Finite Combinatory Processes-Formulation I,” *Journal of Symbolic Logic* 1, 103-105.

Post, E. L. [1947]. “Recursive Unsolvability of a Problem of Thue,” *Journal of Symbolic Logic* 12, 39022.

Prather, R. E. [1976]. *Discrete Mathematical Structures for Computer Science*, Houghton Mifflin, Boston.

Pratt, T. W. [1982]. “The Formal Analysis of Computer Programs,” in Pollack, S.V. (ed.) *Studies in Mathematics*, The Mathematical Association of America, 169-195.

Priese, L. [1979]. “Towards a Precise Characterization of the Complexity of Universal and Nonuniversal Turing Machines,” *SIAM Journal on Computing* 8, 508-523.

Pugsley, D. [2000]. *Justinian's Digest and the Compilers,* University of Exeter, Faculty of Law, Exeter.

Rabin, M. O. and Scott, D. [1959]. “Finite Automata and Their Decision Problems,” *IBM Journal of Research and Development* 3, 115-125.

Rado, T. [1962]. “On Noncomputable Functions,” *Bell System Technical Journal* 41, 877-884.

Revesz, G. E. [1983]. *Introduction to Formal Language Theory*, McGraw-Hill, New York.

Reynolds, J. C. [1981]. *The Craft of Programming*, Prentice Hall, Englewood Cliffs, New Jersey.

Rice, H. G. [1953]. “Classes of Recursively Enumerable Sets and their Decision Problems,” *Transactions of AMS* 74, 358-366.

Rogers, Jr., H. [1967]. *The Theory of Recursive Functions and Effective Computability*, McGraw-Hill, New York.

Rosenkrantz, D. J. [1967]. “Matrix Equations and Normal Forms for Contex-Free Grammars,” *Journal of the ACM* 14, 501-507.

Rosenkrantz, D. J. [1969]. “Programmed Grammars and Classes of Formal Languages,” *Journal of the ACM* 16, 107-131.

Rosenkrantz, D. J. and Stearns, R. E. [1970]. “Properties of Deterministic Top-Down Grammars,” *Information and Control* 17, 226-256.

Rosenkrantz, D. J., Stearns, R. E., and Lewis, P. M. [1977]. “An Analysis of Several Heuristic for the Travelling Salesman Problem,” *SIAM Journal on Computing* 6, 563-581.

Rozenberg, G. [1973]. “Extension of Tabled 0L Systems and Languages,” *International Journal of Computer and Information Sciences* 2, 311-334.

Rozenberg, G. [1977]. “Selective Substitution Grammars (Towards a Framework for Rewriting Systems), Part I: Definitions and Examples,” *J. Inform. Process. Cybern.* 13, 455-463.

Rozenberg, G. and Salomaa, A. (eds.) [1997]. *Handbook of Formal Languages, Volume 1 through 3*, Springer.

Rozenberg, G. and Salomaa, A. [1980]. *The Mathematical Theory of L Systems*, Academic Press, New York.

Rozenberg, G. and Solms, von S. H. [1978]. “Priorities on Context Conditions in Rewriting Systems,” *Information Sciences* 14, 15-50.

Rustin, R. (ed.) [1972]. *Formal Semantics of Programming Languages*, Prentice Hall, Englewood Cliffs, New Jersey.

Salomaa, A. [1969]. *Theory of Automata*, Pergamon Press, London.

Salomaa, A. [1973]. *Formal Languages*, Academic Press, New York.

Salomaa, A. [1985]. *Computation and Automata*, Cambridge University Press, Cambridge, England.

Sampaio, A. [1997]. *An Algebraic Approach to Compiler Design*, World Scientific, London.

Savitch, W. J. [1970]. “Relationships between Nondeterministic and Deterministic Tape Complexities,” *Journal of Computer and System Sciences* 4, 177-192.

Savitch, W. J. [1982]. *Abstract Machines and Grammars*, Little, Brown, Boston.

Scheinberg, S. [1963]. “Note on the Boolean Properties of Context-Free Languages,” *Information and Control* 6, 246-264.

Schutzenberger, M. P. [1963]. “On Context-Free Languages and Pushdown Automata,” *Information and Control* 6, 246-264.

Scott, D. [1967]. “Some Definitional Suggestions for Automata Theory,” *Journal of Computer and System Sciences* 1, 187-212.

Scowen, R. S. [1983]. “An Introduction and Handbook for the Standard Syntactic Metalanguage,” *National Physical Laboratory Report* DITC 19/83.

Sharp, R. [2004]. *Higher Level Hardware Synthesis,* Springer, London.

Shannon, C. E. [1956]. “A Universal Turing Machine with Two Internal States,” in *Automata Studies*, Princeton University Press, Princeton, New Jersey, 129-153.

Shannon, C. E. and McCarthy, J. (eds.) [1956]. *Automata Studies*, Princeton University Press, Princeton, New Jersey.

Shepherdson, J. C. and Sturgis, H. E. [1963]. “Computability of Recursive Functions,” *Journal of the ACM* 10, 217-255.

Shyr, H. J. [1991]. *Free Monoids and Languages*, Hon Min Book Comp., Taichung.

Sippu, S. and Soisalon-Soininen, E. [1987]. *Parsing Theory,* Springer-Verlag, New York.

Sippu, S., Soisalon-Soininen, E., and Ukkonen, E. [1983]. “The Complexity of LALR(k) Testing,” *Journal of the ACM* 30, 259-270.

Smith, A. R. [1984]. “Plants, Fractals, and Formal Languages,” *Computer Graphics* 18, 1-10.

Solow, D. [1982]. *How to Read and Do Proofs*, John Wiley & Sons, New York.

Srikant, Y. N. [2003]. *The Compiler Design Handbook: Optimizations and Machine Code Generation*, CRC Press, London.

Stepney, S. [1992]. *High Integrity Compilation: a Case Study*, Prentice Hall, London.

Stepherdson, J. C. [1959]. “The Reduction of Two-Way Automata to One-Way Automata,” *IBM Journal of Research and Development* 3, 198-200.

Stockmeyer, L. J. and Chandra, A. K. [1979]. “Intrinsically Difficult Problems,” *Scientific American* 240, 5, 140-159.

Stone, H.S. [1973]. *Discrete Mathematical Structures and Their Applications*, SRA, Chicago, Illinois.

Sorenson, P. G. and Tremblay, J. P. [1985]. *The Theory and Practice of Compiler Writing*, McGraw-Hill, New York.

Sudkamp, T. A. [1988]. *Languages and Machines,* Addison-Wesley, Reading, Massachusetts.

Tarjan, R. E. [1981]. “A Unified Approach to Path Problems,” *Journal of the ACM* 28, 577-593.

Tennet, R. D. [1981]. *The Denotational Semantics of Programming Languages*, Prentice Hall, Englewood Cliffs, New Jersey.

Thatcher, J. W. [1967]. “Characterizing Derivation Trees of a Context-Free Grammar through a Generalization of Finite-Automata Theory,” *Journal of Computer and System Sciences* 1, 317-322.

Thatcher, J. W. [1973]. “Tree Automata: An Informal Survey,” in Aho, A. V. (ed.), *Currents in the Theory of Computing*, Prentice Hall, Englewood Cliffs, New Jersey, 143-172.

Thompson, K. [1968]. “Regular Expression Search Algorithm,” *Communications of the ACM* 11, 419-422.

Thue, A. [1906]. “Uber unedlische Zeichenreihen,” *Skrifter utgit av Videnskapsselakapet i Kristiania* 1, 1-22.

Thue, A. [1914]. “Probleme uber Veranderungen von Zeichenreihen nach gegebenen Regeln,” *Skrifter utgit av Videnskapsselakapet i Kristiania* 10.

Tseng, P. [1990]. *A Systolic Array Parallelizing Compiler*, Kluwer Academic, London.

Tremblay, R. E. and Manohar, R. P. [1975]. *Discrete Mathematical Structures with Applications to Computer Science*, McGraw-Hill, New York.

Turing, A. M. [1936]. “On Computable Numbers with an Application to the Entscheidungs-Problem,” *Proceedings of the London Mathematical Society* 2, 230-265.

Ullman, J. D. [1984]. *Computational Aspects of VLSI*, Computer Science Press, Rockville, Maryland.

Urbanek, F. J. [1983]. “A Note on Conditional Grammars,” *Rev. Roum. Math. Pures Appl.* 28, 341-342.

Valiant, L. G. [1975]. “General Context-Free Recognition in Less than Cubic Time,” *Journal of Computer and Systems Sciences* 10, 308-315.

van Leewen, J. [1974a]. “A Generalization of Parikh's Theorem in Formal LanguageTheory,” *Proceedings of ICALP* '74, Springer-Verlag Lecture Notes in Computer Scince 14, 17-26.

van Leewen, J. [1974b]. “Notes on Pre-Set Pushdown Automata,” Springer-Verlag Lecture Notes in *Computer Science* 15, 177-188.

van Wijngaarden, A. (ed.) [1969]. “Report on the Algorithmic Language ALGOL 68,” *Numerische Mathematik* 14, 79-218.

van Wijngaarden, A., Mailloux, B. J., Peck, J. E. L., Koster, C. H. A., Sintzoff, M., Lindsey, C. H., Meertens, L. G. L. T., Fisker, R. G. (eds.) [1974]. “Revised Report on the Algorithmic Language ALGOL 68,” *Acta Informatica* 5, 1-236.

Vaszil, G. [2004]. “On the Number of Conditional Rules in Simple Semi-Conditional Grammars,” *Theoretical Computer Science*.

Vere, S. [1970]. “Translation Equations,” *Communications of the ACM* 13, 83-89.

Waite, W. [1993]. *An Introduction to Compiler Construction*, HarperCollins, New York.

Walt, van der A. P. J. [1972]. “Random Context Languages,” *Information Processing* 71, 66-68.

Watanabe, S. [1960]. “On a Minimal Universal Turing Machine,” *MCB Report*, Tokyo.

Watanabe, S. [1961]. “5-Symbol 8-State and 5-Symbol 6-State Universal Turing Machines,” *Journal of the ACM* 8, 476-483.

Wegner, P. [1972]. “Programming Language Semantics,” in Rustin, R.(ed.), *Formal Semantics of Programming Languages*, Prentice Hall, Englewood Cliffs, New Jersey, 149-248.

Wegner, P. [1972]. “Programming Language,” *Computing Surveys* 4, 5-63.

Wilhelm, R. [1995]. *Compiler Design*, Addison-Wesley, Wokingham.

Wirth, N. [1973]. *Systematic Programming: An Introduction*, Prentice Hall, Englewood Cliffs, New Jersey.

Wirth, N. [September 1984]. “Data Structures and Algorithms,” *Scientific American* 251, 60-69.

Wirth, N. [1996]. *Compiler Construction*, Addison-Wesley, Harlow.

Wise, D. S. [1976]. “A Strong Pumping Lemma for Context-Free Languages,” *Theoretical Computer Science* 3, 359-370.

Wolfram, S., Farmer, J. D., and Toffoli, T. (eds.) [1984]. “Cellular Automata,” *Proceedings of an Inter-Disciplinary Workshop*, Physica 10D, Nos. 1 and 2.

Wood, D. [1969]. “The Normal Form Theorem—Another Proof,” *Computer Journal* 12, 139-147.

Wood, D. [1969]. “The Theory of Left-Factored Languages,” *Computer Journal* 12, 349-356 and 13.

Wood, D. [1984]. *Paradigms and Programming with PASCAL*, Computer Science Press, Rockville, Maryland.

Wood, D. [1987]. *Theory of Computation*, Harper and Row, New York.

Yentema, M. K. [1971]. “Cap Expressions for Context-Free Languages,” *Information and Control* 8, 311-318.

Younger, D. H. [1976]. “Recognition and Parsing of Context-Free Languages in Time n 3,” *Information and Control* 10, 189-208.

Zima, H. [1991]. *Supercompilers for Parallel and Vector Computers*, ACM Press, New York.