Elements of Compiler Design

Alexander Meduna

Auerbach Publications
Taylor & Francis Group

Elements of Compiler Design

Other Auerbach Publications in Software Development, Software Engineering, and Project Management

Accelerating Process Improvement Using Agile Techniques
Deb Jacobs
0-8493-3796-8

Antipatterns: Identification, Refactoring, and Management
Phillip A. Laplante and Colin J. Neill
0-8493-2994-9

Business Process Management Systems
James F. Chang
0-8493-2310-X

The Complete Project Management Office Handbook
Gerard M. Hill
0-8493-2173-5

Defining and Deploying Software Processes
F. Alan Goodman
0-8493-9845-2

Embedded Linux System Design and Development
P. Raghavan, Amol Lad, and Sriram Neelakandan
0-8493-4058-6

Global Software Development Handbook
Raghvinder Sangwan, Matthew Bass, Neel Mullick, Daniel J. Paulish, and Juergen Kazmeier
0-8493-9384-1

Implementing the IT Balanced Scorecard
Jessica Keyes
0-8493-2621-4

The Insider's Guide to Outsourcing Risks and Rewards
Johann Rost
0-8493-7017-5

Interpreting the CMMI®
Margaret Kulpa and Kent Johnson
0-8493-1654-5

Modeling Software with Finite State Machines
Ferdinand Wagner, Ruedi Schmuki, Thomas Wagner, and Peter Wolstenholme
0-8493-8086-3

Optimizing Human Capital with a Strategic Project Office
J. Kent Crawford and Jeannette Cabanis-Brewin
0-8493-5410-2

A Practical Guide to Information Systems Strategic Planning, Second Edition
Anita Cassidy
0-8493-5073-5

Process-Based Software Project Management
F. Alan Goodman
0-8493-7304-2

Project Management Maturity Model, Second Edition
J. Kent Crawford
0-8493-7945-8

Real Process Improvement Using the CMMI®
Michael West
0-8493-2109-3

Reducing Risk with Software Process Improvement
Louis Poulin
0-8493-3828-X

The ROI from Software Quality
Khaled El Emam
0-8493-3298-2

Software Engineering Quality Practices
Ronald Kirk Kandt
0-8493-4633-9

Software Sizing, Estimation, and Risk Management
Daniel D. Galorath and Michael W. Evans
0-8493-3593-0

Software Specification and Design: An Engineering Approach
John C. Munson
0-8493-1992-7

Software Testing and Continuous Quality Improvement, Second Edition
William E. Lewis
0-8493-2524-2

Strategic Software Engineering: An Interdisciplinary Approach
Fadi P. Deek, James A.M. McHugh, and Osama M. Eljabiri
0-8493-3939-1

Successful Packaged Software Implementation
Christine B. Tayntor
0-8493-3410-1

UML for Developing Knowledge Management Systems
Anthony J. Rhem
0-8493-2723-7

AUERBACH PUBLICATIONS

www.auerbach-publications.com
To Order Call: 1-800-272-7737 • Fax: 1-800-374-3401
E-mail: orders@crcpress.com

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: 20131031

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

in memory of St. John of the Cross

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

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.

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 $\varnothing$; note that *card*($\varnothing$) = 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, $\Sigma = \{a_1, a_2, \ldots, a_n\}$ 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 $\Sigma \cap \Omega = \varnothing$, Σ and Ω are *disjoint*. More generally, n sets $\Delta_1, \Delta_2, \ldots, \Delta_n$, where $n \geq 2$, are *pairwise disjoint* if $\Delta_i \cap \Delta_j = \varnothing$ 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 $i = 1, \ldots, |x|$. 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 $\varnothing$ and $\{\varepsilon\}$ are finite; noteworthy, $\varnothing \neq \{\varepsilon\}$ because $card(\varnothing) = 0 \neq card(\{\varepsilon\}) = 1$. 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 $a_1a_2\ldots a_n$ instead of $(a_1, a_2, \ldots, a_n)$.

■

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 $L^* = L^0 \cup L^1 \cup L^2 \cup \ldots$, and the *positive closure of L*, L^+, is defined as $L^+ = L^1 \cup L^2 \cup \ldots$. 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 $prefixes(L) = \{x |\ x \in prefixes(y) \text{ for some } y \in L\}$. For $i = 0, \ldots, |y|$, *prefix*(y, i) denotes y's prefix of length i; notice that $prefix(y, 0) = \varepsilon$ 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 $suffixes(L) = \{x |\ x \in suffixes(y) \text{ for some } y \in L\}$. For $i = 0, \ldots, |y|$, *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) = $\{\varepsilon, 1, 10, 101, 1010\}$. Strings 010 and ε are suffixes of 1010. String 010 is a proper suffix of 1010 while ε is not. We have *suffixes*(1010) = $\{\varepsilon, 0, 10, 010, 1010\}$ and *substrings*(1010) = $\{\varepsilon, 0, 1, 01, 10, 010, 101, 1010\}$.

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*$(L) = \{\varepsilon, 1, 0, 01\}$, *suffixes*$(L) = \{\varepsilon, 1, 01\}$, and *substrings*$(L) = \{\varepsilon, 0, 1, 01\}$.

■

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 $(a, b) \in \phi)\} \le 1$. If *domain*$(\phi) = A$, ϕ is *total*; otherwise, ϕ is *partial*. If for every $b \in B$, *card*$(\{a |\ a \in A$ and $(a, b) \in \phi)\} \le 1$, 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 $k \ge 1$, 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 $k \ge 2$. 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 $k \ge 1$, 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 $k \ge 0$.

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 $\tau(a_1 a_2 \ldots a_n) = \tau(a_1)\tau(a_2) \ldots \tau(a_n)$, where $a_i \in T$, $1 \le i \le n$, for some $n \ge 1$, 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 $\Xi = \{0, 1, \ldots, 9\}$ and $\Psi = (\{A, B, \ldots, Z\} \cup \{|\})$ 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, $(a_0, a_1, \ldots, a_n)$, 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 $(a_0, a_1, \ldots, a_n)$ 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.

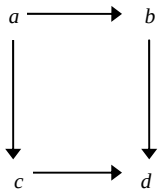

Figure 1.1 *Graph*.

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.

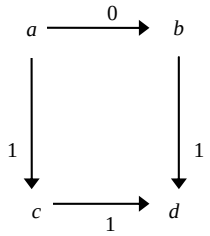

Figure 1.2 ***Labeled Graph*****.**

■

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 $\varnothing \subset B \subseteq A$, $\upsilon \subseteq \rho \cap (B \times B)$, 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, $\hat{\mathbf{r}}T$ 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 $c\langle d\rangle$.

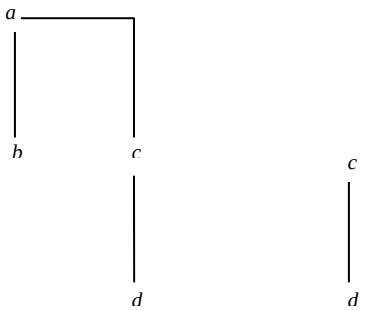

Figure 1.3 ***Tree and Subtree.***

■

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 $n = p_1 p_2 \ldots p_k + 1$. Observe that *n* is not divisible by any p_i, $1 \le i \le k$. 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 $i \ge b$, 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 $n \ge b$ and s_m is true for all $b \le m \le n$.
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

$$1 + 3 + 5 + \ldots + 2i - 1 = i^2$$

for all $i \ge 1$. 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 $1 \le m \le n$, where *n* is a natural number.
Inductive Step. Consider

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

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.

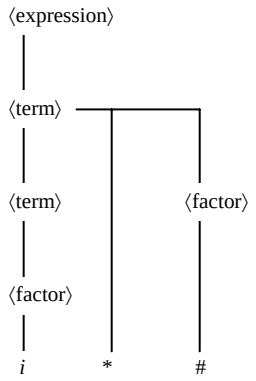

Figure 1.6 ***Parse Tree*****.**

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).

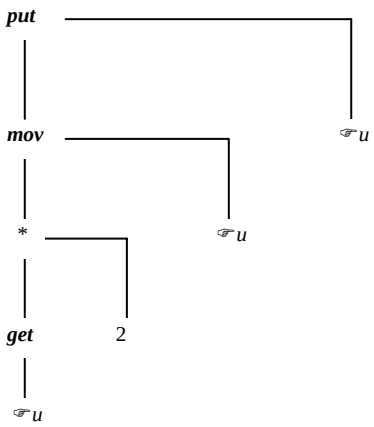

Figure 1.7 ***Syntax Tree*****.**

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.

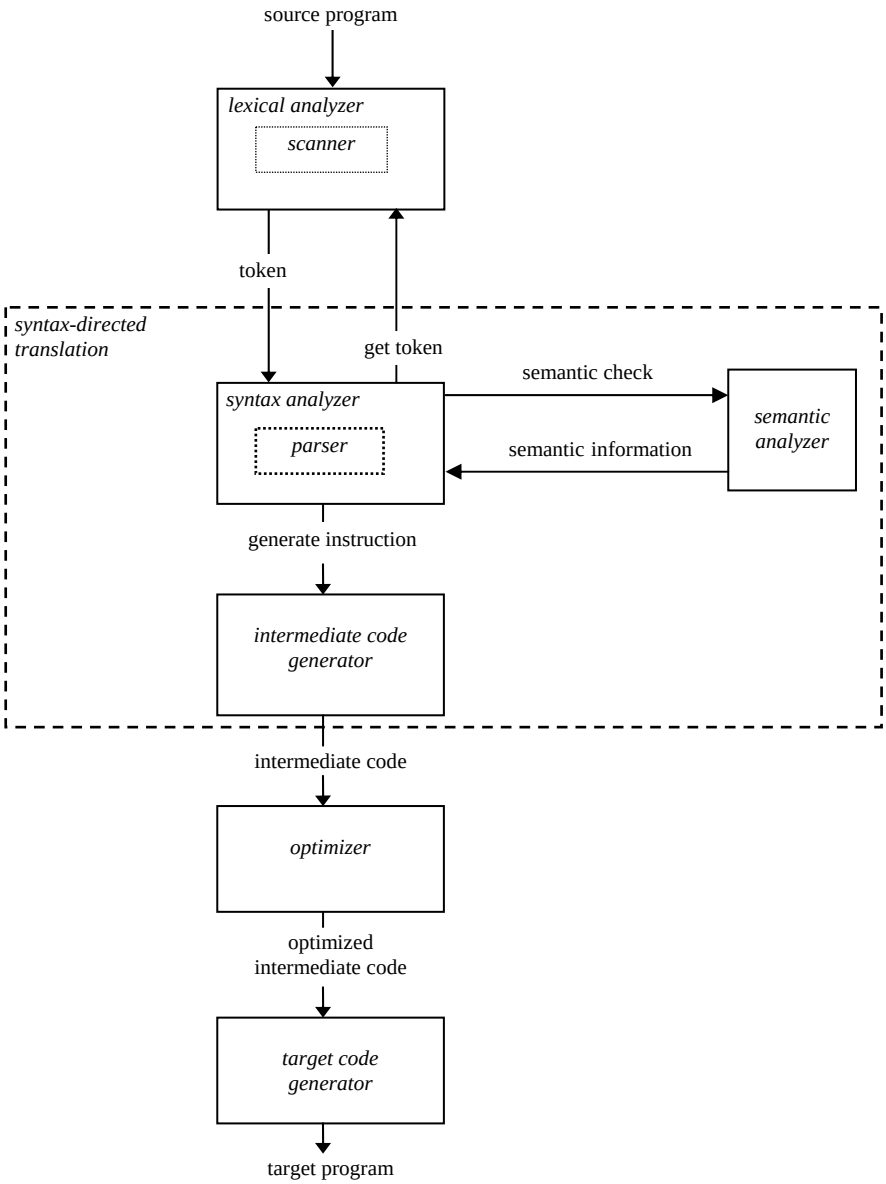

Figure 1.8 *Compiler Construction*.

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 $x \to y$ throughout this book. For brevity, we often denote $x \to y$ with a label r as r: $x \to y$, and instead of r: $x \to y \in R$, we sometimes write $r \in R$. For r: $x \to y \in R$, 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 $\rho = (r_1, r_2, \ldots, r_n)$, we simply write ρ as $r_1 r_2 \ldots r_n$. To explicitly express that Σ and R represent the components of M, we write ${}_M\Sigma$ and ${}_M R$ 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 $x \to y \in R$ 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: $x \to y \in R$. 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, $w_0, w_1, \ldots, w_n$ 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* $r_1 r_2 \ldots r_n$, symbolically written as $w_0 \Rightarrow^n w_n$ $[r_1 r_2 \ldots r_n]$ 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.