This page intentionally left blank
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  
0-8493-8066-3

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

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
Elements of Compiler Design

Alexander Meduna
in memory of St. John of the Cross
Contents

Preface ix
Acknowledgement xi
About the Author xiii

1 Introduction 1
1.1 Mathematical Preliminaries 1
1.2 Compilation 8
1.3 Rewriting Systems 15
Exercises 16

2 Lexical Analysis 21
2.1 Models 21
2.2 Methods 32
2.3 Theory 42
   2.3.1 Transformation of Regular Expressions to Finite Automata 42
   2.3.2 Simplification of Finite Automata 49
   2.3.3 Non-Regular Lexical Constructs 56
   2.3.4 Decidable problems 65
Exercises 68

3 Syntax Analysis 75
3.1 Models 75
3.2 Methods 89
3.3 Theory 103
   3.3.1 Power of Parsing Models 103
   3.3.2 Verification of the Grammatical Syntax Specification 104
   3.3.3 Simplification of Grammars 106
   3.3.4 Grammatical Normal Forms and Parsing Based on Them 116
   3.3.5 Syntax that Grammars cannot Specify 121
   3.3.6 Decidable Problems 128
Exercises 129

4 Deterministic Top-Down Parsing 139
4.1 Predictive Sets and LL Grammars 139
4.2 Predictive Parsing 145
Exercises 154

5 Deterministic Bottom-Up Parsing 159
5.1 Precedence Parsing 159
5.2 LR Parsing 168
Exercises 180
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 Csuhaı́j-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
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.

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

A finite set, $\Sigma$, 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 $\Sigma$. An infinite set, $\Omega$, is usually specified by a property, $\pi$, so that $\Omega$ contains all elements satisfying $\pi$; 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 $\Sigma$ and $\Omega$ be two sets. $\Sigma$ is a subset of $\Omega$, symbolically written as $\Sigma \subseteq \Omega$, if each member of $\Sigma$ also belongs to $\Omega$. $\Sigma$ is a proper subset of $\Omega$, written as $\Sigma \subset \Omega$, if $\Sigma \subseteq \Omega$ and $\Sigma$ contains an element that is not in $\Sigma$. If $\Sigma \subseteq \Omega$ and $\Omega \subseteq \Sigma$, $\Sigma$ equals $\Omega$, denoted by $\Sigma = \Omega$. The power set of $\Sigma$, denoted by $\text{Power}(\Sigma)$, is the set of all subsets of $\Sigma$.

For two sets, $\Sigma$ and $\Omega$, 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 $\Sigma$ is a set over a universe $U$, the complement of $\Sigma$ is denoted by $\text{complement}(\Sigma)$ and defined as $\text{complement}(\Sigma) = U - \Sigma$. The operations of union, intersection, and complement are related by DeMorgan’s rules stating that $\text{complement}(\Sigma) \cup \text{complement}(\Omega) = \Sigma \cap \Omega$ and $\text{complement}(\Sigma) \cap \text{complement}(\Omega) = \Sigma \cup \Omega$, for any two sets $\Sigma$ and $\Omega$. If $\Sigma \cap \Omega = \emptyset$, $\Sigma$ and $\Omega$ are disjoint. More generally, $n$ sets $\Delta_1, \Delta_2, \ldots, \Delta_n$, where $n \geq 2$, are pairwise disjoint if $\Delta_i \cap \Delta_j = \emptyset$ 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 $\varepsilon$, is the sequence consisting of no
element; that is, \(|x| = 0\). 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 \(\Sigma\) is a finite non-empty set, whose members are called symbols. Any non-empty subset of \(\Sigma\) is a subalphabet of \(\Sigma\). A finite sequence of symbols from \(\Sigma\) is a string over \(\Sigma\); specifically, \(\varepsilon\) is referred to as the empty string. By \(\Sigma^*\), we denote the set of all strings over \(\Sigma\); \(\Sigma^* = \{\varepsilon\}\). 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 \(a\) in \(x\), so \(\text{occur}(x, a)\) always satisfies \(0 \leq \text{occur}(x, a) \leq |x|\). Furthermore, if \(x \neq \varepsilon\), symbol\((x, i)\) denotes the \(i\)th symbol in \(x\), where \(i = 1, \ldots, |x|\). Any subset \(L \subseteq \Sigma^*\) is a language over \(\Sigma\). Set symbol\((L, i) = \{a \mid a = \text{symbol}(x, i), x \in L - \{\varepsilon\}, 1 \leq i \leq |x|\}\). 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, \(\Sigma^*\), which is called the universal language over \(\Sigma\), is an infinite language while \(\emptyset\) and \(\{\varepsilon\}\) are finite; noteworthy, \(\emptyset \neq \{\varepsilon\}\) because \(\text{card}(\emptyset) = 0 \neq \text{card}(\{\varepsilon\}) = 1\). 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 \(a_1 a_2 \ldots a_n\) instead of \((a_1, a_2, \ldots, a_n)\).

Operations. Let \(x, y \in \Sigma^*\) be two strings over an alphabet \(\Sigma\), and let \(L, K \subseteq \Sigma^*\) be two languages over \(\Sigma\). 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, \(\Sigma^*\) and \(\Sigma^+\) are the free monoid and the free semigroup, respectively, generated by \(\Sigma\) under the operation of concatenation. Notice that for every \(w \in \Sigma^*\), \(we = ew = w\). The concatenation of \(L\) and \(K\), denoted by \(LK\), is defined as \(LK = \{xy \mid x \in L, y \in K\}\).

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 \(\overline{L}\) and defined as \(\overline{L} = \Sigma^* - L\). The reversal of \(x\), denoted by \(\overline{x}\), is \(x\) written in the reverse order, and the reversal of \(L\), \(\overline{L}\), is defined as \(\overline{L} = \{\overline{x} \mid x \in L\}\). For all \(i \geq 0\), the \(i\)th power of \(x\), denoted by \(x^i\), is recursively defined as \((1) x^0 = \varepsilon\), and \((2) x^{i+1} = xx^i\), for \(i \geq 1\). Observe that this definition is based on the recursive definitional method. To demonstrate the recursive aspect, consider, for instance, the \(i\)th power of \(x\) with \(i = 3\). By the second part of the definition, \(x^3 = xx^2\). By applying the second part to \(x^2\) again, \(x^3 = xx^2\). By another application of this part to \(x^2\), \(x^3 = xx^2\). By the first part of this definition, \(x^0 = \varepsilon\). Thus, \(x^1 = xx^0 = xx = x\). Hence, \(x^2 = xx^1 = xx\). Finally, \(x^3 = xx^2 = xxx\). By using this recursive method, we frequently introduce new notions, including the \(i\)th 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 \(L^* = L^0 \cup L^1 \cup L^2 \cup \ldots\). 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, \(x \not\in \{\varepsilon, y\}\), \(x\) is a proper prefix of \(y\). By prefix\((y)\), we denote the set of all prefixes of \(y\). Set \(\text{prefixes}(L) = \{x \mid x \in \text{prefix}(y)\}\) for some \(y \in L\). For \(i = 0, \ldots, |y|\), prefix\((y, i)\) denotes \(y\)'s prefix of length \(i\); notice that \(\text{prefix}(y, 0) = \varepsilon\) and \(\text{prefix}(y, |y|) = y\). If \(xz = y\), \(x\) is a suffix of \(y\); if in addition, \(x \not\in \{\varepsilon, y\}\), \(x\) is a proper suffix of \(y\). By suffix\((y)\), we denote the set of all suffixes of \(y\). Set \(\text{suffixes}(L) = \{x \mid x \in \text{suffix}(y)\}\) for some \(y \in L\). For \(i = 0, \ldots, |y|\), suffix\((y, i)\) denotes \(y\)'s suffix of length \(i\). If \(wx = y\), \(x\) is a substring of \(y\); if in addition, \(x \not\in \{\varepsilon, y\}\), \(x\) is a proper substring of \(y\). By substrings\((y)\), we denote the set of all substrings of \(y\). Observe that for all \(v \in \Sigma^*\), \(\text{prefixes}(v) \subseteq \text{substrings}(v)\), \(\text{suffixes}(v) \subseteq \text{substrings}(v)\), and \(\{\varepsilon\} \subseteq \text{substrings}(v)\).
\( v \in \text{prefixes}(v) \cap \text{suffixes}(v) \cap \text{substrings}(v) \). Set \( \text{substrings}(L) = \{ x \mid x \in \text{substrings}(y) \text{ for some } y \in L \} \).

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

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 \( L \), \( \text{complement}(L) = \Sigma - L \), so every binary string is in \( \text{complement}(L) \) except 1 and 01. Furthermore, \( \text{reversal}(L) = \{ 1, 10 \} \) and \( L^2 = \{ 11, 101, 011, 0101 \} \). The strings in \( L^2 \) that consists of four or fewer symbols are \( e, 1, 01, 11, 011, 101, \) and 0101. \( L^4 = L^2 \cup L^2 \cup L^2 \cup \ldots \); the strings in \( L^4 \) have at least 4 symbols.

Notice that \( \text{prefixes}(L) = \{ e, 1, 0, 01 \} \), \( \text{suffixes}(L) = \{ e, 1, 01 \} \), and \( \text{substrings}(L) = \{ e, 0, 1, 01 \} \).

### Relations and Translations

For two objects, \( a \) 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 \( A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\} \). A binary relation \( \rho \) from \( A \) to \( B \) is any subset of \( A \times B \); that is, \( \rho \subseteq A \times B \). If \( \rho \) represents a finite set, then it is a finite relation; otherwise, it is an infinite relation.

The domain of \( \rho \), denoted by \( \text{domain}(\rho) \), and the range of \( \rho \), denoted by \( \text{range}(\rho) \), are defined as \( \text{domain}(\rho) = \{ a \mid (a, b) \in \rho \text{ for some } b \in B \} \) and \( \text{range}(\rho) = \{ b \mid (a, b) \in \rho \text{ for some } a \in A \} \). If \( A = B \), then \( \rho \) is a relation on \( \Sigma \). A relation \( \sigma \) is a subrelation of \( \rho \) if \( \sigma \subseteq \rho \). The inverse of \( \rho \), denoted by \( \text{inverse}(\rho) \), is defined as \( \text{inverse}(\rho) = \{(b, a) \mid (a, b) \in \rho \} \). A function \( \phi \) from \( A \) to \( B \) is a relation \( \phi \) from \( A \) to \( B \) such that for every \( a \in A \), \( \text{card}(\{b \mid (a, b) \in \phi \}) \leq 1 \). If \( \text{domain}(\phi) = A \), \( \phi \) is total; otherwise, \( \phi \) is partial. If for every \( b \in B \), \( \text{card}(\{a \mid a \in A \text{ and } (a, b) \in \phi \}) \leq 1 \), \( \phi \) is an injection. If for every \( b \in B \), \( \text{card}(\{a \mid a \in A \text{ and } (a, b) \in \phi \}) = 1 \), \( \phi \) is a surjection. If \( \phi \) is both a surjection and an injection, \( \phi \) represents a bijection.

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

Let \( K \) and \( L \) be languages over alphabets \( T \) and \( U \), respectively. A substitution from \( K \) to \( L \) is a relation \( \sigma \) from \( T^* \) to \( U^* \) with \( \text{domain}(\sigma) = K \) and \( \text{range}(\sigma) = L \). A total function \( \tau \) from \( T^* \) to \( \text{Power}(U^*) \) such that \( \tau(uv) = \tau(u) \tau(v) \) for every \( u, v \in T^* \) is a substitution from \( T^* \) to \( U^* \). By this definition, \( \tau(e) = \{e\} \) 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 \( \tau \) is completely specified by defining \( \tau(a) \) for every \( a \in T \).

A total function \( \upsilon \) 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 \( \nu \) by defining \( \nu(a) \) for every \( a \in T \). If for every \( a, b \in T \), \( \nu(a) = \nu(b) \) implies \( a = b \), \( \nu \) 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 \( \Omega \) be a set of binary operators, and let \( \Sigma \) 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 \((a + b) * c\) is \( ab+c* \).

The prefix notation is defined analogously 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 \( \Xi \) to \( \Psi \) defined as \( h(0) = \text{ZERO}, h(1) = \text{ONE}, \ldots, h(9) = \text{NINE} \). For instance, \( h \) maps 91 to \text{NINE}||\text{ONE} \). Notice that \( h \) is an injective homomorphism. Making use of \( h \), define the infinite substitution \( s \) from \( \Xi \) to \( \Psi \) as \( s(x) = \{h(x)\} \{\}\)'. As a result, \( s(91) = \{\text{NINE}\} \{\}\|\{\text{ONE}\} \{\}\)'. Including, for instance, \( \text{NINE}||\text{ONE} \) and \( \text{NINE}||\text{ONE} \).

### Graphs

Let \( A \) be a set. A *directed graph* or, briefly, a *graph* is a pair \( G = (A, \rho) \), where \( \rho \) is a relation on \( A \). Members of \( A \) are called *nodes*, and ordered pairs in \( \rho \) 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 \( \text{card}(\{b \in A \mid (b, a) \in \rho\}) \) and \( \text{card}(\{c \in A \mid (a, c) \in \rho\}) \). 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.](image)

**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.