Synthesis Lectures on Computer Science

The series publishes short books on general computer science topics that will appeal to advanced students, researchers, and practitioners in a variety of areas within computer science.

Paul A. Gagniuc

An Introduction to Programming Languages: Simultaneous Learning in Multiple Coding Environments

Springer

Paul A. Gagniuc
Department of Engineering in Foreign
Languages
Faculty of Engineering in Foreign Languages
University Politehnica of Bucharest
Bucharest, Romania

ISSN 1932-1228 ISSN 1932-1686 (electronic)
Synthesis Lectures on Computer Science
ISBN 978-3-031-23276-3 ISBN 978-3-031-23277-0 (eBook)
https://doi.org/10.1007/978-3-031-23277-0

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

On the occasion of his 85th birthday, I dedicate this work to my best friend, science partner and father figure, Constantin Ionescu-Tirgoviste. You are the greatest man I know! You are intelligence, wisdom, kindness, patience, verticality, diplomacy, morality and inspiration, in one single package.

Acad. Prof. Dr. Constantin Ionescu-Tirgoviste

Preface

This work is an introductory textbook in several computer languages. It describes the most well-known and popular programming environments such as: C#, C++, Java, JavaScript, PERL, PHP, Python, Ruby, and Visual Basic (VB) or Visual Basic for Applications (VBA). Therefore, the main objective of this unique guide is to provide code examples reflected in these nine computer languages. Readers can easily understand the connection and universality between the syntax of different environments and be adept at translating code. This learning experience can be ideal for upper-undergraduate introductory courses, researchers, doctoral students, and sociologists or engineers charged with implementing data analysis. Graphical illustrations are used for technical details about the computation examples to aid in an in-depth understanding of their inner workings. Moreover, the book contains original material that has been class-tested by the author and numerous cases are examined. Readers will also benefit from the inclusion of: (1) Historical and philosophical perspectives on the past, present and future of computer languages. (2) A total of 448 additional files are freely available online, from which a total of 44 files are poster presentations (i.e. PowerPoint and PDF files). (3) A total of 404 code examples reflected in nine computer languages, namely: C#, C++, Java, JavaScript, PERL, PHP, Python, Ruby and VB. This work first begins with a general introduction to history and presents the natural inevitable pathway from mechanical automation to present electronic computers. Following this historical introduction, an in-detail look is made at philosophical questions, implementations, entropy and life. More often than not, there is a genuine amazement of the younger generations regarding the advancement of computer technology. Historical events that led to the development of technologies have been distilled down to the essence. However, the essence of any story is made with a massive loss of detailed information. The essence of essences loses all the more information. Over time, the lack of detail leads to a collective amnesia that can prevent us from understanding the naturalness by which technology has evolved. Thus, new constructs are always built upon older constructs to fit the evolutionary chain of technological progress, which boils down to the same fundamental rules as biological evolution. In the first stage, this book discusses the natural path of programming constructs by starting from time immemorial and ending with examples up to the present times. In the end, naturally driven constructs of all

kinds also drive our society today. In the second part, the emphasis is made on the technical side where a total of nine computer languages are used simultaneously for mirrored examples. Simultaneous learning of multiple computer languages can be regarded as an asset in the world of science and technology. Thus, the reader can get used to the majority of known programming or scripting languages. Moreover, a basic knowledge of software implementation in several computer languages, even in an introductory way, helps the versatility and adaptability of the reader to new situations that may arise in industry, education, or research. Thus, this work is meant to bring a more concrete understanding of the similarities and differences between computer languages.

Ionel Bujorel Păvăloiu
Department of Engineering in Foreign
Languages, Faculty of Engineering
in Foreign Languages
University Politehnica of Bucharest
Bucharest, Romania

Acknowledgements

I wish to thank my friend Andrei Vasilateanu for a wonderful and precise review. His background in programming languages made him the perfect reviewer for this work.

Personal Words

I understand from confirmed sources that *42 is the answer to all things, the universe and everything*. Today I start to believe that myself since I rapidly approach this age of wisdom. Much of this book is based on personal experience that comes from a time period of rapid technological change. In my childhood, I have seen punch cards in use on the "FELIX computers" at the very beginning of the 90s, and my personal experience in the world of computer software started with the "Z80" processor. I know what it means to see red after a few hours spent on the phosphorescent green tube of the monitor. I remember the unmistakable sound of software, namely the incoming or outgoing data. I also remember how to save and load the source code to and from a magnetic tape of a cassette. I know what it means to switch from the "Z80" microprocessor and "BASIC" functional keys to "286" computers equipped with DOS operating systems and "QuickBASIC". I am a first-hand witness of the novelty called the mouse, and the perfection of the rubber sphere that is supposed to be cleaned of dust from time to time. I lived to see and feel the romance portrayed by all stages of the Internet and I was there to see the evolution of programming languages since the mid-90s. When I switched to languages like "C", "Turbo Pascal", or "Delphi", I remember the mystery and the potential I felt in regard to the "486 CPU" computers. Later, on "586", I was amazed by the "Visual Studio 6.0" package, and especially amazed by the "Visual Basic 6.0" programming language. Of this package, I am still amazed to this very day. I was fortunate enough to see the ups and downs of tech companies and the radical changes of the Internet, and because I am a Romanian, I witnessed the highest Internet connection speeds on the planet. I was born at the right time to experience punch cards, magnetic tapes, cassette tapes, floppy disks, songs/sounds of the modem, hard drives, CDs, DVDs, Blu-ray discs, USB drives and SSD drives. Computers shaped me! The journey made me a happy young-old man. But, could four decades encompass so much? It appears so! Well, these were the times, the best times.

Contents

1 Historical Notes 1
- 1.1 Introduction 1
- 1.2 The Ultimate Foundation 2
  - 1.2.1 Closer to Our Times 2
  - 1.2.2 Universality at the Crossroads 2
- 1.3 On the Recent Origin of Computers 3
  - 1.3.1 Automatons and the Memory of the Soul 4
  - 1.3.2 Mechanical Computers 5
  - 1.3.3 Electronic Computers 5
  - 1.3.4 American Standard Code for Information Interchange 6
  - 1.3.5 A Conspiracy for Convergence 7
- 1.4 History of Programming Languages 7
  - 1.4.1 The Making of an Advanced Civilization 8
  - 1.4.2 The Dark Age of Computer Languages 9
  - 1.4.3 The Extraordinary Story of ActiveX 11
  - 1.4.4 Killed on Duty by Friendly Fire 11
  - 1.4.5 The Browser: Resistance is Futile, You Will be Assimilated . . . 12
- 1.5 Conclusions 12

2 Philosophy and Discussions 15
- 2.1 Introduction 15
- 2.2 The Entropy of Software 16
  - 2.2.1 Entropy of Codes and Human Nature 16
  - 2.2.2 Raw Versus Fine-Grained Entropy 16
  - 2.2.3 How Does Software Entropy Increase? 17
- 2.3 The Operating Systems and Entropy 17
  - 2.3.1 The Twins 18
  - 2.3.2 Rejection of Equilibrium 18
  - 2.3.3 The Third Party Software 18
  - 2.3.4 Examples of Universality 19

2.4 Software Updates and Aging 20
2.5 Universality Supports Self-reflection 21
2.5.1 The Evolution of Large Brains Versus Entropy 21
2.6 From Computer Languages to Art and Sports 22
2.6.1 The Art 23
2.6.2 The Sport 24
2.7 Compiled Versus Interpreted 25
2.7.1 Programming Languages 25
2.7.2 Scripting Languages 27
2.7.3 Source Code Encryption 27
2.7.4 The Executable File 28
2.7.5 Executable Files and Scripting Languages 28
2.8 The Unseen and Unspoken 29
2.8.1 Witch Hunting Shows Weakness 30
2.8.2 No Secrets for the Emeritus 30
2.8.3 The War Against the Executable File 31
2.8.4 We Decide What Product Comes About 31
2.9 Psychological Warfare 32
2.9.1 Removal by Threat 32
2.9.2 Removal by Advertising 33
2.9.3 Handling of Terms 33
2.9.4 Battle of Computer Languages 34
2.9.5 Uniformity Means Death 35
2.9.6 Modern Does Not Mean Better 35
2.9.7 Market Share Demands Responsibility 36
2.10 Human Roles and Dilemmas 37
2.10.1 The Identity Crisis 37
2.10.2 Work Environments 38
2.10.3 Genus: *Homo* 38
2.11 Worst Professors Are Those Who Assume 39
2.12 Conclusions 40

3 Paradigms and Concepts 41
3.1 Introduction 41
3.2 The Story of Programming Paradigms 42
3.2.1 Imperative Programming 42
3.2.2 Declarative Programming 45
3.2.3 The in Between 46
3.2.4 The Foundation 46
3.3 Computer Languages Used Here 47
3.3.1 C# 47
3.3.2 C++ 47

3.3.3 Java . . . 48
3.3.4 JavaScript . . . 48
3.3.5 Perl . . . 48
3.3.6 PHP . . . 49
3.3.7 Python . . . 49
3.3.8 Ruby . . . 49
3.3.9 Visual Basic . . . 49
3.4 Classification Can be Misleading . . . 50
3.4.1 A Critique . . . 50
3.4.2 Which Computer Language is Better? . . . 51
3.4.3 The Operating System Versus the Application Makeup . . . 53
3.4.4 The Virtual Machine: A CPU for Bytecode . . . 54
3.4.5 Compiled Languages . . . 54
3.4.6 Interpreted Languages . . . 54
3.4.7 Just in Time Compilation . . . 55
3.4.8 Another Critique . . . 55
3.4.9 A Security Thought Experiment . . . 55
3.4.10 About Security Privileges . . . 56
3.5 The Quick Fix . . . 57
3.6 Conclusions . . . 59

4 Operators and Expressions . . . 61
4.1 Introduction . . . 61
4.2 Operators . . . 62
4.2.1 Arithmetic Operators . . . 62
4.2.2 Assignment Operators . . . 62
4.2.3 Relational Operators . . . 63
4.2.4 Concatenation Operators . . . 63
4.2.5 Logical Operators . . . 63
4.3 Operator Symbols . . . 63
4.3.1 Power Operator: The Curious Case of Exponentiation . . . 64
4.3.2 The Modulo Operator . . . 65
4.3.3 Unitary Operators . . . 67
4.3.4 The String Operator . . . 67
4.3.5 The Repetition Operator . . . 67
4.3.6 The Concatenation Operator . . . 68
4.3.7 Relational and Logical Operators . . . 69
4.4 Assignments . . . 71
4.4.1 Simple Assignments . . . 71
4.4.2 Aggregate Assignments . . . 72
4.4.3 Multiple Assignments . . . 72

4.5 Operator Precedence and Associativity . . . 73
4.6 Conclusions . . . 78

5 Data Types and Statements . . . 79
5.1 Introduction . . . 79
5.2 Data . . . 79
5.2.1 Bits and Bytes . . . 80
5.2.2 Symbol Frequency Matters . . . 82
5.2.3 The Encoding . . . 85
5.2.4 A Hypothetical System of Reference . . . 85
5.2.5 The Bytes of an Alien World . . . 86
5.3 Data Type . . . 88
5.3.1 The Curious Case of the String Data Type . . . 89
5.3.2 Experimental Constructs . . . 91
5.4 Statements . . . 92
5.4.1 ASCII Symbols . . . 92
5.4.2 Unicode Transformation Format . . . 92
5.4.3 Sentences are Made of Constructs . . . 93
5.4.4 The Root of Behavior . . . 93
5.4.5 The End of the Line . . . 93
5.4.6 Statements and Lines . . . 94
5.4.7 Multiple Statements and Line Continuation . . . 96
5.4.8 Recommended Versus Acceptable Statements . . . 98
5.5 The Source Code . . . 101
5.5.1 Indentations . . . 101
5.5.2 Comments . . . 102
5.6 Conclusions . . . 104

6 Classic and Modern Variables . . . 105
6.1 Introduction . . . 105
6.2 Variables . . . 105
6.2.1 Literals . . . 106
6.2.2 Naming Variables . . . 109
6.2.3 Variables: Explicit and Implicit . . . 110
6.2.4 Statically Versus Dynamically Typed Languages . . . 110
6.3 Evaluations of Expressions . . . 115
6.3.1 Details by Language . . . 116
6.4 Constants . . . 118
6.5 Classes and Objects . . . 120
6.5.1 About Design Patterns . . . 120

6.6 Arrays ... 121
6.6.1 Creating an Empty Array ... 121
6.6.2 Creating an Array with Values ... 123
6.6.3 Adding Elements ... 123
6.6.4 Accessing Array Elements ... 126
6.6.5 Changing Values in Array Elements ... 131
6.6.6 Array Length ... 131
6.6.7 Nested Arrays ... 137
6.6.8 Multidimensional Arrays ... 139
6.7 Conclusions ... 139

7 Control Structures ... 147
7.1 Introduction ... 147
7.2 Conditional Statements ... 148
7.3 Repeat Loops ... 153
7.3.1 The While Loop ... 153
7.3.2 The For Loop ... 159
7.3.3 Nested Loops ... 170
7.3.4 Multidimensional Traversal by One For-Loop ... 170
7.4 Conclusions ... 184

8 Functions ... 187
8.1 Introduction ... 187
8.2 Defining Functions ... 187
8.2.1 Simple Arguments ... 189
8.2.2 Complex Arguments ... 192
8.2.3 Nested Function Calls ... 196
8.2.4 Chained Function Calls ... 200
8.2.5 Relative Positioning of Functions ... 200
8.2.6 Recursive Calls ... 208
8.2.7 Global Versus Local Variables ... 214
8.2.8 Functions: Pure and Impure ... 218
8.2.9 Function Versus Procedure ... 218
8.2.10 Built-In Functions ... 223
8.3 Conclusions ... 228

9 Implementations and Experiments ... 233
9.1 Introduction ... 233
9.2 Recursion Experiments ... 234
9.2.1 Repeat String *n* Times ... 234
9.2.2 Sum from 0 to *n* ... 234
9.2.3 Factorial from 0 to *n* ... 254
9.2.4 Simple Sequence Generator ... 254

9.2.5 Fibonacci Sequence 255
9.2.6 Sum All Integers from Array 255
9.3 Interval Scanning 256
9.4 Spectral Forecast 265
9.5 Conclusions 274

Correction to: Historical Notes C1

References 275

List of Figures

Fig. 2.1 From entropy to art and back. The word “entropy” is written on the beaches of Golden Sands in Bulgaria. The word written in the sand indicates low entropy, which is quickly increased by noise represented by the waves of the Black Sea. The top-right panel shows a viral capsid close to a cell wall, which is portrayed by ASCII art 24

Fig. 2.2 Types of computer languages and their relationship to terms. It presents the relationship between scripting languages and programming languages and tries to highlight the relationship with the notions of interpreters and compilers. The first column from the left shows the classic case of a scripting language in which the source code is directly interpreted by an interpreter application. The middle column shows the situation often encountered today, where the source code is converted to bytecode, and then the bytecode is interpreted by an interpreter application for compatibility with the operating system and then compiled into machine code. On the right column, the classic programming languages are OS-specific, where the source code is directly converted into machine code. Note that Bytecode is a form of P-code, and it means pseudo code. Also, JIT is the Just-In-Time interpretation and compilation that a virtual machine does depending on the operating system 27

Fig. 3.1 Paradigms, computer languages and their syntax. It shows the link between hardware, computer languages, paradigms and syntax styles. Notice that low level computer languages are imperative and unstructured. Some older high-level computer languages that are equiped with the absolute jump commands, are in fact imperative and unstructured (ex. QBASIC). The bridge from unstructured to structured also exists. Some of the most recent higher-level computer languages, are equipped with absolute jump commands and functions at the same time (ex. VB6). Note that absolute jump commands are

known as “GOTO” in most high-level computer languages of the past, where this keyword was able to move execution from the current line to an arbitrary line (eg. Inside a 100-line implementation, “GOTO 10” can move execution to line 10, regardless of where the statement is made). In the assembly language, the most well-known unconditional jump command is the “JMP” mnemonic of Intel CPU’s. There are other types of jumps that represent conditional jumps, and these represent a myriad of mnemonics in groups of two to four characters that all begin with the letter “J” (eg. “JL”—Jump if Less, “JGE”—Jump if Greater or Equal, “JNLE”—Jump if Not Less or Equal, and so on). In other CPUs, like Z80, the mnemonic for the absolute jump command is “JP”. From firmware to firmware, these notations, or mnemonics, can be represented by different sets of characters. However, because the world works on Intel CPU designs, the word Assembly language is often associated with Intel CPUs. Note that mnemonics means “memoria technica” or “technical memory”, and it refers to how information is written in the shortest way in order to be remembered without information loss. In short, it is optimization of notation 44

Fig. 3.2 Bytecode portability and compilation versus interpretation. In an abstract fashion, it shows how most interpreted computer languages work today. It starts from the source code written by the programmer, which is assumed to be compiled to bytecode. The bytecode represents an abstraction of the initial source code. Bytecode is then used as it is on any platform, because there, whatever the platform is, it is met by an adaptation of the same virtual machine. This virtual machine makes a combination between interpretation and sporadic compilation (Just In Time compilation—JIT) to increase the execution speed of the software implementation. Note that “native code” and “machine code” have the exact same meaning across all figures that are alike. This particular figure contains the words “Native code” instead of “Machine code” in order to fit the text inside the horizontal compressed shapes. Note also that in a different context, “native code” may refer to the only language understood by some abstract object. For instance, Java bytecode is the “native code” to the Java Virtual Machine. As it was the case in the old days, some interpreters of lower performance (not necessarily VMs) made a direct interpretation of source code, without an intermediate step like the use of bytecode. In principle, virtual machines could be designed to directly interpret high-level source code, short circuiting the source code security

through obscurity or the multi-step optimization, or both. Thus, in such a case the "native code" would be the Java high-level source code. Also, please note that the abstract representation of the modules shown in the figure indicates a lack of extreme contrast between what is commonly called an interpreter or a compiler. That is, the compiler also does a little bit of interpreting and the interpreter also does a little bit of compiling 53

Fig. 4.1 Operator precedence and associativity symbols by computer language. In this table, operators enclosed in the same border have equal precedence and their associativity is shown on the column next to the symbols. The pink color of a cell indicates a group of operators and the light yellowish color indicates single operators per level. Note that the abbreviation OP means Order of Precedence; A = Associativity; N = Order of direction is not applicable here—non-associative; L = left-to-right; R = right-to-left. Some lesser known and used operator symbols are not shown here. The plus and minus signs belonging to addition and subtraction can be seen immediately below multiplication and division. Other plus or minus symbols present either above or below that position have dual roles, such as the plus sign in JavaScript which uses the symbol for both concatenation and addition. Other interesting observations are: In VB the "\" means integer division; in Ruby "=~" means matching operator; also in Ruby "!~" means NOT match. In C# the "^" means bitwise XOR, whereas in VB it means exponentiation 75

Fig. 4.2 Examples of operator precedence and associativity. At the top, the two panels show one example each for operator precedence or operator associativity. A mixed example is given at the bottom of the figure showing the relationship between operator precedence and operator associativity. In the lower right part there is a short list with symbols for only a few operators. In this list, the vertical order of the operators indicates operator precedence and the symbols found on the same level have equal precedence. Notice that in all panels there is a well-established and numbered sequence of computations that is based on precedence and associativity 76

Fig. 5.1 ASCII and UTF-8. It shows the back compatibility of UTF-8. On the vertical axis, the first half of the figure shows the structure of ASCII, which encodes for symbols using 8-bit sequences (1 byte). A schematic of UTF-8 is unrivaled in the second half of the figure. The UTF-8 relationship with ASCII is preserved for encoding positions starting from 0 to 127. However, starting from position 128 up to 255,

ASCII and UTF-8 use different encodings. Namely, ASCII uses 1 byte for this range, whereas UTF-8 uses 2 bytes. Outside the ASCII range, UTF-8 uses 2 bytes up to 4 bytes to encode new arrivals in the symbol set. UTF-8 may stop at 32 bit (4 bytes) representations, as all symbols with meaning in all human history, does not exceed 4.3 billion, as 4 bytes can encode 83

Fig. 5.2 Size of text according to UTF-8. It shows the size of text " ☀sunny" under UTF-8. This further includes the code points (the whole number associated with a symbol), the corresponding bit sequences and the actual symbols associated with these abstract representations. Note that boxes indicate abstract regions of physical memory. The space character and the letters that make up the word "sunny" take up a total of 6 bytes, however, the sun symbol is new and is encoded in 3 bytes instead of 1 byte. This observation is in fact very important. Usually, the most necessary symbols were those that were first introduced as characters in the development of computers over time. Consequently, time precedence of characters is directly proportional to their frequency of occurrence in data. Thus, preservation of the initial encoding for the most frequent symbols dictates the conservation of file size. UTF-8 characters can be represented by 1 byte for older legacy symbols, up to 4 bytes for newer symbols. This is one of the main reasons why UTF-8 is crucial to the future of technology when compared to other character encodings 84

Fig. 5.3 The alien text measured in alien bytes. The top of the figure shows five hypothetical characters in a 2D formation of 5×6 bits. Below the representations are the 3-bit codes that can be associated with these object characters. Just below the 3-bit codes, the characters are displayed using colors instead of 0 and 1s. The abstract box representation shows the 3-bit code and the character code associated with the symbols. On the bottom of the figure, an "alien" phrase of 20 characters is shown. The meaning of the phrase is not important. There, the comparison is made between the size of the 20 characters (200 bytes) and the size of the encoding (20 bytes). Thus, the "alien" example indicates the role of character encoding in reducing size without information loss. Note that in this example, an "alien" byte represents a 3-bit sequence 87

Fig. 5.4 Data Type representation. It describes the general constructs used by computer languages to represent data. The data type constructs shown here are normally divided into two, primitive data types and non-primitive data types. Primitive data types in turn are divided into two other categories, namely numeric and non-numeric data. Non numeric data contains the character type and the boolean type, whereas the numeric category contains the weight of the constructs. Namely, for integers, there is the byte type, the integer type, the long type and the short type. In the case of the floating point category there is double type and the float type. Among the non-primitive categories the array type, the string type and the object type are listed. The object type also implies the possibility of creating other new data types. Note: there are many computer languages today that no longer use primitives in the true sense of the word, but objects that simulate primitives, such as pure object-oriented languages, like Ruby 90

Fig. 5.5 Examples of multiline comments are presented in the case of Python, which show the connection between the one-dimensional patterns from the previous table and the two-dimensional representation from the source code. Note that the source code is in context and works with copy/paste 104

Fig. 6.1 One-dimensional array variables. It presents two different representations of array variables. The first approach from above shows how the lower bound starts from zero ($0 \ldots n$). Notice that the total number of elements in the array is $n + 1$. This is the case with many modern computer languages. The second approach shows the case of VB, were the index of an array variable may start from any value and end with any value that is bigger than the first ($n \ldots n + m; m > n$). Notice that the total number of elements in the array is $m + 1$ 137

Fig. 6.2 Multi-dimensional arrays. It shows two diagrams that represent array variables with two dimensions. The first diagram shows a lower bound that starts at zero for both dimensions, and the second (bottom) diagram representing an array variable with an arbitrary lower bound position for each dimension. These two representations can be given in three dimensions by providing another row in the diagram. This is true for any dimensions, in wich each dimension can be represented by boxes positioned linearly in this figure 146

List of Tables

Table 4.1 Critical Arithmetic Operators. These operators can be safely called the primitive operators as they are fundamental to every operation (especially addition and subtraction). The symbols for Addition, Subtraction, Multiplication, Division and Exponentiation, are shown for each computer language used in this work 64
Table 4.2 Concatenation, repetition and non-critical arithmetic operators. Some of these operators can be considered advanced operators because they are borderline constructs with built-in functions (notably these operators are: Modulus, Concatenation, Repetition). The Increment and Decrement operators are part of the list of primitive operators continued from the previous table. Briefly, symbols for Modulus, Concatenation, Repetition, Increment, Decrement, are shown for each computer language used in this work 65
Table 4.3 Relational operators. Relational operators which are also known as comparision operators, are used for comparing the values of two operands. Briefly, symbols for equality, inequality, less than, greater than, less than or equal to, greater than or equal to, are shown for each computer language used in this work. The square brackets in the table cells indicate the optional representation of the operands . . . 69
Table 4.4 Logical operators. Relational operations can only be linked together by using logical operators. Briefly, symbols for *Logical Not*, *Logical And, Logical Or*, are shown for each computer language used in this work. The square brackets in the table cells indicate the optional representation of the operands 70
Table 5.1 From bits to encoding possibilities and bytes. It shows the number of encoding possibilities for bit sequences between 1 and 64. For each bit sequence considered here, the number of bytes is shown from the octet perspective. Namely, the last column in the table shows

that bytes are no longer represented by an integer value when bit sequences are smaller or larger than multiples of 8. It can be seen that 32-bit sequences allow close to 4.3 billion coding possibilities $\left(2^{32} = 4.294967296e + 9\right)$. Likewise, 64-bit sequences cover the unthinkable, because there are not enough meanings in this world to fill the space of coding possibilities $\left(2^{64} = 1.84467440737e + 19\right)$ 82

Table 5.2 Example of primitive data types in Java. A primitive data type specifies the size and type of information the variable will store. There are eight primitive data types that are fundamental to programing. Note that 1 byte is 8 bits. Also, short is the inherited integer. Depending on the computer language, the integer data type may be either the old one (−32,768 to 32,767) or the new one (−2,147,483,648 to 2,147,483,647). Note that from one computer language to another, the ranges of the values associated with these data types vary greatly. Due to the increase in hardware capabilities over time, the range of values for data type constructs has naturally increased as well 86

Table 5.3 List of primitive data types and composite data types. The table lists primitive data types and composite data types for each computer language used in this work. Note that in one way or another all computer languages have data types that lend themselves into modern programming by necessity because of the inheritance from the past, such as array, string, integer, boolean and so on. Without these, the paradigm changes automatically 91

Table 5.4 Line feed, carriage return and the ASCII conversions. Representations for some of the non-printable ASCII characters are shown here for all computer languages used in this work. Note that "LF" stands for line feed, "CR" stands for carriage return, and "CR & LF" represents the two ASCII characters as a unit. The last two columns show the methods by which a character can be obtained based on the ASCII code or, how the ASCII code can be obtained based on a given character. The statements in the fifth column return a character, while those in the last column return an integer. Letter "a" represents an integer between 0 and 255, while "b" represents one character 95

Table 5.5 Multiple statements and Line continuation. Continuing a statement over multiple lines or putting multiple statements on one line is critical in some instances were complexity is high. The second column shows the pattern of positioning the code lines, labeled *a*, *b* and *c*, one after the other through a delimiter, namely the ":" symbol, or more frequently the ";" symbol. The third column shows a pattern that indicates the rules according to which a very long statement can be broken into multiple lines. In this case the example is made for assignments, namely on expressions placed at the right of the equal operator. The letters *a*, *b*, and *c* represent values of different data types. Note that, only in this example, the "■" character indicates the action of pressing the *Enter* key 97

Table 5.6 Comments and symbols. For exemplification, ASCII characters used to start a line of comment are shown for each computer language. Perhaps because of historical reasons, some characters are shared between languages. On the third column, a series of one-dimensional models show ways to write multi-line comments for each computer language. In these patterns, the letters *a*, *b* and *c* may represent any line of text. Only in this example, the "■" character indicates the action of pressing the *Enter* key 103

List of Additional Algorithm

Additional algorithm 3.1 It shows the "Hello world" example for all computer languages used in this work. This is intended as a positive first introduction. Note that the source code is in context and works with copy/paste 57

Additional algorithm 4.1 Examples of assignments are shown for multiple computer languages. An important observation is that VB refers to Visual Basic 6.0 (VB6) and VBA syntax, namely the last version of Visual Basic. Thus VB6 lacks aggregate assignment as this style is a relatively new addition to computer languages. VB6 can explicitly declare multiple variables for a certain data type (Dim a, b, c As Integer), however, it lacks the posibility for multiple assignment. Note that the source code is out of context and is intended for explanation of the method 71

Additional algorithm 5.1 The first line of each computer language in the above list, shows an extraction of an ASCII character on the basis of an ASCII code. The second line shows the extraction of the ASCII code based on a given ASCII character. The output for any of the above statements is "Code 65 is the: 'A' letter" and "Letter A has the code: '65'". Note that the source code is in context and works with copy/paste 95

Additional algorithm 5.2 It shows basic good practices in JavaScript, such as: what is recommended, acceptable, and wrong. Note that the source code is out of context and is intended for explanation of the method 98

Additional algorithm 5.3 It demonstrates multiple statements made on one line, and a line continuation for long statements. The statements shown here are very short, but the point of the exercise remains valid. Note that the source code is out of context and is intended for explanation of the method 99
Additional algorithm 6.1 It shows a few examples of literals. The examples bring a series of known data types, namely an integer literal (42), a floating point literal (3.1415), and two string literals ("a" and "this text"). Thus, anything that is written data is a literal. Note that the text is out of context and is intended for explanation of the method 106
Additional algorithm 6.2 It shows how values (literals) of different data types are assigned to variables. Please note that C#, C++, Java and VB use the data type explicitly, i.e. the type of the variable is declared before assignment. On the other hand, notice that all the other environments use implicit data type, that is, the value is able to explicitly declare the variable type. Judging by the trends, it is possible that in the future explicit assignments may be less frequent. Note that the source code is in context and works with copy/paste 107
Additional algorithm 6.3 It shows explicit and implicit declarations of variables as well as examples of expressions and their evaluations for all computer languages used here. It mainly shows the connection between operators and data types. Note that the source code is in context and works with copy/paste 112
Additional algorithm 6.4 Example of interesting evaluations in PERL showing that concatenations that use the "+" operator instead of the "." operator, lead to the elimination of the string value from the result, with no error in sight. Note that the source code is out of context and is intended for explanation of the method 117
Additional algorithm 6.5 It shows how constants are declared in different computer languages. Moreover, it shows the difference between constant declaration (second column) and variable declaration

(third column). Some computer languages use special keywords and data type declarations, while other computer languages do not. Notice how in certain computer languages where there are no special keywords for defining constants, the difference between constant and variable is made by convention; namely a variable written with an uppercase letter means a constant and a variable written with a lowercase letter means a simple variable whose content can be changed at will. Note that the source code is out of context and is intended for explanation of the method 119

Additional algorithm 6.6 It shows two methods of declaring an empty array. For declaration purposes, computer languages use either square brackets or round brackets to indicate that the variable represents a group of "internal subvariables". On the second column is the array square parentheses type of declaration. On the third column is the array constructor type of declaration. Most computer languages that use the array constructor statement are usually object-oriented. But not all of them; for example Python does not have a special keyword of this kind, preferring the array square parentheses notation. Those declarations that explicitly write the data type for the array, can obviously take any data type. Here the example was given on a string data type for computer languages such as C++, C#, Java or VB6. Note that the source code is out of context and is intended for explanation of the method 122

Additional algorithm 6.7 It shows how to create a multi-valued one-dimensional array variable using literals. In this example an array variable *A* is used to store only string literals and an array variable *B* is used to store integer literals. In languages such as Javascript, PHP, PERL, Ruby or Python, array variables can store several types of literals, including objects. In languages such as C++, C#, Java or VB6, array variables can store only one type of literal. Note that the source code is in context and works with copy/paste 124

Additional algorithm 6.8 It shows the statements by which an array variable A is declared and the statements by which literal values are subsequently inserted into the elements of the array variable. It should be noted that some computer languages such as Javascript, PHP, PERL or Ruby allow the declaration of an empty array variable, after which the values can be inserted into newly declared elements. On the other hand, in other computer languages such as C++, C#, Java, VB6 and Python, the number of elements in the array variable must be declared before the assignment of values. Note that the source code is in context and works with copy/paste 127

Additional algorithm 6.9 It shows how to access the values stored in the elements of an array variable. An array literal is declared, in which three string values (three separate characters, namely "a", "b", "c") are stored. Then, two variables *x* and *y* are declared, which take values from the elements of the array variable *A*. Then, once assigned to the *x* and *y* variables, the string values are displayed in the output for visualization. As it can be observed, the result obtained after the execution is "bc". Note that the source code is in context and works with copy/paste 129

Additional algorithm 6.10 It shows how to change values in existing array elements. An array variable *A* is declared. String literals are assigned to each element of *A*. The value from the first element of the array variable *A*, is assigned to a variable *x*. Then, a literal string value (i.e. "d") is assigned to the second element of variable array *A*, thus erasing the previous value (i.e. "a") from this element. Next, the value from the third element of *A* is assigned to the second element of *A*, thus deleting the initial value (i.e. "b") from the second element. The value stored in variable *x* is assigned to the third element of array *A*. At the end, the values from each element are displayed in the output for inspection. Here, the initial sequence "abc" was transformed

into the sequence "dcb". Note that the source code is in context and works with copy/paste 132
Additional algorithm 6.11 It shows how to get the total number of elements from an array. First an array literal *A* is declared, that contains three elements, each with a string literal (one character). Next, a variable *x* is declared and a value is assigned to it. The value in question represents the number of elements in array *A* and is provided either by an in-built function or by a method of the array object, depending on the computer language used. Finally, the content of variable *x* is displayed in the output for inspection. One thing to note is that in VB, the ubound internal function returns the last index in the array and not the total number of elements as expected from the other examples. Note that the source code is in context and works with copy/paste 135
Additional algorithm 6.12 It presents nested arrays in Javascript, Ruby and Python. Three array variables *A*, *B* and *C* are declared here, each with three literal values. To represent the notion of nested, three other array variables are declared, namely *D*, *E* and *F*, each with three elements that hold one of the arrays *A*, *B* or *C*. To provide yet another level in the nest, a last three-element array variable is declared (i.e. *G*), in which each element takes one of the recently mentioned arrays (i.e. *D*, *E* or *F*). Note that the source code is in context and works with copy/paste 138
Additional algorithm 6.13 It shows the way in which multidimensional array variables can be declared. An interesting difference can be observed between two groups of computer languages. A group involving Javascript, PHP, PERL, Ruby or Python and another group involving classic computer languages, namely C++, C#, Java or VB6. The first group (i.e. Javascript, PHP, PERL, Ruby or Python) uses largely the same type of declaration for several dimensions. The Javascript example shows how to declare two-dimensional and three-dimensional array variables, where the pattern can be followed for any

higher dimensions (i.e. 4D, 5D, 6D, and so on). In PHP, PERL, Ruby or Python, the exemplification is only repeated for two dimensions and it assumes that for more than two dimensions the declarations can be made in the same way as in Javascript. The second group is more different, where Java, C# and VB are radically different in the way statements are made. Obviously, Java and C# have common syntax elements, but they differ a little in the way the declarations for arrays are made. In VB, the number of dimensions and the number of elements in each dimension are initially declared. Only then these elements in their respective dimensions can receive values by assignment. VB is so radically different when compared to other computer languages, that array variables have a lower bound (read through the *LBound* function) and an upper bound (read through the *UBound* function), a property that can open paths in prototyping (especially in science). In the VB examples, each *Debug.Print* statement line corresponds to a row in the output. Note that the source code is in context and works with copy/paste 140

Additional algorithm 7.1 Demonstrates the implementation of conditional statements. Three variables *a*, *b* and *c* are declared and assigned to different values. A condition triggers a statement to increment the value of variable *c*, only if the value of variable *a* is less than the value of variable *b*, otherwise a decrement is applied to the value of *c*. Note that the source code is in context and works with copy/paste 149

Additional algorithm 7.2 Demonstrates the implementation of conditional statements on array variables. Three elements of an array variable (*A*) are declared and filled with values. A condition triggers a statment to increment the value of the last element of the array (i.e. “A[2]”), only if the value of the first element (i.e. “A[0]”) is less than the value of the second element (i.e. “A[1]”), otherwise a decrement is applied to the value

of the last element of the array. Note that the source code is in context and works with copy/paste 151
Additional algorithm 7.3 Here the positive increment while-loop structure is demonstrated. A variable *i* is declared and set to zero. A while loop structure increments variable *i* from its initial value to its upper limit (number five). At each iteration, variable *i* is printed in the output. The result is an enumeration of values from 0 to 4. Note that the source code is in context and works with copy/paste 155
Additional algorithm 7.4 It demonstrates the traversal of a one-dimensional array. An array variable is declared with string literals. The implementation also uses two other variables. A variable *t* stores string values and is initially set to empty. Another variable (i.e. *i*) initialized with value zero is the counter of a while-loop. The while-loop traverses the elements of array *A* by using the counter *i* as an index. At each iteration, the value from an element is added together with other string characters to the variable *t*. Once the end of the while-loop cycle is reached, the value collected in the variable *t* is printed in the output for inspection. Note that the source code is in context and works with copy/paste 160
Additional algorithm 7.5 The for-loop cycle for incrementing some simple variables is demonstrated. Specifically, two variables *a* and *b* are declared and initialized. The variable *a* is initialized to the integer five and the variable *b* is set to zero. The for-loop is then declared to start at the initial value of *i* and end at the value indicated by variable *a*. At each increment, the value in variable *i* is added to the numeric value stored in variable *b*. At the end of the loop, the final value stored in variable *b* is printed to the output for inspection. Note that the source code is in context and works with copy/paste 166

Additional algorithm 7.6 It demonstrates the use of a for-loop for the traversal of a one-dimensional array. An array variable is declared with string literals. The implementation also uses two other variables. A variable *t* stores string values and is initially set to empty. Another variable (i.e. *i*) initialized with value zero is the counter of a for-loop. The for-loop traverses the elements of array A by using the counter *i* as an index. At each iteration, the value from an element is added together with other string characters to the content of variable *t*. Once the end of the for-loop cycle is reached, the value collected in variable *t* is printed in the output for inspection. Note that the source code is in context and works with copy/paste 171

Additional algorithm 7.7 It demonstrates the use of nested for-loops. It shows the traversal of a two-dimensional array by a nested for-loop structure. A 2D-array variable (*A*) is declared with mixed datatypes, namely with string literals and number literals. A string variable *t* is initially set to empty. Another two variables (i.e. *i* and *j*) are initialized with value zero and are the main counters of nested for-loops. The upper limit of each for-loop is established by the two dimensions, namely the number of rows and columns from matrix *A*. The two for-loops traverse the elements of array *A* by using the counters *i* and *j* as an index. At each iteration, the value from an element is added to the content of variable *t*. Once the end of the nested for-loops is reached, the value collected in variable *t* is printed in the output for inspection. The end result is the enumeration of each value in the output, in a linear manner. Note that the source code is in context and works with copy/paste 174

Additional algorithm 7.8 It demonstrates the use of a single for-loop for two-dimensional arrays. It shows the traversal of a two-dimensional array by one for-loop structure. A 2D-array variable (*A*) is declared with mixed datatypes as before, namely with string literals and number literals. A string variable *t* is initially set to empty. A variable *v* is set to zero and it represents the main counter of the for-loop. Another two variables (i.e. *i* and *j*) are initialized with value zero and are the main coordinates for element identification. Each dimension of array *A* is stored in variables *n* and *m*, namely the number of rows in *n* and the number of columns in *m*. The upper limit of the for-loop is calculated based on the two known dimensions *n* and *m*. Thus, *m* times *n* establishes the upper limit of the for-loop. Here, the value of the counter *v* from the for-loop is used to calculate the *i* and *j* values that are used as an index to traverse the array variable *A*. The value of variable *j* is computed as the *v* % *m* and the result of this expression indicates the reminder (ex. 5 mod 3 is 2). The secret to this implementation is a condition that increments a variable *i* (rows) each time *j* (columns) equals zero. Thus, in this manner this approach provides the *i* and *j* values that a nested for-loop provides. At each iteration, the value from an element is added to the content of variable *t*. Once the end of the for-loop is reached, the value collected in variable *t* is printed in the output for inspection. The end result is the enumeration of each value in the output, in a linear manner. Note that the source code is in context and works with copy/paste 179

Additional algorithm 7.9 It demonstrates the use of a single for-loop for three-dimensional arrays, with an extrapolation to multidimensional arrays. Note that the example shown here is done only for Javascript in order to preserve paper. One can port this in any other

language as previously shown. The traversal of a 3D array using only one for-loop structure, is based on the previous example. A 3D-array variable (A) is declared with mixed datatypes, namely with string literals and number literals. The 3D-array is represented by five matrices, in which the columns represent one dimension, the rows represent the second dimension, and the number of matrices, represents the third dimension. Thus, this array can be understood as a cube-like structure. A string variable t is initially set to empty. A variable v is set to zero and it represents the main counter of the for-loop. Another three variables (i.e. i, j and d) are initialized with a value of zero and are the main coordinates for array element identification. Each dimension of array A is stored in variables s, m and n, namely the number of matrices in s, the number of rows in m and the number of columns in n. The upper limit of the for-loop is calculated as $s \times m \times n$. Here, the value of the counter v from the for-loop is used, as before, to calculate the i, j and d values that are used as an index to traverse the array variable A. The value of variable j is computed as the v % m. A condition increments a variable i (rows) each time j (columns) equals zero. Thus, both i and j values are computed. However, the value for variable d (matrix number) is calculated as v % ($m \times n$), which provides a value of zero each time a matrix was traversed. Thus, a condition increments variable d and resets variable i, each time the value of k equals zero. At each iteration, the value from an element (d, i, j) is added to the content of variable t. Once the end of the for-loop is reached, the string value collected in variable t is printed in the output for inspection. The end result is the enumeration of each value in the output, in a linear manner. Note that the source code is in context and works with copy/paste 185

Additional algorithm 8.1 It shows the use of functions that take simple arguments. An integer literal is assigned to a variable *a*. Variable *a* is then used as an argument for a function called "compute". Function "compute" takes the argument and uses its value in a mathematical expression. The returned value of function "compute" is then assigned to a variable *b*, which is then printed into the output for inspection. Note that the source code is in context and works with copy/paste 189

Additional algorithm 8.2 It shows the use of functions by considering complex arguments. Such complex arguments can be strings, array variables, or complex objects. In this specific case, a string and an array variable are used as arguments to a function called "compute". An array variable containing five elements is declared using string literals. Then a string variable *t* is declared and set to empty. The two variables are passed to the "compute" function. Inside the "compute" function, a for-loop traverses each element of the array *a*, and it adds the value from it to the accumulator variable *t*. At the end of the for-loop, the "compute" function returns the value of *t*, which is assigned to a string variable *b*, that is further printed onto the output for inspection. Note that the source code is in context and works with copy/paste 193

Additional algorithm 8.3 It shows the principle of nested function calls in which the return value of the most inner function becomes the argument for the most immediate outer function call, and so on. An integer literal is assigned to variable *a*. Then, the final return value of a group of nested function calls is assigned to a variable *b*, which in turn is printed to the output for inspection. Initially, the value stored in variable *b* is a negative value (i.e. -756029). Thus, for demonstration purposes, the minus sign is inserted in front of variable *b* in order to change the sign of the stored integer value (i.e. $b = -b$). Note that the source code is in context and works with copy/paste 197

Additional algorithm 8.4 It shows how functions may use other functions in a chain of calls. Another important observation made here, is related to the position of functions relative to the main program. In some computer languages function must be declared before the main program, whereas in other computer languages the order of the functions or the position of the functions relative to main, is not important. This fact indicates how the source code is treated by the compiler. That is, in some computer languages, execution is immediate, regardless of whether the functions are loaded or not, while in other computer languages, execution begins once all the code is loaded. The example from above shows how two variables become the arguments of a function *c*1, which pass their values to other functions in a chain that ends in a function *c*5. This trip of the arguments shows different types of additions until the last level is reached, such as additions of values, either literals, returned values from other functions or values from new variables. Function *c*5 uses a for-loop to traverse the elements of the array variable in order to sum up the values in the accumulator variable *t*. Once the for-loop finishes the iterations, the value from variable *t* is returned to function *c*4, which adds some other value to the this response. In turn, function *c*4 returnes the value to function *c*3, until it reaches the path to function *c*1, which assigns the final response value to a variable *b*. Variable *b* in turn is printed into the output for inspection. Notice that, in the case of C++, variable *t* holds the total number of elements of array *a*, until the chain of calls reaches function *c*5. There, the content of variable *t* is assigned to a new variable (i.e. *l*), and variable *t* is set to zero to take the role of an accumulator variable for calculating the sum. It should be noted that pointers can be used, namely, the parameter “int a[]” can be written as a pointer, namely “∗a”, which will provide the same result because the number of elements

in array *a* is calculated before any function is called. Note that the source code is in context and works with copy/paste 201

Additional algorithm 8.5 It shows how a recursive function call can be a replacement of a for-loop statement. Thus, a function called "for-loop" is capable of receiving three arguments. An argument for *a*, which is the counter for the number of self-calls, another argument for *b*, which indicates the upper limit of recursive calls (self-calls), and finally an argument for *r*, which accumulates an integer literal (i.e. 5) at each iteration/recursion. Inside the function a condition checks if the value of *a* is higher or equal to the value of the limit, namely *b*. In cases that *a* is less than *b*, the recursion continues, whereas if *a* is higher or equal to *b*, the value of *r* is returned back to the original caller. Once the final return value arrives to the caller, it is immediately assigned to variable *a* in the main program, an then the content of the *a* variable is printed into the output for inspection. Note that the source code is in context and works with copy/paste 210

Additional algorithm 8.6 This example shows the meaning of constants and global variables. A constant (i.e. *a*) and a global variable (i.e. *b*) are declared, either in the main routine (e.g. in Javascript, PHP, PERL, Ruby and Python) or outside the main routine/program (e.g. like in C++, C#, Java and VB/VBA). In the main routine a function named "compute" is called to provide a return value for a variable named *b*. Once the thread of execution moves to the "compute" function, the value from the global variable *b* is visible inside the function and is assigned to a local variable *x*. The content of variable *x* is then used inside a mathematical expression and the result is returned to the caller. Once the returned value is assigned to variable *b*, the content of the variable and that of the constant is then printed into the output for inspection. In the C++ computer language, one can see a comment declaring the constant and the global variable

between the two functions. For testing, the activation of those declarations will result in an error because in C++ or VB, constants and global variables are written at the beginning of the program because the compiler needs to know the context before execution. In PHP and Python, global variables have visibility inside a function only if they have a special declaration (i.e. Global $name_of_variable;). Also notice that in Ruby, global variables are denoted using the dollar sign in front of the name of the variable (ex. $b). Note that the source code is in context and works with copy/paste 215

Additional algorithm 8.7 It shows the meaning of pure and impure functions. A function named "pure" receives an argument for x and returns a value that is the result of the evaluation of a mathematical expression. This function is pure because it does not change anything outside the function. On the other hand, a function called "impure" receives the same argument for x that is used in the same mathematical expression as in the "pure" function. However, the "impure" function, modifies the value of a global variable a. This modification made outside the function makes the function impure. Notice that both functions return the same result in the initial call. However, in the third call the returned value differs, as the global variable a that is modified by the "impure" function is in fact the argument for the next calls. Note that the source code is in context and works with copy/paste 219

Additional algorithm 8.8 It shows the difference between functions and procedures. A pure function named f takes an argument and returns a value based on a mathematical expression. A procedure named "p" that takes no arguments and gives no return values, is used to assign the result of a subtraction to a local variable x (i.e. $x = a - 11$). Next, the result of a mathematical expression is assigned to a global variable b, after which the execution thread returns automatically to the caller. Notice

that in PHP and Python, global variables have visibility inside a function only if a special declaration exists (i.e. Global $name_of_variable;). Also, notice that VB has a special keyword for procedures. The distinction between functions and procedures is made by using the keyword "function" and the keyword "Sub", respectively. Moreover, in VB, a sub is not called by using the round parenthesis as "p()", but the name of the procedure is simply stated, like "p". Single letter names for procedures can be confusing in case of VB, and procedure names with more than two characters are adviseable. Note that the source code is in context and works with copy/paste 224

Additional algorithm 8.9 This shows an example of using the built-in functions. In this specific case, it shows how to check for the presence of a string above another string. A string literal is assigned to variable "a" and a string literal representing the target is assigned to a variable "q". The number of characters found in *a*, is assigned to a variable *b*. Next, in a function chain all *q* encounters found in the string of *a*, are replaced with nothing. If the *q* string exists in variable *a* than the result is a shorter string than the original. Next in this function chain, the result is passed directly to the length function, which provides the total number of characters in the procesed string. This last result is then assigned to variable *c*. In a condition statement the value of *c* is compared with the value from *a*. If the two values are different, it means that *q* was present in the original string of *a*. Note that the replacement is made by using two methods: (1) The *split* function that uses *q* as a delimiter, provides an array wich in turn is converted into a normal string again, without any instances of *q* (this can be seen in Javascript and VB). (2) The replace function which is able to replace all instances of *q* found in *a*, with an empty string (eg. it deletes *q* from *a*). Note that the source code is in context and works with copy/paste 229

Additional algorithm 9.1 It shows different experiments on recursive functions. A total of six examples are shown, in which: (1) A recursive function repeats one (or a group) of characters n times and returns a string of length *n*. (2) A recursive function sums integers from zero to *n*. (3) A recursive function computes the factorial for an integer *n*. (4) A function generated a sequence of numbers based on various rules. (5) A recursive function provides the Fibonacci sequence. (6) A recursive function sums all the integers stored in the elements of an array variable. Note that the source code is in context and works with copy/paste 235

Additional algorithm 9.2 It shows how a distribution can be calculated for a range of integers. This example uses a mathematical expression shown across the chapters. The mathematical expression takes an input value and, as expected, provides an output value. In this particular example, an implementation takes a range of integers and returns a corresponding range of values calculated using the mathematical expression. For each computer language there are two examples. One example that uses a string variable to store the results, and another example that uses an array variable to store the results. The two examples per computer language show the malleability of code, that points out the possibility of multiple solutions to one problem. Note that the source code is in context and works with copy/paste 257

Additional algorithm 9.3 It shows the implementation of the *Spectral Forecast* equation on two signals. Two signals are represented by a sequence of numbers each. This sequence of numbers is stored as a string value in two variables *A* and *B*. These two values are then decoded into individual numbers inside the elements of the array variables (*tA* and *tB*). The maximum value found over the elements of the two array variables is calculated and stored before switching to the computation of *Spectral Forecast*. The array variables *tA* and *tB* are then used inside a for-loop to calculate a third signal *M* using the *Spectral Forecast* equation for a predefined index *d*. The index *d* determines how similar the third signal is to signal *A* or signal *B*. The method shown here allows for a useful protocol to manage and process numeric data stored as simple text, a case that is often encountered in science and engineering. Note that in the case of C++ some new built-in functions can be applied to a value inside a variable *v*, such as: the "substr" function that cuts a certain portion of a string, or the "strtof(v)" which converts a string to float. Other functions of interest not used here are: the "strtod(c)" function that converts a string to a double, or the "v.c_str()" method that converts a numeric value to a string. Also, in C++ the example uses vectors, and the number of components is given by the "size()" method. Again, the source code is in context and works with copy/paste 267