Synthesis Lectures on Computer Science

SYNTHESIS
COLLECTION OF TECHNOLOGY

Paul A. Gagniuc

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

MOREMEDIA

Springer

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

List of Figures

List of Tables

List of Additional Algorithm

Historical Notes

1

1.1 Introduction

The need for artificial computation always existed in the history of our species. Here, the history of programming languages is pushed far into the past, much further than perhaps expected. This chapter explains the broader meaning of programming languages. It starts from the origin of life and continues by pointing out the natural development of technology that has led to the general purpose electronic computers of today. In order to understand the lack of mystery, the milestone achievements from the last centuries are presented in sequence. Namely, automatons are presented as the main route from which mechanical computers emerged. Next, mechanical computer models are described as the natural predecessors of electromechanical computers. In turn, the first electronic computers based on vacuum tubes are described as the main step forward when compared to the electromechanical relays of previous computer models. The era of vacuum tube computers is further presented as a short-lived dream replaced by the era of transistors and contemporary general purpose computers based on integrated circuits. Towards the end of the chapter, a detailed presentation is made about the very first high-level computer languages and their evolution to the present day.

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/978-3-031-23277-0_1.

P. A. Gagniuc, *An Introduction to Programming Languages: Simultaneous Learning in Multiple Coding Environments*, Synthesis Lectures on Computer Science,
https://doi.org/10.1007/978-3-031-23277-0_1

1.2 The Ultimate Foundation

In all aspects, life is a phenomena based on discrete components and quantities [1]. By using a stretch of the imagination, one can assume that programming languages are connected to this phenomenon. It is perhaps unusual to push the origins of computations at the very origin of life. However, from our human perspective, two main kinds of programming languages can be identified with ease, one class that applies to all biological life forms and the other class that applies to machines, by extension. It can be argued that the earliest possible language was the biochemical feedback between living organisms in the earliest forms of life [2, 3]. Thus, action and reaction where the primary roots of computation. In other words, behavior induced behavior. Since the beginning of life, organisms were able to signal each other, or better said, program each other by using different means of communication [4]. At the root, communication is the action of exchanging raw or encoded information between two biological or/and artificial agents. Thus, either chemical, mechanical, electromagnetic or another type of communication environment, the output of one it is the input of the other [1].

1.2.1 Closer to Our Times

When hominides (great apes) came to be, programming reached new levels of complexity. Prior to everything it was the sign, namely the gesture, that was used in order to communicate instructions [5, 6]. So it is presumed; because of our observations on other animals [7]. Later, in modern humans, sounds evolved into more meaningful channels of communication that were capable of encoding complex information. Thus, those sounds in turn became the way to communicate more discretised and sophisticated instructions [8, 9]. The written language came to be, when the meaning of sounds was associated with drawn symbols [10]. Thus, computer programming languages really are an extension of written symbols with a meaning. For some time, sequences of symbols have been the only practical method of coding and recording meaning. All writings, like books, articles or letters are in fact a form of software, in which grammar has set the rules on how instructions can be represented.

1.2.2 Universality at the Crossroads

Let us imagine a piece of text written by an individual, which later is read by another individual. Is that text a software executed in the mind of the reader? It could be argued that indeed this is the case. Reading a book is the process of execution of a program. For instance, this is the main reason for which the act of written propaganda works as it is intended [11]. The correct sequence of words allows for simple programs that are

universal between the receivers. Thus, they work accordingly. In such primitive programs, the written truth was always the main application and the written lie represented the malware. In order to make an association, we can point out that even in our current programming environments on computers, the source code can be implemented in a highly language-specific manner, or, in a general widely compatible way which can be easily portable into other environments. The point of the above discussion is to show different forms of programming languages and their universality. In modern times, next to the written language it was the written language of mathematics and the written language of music (the musical notation). Thus, programming languages take different forms across time but they are an integral part of life, regardless of the form that living beings can take. To put things into perspective, these fundamental laws are also the reason for which some classical expectations are being drawn in regard to extraterrestrial life forms [12, 13]. Above the kingdoms of life on our planet, all programming paradigms can be found in one way or the other. As a side note, programming paradigms are different styles in which a biological or artificial machine can be programmed.

1.3 On the Recent Origin of Computers

Many inventions, discoveries and innovations have been tried in the development of computers. In this subchapter a short discussion points out the main steps that led to the computer revolution. This story begins with automatons (robots) and ends with the electronic computers of today. Moreover, a description is made on the main steps and the technologies that were adopted for achieving the final goal of a fully programmable, general-purpose computer. The following points out how automatons were the precursors to mechanical computers and then later to the electromechanical computers and then to the thermionic valve (vacuum tubes) computers, and last the precursors to the transistor-based computers and the Integrated Circuits computers. Please note that analogue computers are not discussed here. The difference between analogue computers and digital computers is, well, binary. As the analogue computers become more complex, these kind of machines are always under the umbrella of chaos theory because of noise, which leads to a higher or lower offset for values over time. In other words, the same input into a complex analogue computer always leads to a slightly different output. Digital computers on the other hand are noiseless, and are in fact able to generate a separate universe from our own, in which the same input always leads to the same output [1]. Thus, the main story begins with the concepts that lead to digital machines.

1.3.1 Automatons and the Memory of the Soul

The ability to start a discussion about mechanical computers it is given by the quest of the ancients for immortality and artificial life [14, 15]. The fundamental ideas behind our electronic computers of today comes from a class of machines named automatons. An automaton is a self-operating machine that follows a sequence of operations which may represent predetermined instructions. Usually, such machines where mechanical robots used in the past to mimic humans and animals [16]. The idea of digital comes from a desire to discretize actions. Analogue machines started to incline more and more in favor of discretization in this time period. One may imagine a rotating cylinder with hard solid bumps that push against a series of rods. The way the bumps on the cylinder are positioned determines the sequence in which the rods are pushed. In turn, the rods trigger different mechanical events. This kind of mechanism is in fact at the core of a primitive mechanical computer with a memory. The amount of information, namely the amount of bumps that can be placed on the surface of a cylinder, is limited. The more information is needed, the more the diameter of the cylinder has to increase or the density of bumps must be increased by decreasing their size. To avoid these constraints, the solution was an unfolded cylinder. The punch card system (paper tape with holes) was invented by Basile Bouchon and Jean-Baptiste Falcon in 1725–1728 as a method to store patterns that in turn control a loom [17, 18]. Note that a loom is a machine that is capable of weaving cloth and tapestry. In 1741, Jacques de Vaucanson improved on the Bouchon-Falcon design and simplified the mechanism to be useful in automatons [19]. In 1804, Joseph Marie Jacquard made use of these "memory units", namely the punch cards. He patented a design that today is known as the Jacquard machine. These memory units called "punch cards" were based on patterns of holes made on a piece of thick paper or even on thin boards of wood. The punch cards were able to control the sequence of operations performed by a mechanical machine. A hole in the card can be understood as binary one, and the absence of the hole as a binary zero. By following the positions of the holes on these cards the Jacquard machine was able to precisely construct different pictures on textile materials [17]. The Jacquard machine represents probably one of the most nodal points in the history of computer hardware and computer software [18, 20]. The ability to change the "memory" of the loom by replacing the punch cards was the most important conceptual precursor of data entry and computer programming [21]. In order to make a last obvious association between the past and present, one can force an equivalence between the Jacquard machine and the computers of our time [20]. What Jacquard made, is perhaps the oldest graphic card, in which the punch cards represented the way a picture was encoded into memory, and the resulting textile material can be viewed as the monitor display [18].

1.3.2 Mechanical Computers

The step-by-step evolution from punch cards was slowly replaced with the magnetic-core memory, the magnetic tape technology, optical discs and so on. However, the conceptual ideas born in France are used today regardless of technology that is being used as a support [19]. In short, these are the same binary one represented by "something", and the same binary zeros represented by the lack of "something". Worldwide, punch cards have been used to load software into computers up to the end of the 80s. Nonetheless, coming back to 19th Century we can discuss further about other two important key points in the history of mechanical computers. The first key point is related to the Analytical Engine [22]. The Analytical Engine was a concept with a design, and it was never realized in practice. The Analytical Engine was envisioned with an Arithmetic Logic Unit (ALU), a basic flow control, and of course, punch cards. The second key point is related to the actual design and construction of a mechanical computer. Between 1935 and 1938 the very first mechanical digital computer called "Z1" was built in Germany, Berlin [23, 24]. The "Z" series of machines, were also the beginning of a transition from mechanical computers to electronic computers. Next, the "Z3" electromechanical computer was completed in 1941. The "Z3" machine was based mainly on electrical relays and it was a programmable, fully automatic electromechanical digital computer. The "Z1" and "Z3" machines were not equipped with conditional branching as it was the case in later universal computers. It is worth mentioning that in this series of machines, the "Z10" version was the first commercially-sold computer anywhere in the world. Next, the "Z22" version was the first computer to use magnetic storage for memory.

1.3.3 Electronic Computers

The Atanasoff–Berry machine made in 1939, was the first vacuum-tube computer [25]. However, this prototype was not a general-purpose computer. In 1945, the Electronic Numerical Integrator and Computer (ENIAC) machine was the first vacuum tube general-purpose digital computer [26, 27]. By using the principles of punched tapes (perforated paper tape), ENIAC was able to receive what today we call software [28]. A crucial point in the history of civilization was the invention of the transistor in 1947 [29, 30]. At the end of the 40s, the high expectations of a new technological revolution was on the rise on good grounds: Semiconductors [30]. The vacuum tube and the transistor basically did the very same thing, namely amplification and switching [27]. The only differences between the two where relevant properties such as power consumption, temperature and size [31]. Thus, the first computer to use discrete transistors instead of vacuum tubes, was the "Transistor Computer" build in 1953 [32]. Next, a few months later (January 1954), the Bell Laboratories TRADIC (TRAnsistor DIgital Computer) was announced [33, 34].

Throughout the 50s, transistor components gradually replaced vacuum tubes in all computer designs. At the end of the 50s, not only that computers used only transistors, but the revolution continued with miniaturization. The very first prototype for an Integrated Circuit (IC) was made in 1958 [35, 36]. From there the entire path of computer development that leads to our times is most likely well-known and highly familiar to all of us. The list of achievements from the past is large. Here, I mentioned only a few in order to show the smooth step-by-step innovations that lead to the modern computers. The point here is that all these developments seem natural from our perspective. Of note is that none of these machines were only mechanical or only electromechanical, or only electronic with vacuum tubes, or only electronic by using transistors. The majority of these unique machines have used hybrid technologies. For instance, the mechanical computer had an electrical engine to power it, the electromechanical computer still had mechanical parts in it, the vacuum tube computers still had some relays in there for different processes. Last but not least, the transistor computers still made use of previous components like vacuum tubes in some places. Of course, all these converged quickly to fully transistorized computers and more miniaturization by using microchips, that is, until we reached today.

1.3.4 American Standard Code for Information Interchange

With a rapid technological evolution came an acute need for data standardization for computers [37, 38]. Thus, in 1963 the first character encoding standard for electronic communications appeared, and it was known as the American Standard Code for Information Interchange (ASCII). Far into the past, prior to any kind of real technology, people have used digital encodings by blocking or unblocking the light from fires to signal to another individual the presence of an enemy and details about their position [39]. This was done in all recorded history, from the ancients to the time of *Stefan the Great and Saint* up to recently in the World War II [40–43]. Thus, the medium was the photons of visible frequencies emitted from the fires [43–45]. This signaling principle has led to the telegraph, which used a different medium (electrons), but the exact same principles of communication [46, 47]. As the past always follows the current times, ASCII was based on the telegraph code. Telegraph codes are character encodings extensively used in the past to transmit information optically or by wire [48]. As it happens, in these one dimensional environments codification was best done digitally, by using dots and lines, like in the Morse code [49, 50]. Thus, their direct use for digital computers was as natural as any other development presented above.

1.3.5 A Conspiracy for Convergence

This section is the part of computer history that not many people are aware of, in order to acknowledge the real heroes of our civilization. The 60s, where the most important years for our current technology. The graphical interaction with a computer machine is made earlier than expected by many. The first graphical user interface (GUI) was developed in 1963 [51–53]. This prototype demonstrated object oriented approaches and it included vector graphics, 3D modelling, touch screen capabilities and even a flowchart compiler (assembler) [52]. In 1968, the computer mouse and fundamentals of modern computing were fully demonstrated on the "SDS 940" computer, like for instance the file revision control, hypertext linking, real-time text editing, multiple windows with flexible view control, cathode display tubes, and shared-screen teleconferencing [54–56]. The "Xerox Alto" (1973–1975) machine was built on the concepts of modern computing that were demonstrated years earlier on the "SDS 940" computer [57]. Moreover, "Xerox Alto" was the very first computer desktop as it is understood today. "Xerox Alto" was equipped with a graphical operating system, using graphical icons and a mouse to control the system [57]. It had current day concepts like document layout, bitmaps, vector graphics editing, and even Object Oriented Programming (i.e. OOP in the "Smalltalk" programming language) [57–59]. In 1981, the Xerox Star computer represented objects and applications with desktop icons in the same manner we are used today. All advanced modern features were already a reality embedded in the Xerox Star machine (official name: Xerox 8010 Information System) [60]. For instance it used what today we call a desktop interface, containing window-based graphical user interface, desktop icons, file types, folders, copy/paste, bitmapped display, fonts, multimedia documents, printers, ethernet connection and even e-mail capabilities [61]. From this point in time up to the present day, the following historical events are well known by many. Therefore, these events will not be discussed further.

1.4 History of Programming Languages

The history of electronic computers is almost indistinguishable from the history of programming languages [62, 63]. To mention one without the other means to discuss the chicken without the egg. The very first computers where indeed instructed directly by using the machine code without the help of any other more sophisticated programming languages. The reason for which programming languages exist today and existed early on in the history of computers was to provide an easy interaction with these machines [64]. Under the umbrella of the slogan "*without computers we are nothing*", we can continue with a compressed history of programming and scripting languages which provided a real impact over our society.

1.4.1 The Making of an Advanced Civilization

The mid 40s brought Plankalkül (Plan Calculus), the proposal for the first high-level programming language (before WWII, somewhere between 1933 and 1945) [64–66]. However, the real beginning of the history of programming languages starts in the late 40s when the assembly language was first used as a type of computer programming language in order to simplify the interaction between scientists and computers (after WWII) [67]. This was the very first time a computer language was an intermediary between humans and the machine code [68–70]. Once assembly language was born, the evolution of programming languages was almost exponential for two or three decades. Early in the 50s, "Autocode" was the name of the first computer programming language that was able to compile the high level instructions into machine code [71, 72]. Late in the 50s, it was an explosion of important programming languages with historical implications. A computer language called "FORTRAN" (Formula Translator) was made in order to be useful in scientific research and engineering [73]. Also, in 1958 "Algol" appeared as the first algorithmic programming language, and it is known today as the main root for computer languages such as "Java" and "C" [74]. However, it appears that important parts of "Algol" came from the "Plankalkül" programming language, proposed in the 30 s and the beginning of the 40s [75, 76]. At the very end of the 50s, namely in 1959, two other computer languages where born. The first one to be mentioned is the computer language "COBOL" (Common Business Oriented Language) designed with the intent to be universal among all computers [77]. The second one, is a programming language called "LISP" which was made with the intent of being useful in a special field of research called artificial intelligence [78]. The 60s was the quiet decade, nevertheless, it witnessed the birth of a historical programming language called "BASIC" (1964). The "BASIC" (Beginner All-purpose Symbolic Instruction Code) family of languages is the main line of programming and scripting languages that has led to our civilization of today, as the "BASIC" family made computers highly popular and accessible to all [79]. The world owes much to "BASIC", if not all. The 70s, was perhaps the most important decade, with implications that echo in our current times. In 1970 the programming language "PASCAL" was developed [80]. Also born in the 70s, it is one of the most respected high-level programming languages, which has set a number of standards in the field of software development. In 1972, this programming language it was called "C", and it was considered a true intermediary between machine code and humans [81]. In the same year, namely 1972, the "SQL" computer language appears as a solution for data manipulation in databases [82]. In the late 70s, a computer language called "MATLAB" is released as a solution for science and education [83]. In the 80s, with small exceptions, programming languages start to look like and feel like the programming languages of today. In 1983, two important programming languages appear on the stage, one is the programming language "C++" and the other one is the programming language "Objective-C" [84, 85]. In the late 80s, namely in

1987 to be more precise, the scripting language "PERL" appears as a solution for reporting automatization [86, 87]. The 90s where the golden years of computer languages. The first half of the 90s, has seen an explosion of new computer languages that are used and continuously developed up to this day. The programming language "Haskell" was developed in 1990 as a functional computer language for higher level mathematics [88, 89]. The year 1991 witnessed the addition of two important computer languages, namely the scripting language Python and the programming language Visual Basic [90–94]. Of note is the fact that the two computer languages contain a similar kind of syntax. Also, the computer language "R" is published in 1993 as a solution for mathematicians and statisticians [95]. From the perspective of our current time, the year 1995 was perhaps the most important year of the 90s, mainly because it witnessed the addition on the market of four different and important computer languages. In 1995 the programming language "Java" was born [96]. In the same year, the scripting language "PHP" appears as a solution for web development [97]. The computer language "Ruby" appears also in 1995 as a general purpose language [98]. Most important of all, the year 1995 brings the scripting language JavaScript which today is the main scripting language for the Internet browsers [99–101]. Worth mentioning here, is the fact that JavaScript quickly becomes important today as an environment independent from the internet browsers, step-by-step taking more and more visible and diverse roles in science and all over the industry. Between the year 1995 and the year 2000 nothing spectacular happened in regard to the history of programming languages, except perhaps the publication of the last true programming language in the basic family of programming languages, namely Visual Basic 6.0 [102, 103]. In the year 2000, "C Sharp" or "C#" comes into existence as a version of "Java". Since 2000, there have been several programming and scripting languages worth mentioning, namely "SCALA" and "Groovy" in 2003, the computer language "GO" in 2009, and "Swift" in 2014 [104].

1.4.2 The Dark Age of Computer Languages

Nevertheless, the period after 2000 can be considered the great void in the history of programming languages. The reason for this void was the lack of impact on the market for any new spectacular innovation. However, in order to give historical credit for the time period between 2000-present, one can say that programming and scripting languages that appeared in the 90s have been perfected and improved according to the needs of the industry and the needs of scientific research. One example of such a computer language is JavaScript which by far it had a constant improvement over the years; some improvements that were necessary with the times, and other improvements that were simply intelligent and visionary. The good practices of JavaScript development across the years consist in the way the syntax was strongly preserved. On the other side of history is a computer language called Visual Basic. By the late 90s, Visual Basic 6.0 (VB6) was on the rise, quickly becoming the favorite package for Rapid Application Development (RAD). The

Visual Basic syntax strategically covered a few key environments, namely Visual Basic Scripting (VBS), Visual Basic for Applications (VBA) and of course the "VB 6.0" programming language, which appeared on the market in 1998. Between 1998 and 2002, Visual Basic 6.0 quickly became the most well received and used programming language around the world. Visual Basic 6.0 is a programming language that was so advanced at the time and so perfectly designed in the mid 90s, that it needed no update across the decades. Moreover, it is used today by a large group of programmers for scientific research and advanced technologies in the industry. Unfortunately, Visual Basic 6.0 was the very last version of BASIC up to this day, the last programming language capable of true compilation in this family line. In 2002, VB.NET was presented as the successor of the Visual Basic 6.0 programming language. Unfortunately, VB.NET radically changed the BASIC syntax and lacked any back compatibility with "VB6" projects. For these reasons, VB.NET has never been well received or used as much as Visual Basic 6.0. To this very day, Visual Basic 6.0 eclipses all subsequent versions of VB.NET, both by the amount of open source projects found online and the pool of programmers available. The paradigm change that happened in 2002 was so important and radical that it allowed the rise of Python and JavaScript. Back then, the Internet browsers allowed two scripting languages to be used for HTML (HyperText Markup Language) pages. The first language was Visual Basic Scripting (VBS) and the second one was JavaScript. With the switch made from Visual Basic 6.0 to VB.NET, the internet order was affected. The "VBS" environment collapsed and it was replaced entirely by JavaScript. Next, the market void made by the announcement of VB.NET allowed the rise of Python and other programming and scripting languages. This partial takeover was not immediate, however, the close syntax resemblance between Visual Basic 6.0 and Python made this transition possible. To point out the importance of the Visual Basic 6.0 engine, one must look no further than Office Excel or other packages from MS Office that use the Visual Basic for Applications (VBA) scripting language. The example in regard to Visual Basic 6.0 is a lesson for the future in which decision makers must exploit perfect technologies instead of just destroying them in the name of vanity (or trying to). Once the 90s ended, different confusions appeared in regard to what a programming language is. These misunderstandings coming from the industry into the realm of education have been the main reason for the appearance of bloated, memory consuming and slow software development tools. One thing to take away from the above, is that new computer programming languages of today are constructed on the concepts designed for the programming languages of the 90s and even earlier than that. Many old programming and scripting languages from the 90s are used today. Some of these languages are updated and others are not. Time has demonstrated that well designed programming languages or scripting languages rarely require updates, if any. That is, because they are fundamental to all computer related fields. Unfortunately, the will of the industry to increase the number of programmers in the name of equality, leads to new computer languages that have the aim to simplify the work of programmers. Simplification is done to such an extent that engineers become mere users limited by

environmental constraints. This is a generational issue to which a solution must be found in order to avoid stagnation or regression.

1.4.3 The Extraordinary Story of ActiveX

Another ferocious competition in the late 1990s that greatly disrupted the order of the Internet, was the battle for the monopoly of dynamic web pages. At that time, the concept of dynamic HTML page was in its infancy. The most straightforward and logical approach for dynamic web pages was the inclusion of compiled applications and their GUI as foreign objects in the Internet browser window. In other words, these were a kind of primitive-like nested objects with independent execution from the Internet browser. This approach obviously had security issues and the attempt at regulation and standardization lasted over a decade after. Here, two technologies were strongly competing at that time, namely ActiveX technology (".ocx" files) and Java Applets technology (".class" files). Both were compiled objects. However, ActiveX objects were injected as they were into the Internet browser window with a separate execution from the browser. On the other hand, Java Apples were emulated by virtual machines, which in turn were injected into the browser window as objects executed separately from the main browser. Emulation by using virtual machines takes more execution time than the ActiveX approach, however, it provides security.

1.4.4 Killed on Duty by Friendly Fire

With the forced replacement of VB 6.0 and other technologies that produced ActiveX objects, the battle was automatically lost in favor of Java Applets. This unexpected gain of Java Applets following the disappearance of ActiveX, propelled both Java and JavaScript into the future of the Internet. The virtual machine approach has been adopted to date in all Internet browsers because of the security umbrella it automatically provides. The level of power, control and speed offered by ActiveX was obviously similar to any compiled application that runs directly on the Central Processing Unit (CPU). Up to date, no other Internet technology could be as fast as this approach was. We will probably come back to this ActiveX-like methodology someday, both for its simplicity and for the extraordinary speed of execution. At this moment in time, WebAssembly (".wasm" files) seems to be the closest secure version of what ActiveX used to be. WebAssembly is a promising technology, advertising a near-native speed of execution and JavaScript control by using API.

1.4.5 The Browser: Resistance is Futile, You Will be Assimilated

Later in this development, other technologies appeared to be similar to Java Applets, like for instance "Flash" Applets (".swf" files) that used a dedicated scripting language called "ActionScript". In 2008, the importance of Flash and Java Applets diminished greatly when standardisation allowed for such virtual machines to be embedded by default into the HTML5. Thus, in the end, Cascading Style Sheets (CSS) and JavaScript replaced all these pioneering technologies by integration of these ideas into the HTML standards. Third-party object extinction was perhaps an inevitable outcome. The first indirect blow to applet technology was a method called Asynchronous JavaScript and XML (Ajax), which allowed an exchange (still does) of data between browsers and servers without a HTML page reload. Until the advent of Ajax in 2005, all data exchanges between Internet browsers and Internet servers were accomplished through a complete reload of the HTML source code. The above is perhaps one of the short presentations on the process of technological evolution that shows how the world's most complex applications came to exist by natural selection, namely the great Internet browsers. In other words, what worked it was adopted into the standards, and what didn't was forgotten. Ultimately, our technology is an extension of our evolution. Into the future, a possible assimilation of the operating systems by Internet browsers will not surprise anyone. I personally wonder, do we really need the whole operating system to be a separate entity? Or can we boot up the browser and have a desktop tab inside of it? Of course, there have been several successful attempts to develop Internet browser operating systems in the past, but will this be the standard? We shall see. In the following chapters of this work, discussions based on experience will continue these observations related to software on several levels, from entropy to psychology and finally to discussions about the human nature.

1.5 Conclusions

The whole reason for the existence of any high-level programming language is to make a bridge between humans and the machine code. Thus, the code is readable by other programmers and can be maintained more easily over economically driven time periods. This chapter revealed a path that highlighted the main fundamental concepts, starting from primitive mechanical technology and arriving to the advanced general-purpose computers of today. In this compact journey, automatons have been described as the backbone of early mechanical computers which laid the foundation of later developments. With the advancements in electromagnetism, mechanical switches become relays. Thus, the electromechanical computers were then presented as the main bridge from mechanical computers to electronic computers. Several stages have been mentioned in the case of electronic computers, namely the short era of thermionic valves (vacuum tubes), the era of transistors and finally the era of integrated circuits that extends to the present day. Next,

a detailed presentation was made about the history of programming languages and their evolution. At the very end, the future dangers of hyper simplification of programming languages are discussed. In this regard, our dependency on software technology in all aspects of life, more likely will only increase the interest on programming and scripting languages of the past in order to avoid a loss of know-how.

Philosophy and Discussions

2

2.1 Introduction

Philosophical discussions are becoming increasingly important for understanding the impact of computer languages on the future evolution of our civilization. Much time has passed since software became an integral part of the environment in which we live. Through many observations over decades, clear conclusions have been drawn regarding software applications, which could not have been initially foreseen. Mainly, it was possible to observe the entropy that is applied both in the construction and maintenance of software applications and especially the entropy resulting from software activity over time (i.e. operating systems). Here we discuss in detail the communality of systems, be they electronic or biological. The chapter explains how the human condition introduces noise into the structure of complex software applications. By the middle of the chapter, the life cycle of software applications is discussed in the context of harsh realities of the Internet. Thus, based on experience, rare topics are described, which refer to technical measures and psychological warfare used in unfair business practices. Towards the end of the chapter, the roles and dilemmas in the industry regarding the pool of human resources are explained in reasonable detail.

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/978-3-031-23277-0_2.

P. A. Gagniuc, *An Introduction to Programming Languages: Simultaneous Learning in Multiple Coding Environments*, Synthesis Lectures on Computer Science,
https://doi.org/10.1007/978-3-031-23277-0_2

2.2 The Entropy of Software

The life cycle of software is a very interesting phenomenon related to the increase of entropy into the system (i.e. of the software application). In other words, we naturally witness the phenomenon of aging even in the case of simple systems such as software. From everyone's experience, we have been able to observe how later versions of software applications are becoming slower and larger in size, with increasing computing power requirements. The question that arises in these situations would be: how can an application that does primarily the same thing in all versions, require more and more hardware resources? This paradox has been observed from the Integrated Development Environments (IDEs) of programming languages up to Internet browsers. Of course there are many factors that participate to this phenomenon. The bottom line is that software applications are aging with us. Their entropy has two major causes: the aging of programmers and the number of programmers working on that project over time.

2.2.1 Entropy of Codes and Human Nature

When the frequency of software errors increases for an application, it is then clear that entropy is increasing beyond comprehension. The "cancer" that can occur over time in any software project consists in the intimate intersection of functionalities between the original modules, conceived at the beginning of the project in a visionary way by the software designers. This intersection of functionality occurs when there is either a lack of interest or a lack of time, or, too little experience or ability on the part of the programmers. There is also the lack of attachment to projects, which of course participates in the increase of entropy. Functional mixing (code shortcuts) done for the sake of quickly finishing a task in an application, leads to the appearance of unnecessary code and the loss of clear functional identity among the software modules, which in turn leads to the uncontrollable increase in entropy, and inevitably to the end of the project.

2.2.2 Raw Versus Fine-Grained Entropy

There is a Romanian saying that states: "*beat the iron while it is hot*". The same is true with the minds of programmers. Those who start a software project are able to complete it with the lowest entropy. Also, it is widely known that "*adding manpower to a late software project makes it later*" [105]. However, time is only one factor in this enterprise. What is worth adding to the above is that programmers who later resume pieces of projects from others, will inevitably add imperceptible noise into the source code. Over time, this noise is produced by small inconsistencies between the original meaning of the project initiators and the wishful interpretations made by the project successors.

Moreover, the heterogeneous coding style of each individual and the subjectivity of code interpretation between individuals, increases the entropy from one task to another. To reduce the entropy of software applications, teams of programmers apply a process that explains itself by name, namely: *code refactoring*, where optimization is usually done by eliminating redundancies. In short, *code refactoring* means changing the structure of the code without changing the external behavior of the software [106]. However, this process locally lowers the overall entropy and introduces a fine-grained entropy into the system as a whole [107, 108].

2.2.3 How Does Software Entropy Increase?

In a complex project, programmers are interested, as they should be, in the input–output part of the code they are modifying. However, the problem arises and amplifies with the complexity of the code. A line is drawn for the structure of a software application from the beginning by the initial designers. This line cannot be perceived by other programmers working on that application over time. Thus, additions, deletions or code changes will depend on the immediate functionality of these applications which is understood locally by the programmer. A local understanding of the problem automatically leads to a continuous deviation from the original plan thought by the designers. Also, in the case of teams, desynchronizations that appear in the communication between programmers, also participates in this increase of the entropy. These subtle deviations as well as the attempt to correct them, do nothing but increase the entropy of the application. Clarity of mind is the most important added value that a person can bring to any project. But even this extraordinary quality of people cannot stop the software entropy due to unknowns in the system, which require time to be known. Unfortunately, time is one of the main currencies that balances the economy. Therefore, the pressure exerted by managers on programmers leads to a certain rise in the entropy of software applications. For these reasons noticeable effects can be observed, like the increase in consumption of resources and a lower and lower speed of these applications. In time, such an undesirable effect leads to less users that are interested on the software in question. In turn, a declining user base leads to the extinction of the product from the market, namely it leads to the end of the application lifecycle.

2.3 The Operating Systems and Entropy

Entropy manifests itself in everything at any scale above our frame of reference. Thus, there is also a kind of entropy that is related to the use of software rather than an induction of entropy through the programming process. An example that can clarify

the phenomenon of software entropy is related to complex operating systems (OS). The example discussed below was at least once experienced by each of us.

2.3.1 The Twins

Let us imagine two computers alike, and two operating systems alike. The installation of an operating system on the first computer is followed by a shutdown for a period of two years. After the installation of the operating system on the second computer, the machine is continuously used by a user for two years. At the end of this two-year period, the first computer is turned on. Which of the two computers will be faster? Obviously, the first computer with low entropy will respond to commands faster than the second computer. Another question that arises is: Which of the two computers is more important. Here we can clearly say that the second one with high entropy is the important one, because it includes the user experience and data. Here again we can make the association with us, with the people. Young individuals with low entropy are alert but useless without experience, only with potential. Where, individuals who have gone through life have higher entropy and experience, but are not as responsive as the former.

2.3.2 Rejection of Equilibrium

The universality of systems can also be observed again through case association. A good start is the comparison between artificial systems and living organisms. For instance, a question that arises would be: Are biological organisms in equilibrium or not? The answer may initially surprise the reader, but biological organisms are in a permanent disequilibrium. The body of an organism uses energy to maintain itself below the maximum entropy. In contrast, equilibrium occurs with the death (maximum entropy) of the respective organism. The same is true in the case of operating systems as well as in the case of software programs that require updates.

2.3.3 The Third Party Software

Endless temporary files, log files and growing reminiscent folder paths lead to memory granulation until the system makes noticeable lags and maybe occasional crashes. This has happened from the most primitive operating systems of the past to the most complex ones of our time. Thus, entropy problems start early, even for low levels of complexity. An operating system is the environment on which all other software depends on. The primary source of entropy over all operating systems is the human from the “software cast”. Imagine the installation of a number of software applications on the operating

system. Their use on the operating system leaves traces in the registry, temporary files, log files, folder paths and it leaves changes even in the operating system files. Once these software are uninstalled, the entropy of the system lowers only a little bit, but the inevitable harm to the OS is done. It is hard for different teams of developers in this world to keep track of all the changes their software makes to the operating system over time. A proper staged uninstall that can be planned by programmers is extremely difficult or even impossible. Once changes are made to the operating system, other changes will accumulate on top of the old ones in the future due to other software applications. Thus, a correct uninstall can be either extremely complex, or it can be very direct and in the process the OS can be broken (like the uninstall of antivirus engines of the past, where computers were left unbootable). For many, the smartest uninstall solution so far was to leave the traces in the system in order to avoid complications like total OS failure. The use of third party software is like smoking cigarettes, it is useful but also harmful. If one stops using them, the harm is done up to that point in time, because all traces are there. **Note**: What is the meaning of "*third party software*"? The "*first party software*" is the operating system and all applications accompanying it. Thus, the company that designs the operating system makes the "*first party software*". The "*second party software*" refers to the software made by the user of the operating system. The "*third party software*" is any software that is not made by the company of the operating system or by the user of the operating system [109]. Thus, a "*third party software*" is anything outside the operating system or the software built locally on it [110, 111]. This is why the phrase "*third party software*" is so common and the other two phrases are not.

2.3.4 Examples of Universality

In society, this universality of laws is also observable. For example, social pressure leads to an increase in the entropy of the system, that is, to what we call corruption. A clearer example would be the emergence of new taxes and their increase (pressure), which inevitably leads to corruption (raising entropy). Furthermore, the rise of entropy leads to the dilution of identity and finally, destruction. Another simple example is related to cars, the more often the driver accelerates, the shorter the life of the engine. Obviously, these considerations echo in any system because they are governed by universal laws. Unlike biological systems, the destructive journey of any software system can be stopped at any point in time. However, if you are a manager, are you trying to correctly estimate the deadline of a task to reduce the entropy in the software applications that the programmer is working on? The point of the above is to show that entropy in software has multiple and intertwined sources.

2.4 Software Updates and Aging

Software aging is a complex phenomenon that involves many considerations, and one of those considerations is related to the upgrade process [112–114]. Some applications have no requirements for updates. For instance, there are video games or programming languages without any update over time, that are still used today after 20–30 years from their initial design. However, for security applications that depend on malware signatures to keep up with the times, perhaps such updates are indeed a necessity. The issue that appears is not necessarily related to data updates, but with methodology updates, algorithm updates and so on. Moreover, the word "update" is both a psychological word and one of the methods by which applications age. An upgrade consists of a series of steps that include, among many others, processes such as *continuous integration* (CI). *Continuous integration* is the term used for a periodic merger of close versions of source code into a final version [115, 116]. Thus, this process (i.e. CI) is one of the layers accumulating entropy. The more updates one makes, the faster the aging of a software application. In order to make an association, this phenomenon is directly equivalent to the metabolism of the human body or any other living organism. The faster the metabolism, the faster the body ages. A correct dosage of nutrients over the lifetime of an organism allows for the extension of vitality and of life. The point here is that moderate hunger leads to a slower aging of the body. To bring this universality to the extreme, the same happens with the stars. The faster they consume themselves, the brighter they are. Thus, these stars age and disappear faster than others. In the case of software applications, every time something new is added and integrated into the source code, the entropy of the application increases. This entropy translates into higher resource consumption and a noticeable slowdown of the application. Of course, *code refactoring* and other methods, further decrease entropy, but the remaining entropy becomes fine-grained. The fine-grained entropy is even harder to manage in the subsequent additions or modifications of a software. Thus, further elimination of entropy in later code changes becomes increasingly difficult. When it comes to the frequency of software updates, the above examples become relevant. Psychologically, these updates automatically portray trust and demand respect based on the impression that time and paid human resources are heavily invested in these applications. However, let us have a little reminder of operating system entropy and the third-party software implications. It is well known that a specific digital signature is left by each third-party software. Therefore, the signature is a characteristic to that particular software. Updates of third party software build up new signatures over the old signatures of the same software application until entropy is raised to the highest levels. Thus, this is just another mechanism by which entropy is risen. In other words, all interactions with a software that outputs new objects or adds information to old ones, will increase entropy.

2.5 Universality Supports Self-reflection

Perhaps intuition may suggests that software implementations can become noisy when greater complexity is involved. However, please note that noise does not exist in digital systems regardless of complexity, because noise involves the element of randomness which a computer lacks entirely without an input from the outside of the machine (random variables from our universe). I would like to emphasize here that low or high entropy is very different from the meaning of low or high noise. Please read the material indicated in the main text for deeper clarifications on the issue.

The manner in which software has evolved since the dawn of the computer age, may have a direct analogue in our considerations on the emergence of life on our planet. All systems in our universe have common laws that can be observed from any frame of reference. Perhaps a study of how computers have evolved over time could shed light on how life on earth originated. Although life and modern computers are different beasts, they still have one thing in common and one main difference. Commonality is related to discretization. Both biological and digital electronic systems use discrete elements. Atoms, complex molecules and cells are discrete elements. Even thermal energy is discrete (photons). On the other hand, the main difference is that digital systems are noiseless, while life is inherently noisy. This crucial property of digital systems allows us humans to obtain simulations totally separated from the noise-related influences of our universe. Any technology is an evolutionary extension of a natural process, namely biological evolution. A look at the technological evolution directly reflects our own emergence on earth. For more on this subject I refer the reader to a chapter on *Philosophical Transactions* published in "*Algorithms in Bioinformatics: Theory and Implementation*", where these links between biology and software are fully drawn in great detail [1].

2.5.1 The Evolution of Large Brains Versus Entropy

The most basic natural link between human behavior and computers, consists in our ability to see the extremes. Constructs such as Yin & Yang, good and evil, or duality in nature, like pray and predator, light and darkness, are in fact an ancestral behavioral reminiscence of natural repetitive cycles. Perhaps, for this very reason our computers are binary and not otherwise.

Our brains are directly responsible for the amount of entropy introduced into software programs. Since people are different, so the levels of entropy into software are different. In

order to explain the connection between software entropy and people, some assumptions will be made here. We must first deduce how and why we emerged as a species to then understand how entropy is introduced into software technology, which is also an extension of the biological evolutionary process. In short, the smarter the human the lower is the entropy injected into these applications when they are modified. How brains evolved in time and what factors led to their current dimensions is still a mystery. There are a number of hypotheses, but they are all guesses since no one has been able to prove with certainty the reason behind the large skull enlargement. Nevertheless, a short plausible explanation can be given here. Around the disappearance of the large dinosaurs 65 million years ago, mammals experienced a relaxed environment in which to evolve [1]. For some mammals that had multiple predators, the advantageous evolutionary adaptation was an increase in cranial capacity. Perhaps, such an adaptation was made at the expense of other physical traits, with the role of defensive/offensive mechanisms for the countless predators that existed in the past. Thus, greater brain capacity led to the replacement of defensive and offensive physical traits with complex behaviors. Mimicry was the main trait, where for instance the behavior of the predator was copied in order to catch its type/model of prey. Over time, across generations, the preservation of specific mimicry behaviors has been conditioned by survival and cemented in DNA as self-traits. Thus, among other mechanisms of behavior, our brain works by mimicry and association. Later, complex behavior involved stone-throwing, spear-throwing, understanding the importance of a position, designing traps, reading animal behavior. Most important of all in this behavior, was a correct understanding of the complex implications of the positions of other individuals in a given situation, which ultimately led to what we call teamwork. Moreover, each animal was associated with a series of context-independent characteristics, which were further associated with a series of defensive events to ensure survival. Then, these associations turned into meaningful symbols, drawn and then recognized by other individuals. Therefore, the full context of the events is intentionally forgotten because it is irrelevant to the sequence of steps that lead to the desired outcome: survival. In other words, a pattern molded on the situation was maintained. A return to our days may perhaps indicate more about the current setup. When a software is modified by programmers other than the original ones, the context is also removed, due to the way humans have evolved. Thus, among many other elements, this contextless evolutionary path may be one of the roots of entropy for software applications. In contrast to the above, context keeps entropy low.

2.6 From Computer Languages to Art and Sports

The beginning of the computer age was a wide period where mathematics and art intersected, if they were at all different. As stated in the previous chapter, the start of computer era is indeed foggy and it makes more sense with automatons and looms. In the modern

post-transistor era, the art starts to clearly develop with programming languages. Note that I purposely write about programming languages when it comes to art, because scripting languages in the past were not as important or used as they are now. In the past, the hardware requirements for many software applications were truly demanding. This happened not because the applications were complicated, but because the hardware was primitive/limited. To maximize the hardware capabilities of the time, these higher-order software projects required a compilation of the source code into machine code. Once the hardware power increased enough, scripting languages started to be taken seriously. Thus, the art of codes appeared early in the history of programming languages.

2.6.1 The Art

Language produces art, like poetry or other pieces of literature. Thus, computer languages can also produce art by association. The art part has several faces. One of the faces is related to the way a code is formatted. The empty space until the beginning of a line of text is called indentation. The role of indentation is to build nested structures from lines of code in order to be readable by the human brain. Indentations are used as the main form of art in all programming and scripting languages. Another face of art is the optimization of a code in such a way that it is as short as possible and as computationally advantageous as possible. Another face of art allows programmers to actually draw using ASCII characters inside comments (Fig. 2.1). This is the oldest art form in programming languages and it originated in the past when graphical interfaces were only a beautiful dream. The most important aspect is the balance in the art of programming. A source code should maintain a balance between how short it is and how easy it is to be understood by another human. This is really the most subjective and interesting art form where every programmer has a personal style or signature when coding. The same way a writer or a painter is distinguishable by a clear personal style, the same is the case with programmers. Another type of art is the design of adaptable software. That means a visionary programmer is able to predict the future environment of the software application without trial and error. Such applications are simply adaptable to the unforeseen element, like new variables in the medium, or new data, or new type of data in the input of the application. In short, this is the art of dealing with partially unknown input data. This form of art is done mainly by the most experienced programmers. Without the artistic talent, a number of coding strategies that enforce good practices (or safe programming) can be applied to ensure longer lifetimes for software applications. Nevertheless, we will talk about some of these good practices throughout the book. At the “Guru” level of art, this prediction of technology goes beyond a simple dynamic input. It may reach the point where the programmer can intuitively predict the fundamental changes in the future versions of the operating systems that could affect the lifetime of the application (i.e. future API changes in the next versions of the operating system). That, is indeed the gift of logical foresight.

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

2.6.2 The Sport

Just as the gladiators had the *Roman Arena* to gain sympathy and fame, so the programmers have such things in their corner of the world. In the past, computer languages went hand in hand with mathematics and logic competitions. In such competitions, a problem had to be solved as quickly as possible. In other situations, the optimal solution was required within a limited time. At other times, the competitions tested the management capability and cohesion of the teams involved. To be able to complete the project before others did, a task was divided by the team leader into subtasks as best as possible. These types of competitions obviously train young minds and prepared them for the industry in a way. However, these are classical intellectual proprieties of the human mind that were explored in these competitions. The sport in programming languages evolved much since the old days. In modern times, collectively, the upper echelons of civilization wish, as it

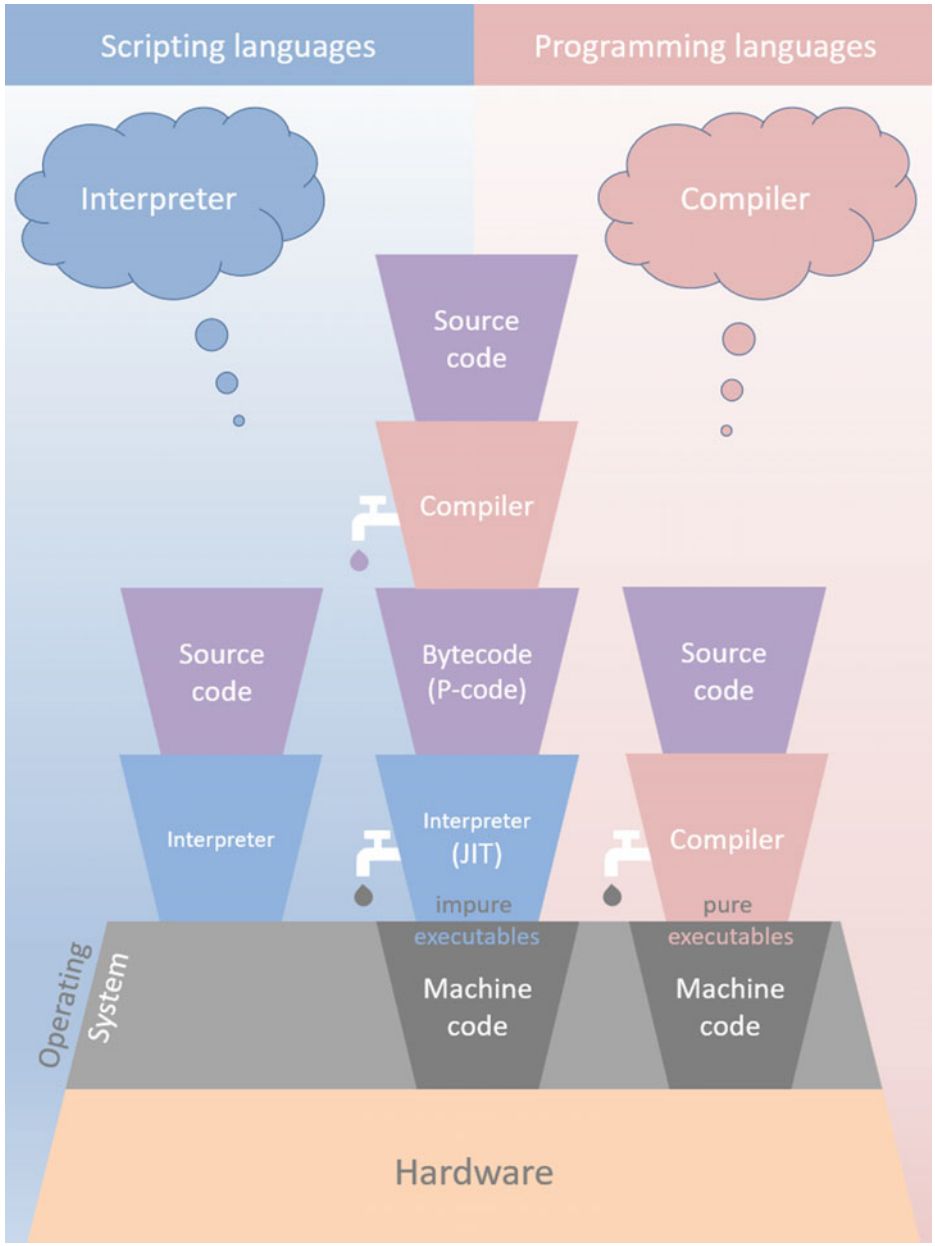
is normal, to finance and see originality in order to gain added value, which can later be used for something that leads to progress. What the industry and education understand about humans in general is that certain conditions appear to be great advantages in this modern era, especially for original thinking and uniqueness. Such conditions like subtle forms of autism or the Tourette syndrome, appear to be provided with new intellectual abilities never fully appreciated until the era of computers.

2.7 Compiled Versus Interpreted

Higher-order computer languages such as C, C++, VB6, Erlang, Haskell, Rust, and Go are some of the examples of compiled languages. On the other hand, computer languages such as JavaScript, Java, C#, Python, VBA, PHP, Ruby, and Perl are examples of common interpreted languages. The main difference between programming languages and classical scripting languages is fundamental. In short, programming languages are able to convert high-level source code into machine code, which in turn is structured further into a binary file (OS specific). Scripting languages use an interpreter application that reads and executes instructions from a high-level source code, which is usually stored further into a simple text file. In other words, an interpreted (scripting) language is unable to protect the source code, while compiled (programming) languages protect the source code by conversion to machine code, which is more difficult for the average programmer to understand. Also, the machine code produced by programming languages is extremely fast, while scripting languages are interpreted and are slower in execution. Such a radical difference between scripting languages and programming languages requires a more extensive discussion from which conclusions can emerge (Fig. 2.2).

2.7.1 Programming Languages

Higher-order compiled languages are platform dependent, that is, hardware and operating system dependent. Compiled languages were always seen as converters, because the compilation process is in fact a dynamic conversion process. Higher-order programming languages are very much like LEGO bricks in which pieces of standard machine code are added dynamically together in order to perform highly complex operations. Thus, programming languages compile/convert source code into machine code, later to be executed directly by the CPU. Programming languages provide source code security through obscurity. In other words, the lack of know-how and complexity of machine code ensures the security of an application even if it is not encrypted. Furthermore, programmers who can reverse engineer machine code for know-how, can make a complex algorithm from scratch without much effort.


Scripting languages
Programming languages
Interpreter
Compiler
Source code
Compiler
Source code
Bytecode (P-code)
Source code
Interpreter
Interpreter (JIT)
Compiler
impure executables
pure executables
Operating System
Machine code
Machine code
Hardware

◀**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

2.7.2 Scripting Languages

Scripting languages contain a module that interprets the source code. This module is widely known as the "interpreter". Most of the time the source code is visible to those interested in it. The source code can obviously be obfuscated in such a way to make theft difficult. Obfuscation is the process of changing the appearance of text while preserving the intended meaning, thereby making the source code difficult to understand. However, obfuscation still allows the source code to be clearly visible. Encryption methods can be used to hide the source code, but ultimately, with or without encryption the source code is visible to an experienced programmer. Interestingly, most of the clever source code protection methods were invented by malware creators. In the case of malware, the idea of obfuscating the source code for scripts was to avoid static signatures of antivirus engines. These engines can be tricked by clever code obfuscation. Many companies were inspired by these methodologies. Therefore, in order to maintain the proprietary inner workings a secret, their software products were sufficiently obfuscated and/or encrypted.

2.7.3 Source Code Encryption

Both programming and scripting languages can encrypt their code. Programming languages can encrypt important pieces of code that the machine can decrypt only for immediate execution, and scripting languages can encrypt source code that can be decrypted piece by piece during execution. Interestingly, the methods by which an important application can protect its code originate from the need of virus designers in the 90s to hide their code from antivirus engines. Moreover, those methods gave birth to polymorphic viruses that are able to encrypt their own body in such a way that it looks different with each infection. Since the body of a virus looks different from infection to infection, malware specialists could provide only a subjective signature to antivirus engines. As a small parenthesis, the core of unencrypted code that had the role of decryption, was usually also the signature used for detection. However, the development of this subject requires an independent book. Obviously, these innovative tactics taken from malware

were finally adopted for security where the source codes were better protected from the curious minds.

2.7.4 The Executable File

An executable file is a computer file that contains data and a sequence of instructions that can be executed on the machine they are activated on. Such instructions can be either in the form of machine code with direct execution on the CPU or in the form of high-level interpreted scripts. Thus, binary machine code files that take the role of executable files can run on any compatible OS without requiring the existence of third party applications. In short, executable files that contain binary machine code are compiled by different programming languages based on the higher-level source code. At execution time, the binary file is interpreted directly by the CPU which in turn drives the other hardware modules. An executable file can be run from a graphical user interface-based operating system (simple events like a double-click) or by a command-line interface-based OS. In some cases, an executable file can be triggered and run passively by other software applications. Depending on the OS manufacturer or OS versions of the same manufacturer, the executable files include file extensions like: .exe, .bat, .com, .cmd, .inf, .ipa, .osx, .pif, .run, .wsh, .app and many other extensions over the existing OSs. However, among these executable file formats, the ".exe" extension is the most known to the general public. Thus, executable files usually refer to the ".exe" file format because of the market presence that Microsoft Windows had over many decades around the world.

2.7.5 Executable Files and Scripting Languages

Classical scripting languages lack any kind of compiler, so they are unable to write actual executables. However, classical scripting languages can simulate/mimic a kind of executable files. The same executables, on the other hand, can deceive the eye at first impression. These files that can be produced by scripting languages are a composite made from the executable of the interpreter and the script itself, both of which are physically part of the final executable file. The method of producing executables by using scripting languages is one that comes, again, from malware designers. There are many classes of viruses. One of these classes is the append or pack method. This version of the virus infects by placing the virus file in front of the infected executable files. Post infection, the icon of the infected file is then injected by certain techniques into the executable file of the virus, to avoid a suspicious look regarding the infected file. When the infected file was executed, the virus was executed first. The virus file then made a copy of the original file from its own body into a temporary file that was automatically executed without the knowledge of the user. Usually this temporary file, which contained the original file of

the user, was written in the same path as the file of the virus (virus + original file). From this basic method of infection it was only a step to the making of executable files through scripting languages. In the same way, the text file containing the source code is appended to a copy of the interpreter, where both make a single executable file at the end. Therefore, scripting languages are unable to write proper executable files. Another example that has its roots in methods invented by malware designers is the Self-extracting archive (SFX or SEA) file produced by WinZip or WinRAR. This type of executable contains the unpacker and the data to be unpacked in a single file without the need to install additional applications. In the end, it really matters how one uses a method and for what purpose.

2.8 The Unseen and Unspoken

Note: There are two main types of approaches to executable files. Third-party applications can be designed either as simple standalone executables with no dependencies (a rare situation) or as a standalone executable installers that can unpack files (including other executable files) and configure changes to the operating system. A standalone executable does not require dependencies on external libraries or components, or entries in the operating system registry. Executable files that require inclusion in the standalone installer usually have dependencies that are present in the same package and requests for changes to the operating system need security privileges.

A discussion about monopoly, centralized power and the real source of many innovations is addressed in this subchapter. Much of what is discussed here is valid for the period 2005-present, and it refers both to human nature in general and to risky strategies put into practice by third parties. The continuous battle of antivirus companies against executable files and the inevitable consequence of revealing software methods through the intentional or unintentional promotion of scripting languages is widely discussed. This push towards scripting languages further led to a full disclosure of "the" work which radically reduced the economic value of many algorithm implementations. The release of software products into the market is discussed from the perspective of the past, and then presented as an experience for the near future. Before much criticism that can be found below, it should be mentioned that both antivirus engines, digital certificates or Internet browser filters are extraordinary tools that lead to progress. However, these methods remain beneficial instruments of order only if there are good intentions of the people behind them, otherwise they become instruments of centralized control. Towards the end of the subchapter, the manipulation of fundamental terms is discussed in relation to different tactics seen often in the field of sociology.

2.8.1 Witch Hunting Shows Weakness

The first question that arises in the case of any software application is: *Cui bono?* or *cui prodest?* The answer to this question seems to show market needs and also it seems to make a prediction for the trajectory of a software project. But is it? Experience can tell unknown stories related to software, incompetence and politics. In order to begin a life cycle, software applications hit various bumps or even entire walls when they are released on the Internet. These obstacles are relatively undiscussed. For example, by using aggressive unscientific or ill-conceived policies, many anti-virus/anti-malware companies have largely succeeded in discouraging programmers from compiling executable files almost entirely. Moreover, the so-called strict control and erroneous detection of normal files as malware, culminated in a lack of trust in antivirus companies. This mistrust was amplified because in the case of these false detections it was impossible even for programmers to distinguish between malicious intent, mistake or pranks made by security companies. In the past, it got so far that anti-viruses were no longer used by people who knew even a little about these security issues. The mockery of the public and the civilized world was so great that the executable files that were compiled by certain programming languages were directly detected as malware without any reason. This is also the case with the *Visual Basic 6.0* programming language, which by default produces 16 Kb executable files that show a simple interface. Even those default 16 Kb executables have been classified as malware at some point. Another problem that led to disaster was the introduction of the possibility for antivirus users to decide for themselves which executable file is suspicious and which is not. This approach has led to a deepening of distrust in antivirus companies.

2.8.2 No Secrets for the Emeritus

Antivirus companies are responsible for raising the status of scripting languages and therefore for the lack of secrecy of new algorithms and new ideas developed for businesses. Until the destruction of executables as entities with a long history and high importance, all implementations could be, at least in theory, hidden from the eyes of opportunists who wanted to make money out of nothing. After a partial transition to scripting languages, all source code was visible and easy for opportunists to grab and use. Thus, the situation created a vicious circle from which opportunists gained more than deserving people. On the other hand, this visibility of codes led to easy access for know-how without solid foundations. Many could see and use pieces of implementations without knowing much about the path that led to those designs. This approach to technology is similar to a nineteenth century cart that uses German automobile wheels. What can we say? is a definite improvement for the cart!

2.8.3 The War Against the Executable File

On one hand, the war on executable files was waged out of greed and a lack of vision for the future. In the past, antivirus companies have unofficially requested software developers for the purchase and attachment of digital certificates to executables. These certificates indicated to antivirus engines that an executable file should not be scanned. In the end, a digital certificate shows that a software application really originates from a certain verified source. Thus, digital certificates also have advantages. However, this push for the use of digital certificates had numerous consequences for the user base of some programming languages, where their newly compiled executable files were automatically detected as malware for no reason. In a first instance, the positive part about digital certificates would have been a standardization and a gradual optimization of malware signature extraction. Additional grounding of certificates would have led eventually to the elimination of the need for security and security programs in general. Nevertheless, this approach to the use of digital certificates was a utopian one that ultimately led to disaster. Moreover, digital certificates led to a taxation of the software companies and of regular programmers without any legal or moral basis. By association, this is very similar to the mafia demanding protection fees. But what may be the implications of such tactics? In order for progress to continue, the existence of a software on the Internet should perhaps not depend on anyone. The push for the purchase of certificates to sign executable files was ultimately the main action by which antivirus companies drastically reduced their dominance over the Internet. How did this happen? In order to run their own commercial executables in certain isolated ecosystems, people gradually started to uninstall antiviruses. Also, false detections of newly compiled files that had default and clean functionality, eventually led to a mass awareness of the uselessness of many antivirus engines. On the other hand, the war on executable files had a positive effect also. The "strict control" of executable files has minimized direct access to the CPU, memory and the setup of the OS, which has led to a reduction or total elimination of errors generated by third party software applications. However, the downside of this positive effect was the loss of low-level knowledge of operating systems by the general population of programmers. In other words, less people know today how to interact with the internals of operating systems because the executable file is more exclusivist now than ever before. It is true that OS also have evolved and become more complex, but not complex enough to explain or excuse the above.

2.8.4 We Decide What Product Comes About

Both antivirus engines and browser filters have been deciding who is and who is not allowed on the market for over two decades or more. Often, I have seen harmless and intelligently built applications that were categorized as malware for no reason at all. The previous statement of course included my own analysis of the executables in question.

In order to have a life cycle, an application must be born on the market first. A release on the market of a software application that is prevented by various methods such as "confusion" with malware, can delay the start of the life cycle of an application or even stop it completely. The psychological effect can destroy the work done on a product even if the signatures indicating a malware detection are removed. This activity, either of manipulations or mistakes done by security companies can be further discussed in detail for a better understanding of the market environment.

2.9 Psychological Warfare

Whether it is computer languages or other types of software applications, these are ultimately used by people. Their opinions related to certain applications dictate the future of those applications as well as their financial success. Therefore, an important and little-known link to software companies involves the presence of opinion leaders or even the removal of competition through technical means offered by products already on the market, such as antivirus engines or their extensions, internet browser filters. Moreover, these methods may even resort to sociological propaganda tactics often used in political election campaigns. Threat removal, advertising or distortion of terms are the main strategies used to inflict unfair business practices, by a direct sabotage made on the mind of the customer. To understand how these malicious methods work in the wild, a more extensive discussion is necessary.

2.9.1 Removal by Threat

In the case of Internet browser filters, an announcement like "*this file can damage your computer*" is a disaster for the company producing the file mistakenly considered "malware". The psychological impact on the user who downloads the installation file is radical. Even the subsequent removal of the malware signature from the signature database of antivirus engines can no longer cover the damage and scars previously made by these security companies. It is worth mentioning that even now security companies are not regulated and the harm they can do to small and medium-sized software companies around the world is absolutely colossal. Of course, this kind of behavior has certain consequences for security companies with such practices, namely a natural reduction of the userbase to the level of bankruptcy. But these consequences stretch for years and the harm done to third parties remains done, and unpunished. Such unfair business practices have been observed over time by many specialists. In the case of the author, bad practices were observed against my own security product introduced on the market in 2008, called "Scut

Antivirus". Probably, this story of Scut Antivirus deserves in itself a security book in which the technical and psychological tactics applied in the market can be explained in great detail.

2.9.2 Removal by Advertising

The technical and psychological struggle to remove or sustain new applications on the Internet does not end with these "security" methods like antivirus engines or download filters of Internet browsers. These struggles go to the point where some software companies have specially organized offices that include opinion leaders. They deal with Internet comments, videos and even bots that analyze and comment automatically on different platforms. These "entities" are the unseen links between the company producing a software and current or future users. Since all software applications are used by people, their minds are targeted. Whether it is the ordinary user or the engineer who opts for a software solution, the psychological dangers of persuasion and indoctrination in regard to a product are the same. Masked praises of the one product and the defamation of other products by using false comparisons, are the main tactics behind these opinion bureaus.

2.9.3 Handling of Terms

Software applications can have a well-defined and rather short life compared to the life of a programmer. Defamation of the image of a worthy application has financial implications for the producing company, but not for the entire industry. However, the fundamental terms used in research and industry are laws that support the entire field of computer science. Therefore, the terms used in research and industry must represent the truth because they have a particularly long life. Unfortunately, the manipulation of terms is probably the most serious and disastrous consequence of psychological manipulations in computer science. Because of these competitive manipulation tactics, other areas began to be affected, such as the field of research in computer science. Here, one of the clear examples that can be portrayed is related to the term "*Artificial Intelligence*". Artificial Intelligence (AI) is a field of research and not a reality. Thus, neural networks are often called "*Artificial Intelligence*", but these algorithms are deterministic and their regulatory values are molded according to the desire in the output. This modeling of the neural network environment has nothing to do with intelligence. To look at neural networks by using associations from nature, we can imagine a landscape like a hill, a spring at the top of the hill and a house somewhere next to the hill. Initially, spring water flows naturally along the paths traced by the small imperfections in the landscape of the hill and the water reaches the base of the hill far from the house. Repeated attempts to divert the river towards the house can be associated with the training of a neural network. The position

of the house being the desire from the output and the deviation of the trench can be seen as a training process for this model. Thus, we can understand that neural networks are not intelligent at all. Over time, the term "AI" applied to neural networks began to be used for the sole purpose of economic impact rather than the field of research it represented.

2.9.4 Battle of Computer Languages

The battle of the programming languages is an old one, and it appeals to the frustrations of programmers who have more experience in one computer language than another. Thus, until the age of wisdom, it is about all of us (42 years old = age of wisdom, perhaps). Tactics applied to software products are also used to manage the userbase of computer languages. Thus, frustrated programmers are the victims of it. On the internet one can see titles like "*top X programming languages of 20XX*" or "*Best programming language of 20XX*" or even better, targeted questions such as: "*Which is the best programming language?*". All these titles weight great impact over the new generations, and represent gross psychological manipulations for the intentional push of a certain computer language in front of the others in order to increase the userbase. Of course, these tactics apply to any product, from socks to fighter jets, regardless. However, the prestige of computer languages is also a target of these manipulation methods. More subtle and credible manipulations include well-structured comments on various online platforms with the aim of advertising or defaming other competing products. In such comments, the main ingredients are often sentences like: "*X programming language is a toy language*", or "*X programming language makes spaghetti code*", or "*X is not a modern programming language*", or categorical injections such as "*there is no such thing as…*". After all, programming languages are also software applications and obey the same rules. In my subjective opinion, these tactics started to be used excessively from around 2007–2008. Since that time, these actions have led to competing companies hiring opinion leaders to destroy the userbase of the competition. They most likely took the model from Non-Governmental Organizations (NGOs) that do exactly the same things but usually for other purposes and for longer periods of time. In the future, the excessive continuation of these tactics can destroy not only the Information Technology (IT) field but even the society in which we live, because the tendency is towards monopoly, and monopoly can lead to unitary decisions. The question that naturally arises would be: why unitary decisions are not desirable in this case? In short, without a backup mechanism, the first wrong decision affects the entire system irreversibly most of the time. However, a clear example is best for these kinds of situations.

2.9.5 Uniformity Means Death

The best examples are made by association. Thus, an example related to politics and society is in order. In the case of former communist countries, such as Romania, the fall of the system was dampened by trade with neighboring countries. This was the backup system of all communist countries. However, much more severe was the fall in the case of the massive entity called the Soviet Union, where the neighboring countries hardly cushioned the fall of the system. Considering the above, we can imagine a unitary system at the global level. The question is, what happens if this global system makes a wrong decision that leads to collapse? What will be the damping mechanisms of civilization? The problem is simple, these mechanisms will not exist and civilization could disappear without a replacement. The above example is related to the ecosystem of programming languages, which can be seen as a kind of independent countries. The more programming and scripting languages there are and the more stable they are, the more the IT field will have a backup system for the existing technologies that make our civilization advanced.

2.9.6 Modern Does Not Mean Better

Again, the thread of discussion is pushed towards the meaning of the terms. New versions of software applications are seen as modern compared with those of the past. However, the word "modern" is often confused with the word "better". The two words of course have nothing in common. Thus, a new version of software can actually be worse than an older one. For instance, an example comes to mind regarding the recent history of programming languages. From 1998 to around 2005 (maybe even 2007), Visual Basic 6.0 was the most popular programming language in the world. To destroy the language reputation and the pool of programmers, classic psychological tactics were applied. Over time, one theme often pushed was the old "*VB6 is not a modern programming language*". The main motivation for these tactics was to force VB6 programmers to VB.NET and the.NET Framework environment. The secondary motivation was the gain of specialists for other communities of other programming languages. It is worth noting that Visual Basic.NET (VB.NET) is not Visual Basic. Since 2002 to present, VB.NET has never been recognized by the software community as Visual Basic. Although it bears the name and the event-driven paradigm, VB.NET has a different syntax than BASIC and is not backwards compatible. Moreover, Microsoft even tried a desperate move to mimic the name of Visual Basic 6.0 and injected the name Visual Basic 9.0 for a later version to imply a continuation of VB. Of course it didn't work, because back compatibility with all VB6 projects is paramount. Today, both VB6 and VBA are used more by industry and science than all versions of VB.NET combined; and more. In time, these psychological tactics against VB6 led to a flight of programmers to Java and Python and a total rejection of VB.NET. Nevertheless, the majority of programmers remained in the VB6 community

and now they are multi-language programmers. The most unexpected, was the annual influx of young programmers into the VB6 community up to this day. The reason is the myriad of advanced open source projects found online, that lure the young minds into the VB6 collective. In the end, a two-decade-old language still seems to be clearly superior to the languages that replaced it in the .NET Framework ecosystem. Moreover, the Component Object Model (COM) of the 90s has proven to be so effective and crucial that could not be easily disregarded by the .NET Framework and most likely will be used in the future in the next generations of Windows. From far away, in regard to programming languages, the period between the late 90s and early 2000s, seems to show a change in human resources at Microsoft. In other words, the visionaries of the mid-90s were replaced by presumptuous bad-decision makers of the 2000s. As a generality, spending energy to cover up self-mediocrity or some fatal mistakes of the past out of simple frustration, are mistakes of their own in the end. Using the energy to cover it up with something better, is the healthy way forward. The elimination of frustration removes all evils in the human nature. Thus, perhaps the important humans are those that are frustration-resistant, but, few of us are.

2.9.7 Market Share Demands Responsibility

The term "market share" represents the percentage of total sales in a given market. Such a market may represent a class of products. Thus, the higher the market share, the higher the influence and gain. Any company whose market share on a certain product exceeds a certain percentage, has or must have a responsibility towards civilization. Here, I would like to refer primarily to companies that build and maintain operating systems. Thus, the moral dilemma should be: When is a software company allowed to filter the applications that run on its own operating system? As indicated in the other subchapters, this can be easily done by manipulating security signatures for third party files. The balance between market share and the right to decisions of this kind should perhaps be regulated in the future. A software company should no longer have any right to decide which applications run on its own operating system. Why? A very large market share clearly means that the whole (or a large part of) civilization depends on that OS. The right to run an application can no longer be owned by the company producing the operating system because it can easily affect both the natural development of civilization and the appearance of software solutions that lead to the maintenance and/or development of civilization. This would be the reason why an antivirus engine should not exist by default on the operating system, except at the express request of the user. This moral dilemma can be even further amplified if the security engine and the OS are made by the same company. Ultimately, the market share of a company can be correlated with the responsibility towards civilization and therefore regulated by a specialized consortium.

2.10 Human Roles and Dilemmas

The word "computer programmer" or "programmer", shows old roots in the field of IT and points to highly advanced "Guru" type individuals in the art of software. A computer programmer writes and tests the source code for new software projects, is able to debug and update existing applications, rewrites the source code for different operating systems, and last but not least, it secures the applications against possible cyber security threats. In the past, technological diversity was low enough and relatively easy to understand for most technical people. The increase in demands for software technology, either for automation or for other tasks with economic implications, has led to a granular specialization of technical people and this trend will probably continue with a higher complexity of the civilization.

2.10.1 The Identity Crisis

In current times, the rapid advancement in human resources caused an identity crisis for programmers [117]. Thus, the programmer is subdivided into roles with names and definitions that are relatively difficult to digest. The names of these titles did not come naturally into being and are most often imposed by different companies due to a division of work among larger teams. Thus, today we can hear quite confusing terms for the main title of programmer, such as: coder, software designer, software engineer, software developer, software architect and so on. To better understand what these titles mean, a relative description can be made as follows: ***Coder***—a kind of contractor that can be seen as a Joker, namely as an unpredictable misunderstood rebel. This kind is either unexpected as a beautiful surprise, or inexperienced and useless, but in the making. ***Software developers***—it should represent individuals that follow a design, namely the specifications for a software project. They are the ones that have the ability to understand the problem and implement it as desired. The skeleton of an application is dictated by developers. All world implementations in information technology rest on the shoulders of these developers. ***Software designer***—it should represent an individual who does the design of a software, namely the specification of a software. ***Software architect***—A software architect is responsible for high-level design choices related to the overall system structure, behavior and most of all integration. These are the people who are responsible, among other things, for keeping software entropy at low levels. ***Software engineer***—It is the "modern" term for a programmer. Software engineering involves design, development, testing, and maintenance of software applications. ***Programmer***—this title contains a halo that is above all others. Firstly, it is an old title. Secondly, the word itself refers to natively gifted individuals of the "Guru" type that are well versed in the assembly language and everything algorithm related. Being called a programmer is a big deal, and it refers to all titles from the above in one package. However, all these invented titles have no meaning

in the end because in addition to know-how, the highest quality that can be brought to a software project is clarity of mind. The second extraordinary quality that a programmer can have is to navigate through the code of other programmers in order to fix any errors or to update it further. The first activity is called debugging. Not all programmers can take such a task with ease. Both activities are similar to the verbal communication skills between two individuals. Some are more adept at conversations and others are less well spoken. The same is true in the case of debugging or when creating an update. Titles will come and go, however, the word programmer will stay as it signifies the root and the future of the field.

2.10.2 Work Environments

These titles can be included in different specialized work environments that have the role of decreasing the execution time for software projects. Also, these specialized work divisions are able to help in the fair distribution of the workload among the teams from different departments. Thus, these previously discussed titles for a programmer can fit into work environments such as *front-end*, *back-end* or *full-stack*. But what is the meaning of these terms? First we must start with a simple definition in regards to augmentation. To be relatively broad in meaning, one can say that augmentation is an interactive experience of a living being with an artificial machine. The "front" part refers to man–machine augmentation. This interaction, as the name "front" indicates, means "in front of something". Since all objects that we interact with on a computer are virtual, this is more of an augmented reality. The "back" terminology refers to a remote machine, namely "a machine in the back, far away from this one". Thus, everything in this terminology is seen from the perspective of the front-user. The remote machine is the one that receives, processes and transmits data to the machine that performs the augmentation. This setup is known as *front-end* for the client-side, and *back-end* for the server-side (remote machine). Accordingly, there are role titles such as *Front-end Developers*, *Back-end Developers*, and *Full-stack Developers* which includes both. These terms are usually found in reference to web development, but the categories can be extended to any setup that includes a local computer and a remote computer. For example, on the *front-end* we could include Desktop developers or Mobile developers and so on, whereas any machine with server software can be included on the *back-end* section.

2.10.3 Genus: *Homo*

Before the engineering era we had seen the rise of *Homo abstractus*, then with more and more need for applications we have witnessed the apparition of *Homo discretus.* With the era of computers, we had seen the rise of *Homo computatus*, the latest in the genus. What

the future will reserve for our civilisation is hard to predict, it most likely be either a slow appearance of *Homo ignorantes* and the end of civilisation with a gross loss of knowhow, or a bright future of *Homo ameliorates*. Note that none of these species really exist, they are just a mind-game for the reader.

2.11 Worst Professors Are Those Who Assume

The worst professors are those who assume the student knows. In the past, BASIC source code, on the other hand, assumed nothing and was the best professor to many of us. This subchapter tells a personal story of how a long friendship with computers began, which I wish to share with the reader. Today, when someone starts learning programming, one clearly understands what it is doing and why, under the principle learn-to-gain. But in the days of the past, things were very different. In 1989, just before the Romanian coup d'état happened, the hardware gap between the communist Romania and the West was of about 2–4 years (by my subjective appreciation). In the mid-90s, the gap between the West and us was of almost a decade, and the trend continued as all know-how, factories and social order were almost completely wiped out. What our people worked for in the past was now gone. Anyway, because of this technological gap, I meet the computers of the 70 and 80s based on the Z80 microprocessor with the BASIC programming language as default. In the early 80s, a special version of computers were designed in Romania under the name HC-85. Released on the market in the mid-80s, the HC-85 contained the Z80 microprocessor, 16 Kb Erasable Programmable Read-Only Memory (EPROM) and 48 Kb Random Access Memory (RAM). In short, it was probably inspired by and compatible with the *Spectrum* series (ZX Spectrum) of 8-bit home personal computers developed by *Sinclair Research*. The Z80 CPU clock was around 3.5 MHz, which was a lot. One can just imagine, a capability of 3.5 million instructions per second. In the late 80s, Z80 reached clock ticks of 8 MHz by the end of development of this CPU series. In the 90s, when I had Z80 and BASIC, the rest of the world, even in Romania, already had Pentium (586) computers with CPU clock around 66 MHz, and the MS-DOS operating system. As a small note, "Pentium" was a name indicating the fifth generation of $\times 86$ architecture-compatible microprocessors produced by Intel. Now, with above 3 GHz CPU clock ticks and multiple cores, the speed of processing is inimaginable for the majority of us. A clock speed of 3 GHz means 3 billion operations per second or 3000 million operations per second (3,000,000,000 Cycles Per Second, or "cps"). Today the clock speed of CPUs is so high that nobody even bothers talking about these astonishing facts of our times. Nevertheless, my personal journey into the world of software begins in the mid-90s with a bug in a game called "Savage" (probably made in the early or mid-80s), which was loaded into the computer using audio tapes. In case of errors, the source code of certain games was clearly displayed on the screen and it was also editable. Those who managed to fix the bug, I still remember today, could type *Load ""* and the game would restart with

the changes made over the source code. In my case it was a monkey-type modification in which I deleted the name of the game "Savage" and replaced it with the word "Paul". When restarting, instead of the animation that zoomed in and zoomed out on the word "Savage", there was the word "Paul" instead. From here followed a period of three weeks where I did nothing more than to tap like a monkey in that software, learning with a dictionary, the word "FOR", "TO", "LOOP", "DO", "GOTO" and so on. That "Savage" game changed my life. One addition to my shame, was the human–computer interaction with the HC-85. It was brutal, through punches or love for the keyboard (that was the whole computer) due to my frustration on the lack of knowledge or any kind of literature on the matter. Today, literature exists, and what we need is intellectual ability, will and the calling to it.

2.12 Conclusions

The raising of complexity in the realm of software has required the presentation of a wider context. Explanations about programming languages included philosophical discussions based on experience. The effects of software entropy are visible as software complexity increases. A close connection between biology and software was the one argued here. Thus, this chapter made a link between entropy in software applications and human nature. There was also a talk of the increase and decrease of entropy in third party applications. Using several examples, it has been shown how the update activities and operation of an application can reflect universal rules that are widely observable in different frames of reference. The information technology (IT) world is further presented through the prism of art, which has taken root since the stage of transistor computers. Also, the main difference between programming and scripting languages was described, as well as the role and types of executable files. Towards the end of the chapter, a discussion about politics and civilization sparkled some debates on the implications for the world of software. The same implications were shown for human resources where the role and the environment of the programmer was presented in the context of industry and research.

Paradigms and Concepts

3

3.1 Introduction

Context is important and the meaning of some terms can be of great assistance to the reader. Therefore, this chapter presents a useful introduction to programming paradigms, as well as a brief description of the computer languages that are covered in this work. The focus is made on imperative and declarative programming. Next, the computer languages used here are listed and described in the following order: C#, C++, Java, JavaScript, Perl, PHP, Python, Ruby and Visual Basic. Also, the principles behind computer language classification are presented objectively, with both concrete explanations and some criticism. Consequently, a connection is further made between paradigms, computer languages and their syntax. With reference to security, the fundamental composition of operating systems is weighted against the makeup of standard third-party applications. In order to empathize this contrast in a meaningful way, the differences between compiled languages and interpreted languages are presented at length. Interpreted languages pushed further discussion on virtual machines, bytecode and the meaning of "*Just In Time Compilation*". At the end of the chapter a technical introduction is made by a simple "*hello world*" in all computer languages, which also prepares the reader for the chapters ahead.

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/978-3-031-23277-0_3.

P. A. Gagniuc, *An Introduction to Programming Languages: Simultaneous Learning in Multiple Coding Environments*, Synthesis Lectures on Computer Science,
https://doi.org/10.1007/978-3-031-23277-0_3

3.2 The Story of Programming Paradigms

There are two main approaches to computer programming, namely the style of imperative programming and declarative programming. Best-known imperative computer languages are: ALGOL, Assembler, BASIC, C#, C++, C, COBOL, Fortran, Java, Pascal, Python, Ruby. On the other hand, declarative programming languages are less known to the general public and to some extent to many programmers. Declarative programming languages include names such as: Erlang, Haskell, Lisp, Miranda, Prolog and even SQL. Hybrid approaches that allow at least two paradigms are computer languages like: Perl, Java, Ruby, Scala, JavaScript, and other computer languages, as this seems to be a constant trend for the future. Today, we can witness a timid shift from imperative programming to declarative programming. For instance, even old languages like Java are being updated with the ability to support functional programming.

3.2.1 Imperative Programming

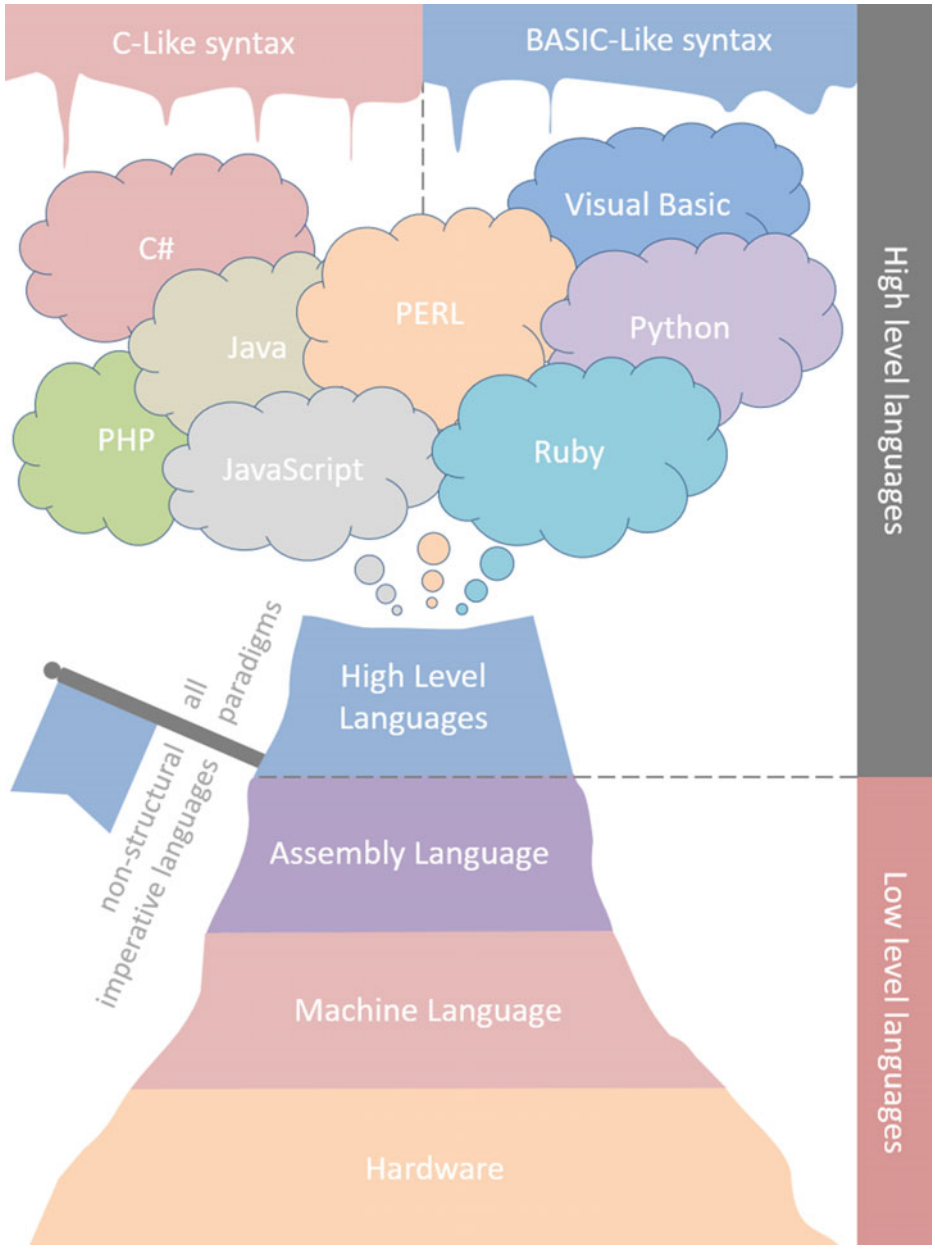
The meaning of the word "imperative" comes from "imperare", the Latin word for "command". Imperative programming is much like kitchen recipes and indicates how to reach the result. Therefore, defines control flow as statements that change the program state. It is the oldest type of programming and the most important that we have. Imperative programming is characterized mainly by the assignment of values to variables. Imperative languages are also divided into non-structural languages and structural languages (Fig. 3.1).

3.2.1.1 Non-structural Imperative Languages

Non-structural imperative languages are dependent on absolute jump commands. Absolute jump commands are instructions that lead to processing continuing at another point instead of the next command. For example, the assembly language is imperative and unstructured (Fig. 3.1).

3.2.1.2 Structural Imperative Languages

On the other hand, structural imperative languages are not dependent on absolute jump commands and make use of control loops and structures. These use modern approaches and their point is to avoid the complexity of jumps. After compilation, these become essentially non-structural.


C-Like syntax
BASIC-Like syntax
C#
Visual Basic
PERL
Java
Python
PHP
JavaScript
Ruby
High level languages
all paradigms
High Level Languages
non-structural imperative languages
Assembly Language
Machine Language
Hardware
Low level languages

◀**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

3.2.1.3 Procedural Programming

Procedural programming includes structural imperative languages and non-structural imperative languages. The procedural programming is the division of algorithms into manageable sections referred to as sub-programs, routines, or functions. Procedural programming was a crucial step in the prevention of human induced redundancies, namely unnecessary code repetitions.

3.2.1.4 Event-Driven Programming

Event-driven programming is somewhere above the classical paradigms mentioned here. However, event-driven including time-driven programming is a very important paradigm that has led to *Rapid Application Development* (RAD). In the *event-driven* programming, hardware events trigger pieces of code that often represent the virtual extension of the user (ex. keyboard, mouse or non-user extensions like timers, etc.).

In other words, hardware events are bound to objects, which is why the paradigm is called *event-driven.* Computer languages like JavaScript, Visual Basic, Visual FoxPro or Visual C++, are only some of the examples that can be given as representative of this programming style.

3.2.1.5 Object Oriented Programming

As the name suggests, instead of logic and function, this approach revolves around objects (data structures) that contain data fields and methods. Some of the principles of object-oriented (OO) programming include what is described as encapsulation, abstraction,

inheritance, and polymorphism. Encapsulation protects data and functions from improper access from outside of the object. In abstraction, only the internal mechanisms that are relevant for the use of the object are shown. Inheritance refers to a hierarchical reuse of code. Pieces of new objects are added to parts of older objects to form a more complex object. The last of these well-known basic principles of object-oriented programming is polymorphism. The term polymorphism has its origins in the Greek word "*polymorphos*", which means "*many forms*" (*poly*: many; *morphos*: shapes). First uses of the word *polymorphism* originated in biology, and the term is now often used in genetics rather than information technology (IT). However, in computer science, namely in object-oriented computer languages, polymorphism represents context-dependent behavior.

3.2.2 Declarative Programming

The name comes from "declarare", the Latin word for "describe". There are two main approaches to declarative programming, namely functional programming and logical programming. Behaviourally, a declarative style is highly satisfying and it gives the programmer the impression he cheated the universe a little bit, which is in fact true.

3.2.2.1 Functional Programming

> To understand and work with functional programming in a hands-on manner, please open an Excel sheet! The relationship made between the formulas inside the cells of an Excel sheet is in fact functional programming.

Functional programming is based on function calls, as the name says. Programming is done by applying arguments to functions. Thus, programming is mainly based on arguments. Arguments are the values injected into the input of the function. This type of programming provides an answer to the question: what result do you want? It focuses on what to execute, defines program logic, but not detailed control flow. The source code of declarative programming is not straight forward and natural for humans, however, it is short. It should be noted that functional programming is based on pure functions that lack the ability to change the state of the program. The previous statement sounds truly vague, however, clarifications are in order: (1) In pure functions the same inputs lead to the same outputs. (2) Pure functions have no side effects. A side effect refers to the change of attributes of the program outside the function, which are made from the interior of the function. In other words, it refers to the change of attributes of the program that are not contained within the function itself. Thus, side effects are changing the state of the program. For instance, a function is impure if it modifies the value of variables that are outside (global variables) the function. Also, a function can be impure if it uses, besides the arguments, some In/Out stream values or some random values for the output of the

function. The lack of side effects allows for a clean software with a lower complexity level. Functional programming, by association, is similar to the structural programming, where the level of complexity is lowered by the removal of absolute jumps. Thus, both paradigms allow in the end for a decrease in entropy of software implementations. Note that at machine level nothing really changes and all of this is helpful for the human mind only. Nonetheless, pure and impure functions are discussed with examples in the next chapters, where functions are presented in detail from simple to complex.

3.2.2.2 Logical Programming

Logic programming is a programming paradigm seen by many as an abstract model of computation. In such computer languages, program statements express facts and rules that are written as logical clauses with a head and a body. Logic programming is valuable for complex reasoning which is difficult for humans to evaluate. PROgramming LOGic (Prolog) is among the programming languages that use this paradigm. Logic programming can be summarized as a method of evaluation for the relationships between abstract objects.

3.2.3 The in Between

The question is: Can functional programming exist in imperative languages? The answer is an obvious yes. Imperative languages have the ability to formulate a set of functions and even have default built-in functions that can be used as functional programming. Thus, imperative languages can be declarative also. The problem of declarative languages is one related to the future, but not in a positive way. Declarative languages and declarative programming in general is particularly useful and greatly optimizes working time, but in the future it can have a very real consequence in terms of know-how for imperative programming. Declarative programming is situated at the borderline between programming languages and regular applications. To force an association, instead of the user pressing the buttons, he writes line by line which buttons should be pressed. Thus, declarative programming is a hard-to-do oversimplification of programming languages. Whether it is a declarative or imperative approach, programming or scripting languages make use of both. Today, in practice programmers no longer realize what the difference is. It should be noted that in-deep discussions about programming paradigms are always highly subjective.

3.2.4 The Foundation

One detail to always remember is that machine code is imperative because the hardware requires it. This is why, fundamentally, everything is imperative. However, having a declarative model as the base of computing would be a possibility, especially for the field of computer research. Hardware can be built to be modeled on declarative programming. Instead of the classic CPU, one can imagine hardware structures that represent different types of complex functions. I can go so far to say that neural networks may represent

pieces of such declarative technology. By extension, our own brains perhaps are a kind of declarative system, and yet, we build imperatively.

3.3 Computer Languages Used Here

Prior to a brief description, some clarifications about computer languages are needed. In general, the syntax of high-level imperative computer languages is divided in two major types, namely: the *C-Like* syntax and the *BASIC-Like* syntax. Two common terms are encountered when it comes to computer languages, namely: *compilers* for programming languages and *interpreters* for scripting languages. Classical compilers convert source code into machine language which is OS-dependent. On the other hand, scripting languages of today are more sophisticated than the older versions. Scripting languages convert their source code to byte code, which is then converted, as the program runs, into machine instructions. Such examples can be found in computer languages like Python, PERL and others. One may notice that Java works in the same way and here it is called a programming language. In the following descriptions, computer languages are referred to as they originally began with their providers, namely as scripting or programming languages. Today, the line between programming and scripting languages is a little more blurry than in the past.

3.3.1 C#

C# (pronounced "C sharp") is a general-purpose programming language first released in 2002. Like Java, C# too is a multi-paradigm language able to support object-oriented and functional programming. C sharp syntax is like Java and therefore is similar to C and C++. Because C# depends heavily on the .NET framework, the language can be described as a cross-platform environment.

3.3.2 C++

The original version of C++ was first released in 1979. "C with classes" was the original terminology used for what we know today as the C++ computer language, and it was an extension of the C programming language. However, "C with classes" was renamed to C++ much later in 1983. The "++" represents the increment operator and it points out that C++ is version C plus one (please see the chapter related to operators). Thus, C++ roughly means "one step higher than C". Today, C++ is a general-purpose programming language

and a multi-paradigm environment able to support object-oriented and functional programming. As expected, C++ is a cross-platform environment and is the root syntax for most languages in use today.

3.3.3 Java

Java is a class-based, object-oriented programming language first created in 1995. Java syntax is similar to C and C++. It is a cross platform environment (Windows, Mac, Linux, Raspberry Pi, etc.), and is able to make a variety of implementations on these platforms. It is used for desktop applications, mobile applications, web applications, server applications, games and so on. Java made history because of Java apps used for web pages.

3.3.4 JavaScript

JavaScript is a scripting language first created in 1995, and today is the most used computer language in the world. The front-end of the Internet covers about 99% of all machines that are considered general purpose computers. Thus, with few exceptions, the Internet rests on the shoulders of JavaScript. This scripting language can be described as a cross-platform and cross-browser environment. JavaScript is also a classic example of multi-paradigm language that supports imperative, object-oriented, and functional programming (it has first-class functions). The basic syntax is similar to both Java and C++.

3.3.5 Perl

Perl is a general-purpose scripting language first created in 1987 for Unix. Today, Perl is a cross platform computer language, used for critical projects in the public and private sectors. Perl syntax is often mistaken with Raku, a new computer language derived from Perl. Perl is procedural in nature, however, it supports object-oriented, procedural and functional programming. Perl syntax has been accused by some of having “line noise”, meaning that it uses special characters in front of variables and when longer expressions are arranged on a single line, it looks like a random sequence of characters. However, this “line noise” can be achieved in any language to a certain extent, and usually the style is pushed by the most advanced programmers. Perl is among the next generation scripting languages along with Python and others, that are in between classical scripting languages and programing languages. Text manipulation capabilities and rapid development cycle were Perl’s trademarks. In the past, it is possible that many of the methods for text manipulations found at the time in Perl, have been tacitly adopted in other environments.

3.3.6 PHP

PHP is a general-purpose scripting language designed in 1994. It is a multi-paradigm language with a C-Like syntax. It encompasses paradigms such as imperative, functional, object-oriented, procedural and reflective styles. PHP runs on all important platforms (Windows, Linux, Unix, MacOS, etc.). Generally, it is used on web servers as a daemon or as a Common Gateway Interface (CGI) executable. The role of PHP is not reserved only for back-end applications. Front-end applications are also possible, but outside of the original scope of PHP. In contrast to JavaScript, PHP is for at least two decades the other pillar of the Internet.

3.3.7 Python

Python is a scripting language first created in 1991. Like the great JavaScript, Python too is a multi-paradigm language able to support object-oriented and functional programming. It is a cross platform environment (Windows, Mac, Linux, etc.) and is able to make a variety of implementations on these platforms. Python has a basic-like syntax with mandatory indentation. The success of Python is owed in part to a vacuum left by Visual Basic 6.0 in mid-2000s. Both Python and Visual Basic were released in 1991. Today, Python is a special scripting language at the borderline with programming languages that uses ancient malware tactics to speed up its execution. It uses the conversion of bytecode into binary machine code.

3.3.8 Ruby

Ruby is a scripting language first released in 1995. The language is developed for GNU/Linux, however, today it is a cross platform environment that works on Windows, MacOS, DOS, Unix, etc. It supports multiple programming paradigms. Ruby uses both functional programming and imperative programming. In Ruby, everything is an object, including primitive data types. Like Python, the syntax of Ruby is very close to the BASIC syntax. The lesson here is: If one knows BASIC than one easily understands Python or Ruby.

3.3.9 Visual Basic

Visual Basic was first created in 1991. It is a multi-paradigm language and is able to support object-oriented programming and multi-threaded applications. Visual Basic comprises of the most recent version, namely Visual Basic 6.0 (VB6) and Visual Basic for

Applications (VBA). Both of them have the same syntax, however, VBA is a scripting language and VB6 is a programming language. VBA is used throughout the Microsoft Office package and is the most valuable scripting language in all companies in the world since its inception. The level of automation that can be achieved with VBA exceeds all expectations, especially for *Excel* or *PowerPoint*. VB6 was for almost a decade the most popular programming language in the world and it was and still partially is used for critical projects in the public and private sectors. Today it is known as the prototyping language because the BASIC syntax allows one to focus on the problem rather than the syntax to be formulated. VB6 is designed for Windows but can also work on Linux through the well known "Wine" environment.

3.4 Classification Can be Misleading

One likes to strictly categorize information for a better exposure of important facts to our peers. This need to categorize is deeply ingrained in the human nature. Therefore, the classification made above, cannot be that contrasting in reality (ex. programming vs scripting languages). Why? one may ask. Fundamentally, all these computer languages are roughly composed of syntax + semantics + toolset. Syntax defines the structure and grammatical rules in a computer language, whereas semantics defines the meaning of different combinations of words and symbols. There are many beautiful definitions for what semantics is. Here a new definition can be tried: Semantics is the assignment of meaning to transitions between groups of symbols in the syntax. The point of the above is that in the end, all that matters is the toolset built around the syntax. If a compiler is built for that syntax, then it is a programming language, and if an interpreter is built instead, then it is a scripting language. If the toolset contains both a compiler and an interpreter, then it can be part of both categories. With knowledge, any computer language that has full access to the Random Access Memory (RAM) and the Solid State Drive (SSD)/Hard Disk Drive (HDD), can create another computer language. Thus, high-level languages can recreate low-level languages or new high-level languages. Moreover, mixed applications can be made using multiple programming languages and intermediate communication media, from strings as messages between processes to whole complex objects. Even files and registry values can be used as messaging media.

3.4.1 A Critique

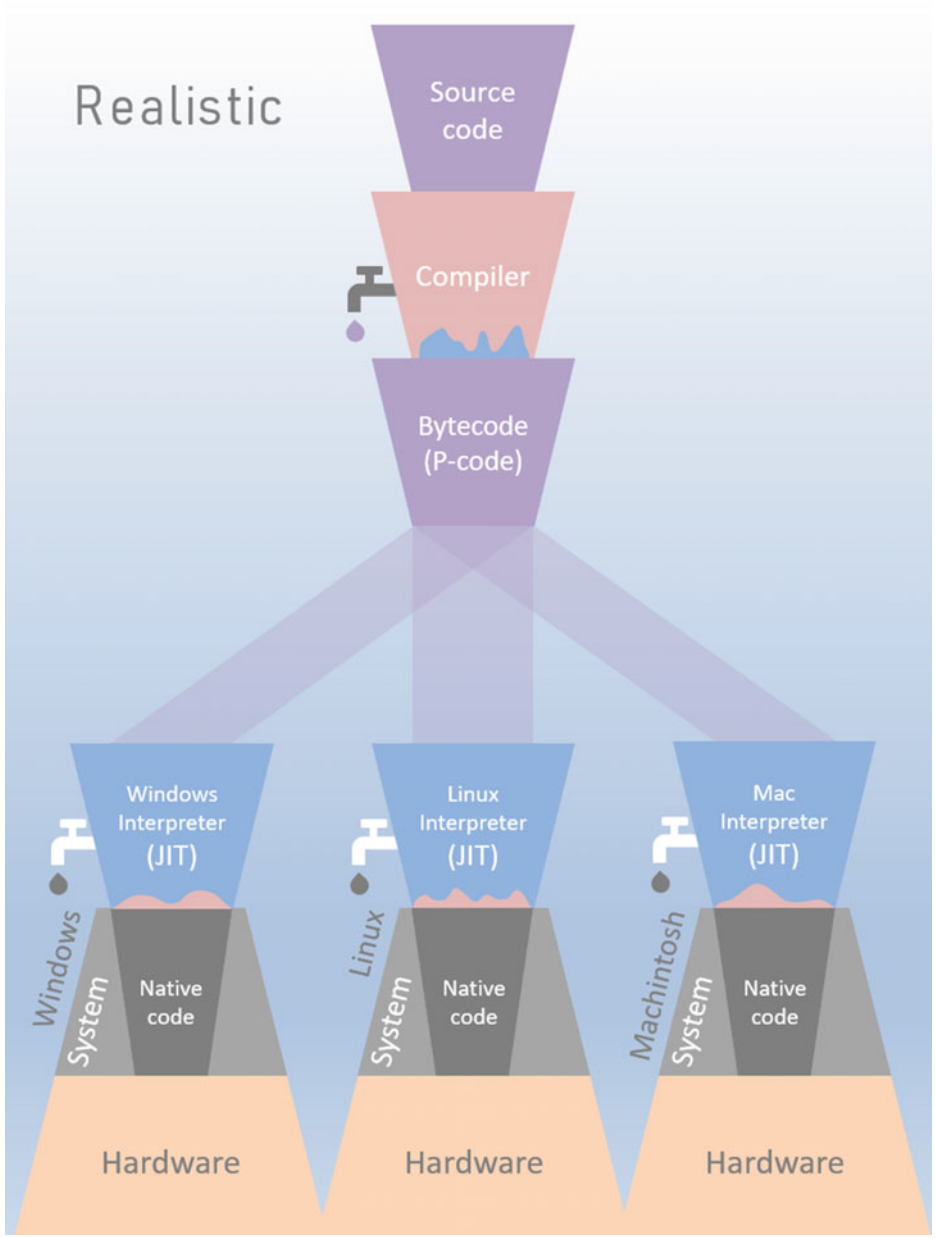
All the power of compiled languages and the respect for these tools comes from the possibility of creating software applications without dependencies or with as few dependencies as possible. In contrast to the above, interpreted languages are fully dependent on the

presence of other applications. On one hand, because of the way operating systems are built and equipped today, these application dependencies that expand from interpreted languages are less and less visible to the average developer. Nevertheless, these dependencies exist under different forms, such as: *Java Virtual Machine* (JVM) of *Java Runtime Environment* (JRE), *Common Language Runtime* (CLR) of *.NET framework*, *Android RunTime* (ART), or other representative names for an application runtime environment (Fig. 3.2). On the other hand, the lack of dependencies allowed compiled languages to be associated with freedom and power. A dependency difference between interpreter-based and compiled-based applications has raised other issues in the programming community, such as the sense of belonging to different sides. Psychologically, it has become ingrained that programming work is done in programming languages due to the similarity of terminology between "programming" and "programming languages".

To avoid this association of terms, companies have started changing titles over time, from programmers to something that cannot be associated with programming or scripting, something neutral like "developer" for example. Moreover, to unfocus the world of software from programming languages, the same situation applies to the word "programming", which is now accompanied by the synonymous word "coding". However, it is good to clarify that in both programming languages and scripting languages, a programmer makes software programming. As a critique, the wannabe scripting languages that constantly aspired to be called programming languages instead of what they are, now are found in a futile position because of JavaScript. The wonderful JavaScript is by far the most powerful computer language in the world. It can be added that it has no direct access to hardware and the security measures isolate it from the operating system. Yet, it continues to be called a scripting language with the most well-deserved pride: It drives the biggest slice from the Internet.

3.4.2 Which Computer Language is Better?

Which computer language is better is a subjective question without a clear answer, because there are particular situations in which one computer language behaves better than another computer language. In general, all computer languages are very close in value and the paradigms used are often screens used as an excuse to demonstrate some kind of superiority. In a more relaxed manner, it can be said that there are no big differences between computer languages, and differences that exist are significant, as previously mentioned, only in certain specific circumstances. Moreover, the access to the Application Programming Interface (API) of the operating system makes these computer languages almost equally powerful. Please note that programming based on APIs is a paradigm in itself. In fact, the API paradigm may reside under declarative programming, because the programmer indicates to the operating system what result it wants. As previously stated, this declarative paradigm is "*cheating the universe*". In the end, perhaps it is better to use


Realistic
Source code
Compiler
Bytecode (P-code)
Windows Interpreter (JIT)
Linux Interpreter (JIT)
Mac Interpreter (JIT)
Windows System
Linux System
Machintosh System
Native code
Native code
Native code
Hardware
Hardware
Hardware

◀**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

the API, than to use a bad implementation that endangers the whole system. It's cleaner that way.

3.4.3 The Operating System Versus the Application Makeup

The world has less programing languages than previously believed. If one wonders why today there are no more fatal errors because of third party software applications, is that all applications are in fact safely interpreted. Interpretation has a positive side indeed. Third party software should not have the same makeup as the operating system files. This is of course debatable. If the prevalence of scripting languages is bad or good, it remains to be seen. However, perhaps it is a very good idea to detach all non-OS applications from the OS make-up. Until a few years ago, both system files and third-party applications were found on equal footing (still are and will be if need be). Thus, many OS errors were frequently seen. With interpreted computer languages these system errors are less frequent today because the make-up of third party applications is not the same as the compiled files of the operating system. With interpreted computer languages, the new configuration (make-up) is the application runtime environment and bytecode, which is then safely compiled or converted step by step to machine code and executed concomitantly.

3.4.4 The Virtual Machine: A CPU for Bytecode

A virtual machine (VM) is a software application that is able to partition the hardware resources of a physical computer. Where the hardware allows, a computer can run multiple instances of virtual machines. Thus, a virtual machine can be called a guest machine and the physical computer it runs on can be called a host machine. The VM mentioned above is of the complex type, from which a nested general purpose computer emerges. Thus, a VM software application is capable of hosting an entire operating system. VMs come in different shapes, forms and separate concepts. Thus, other VMs are much simpler and their existence has a specific purpose, less complex than that discussed above. For instance, such a VM can be a part of a runtime application. VMs can include either emulation, virtualization, or both. Most VMs apply a healthy combination of both where appropriate, by using emulation and/or virtualization. Note that emulation means a software simulation of hardware, while virtualization means a software allocation of hardware resources to a software layer separate from the main operating system. Unlike VMs which deal with complex hardware virtualization and/or emulation, the VM of a runtime application centers around the interpretation of bytecode. The instruction set that makes up the bytecode is universally compatible with this abstract CPU (i.e. VM), which is required to function the same on every physical machine. In a broad sense, the VM is a software processor driven by bytecode, just as a physical CPU is driven by machine code.

3.4.5 Compiled Languages

Classic compilers, or pure compilers, convert high-level source code into object code (machine code) that is inspected and integrated by a linker into an executable file that is platform-specific. Compiled languages need compilation before execution. Any changes require new compilations. Executable files provided by classical compilers run directly on the host machine and provide control over various hardware aspects such as memory management and/or CPU usage.

3.4.6 Interpreted Languages

Impure compilers convert high-level source code into pseudo code, which is then interpreted by an interpreter to run an application. The pseudo code is often abbreviated as P-code and the term has been replaced by the word bytecode. Bytecode is platform neutral and helps in the famous "*write once, run anywhere*" approach for cross-platform execution. A Runtime Environment (RE) usually includes a virtual machine that allows for high portability because it can be designed to exist on multiple operating systems. Thus, the same virtual machine is tailored to the type of platform it runs on. A virtual machine is the bytecode interpreter and runs on a linear instruction flow. To make a reference to

a specific computer language, let us consider Java. Those using Java, will write source code to be ultimately used by the Java Virtual Machine. Nonetheless, some computer languages can have both compiled and interpreted implementations. However, most of the time those with the compiler option are not pure compilers in the classical sense.

3.4.7 Just in Time Compilation

Just In Time Compilation (JIT) represents the virtualization part of VM. JIT is an essential part of the runtime environment and it refers to a performance driven compilation made during the execution of a bytecode-based software application. JIT compilation translates bytecode into machine code instructions that are optimized for the CPU architecture of the host machine. Examples of JIT environments include: Java Virtual Machine (JVM), Common Language Runtime (CLR) or Android RunTime (ART), and others.

3.4.8 Another Critique

Most sources describe the process of converting high-level source code to bytecode as the "compilation process". But is it? The conversion of high-level source code to bytecode is called compilation because the reasoning behind it is that it compiles for an abstract CPU, namely the virtual machine. Compilation classically means "*the end of the line*", as we see it with pure compilers. However, bytecode (pseudocode or P-code) is not "*the end of the line*". Bytecode is the generalized version of the implementation found in the initial high level source code. Thus, the translation from source code to bytecode is a conversion with interpretation rather than a classical compilation. Based on the above rationale, the term "compilation" of the source code to bytecode is relative and improper. The moment of conversion and compression of bytecode into machine code would represent both interpretation and true compilation, as it adheres to the classical meaning of the term found in pure compilers. Thus, as a critique, I would have to say the reference system should be upside down, where the conversion of source code to bytecode can be called "interpretation" and the conversion of bytecode to machine code can be called "interpretation and compilation". At best, computer languages that use this source-bytecode-native approach can be said to be either double-interpreted-compiled, or semi-compiled.

3.4.9 A Security Thought Experiment

Despite a negative position about the attack on binary executables, expressed in the previous chapter (*Philosophy and discussions*), I see some merits in its absence only for certain situations. Bytecode also can be the bridge of security checks without the need for complex implemented privileges in the operating system. In other words, without the pure binary executables, the security privilege check could be done directly by these

application runtime environments. But why do I say this? Pure binary executables provide tremendous power over the operating system when they have administrator permissions. Executable files of this type (i.e. binary) are executed on the operating system as they come from the source without any security checks (with the exception of antiviruses). Let us imagine that in a compiled language, the compiler would be responsible for ensuring the security checks for the operating system on which the binary executable will be run (this is hypothetical). But, this compiler is in the hands of the hacker, who can modify the compiler to bypass these security measures, obviously. Now let us imagine a compilation situation in P-code (bytecode). Let us imagine that in an interpreted language the compiler is also responsible for the security of the operating system on which it will be run (exactly as in the previous example). But, again this compiler is in the hands of the hacker who can modify the compiler to bypass these security measures. However, the P-code (pseudocode) is the end of the line for what the hacker can do. Thus, P-code is the package that will be run on the destination machine, and not a pure executable. The application runtime environment is adapted and made for each operating system. Thus, at the destination, regardless of the operating system, the application runtime environment will interpret the unknown P-code to transform it into compatible machine code for the platform it belongs to. This interpretation of the P-code can no longer be touched by the hacker, and the interpretation made by the application runtime environment from the vendor, can perform security checks before execution. Thus, the security system cannot be manipulated, except by modifying the source code of the application runtime environment at the company that produces it (almost impossible, I should hope).

3.4.10 About Security Privileges

So far an implementation of security privileges over the application runtime has been suggested. But what happens when the privileges are elevated for a pure executable? Pure executables running with administrator privileges have near to absolute power over the operating system. These security considerations are straightforward and administrators can rarely ignore them. The most dangerous situations in security, however, are those of granting privileges by induction. What do I mean by this? In classic scripting languages, the source code is directly interpreted and executed by the application runtime environment without using an intermediate approach like bytecode. If the executable file that represents the application runtime environment holds administrator rights, then any script has full rights to manipulate the operating system through this executable file. An ill-intentioned script application can end up having the powers that any binary executable holds. Thus, a script application with admin rights can be more dangerous than a pure executable with lower privileges.

3.5 The Quick Fix

In order to get more familiar with all the programming and scripting environments used here, a first example is given below. This example is the classical "*Hello World*" shortcut that allows for a quick fix for the very first positive experience. Any of the examples from above should display the '*Hello World*' text in the output of these computer languages (Additional algorithm 3.1). Across the chapters, all the examples will be given in a total of nine programming and scripting languages, namely: JavaScript, C++, C#, VB, PHP, PERL, Ruby, Java, and Python. Depending on the syntax and the constraints imposed by the environment, there are a few variations regarding the way a source code may look like for the same exact result in the output. Note that, unless "<*script*></*script*>" tags are

Lang.	Example

JS	`<script>`
	`  alert("Hello World");`
	`</script>`

C#	`using System;`
	`class HelloWorld`
	`{`
	`  static void Main()`
	`  {`
	`  Console.WriteLine("Hello World");`
	`  }`
	`}`

C++	`#include <iostream>`
	`using namespace std;`
	
	`int main()`
	`{`
	`  cout<<"Hello World";`
	`  return 0;`
	`}`

VB	`Private Sub Form_Load()`
	`  MsgBox "Hello World"`
	`End Sub`

PHP	

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

	<?php echo "Hello World"; ?>
PERL	print "Hello World";
Ruby	puts "Hello World"

Java	public class Main
	{
	public static void main(String[] args)
	{
	System.out.println("Hello World");
	}
	}

Python	print ('Hello World')

Additional algorithm 3.1 (continued)

used, JavaScript implementations in this work were executed using the "*Rhino*" engine (i.e. JavaScript (Rhino) version).

The Integrated Development Environment (IDE) is the main tool of a software developer. It consists of an editor equipped with error handling capabilities and the possibility of two-ways communication with the compiler or/and the interpreter. Any IDE is linked to a console terminal which is able to display the output, namely the errors encountered by the execution process or the data resulting from the execution of the source code. Programming and scripting languages have a specific keyword for displaying data in the output, namely the console of the IDE. For instance, in JavaScript there are two possibilities, either the data can be shown in a pop-up window or in the console log (usually the console log of the browser). In Visual Basic, all versions, the possibilities to show data are similar to those seen in JavaScript, namely a pop-up window or the console.log of the IDE. On the other hand, PHP is a modern scripting language that is usually found on a distant server (a computer). Execution of PHP scripts is done by remote requests or by local executions. Also, the PHP output is displayed either to a local console or to a remote console connected to the server system. In web development, the output of PHP is given to the client, usually an Internet browser, or less often to an IDE of a computer language. In the case of PHP, the keywords used for displaying data in the output, are either *echo* or *print*. The same is true for PERL (*print*), Ruby (*puts*), or Python (*print*). Therefore, with

some exceptions, there is a kind of universality when it comes to the most important keywords across all of these programming and scripting languages. For this reason, source code made in one of the computer languages is portable to other computer languages with ease.

3.6 Conclusions

Most computer languages have the same purpose, namely the instruction of a general purpose computer. However, the substrate on which these high-level computer languages work is fundamental and it makes the difference between freedom and confinement, or more specifically between compiled vs interpreted. Which of the two is primary is a difficult thing to appreciate. On the one hand, freedom (compilation) is risky because it is able to alter established and proven constructs (i.e. the operating system) and push for their improvement or destruction. On the other hand confinement (interpretation) protects established and proven constructs from destruction and is relatively unable to push for their improvement. The real question in this case is: do some constructs require improvement after a certain complexity and reliability is reached? The above context prompted a brief and concise discussion about paradigms and computer languages used in this work, such as: Java, JavaScript, Python, C#, Perl, PHP, Ruby, and Visual Basic. Computer languages have been presented in a subjective light, which casts a shadow over the certainty and contrast of their classification. A brief description of compiled languages outlined the composition of an operating system, whereas a more detailed presentation about interpreted languages showed the makeup of third-party applications. This description made over the interpreted languages covered explanations of runtime applications and their virtual machines, bytecode, and *Just In Time* compilation modules. Also, at the very end of this chapter, the first technical introduction was made with a "*hello world*" approach.

Operators and Expressions

4

In most programming languages, an expression is a combination of operators, constants, and variables (i.e. a piece of code) that evaluates to a value based on a rule of precedence (order of operations). In contrast, a statement refers to action, that is, a piece of code that executes a specific instruction/task.

4.1 Introduction

Our written languages and high-level computer languages obey similar laws because they must be understood by humans. In written languages, formulation of statements requires meaningful words in a well established sequence. In mathematics, some of these meaningful words are represented by specific symbols known as operators, which is why, among others, mathematics is called a language. Computer languages have taken over and used most of these symbols from the field of mathematics, with the goal of formulating statements for machines. Thus, in this chapter, all important operators are discussed in detail and representative examples are provided. Also, the corresponding symbols used in each computer language to represent the operators are shown. More interesting operators are considered for in-depth discussion, such as the power operator, the modulo operator, the unitary operators, the string operator, the repetition operator, and the concatenation operator. Next, three types of assignments are explained, namely the simple assignments, the

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/978-3-031-23277-0_4.

P. A. Gagniuc, *An Introduction to Programming Languages: Simultaneous Learning in Multiple Coding Environments*, Synthesis Lectures on Computer Science,
https://doi.org/10.1007/978-3-031-23277-0_4

aggregate assignments, and the multiple assignments. The chapter further shows examples of operator precedence and associativity, where a table of operator precedence and associativity is built for all general purpose computer languages.

4.2 Operators

Generally, operators are the symbols for action. The word "operator" comes from the Latin word "operari" that means "perform", "action", or "work". Among the normal operators, there are arithmetic, assignment, comparison, logical, and concatenation operators. Of course, there are also special operators, however, these are not discussed here. In the case of operators, special characters are used. These special characters contain symbols used in mathematics that specify the type of calculation to be performed within an expression. Depending on the computer language, the operators may look a little bit different. However, most often than not, these representative characters are chosen by the designers of computer languages; especially if the operators in question are less frequently used or are new.

4.2.1 Arithmetic Operators

The most basic operators are the *arithmetic operators*, which are used to perform frequently needed mathematical operations. This means that the first arithmetic operators to be discussed are: addition, subtraction, multiplication, division, exponentiation, incrementation, decrementation and other special operators like modulo. While all other arithmetic operators use at least two operands, the incrementation and decrementation operators are unary operators that add or subtract 1 from their only operand. As a side note, a unary operation is an operation with only one operand. In the case of numeral systems, a unary numeral system is the simplest numeral system to represent natural numbers. In the case of functions, unary functions are those functions that take only one argument.

4.2.2 Assignment Operators

Assignment operators are used to assign values to variables. The list of all assignment operators is large. With the exception of the equal operator by which the normal assignment is and was done, all other operators are complex and consist of two different characters. Namely, a character usually from the set of arithmetic operators accompanied by the assignment operator with the equal sign. These are called aggregate assignment operators. These compound assignment operators represent a relatively new and sophisticated evolution trait in computer languages, as there are older computer languages that lack support for aggregate assignment (eg. VB6).

4.2.3 Relational Operators

As the name indicates, relational or comparison operators are used to compare two values and return a boolean. Important comparison operators are: *Equal to*, *Not equal*, *Greater than*, *Less than*, *Greater than or equal to*, *Less than or equal to.* With the exception of *Equal*, *Greater than*, and *Less than*, which have deep roots in mathematical notation and are as sacred as the arithmetic operators, the rest of the comparison operators support artistic representations with symbols chosen by the makers of a computer language.

4.2.4 Concatenation Operators

The concatenation operators are able to combine two string values into one. As a method of code optimization, some computer languages use the same symbol for both addition and concatenation. On the other hand other computer languages preserve the separation of the two operations by using different symbols. The two approaches and their important effects are discussed at length in the "*Evaluations of expressions*" subchapter.

4.2.5 Logical Operators

Logical operators are a subset of operators, which are used to connect two or more expressions. For instance, important logical operators are: *equality*, *inequality*, *logical NOT*, *logical AND*, *logical OR*, or conditional selection. These operators are found in the syntax of all programming or scripting languages. The following shows how operators look like in multiple programming and scripting languages.

4.3 Operator Symbols

Up to this point, the types of operators have been presented and discussed without any reference to the representative symbols. The symbols used for these operators in all the computer languages are fully shown in a comprehensive manner in the tables below. In the case of fundamental arithmetic operators, the symbols for addition, subtraction, multiplication and division are very well grounded in mathematics and therefore are shared above all computer languages. Because of their particularities, several interesting operators will be discussed accordingly because of their legacy issues in the field of mathematics or the beginnings of modern computers.

4.3.1 Power Operator: The Curious Case of Exponentiation

Exponentiation is the operation in which a value is multiplied by itself a certain number of times. In high-level computer languages, such an operation is difficult to implement on machines without other "action" operators. As mentioned above, the symbols for basic arithmetic operators are preserved among computer languages and machines (Table 4.1). However, one of the exceptions is the symbol for exponentiation. In mathematics, exponentiation lacks any representative symbol and it is expressed by a base number and the relative top-right position of the power number:

$$base^{power}; b^{p}$$

This is known as "*b* raised to the power of *p*". The lack of an exponentiation operator in mathematics, over time has led to computer languages that represent the exponent by using different types of symbols. It can be rightly said that different symbols for exponentiation create confusion in the minds of all beginners. Most computer languages use the multiplication symbol in a group of two consecutive characters, namely "**". On the other hand, in VB the exponent is represented by a single symbol called a *Caret—circumflex accent*, namely "^". Moreover, computer languages like Java and therefore by automatic extension C#, lack an exponent operator symbol and use the "*Math.pow(base, power)*" method. In all these representations, the BASIC family of computer languages have by far the best representation for this operator.

One thing to note is that the lack of a non-critical operator is usually compensated by a built-in function. Control structures can be used to replace the exponent operator for computer languages that lack built-in functions for such cases (please see the next chapter). Thus, the exponentiation operator can be viewed as a semi-critical operator because it can be computed by using control structures. Other, less critical arithmetic

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

Lang.	Addition	Subtraction	Multiplication	Division	Exponentiation
VB	+	-	*	/	^
JS	+	-	*	/	**
PHP	+	-	*	/	**
PERL	+	-	*	/	**
Ruby	+	-	*	/	**
Python	+	-	*	/	**
Java	+	-	*	/	none
C#	+	-	*	/	none
C++	+	-	*	/	none

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

Lang.	Modulus	Concatenation	Repetition	Increment	Decrement
VB	Mod	&	none	none (+1)	none (-1)
Ruby	%	+	*	none (+1)	none (-1)
Python	%	+	*	none (+1)	none (-1)
PERL	%	.	x	++	--
PHP	%	.	none	++	--
JS	%	+	none	++	--
C++	%	+	none	++	--
C#	%	+	none	++	--
Java	%	+	none	++	--

operators can be discussed, like modulus, increment and decrement (Table 4.2). Here, these are named non-critical operators because some of them can be partially replaced by common mathematical expressions that use critical arithmetic operators. Here, they are called noncritical operators because some of them can be partially replaced by ordinary mathematical expressions that use critical arithmetic operators.

4.3.2 The Modulo Operator

Modulo is one of the most elegant, practical, and less used operators out there. Thus, this beautiful operator deserves some details. The word "modulus" comes from Latin, which means "small measure". Essentially, modulo is actually a measure, as the meaning indicates. Thus, modulo measures the reminder of a division. To provide a graphical example, please consider two sticks: one that is very long and the other one that is very short. The questions that modulo will ask, are: What length remains on the long stick once the short stick is used to measure the long stick?

How many small sticks is the length of the long stick? How many small sticks are needed to cover the long stick? What is the length remaining on the long stick that is less than one small stick? Let us consider that the long stick is 10 centimeters (cm) in length, while the short stick is 4 cm in length. Thus, only two short sticks can cover the long stick, and the remaining length is 2 cm. Please consider a as the long stick and b as the small stick. Then, a relation can be formulated to better explain the modulo operator:

$$r = a - m \times b$$

where r is the remainder and m is the multiplier. Because the order of operations matters, there is no need for parentheses in this expression (the result of the multiplication $m \times b$ is then subtracted from a). Calculation of modulo implies the maximization of the multiplier (m), such that the result of the multiplication ($m \times b$) provides a value as close as possible to a, without exceeding the value of a. Thus, the complex operation described above is written as follows:

$$a \ mod \ b = r$$

Which means:

$$10 \ mod \ 4 = 2$$

Above, number 4 may fit into number 10 only 2 times. Thus, the reminder can be computed as: r = 10 – 2 × 4 = 2. Other helpful examples can be:

$$11 \ mod \ 4 = 3$$

where number 4 may fit into number 11 only 2 times. Thus, the reminder can be computed as: $r = 11 - 2 \times 4 = 3$. In the next example a = 12 and b = 4:

$$12 \ mod \ 4 = 0$$

where number 4 may fit into number 12 exactly 3 times. Thus, the reminder can be computed as: $r = 12 - 3 \times 4 = 0$. In the next example $a = 15$ and $b = 5$:

$$15 \ mod \ 5 = 0$$

where number 5 may fit into number 15 exactly 3 times. Thus, the reminder can be computed as: $r = 15 - 3 \times 5 = 0$. In the next example $a = 3$ and $b = 2$:

$$3 \ mod \ 2 = 1$$

where number 2 may fit into number 3 only 1 time. Thus, the reminder can be computed as: $r = 3 - 1 \times 2 = 1$. In the next example $a = 5$ and $b = 2$:

$$5 \ mod \ 2 = 1$$

In this last example from above, number 2 fits into number 5 only 2 times. Thus, the reminder is: $r = 5 - 2 \times 2 = 1$. The stick example can also be viewed from another angle, namely: Substract b from a until $a < b$: $a = 10$ cm; $b = 4$ cm; $a - b = 6$ cm; $a = 6$ cm; $a - b = 2$ cm; $a = 2$ cm; $a < b$). Notice that a function is required for the above operations, otherwise the modulo operator is difficult to implement. Thus, a self-restriction is required where this operator can be called a semi-noncritical operator

instead of a noncritical operator. In the list of computer languages provided in this work, there are only two representations for the modulo operator. A representation that is closer to the mathematical notation is the 'Mod' character set which stores the symbols for the letter 'M', 'o' and 'd'. The other representation uses only one symbol, namely the percentage symbol "%" (Table 4.2). Sometimes in computer languages not discussed here, the symbol "|" is used, or the "rem" group of characters are used ("rem" is short for the word reminder). In time, the modulo operator was less standardized and less used at the most basic level in different implementations. Because of this, modulo has been expressed in different ways over time. Thus, different individual symbols or groups of symbols were used to represent the modulo operation.

4.3.3 Unitary Operators

Other important non-critical operators are the increment and decrement operators. Some computer languages from the list, like VB, Ruby and Python, take no advantage from the increment and decrement operators, whereas the complementary side of the list does allow from these operators to exist. Computer languages that lack the increment and decrement operators can simply add one or subtract one from a variable (eg. $+1$ or -1 form an integer value). These two antagonist operators can be helpful to avoid the addition of new variables within some professional-like implementations. Examples follow in later chapters.

4.3.4 The String Operator

The last operators that "*are of pure action*", are the concatenation and repetition operators. The concatenation operator is responsible for adding two texts (joining string values) into one and the repetition operator is responsible for duplicating a text (a string value) a number of times.

4.3.5 The Repetition Operator

The repetition operator is a rarity among computer languages, but a future certainty in my subjective opinion. Although any programmer can feel the repetition operator as a naturally occurring operator, it is only present in computer languages such as Ruby, Python and PERL. The repetition operator is represented by multiplication as it makes sense to everyone. However, two different symbols are used, namely: "*" for Ruby and Python, and "x" for PERL, which is actually closer to mathematical notation. Other computer

languages prefer different tactics instead of the repetition operator. For example, a computer language like PHP lacks the repeat operator and instead uses a built-in function (*str_repeat*("*text*",*10*); which repeats the word "*text*" ten times). Computer languages like Java and JavaScript use object methods ("*text*".repeat(10);), instead of the repeat operator. In C#, there is "*string x = new string('text', 10);*", where *x* is a new string variable that contains the word "*text*" ten times. Notice that computer languages that lack an operator use methods to fill the gap. On the other hand, the lack of the repetition operator is understandable, as it has in fact the behavior of a function rather than an operator. Also, is not of critical importance, because it can also be replaced by ordinary mortals which are able to build external functions with control structures inside. However, the repeat operator is very useful to have.

4.3.6 The Concatenation Operator

A string concatenation is the operation of joining one or more sequences of characters into one. In other words, the concatenation of three strings, *a* and *b* and *c*, is the sequence of symbols in *a* followed by the sequence of symbols found in *b*, followed by the sequence of symbols from *c*, and is denoted as:

$$a.b.c$$

Consider that *a* represents the word "*Stephen*", *b* represents the word "*The*", and finally, *c* represents the word "*Great*". This concatenation yields:

$$a.b.c = StephenTheGreat$$

whereas a switch yields:

$$b.c.a = TheGreatStephen$$

Concatenation is also associative, namely:

$$(a.b).c = a.(b.c)$$

Notice that the order of these strings on the left side must be the same as the order of the strings on the right side. This observation is important for the relational operators that are discussed next. The concatenation operator is represented by at least three well known symbols, namely: "and" (&),"dot" (.) and "plus" (+). The "&" symbol is used by the BASIC family of computer languages and is particularly intuitive, while the "." sign has its origins in mathematics and it is used by Ruby and PERL. The other computer

languages use the plus sign "+", both for the addition operation and for the concatenation operation.

4.3.7 Relational and Logical Operators

The last operators discussed here are the relational and logical operators that are crucial for control structures (see next chapter). Relational or comparison operators are able to compare the values of two operands, namely between what is to the left of the operator and whatever is to the right of the operator. On the other hand, logical operators can bind two or more relational operations together. The symbols for the relational operators are universal among computer languages with a few exceptions (Table 4.3). In short, the equality operator is represented in all computer languages (used here) by two characters that encode the equal symbol, namely "==". This is likely done to differentiate between control structures and assignments (Please see the next subchapter for clarification). However, the two symbols side by side have their roots in the field of mathematics and their meaning is "relational operator".

There are two exceptions to this apparent rule. These exceptions point to the BASIC family of computer languages which denote the equality operator by using a single character with the equals symbol. The other exception is the "not equal" relational operator, which in the BASIC family of computer languages is written as "<>" and in other computer languages is written as "!=". In VB, using the *less than* symbol "<" followed by the *greater than* symbol ">" is more intuitive because it implies that if the value is *less than* or *greater than*, it cannot be equal. In contrast to the above meaning, the exclamation point "!" followed the equal symbol "=", directly indicate "not equal". Another

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

Lang.	Equal (E)	Not Equal	Less (L)	Greater (G)	L or E	G or E
PERL	==[eq]	!=[ne]	<[lt]	>[gt]	<=[le]	>=[ge]
VB	=	<>	<	>	<=	>=
JS	==	!=	<	>	<=	>=
C++	==	!=	<	>	<=	>=
C#	==	!=	<	>	<=	>=
PHP	==	!=	<	>	<=	>=
Ruby	==	!=	<	>	<=	>=
Java	==	!=	<	>	<=	>=
Python	==	!=	<	>	<=	>=

feature worth mentioning is that PERL relational operators also have a mirror in groups of two letters. For example, the *equality* operator can be written as "eq", *not equal* as "ne", *less than* as "lt", *greater than* as "gt", *less than or equal to*, as "le", and as probably expected, *greater than or equal to*, is written as "ge". Finally, as mentioned above, the logical operators are the last to be mentioned (Table 4.4). Logical operators either use symbols outside the Latin alphabet or directly use words with the related meaning found in the current language. Thus, for the *NOT operator*, either there is a keyword "Not"/"not" (ex. VB, Python) or the exclamation point is used with the same meaning (ex. all other computer languages in the list). In the case of the *AND operator*, again there is the keyword "And"/"and" (ex. VB, Python, PHP), or the two consecutive "*Ampersand*" symbols "&&", representing the meaning of "and" (ex. all other computer languages in the list, including PHP). Last but not least, the *OR operator* can be represented by the keyword "Or"/"or" (ex. VB, Python, PHP), as well as by the two consecutive vertical line symbols "||" that are semantically linked to the meaning for "or" (ex. all other computer languages in the list, including PHP).

In some dynamically typed languages, such as JavaScript, PHP, and others, the standard equality operator "==" evaluates to true if two values are equal but of different data types. For example, the string value "42" will equal the integer, even if the two values are of a different data type ("42" == 42 evaluates to true). A typed equality operator "===" exists, which is returning true only for values with identical data types (in PHP, 42 === "42" is false, but the evaluation of 42 == "42" is true). In other words, this equality operator ("===") returns true if the two values are exactly the same. Programming languages like VB are able to differentiate between data types and values simply by using the standard equal sign "=". In Ruby, the "===" operator indicates set relationships instead of the meaning it has in JavaScript or PHP.

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

Lang.	Not	And	Or
VB	`Not`	`And`	`Or`
Python	`not`	`and`	`or`
PHP	`!`	`&&[and]`	`||[or]`
JS	`!`	`&&`	`||`
C++	`!`	`&&`	`||`
C#	`!`	`&&`	`||`
PERL	`!`	`&&`	`||`
Ruby	`!`	`&&`	`||`
Java	`!`	`&&`	`||`

4.4 Assignments

In normal circumstances, assignments represent a transfer of responsibility. In computer science, however, assignments are a kind of association between one "thing" and another "thing". In other words, an assignment is an association between a memory address (i.e. where data is found) and a name (i.e. variable, constant, function, and so on). Another angle from which this term can be viewed is like a bridge of meaning between abstract objects. The first question that arises is: How the assignments are normally made in high-level computer languages?, and what other variants of assignments may exist? and why ? Therefore, this subchapter shows three main types of asignments, namely: simple assignments, aggregate assignments and multiple assignments.

4.4.1 Simple Assignments

The assignment of values to variables is at the core of high-level computer languages and of course, it implies the use of operators. In imperative computer languages, the equal symbol "=" is usually the operator that assigns values to variables. The left operand of the "=" operator is the variable name, whereas the right operand is the value stored in the variable. The right operand can take the shape of a value, a variable, an entire expression, and so on. Examples of simple assignments are shown for multiple computer languages in Additional algorithm 4.1.

Lang.	Simple	Aggregate	Multiple
PERL	`$a = 1;`	`$a += 1;`	`$a = $b = $c = 1;`
PHP	`$a = 1;`	`$a += 1;`	`$a = $b = $c = 1;`
JS	`a = 1;`	`a += 1;`	`a = b = c = 1;`
C++	`a = 1;`	`a += 1;`	`a = b = c = 1;`
C#	`a = 1;`	`a += 1;`	`a = b = c = 1;`
Java	`a = 1;`	`a += 1;`	`a = b = c = 1;`
Ruby	`a = 1`	`a += 1`	`a = b = c = 1`
Python	`a = 1`	`a += 1`	`a = b = c = 1`
VB	`a = 1`	`none (a = a + 1)`	none

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

4.4.2 Aggregate Assignments

Over time, computer languages have evolved to be more and more optimized, both in syntax and the semantics of it. Sometimes, such optimizations make the source code of any computer language more beautiful and precise. This is the case of aggregate assignments which provide a shortcut by combining the assignment operator with some other operation. In other words, a character usually from the set of arithmetic operators is accompanied by the assignment operator (the equal sign). For instance, in the computer language Javascript the "+=" operator performs addition and assignment. As an exemplification, the expression "$x = x + 9$" can be rewritten us "x+ = 9", which greatly simplifies the syntax and the amount of information that a programmer must digest in a specific time frame. In the case of "$x = x + 9$", the x variable will hold the current value plus "9". On the other hand, the expression "$x+ = 9$" means add "9" to whatever it is already stored in x. By using the same rules, the "* =" operator performs multiplication and assignment ("$x* = 9$", which is the same as "$x = x * 9$"), or the "/=" operator performs division and assignment ("$x/ = 9$", which is the same as "$x = x/9$"), and so on. These examples from above may take even expressions. Let us consider another two variables, namely variable r and variable t.

For instance, in the expression "$x+ = r * t$", the variables r and t are evaluated ("$r * t$") and the result is added to whatever value it is already present inside variable x. When these combinations are made between normal assignment and other types of operators, such as logical operators, the meaning can become quite complicated to digest, especially when compared to old habits of the past. However, arithmetic operators and the assignment operator ("=") are the ones most often used in practice for aggregate assignment. Even some special operators can be used in some computer languages. For example, PERL contains a repetition operator ("x"), which can be used to repeat sequences of characters multiple times. For aggregate assignment, the duo is written as "x=". Even string concatenation can be used for aggregate assignment (eg ".="). Also, this type of aggregate assignment strongly resembles the way data is handled in the CPU registers by the assembly language, the intermediary language between machine code and the high level programming languages. Thus, aggregate assignments may add a drop of elegance in the way the code is written. Simple and concise examples of aggregate assignments are shown in Additional algorithm 4.1.

4.4.3 Multiple Assignments

What kind of optimization can be implemented for multiple variables that store the same value? The answer to this question is given by multiple assignments. Usually, variables are declared one by one in a separate manner, namely, on separate lines. Computer languages such as Visual Basic 6.0, Python or JavaScript allow the assignment of a single value to

several variables simultaneously. For instance, let us consider three variables, namely *a*, *b* and *c*. In normal cases, by using the simple assignment method, the following statements are true, namely: "$a = 1$; $b = 1$; $c = 1$;". Note that the semicolon indicates the end of the statement just like in JavaScript, C++, C#, Java and so on. However, by using the multiple assignment method, those three different statements can be reduced to one, such as: "$a = b = c = 1$;". Thus, all three variables have the same value, namely 1. By extension, value 1 can be replaced with a variable. Let us consider variable *d*, namely: "$a = b = c = d$;", were variables *a*, *b* and *c* will take the value stored in variable *d*, whatever that value may be. This reduction is not only a cosmetic optimization, but an optimization that is also reflected at the hardware level in most computer languages. That is, because all three variables (*a*, *b* and *c*) are assigned to the same memory location. This multiple assignment optimization works the same whether a real primitive data type for an integer is involved (ex. VB) or an object that simulates a primitive data type for an integer is involved (ex. Ruby). The multiple assignment optimization as discussed above, is actually an old method. What is new instead, is the assignment of multiple objects to multiple variables. In Python for instance, one can assign multiple objects to multiple variables, namely a one-to-one correspondence like: (*a*, *b*, $c = 3$, "Paul", 1). Thus, an integer with the value "3" is assigned to the variables *a*. A string with the value "Paul" is assigned to the variables *b*. An integer with the value "1" is assigned to the variables *c*. This optimization, however, is really more for syntax cosmetics and shortening the source code, which is otherwise very important.

4.5 Operator Precedence and Associativity

Every computer language is equipped with a well-defined set of operators. These operators are associated with an evaluation order called "precedence". This hierarchy of precedence is directly rooted in the mathematical order of operations. Operators that are higher in this order are evaluated first when compared with those of lower order. Please consider a list of operators in which the top operators are the first to be evaluated above an expression and the lowest operators are the last to be evaluated:

higher precedence

[…]
[/*]
[+−]
[…]

lower precedence

where the three dots in square brackets means the list has continuation above or below. An extended list of operator symbols for each computer language can be seen in Fig. 4.1.

C++/C#		Java		JS		PHP		PERL		Python		Ruby		VB	
OP	A	OP	A	OP	A	OP	A	OP	A	OP	A	OP	A	OP	A
()	L	()	L	()	N	()	N	()	N	()	L	[]\[]=	N	()	N
[]	L	[]	L	++	R	[]	N	++	N	**	R	**	R	^	R
.	L	.	L	--	R	!	R	--	N	*	L	!	R	*	L
->	L	++	L	-- N	R	~	R	-	R	@	L	~	R	/	L
++	L	--	L	!	R	++	R	~	R	/	L	+	R	\	L
--	L	++	R	*	L	--	R	!	R	%	L	-	R	Mod	L
++	R	--	R	/	L	*	L	**	R	//	L	*	L	+	L
--	R	+	R	%	L	/	L	=~	L	+	L	/	L	-	L
+	R	-	R	+	L	%	L	!~	L	-	L	%	L	&	L
-	R	!	R	-	L	+	L	*	L	<<	L	+	L	=	R
!	R	~	R	+ C	R	-	L	/	L	>>	L	-	L	=	L
~	R	*	L	<	L	.	L	%	L	<	L	<<	L	<	L
*	R	/	L	<=	L	<<	L	x	L	<=	L	>>	L	>	L
&	R	%	L	>	L	>>	L	+	L	>		&	L	<=	L
*	L	+	L	>=	L	<	N	-	L	>=		^	L	>=	L
/	L	-	L	==	L	<=	N	.	L	==	L	|	L	<>	L
%	L	<<	L	!=	L	>	N	<<	L	!=	L	<	L	And	L
+	L	>>	L	===	L	>=	N	>>	L	is	L	<=	L	Eqv	L
-	L	>>>	L	!==	L	==	N	-e	N	is not	L	>	L	Imp	L
<<	L	<	L	&&	L	!=	N	-r	N	in		>=	L	Or	L
>>	L	<=	L	||	L	===	N	<	L	not in		<=>	L	Xor	L
<	L	>		?:	R	!===	N	<=	L	&	L	==	L	Not	L
<=	L	>=		=	R	&	L	>	L	^	L	===	L		
>	L	==	L	+=	R	^	L	>=	L	|	L	!=	L		
>=	L	!=	L	-=	R	|	L	lt	L	not	R	=~	L		
==	L	&	L	*=	R	&&	L	le	L	and	L	!~	L		
!=	L	^	L	/=	R	||	L	gt	L	or	L	&&	L		
&	L	|	L	%=	R	?:	R	ge	L	=	R	||	L		
^	L	&&	L	**=	R	=	R	==	L	+=	R	.. …	L		
|	L	||	L			+=	R	!=	L	-=	R	?:	R		
&&	L	?:	R			-=	R	<=>	L	*=	R	=	R		
||	L	=	R			*=	R	eq	L	/=	R	+=	R		
?:	R	+=	R			**=	R	ne	L	%=	R	-=	R		
=	R	-=	R			/=	R	cmp	L	//=	R	*=	R		
+=	R	*=	R			.=	R	&	L	**=	R	/=	R		
-=	R	/=	R			%=	R	|	L	&=	R	%=	R		
*=	R	%=	R			&=	R	^	L	^=	R	**=	R		
/=	R	&=	R			|=	R	&&	L	|=	R	&&=	R		
%=	R	|=	R			^=	R	||	L	<<=	R	||=	R		
&=	R	^=	R			<<=	R	..	L	>>=	R	&=	R		
^=	R	<<=	R			>>=	R	? and :	R			|=	R		
|=	R	>>=	R			??=	R	+=	R			^=	R		
<<=	R					and	L	-=	R			<<=	R		
>>=	R					xor	L	*=	R			>>=	R		
,	L					or	L	/=	R			not	R		
						,	L	%=	R			or	L		
								**=	R			and	L		
								,	L						
								not	L						
								and	L						
								xor	L						
								or	L						

◀**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

In such a list, the term "higher precedence" means an operator is above in the operator list when compared to another. In contrast, the term "lower precedence" is used when an operator is below in this list when compared to another. Some of the operators can hold the same position in the list of operators. This situation is called *equal precedence* of two or more than two operators. For instance, multiplication and division have equal precedence. In the case of the expression from below:

$2 * 2 / 2 / 2 * 2 / 2$

What should the evaluation process do? Since the two operators ("/" and "∗") have the same precedence, it is impossible to choose which computation is done first (please also see Fig. 4.2 for reference). Thus, for operators of the same precedence, there is also a direction of evaluation, called *operator associativity*. For the expression from above that contains only two operators ("/" and "∗") of equal precedence, the computation unfolds:

$2 * 2 / 2 / 2 * 2 / 2 =$
$4 / 2 / 2 * 2 / 2 =$
$2 / 2 * 2 / 2 =$
$1 * 2 / 2 =$
$2 / 2 =$
1

Please notice the left-to-right associativity which means a computation made step by step from the left of this expression towards the right, up to the end of it. To put this into perspective, the use of parentheses explains it even better:

$((((2 * 2) / 2) / 2) * 2) / 2 =$
$(((4 / 2) / 2) * 2) / 2 =$
$((2 / 2) * 2) / 2 =$
$(1 * 2) / 2 =$
$2 / 2 =$
1

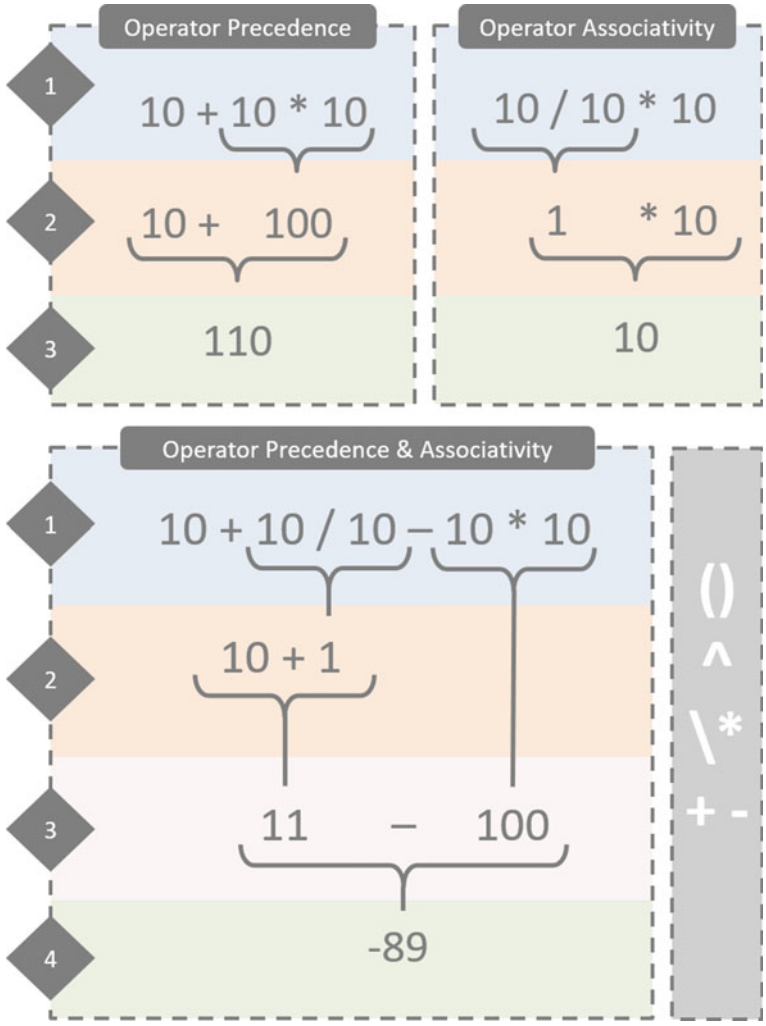


Fig. 4.2 Examples of operator precedence and associativity. At the top, the two panels show one example each for operator precedence or operator associativity. A mixed example is given at the bottom of the figure showing the relationship between operator precedence and operator associativity. In the 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

Also, operators with higher precedence are evaluated before operators with lower precedence. For instance, addition and subtraction have the same precedence. However, division and multiplication have a higher precedence than addition and subtraction. For instance, in the case of:

$2 * 2 / 2 + 2 * 2 - 2$

Division and multiplication are evaluated from the left to right of the expression (left to right associativity). The above expression yields:

$2 * 2 / 2 + 2 * 2 - 2 =$
$4 / 2 + 4 - 2 =$
$2 + 4 - 2$

Next, addition and subtraction are evaluated from the left to right of the expression:

$2 + 4 - 2 =$
$6 - 2 =$
4

In high level computer languages, parentheses force the behavior of evaluations and are at the very top of the precedence list. All other operators have lower precedence. Thus, the above example is equivalent to:

$(((2 * 2) / 2) + (2 * 2)) - 2 =$
$((4 / 2) + 4) - 2 =$
$(2 + 4) - 2 =$
$6 - 2 =$
4

Parentheses and unitary operators hold a higher precedence when compared to all other operators (Fig. 4.1). But why? Why are these first in the list? For the parent expression, parentheses are a bit like a kind of box where one puts something inside and the box morphs into something else based on what was put in. Also, computer languages that are equipped with unary operators require similar expectations from the evaluator.

Evaluation of expressions can begin as long as the terms are known. Unitary operators have priority because the value provided by them is unknown until it is evaluated (the same case as a parenthesis). Thus, the evaluator must obtain a value from the parentheses or/and the unitary operators, prior to the evaluation of the expression. The parentheses are unknowns from which either a value or an error must be obtained. Only then the evaluation of the expression may continue. In other words, an expression in parentheses is an expression inside another expression. E.g:

$a = 2 * 3$
$b = 3 + a$

The above two expressions are the same as:

$b = 3 + (2 * 3)$

Parentheses are their own expressions. In short, operator precedence sets the parsing of operators in relation to each other. Thus, higher precedence operators become the operands of lower precedence operators. Above an expression, multiple operators of the same precedence are evaluated left-to-right or right-to-left. The evaluation examples shown previously were given on the arithmetic operators. However, the list of operators is not limited to arithmetic operators. For example, the way in which comparisons are made between several values through logical operators or the way in which a value is assigned to a variable, are examples that involve an evaluation either from left-to-right or right-to-left. In the figure below, operator precedence is shown from high to low for each computer language on our list. Note that operators of the same level have equal precedence. The rules of precedence and associativity are extremely important, but they cannot be easily remembered for every computer language. As a best practice, the use of parentheses is highly recommended, especially for less frequently used operators (other than "\, $*$, $+$, $-$", which are well known from mathematics). Parentheses allow for nested expressions, or matryoshka doll like expressions, which are safe to work with.

4.6 Conclusions

A brief description was given for all important operators, namely for the arithmetic operators, assignment operators, relational operators, concatenation operators and logical operators. The symbols used to represent each of these operators were shown for all languages used in this book. This chapter further outlined some of the complex operators, discussing their possible origins with real-world examples. Thus, subsequent discussions have considered the power operator, the modulo operator, the unitary operators, the string operator, the interesting case of the repetition operator, the concatenation operator, and finally the relational and logical operators. Next, the meaning of the term assignment was clearly explained. Three types of assignments were discussed, namely: the simple assignments, the aggregate assignments, and multiple assignments. This chapter continued with crucial explanations of the order of operations known in mathematics and further introduced operator precedence and associativity. In order to be clearly understood, examples of precedence and associativity were given. Moreover, a table for operator precedence and associativity was provided for all computer languages used in this work.

Data Types and Statements

5

5.1 Introduction

Information represents the medium from which humans understand the world. Representation of information is fundamental to technology. In the case of machines, data is the form that information takes. Information constructs add different layers to this representation in the form of data structures. Thus, this chapter provides an introduction to units of measurement and the representation of information in binary form. It also gives an overview of important encoding systems and provides examples that show the meaning of the data type. Following this discussion, all data types are presented as basic constructs in a general manner. Namely, data types are discussed in two main points, namely: (i) primitive data types, and (ii) composite data types. The second part of the chapter gives an introduction to computer statements. Again, the ASCII and UTF encodings are briefly mentioned as discrete units of these statements. Next, the rules of writing computer statements are discussed in the light of good practices. The chapter ends with a simple and concise introduction to source code, which is made by expanding the notion of indentation and commenting.

5.2 Data

The word "data" comes from the Latin "*datum*", which means "a given", namely a fact, an observation. In modern times, data is a collection of discrete units of meaning that represent information about observations. For humans, these discrete units of meaning

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/978-3-031-23277-0_5.

P. A. Gagniuc, *An Introduction to Programming Languages: Simultaneous Learning in Multiple Coding Environments*, Synthesis Lectures on Computer Science,
https://doi.org/10.1007/978-3-031-23277-0_5

are the written symbols. However, there are different types of written symbols, some representing letters, others representing numbers, and so on. Consequently, an observation can be described with specific types of symbols, namely letters, or numbers, or other types of symbols. For example, Morse code uses a dot symbol and a line symbol to represent complex information. Thus, different kinds or types of data can result from these observations.

5.2.1 Bits and Bytes

For computers, these discrete units of meaning can be represented by sequences of bits (0 and 1s). Thus, similar to the Morse code, machine code uses the symbol for something and the symbol for the absence of something as the basis of any type of representation. A binary representation of complex symbols is possible in computers by considering the bit combinations that can be made on a sequence of bits of a certain length. For example, the sequence "01,000,001" has 8 bits and means the symbol for the capital letter "A" in UTF-8 (in ASCII it can have 7 bits, namely "1,000,001"). The meaning of the two abbreviations from above is: *American Standard Code for Information Interchange* (ASCII) and *Unicode Transformation Format* (UTF). Another combination of the same length, such as "01,100,001", means the lowercase "a" symbol. On the other hand, in the same format, namely 8 bits, numbers can be represented. The bit sequence "00,110,011" encodes the symbol for the integer "3". Many types of symbols can be encoded under the same bit construct. The 8-bit sequence is the basic representation and many unit conversions have arisen because of it. For example, "octet" is a unit that represents a sequence of 8 bits. Also, a sequence of 8 bits represents 1 byte. Thus, the following units of measurement result:

$$
\begin{aligned}
&8\,bits = 1\,byte = 0.001\,Kilobytes = 0.000001\,Megabytes\\
&80\,bits = 10\,bytes = 0.01\,Kilobytes = 0.00001\,Megabytes\\
&800\,bits = 100\,bytes = 0.1\,Kilobytes = 0.0001\,Megabytes\\
&8000\,bits = 1000\,bytes = 1\,Kilobytes = 0.001\,Megabytes\\
&80000\,bits = 10000\,bytes = 10\,Kilobytes = 0.01\,Megabytes\\
&800000\,bits = 100000\,bytes = 100\,Kilobytes = 0.1\,Megabytes\\
&8000000\,bits = 1000000\,bytes = 1000\,Kilobytes = 1\,Megabyte
\end{aligned}
$$

Abbreviations for the above units of measurement are: 1 b (1 Byte), 1 Kb (1 Kilobyte), 1 Mb (1 Megabyte), 1 Gb (Gigabyte), 1 Tb (Terabyte) and so on. Why was 8 the magic number to represent a whole unit of measurement? A 4-bit sequence is insufficient to store all the critical symbols used by the developed world. The 4-bit sequence allows 16 possibilities (eg. $2^4 = 16$; were two represents the alphabet of the sequence made of 1

and 0's). Thus, since the critical symbol space is known to weigh more than 16 symbols, bit sequences longer than 4 bits were required (Table 5.1). At some point these sequences grew from 3 to 8 bits and finally stayed at 8 bits (eg. $2^8 = 256$ possibilities). The 8-bit sequence represents a balance point. Less than 8 bits cannot encode for the critical character set required for a general purpose computer (which is around 128–256 symbols), while more than 8 bits means that some combinations will lack any association of symbols. For example, a 9-bit sequence has 512 possibilities (eg. $2^9 = 512$ possibilities), however, the critical set of symbols is much smaller than 512. Because of this coincidence between the 8-bit sequence and the number of critical characters, the "byte" became a standard in the early 90s, where 8 bits were defined as a byte. The term "octet" is rarely used mainly because of this standardization, and it has become synonymous with "byte". The main question can be asked again: Why did the octet become a standard for representing bytes? Text storage was the main reason. Because the octet could store one character, it became important for indicating size, especially the size of text files. Any changes that extend the space needed to encode more complex constructs are almost always multiples of 8 (eg. 16, 32, 64) because the byte is a standard (Table 5.1). Over time, as hardware power increased, the symbol set of a general-purpose computer would encompass other alphabets, from those used by each country, to those that are extinct and historically important.

However, the encoding of the ASCII set was established on sequences of 8 bits and was standardized as a unit of measurement. The byte allows 256 possibilities, whereas the symbols of the world can number in thousands or even tens of thousands. To solve the issue of the byte, the Unicode Consortium found a stable solution to reconcile the past with the future. That solution is today known as UTF-8. The UTF-8 solution expanded the critical ASCII character set, remaining back-compatible by dynamically encoding characters using between 1 byte for some of the classical ASCII characters, and 2, 3 or 4 bytes for other special characters (Fig. 5.1).

Note that UTF-8 means that the code unit is 8 bits. Thus, any complex representation is made from multiples of 8 bits and the "byte" is in a sense the shortest unit of addressable memory. Someone may rightfully ask: Then why is there UTF-16 and UTF-32? The answer can be the following: A unicode character in UTF-16 encoding is between 16 bits (2 bytes) and 32 bits (4 bytes), while a unicode character in UTF-32 encoding is always 32 bits (4 bytes). Unlike UTF-16 and UTF-32, UTF-8 is by far the most valuable encoding system because it is back-compatible. UTF-8 extends from 8 bits (1 byte) to 32 bits (4 bytes) and it can do more if needed, as it is infinitely extendable (Fig. 5.1). UTF-16 and UTF-32 can be seen as an experiment rather than a useful strategy, because their existence ignores the past, which is never a good thing.

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 $\left(2^{64} = 1.84467440737e + 19\right)$

Bits	Encoding possibilities	Bytes
1	2	0.125
2	4	0.25
3	8	0.375
4	16	0.5
5	32	0.625
6	64	0.75
7	128	0.875
8	256	1
9	512	1.125
10	1024	1.25
11	2048	1.375
12	4096	1.5
13	8192	1.625
14	16384	1.75
15	32768	1.875
16	65536	2
…	…	…
24	16777216	3
…	…	…
32	4294967296	4
…	…	…
64	18446744073709551616	8
…	…	…

5.2.2 Symbol Frequency Matters

UTF-8 is important for the size of text files because the frequency of ASCII symbols is huge compared to other symbols. Thus, a permanent encoding of a high-frequency symbol on several bytes (i.e. UTF-16 or UTF-32) implies an obvious increase in the size of text files. That can be disadvantageous for storage and the internet bandwidth. In order to understand what was said above in regard to UTF-8, we can consider the text "☀ sunny" as an example (Fig. 5.2). If only ASCII symbols were involved, then such a sequence of characters would need to occupy 7 bytes, because there are 7 symbols in the example. However, the "☀" symbol is encoded into a character positioned well outside the ASCII range.

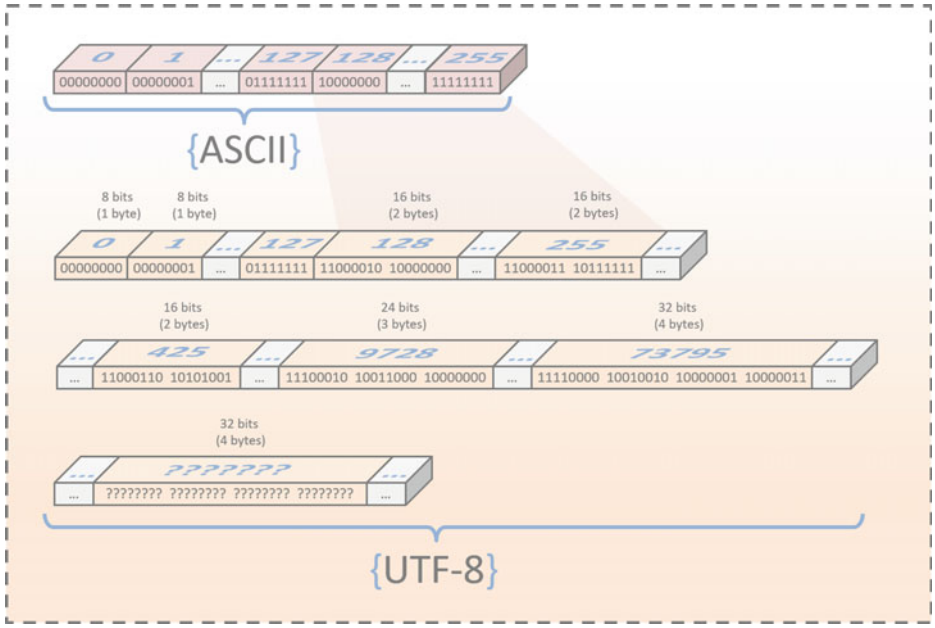


Fig. 5.1 ASCII and UTF-8. It shows the back compatibility of UTF-8. On the vertical axis, the first half of the figure shows the structure of ASCII, which encodes for symbols using 8-bit sequences (1 byte). A schematic of UTF-8 is unrivaled in the second half of the figure. The UTF-8 relationship with 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

Thus, the "☀" character alone is encoded using 3 bytes, while the other 6 characters (including the space character) are normally encoded by using 1 byte each. Thus, in total, the text "☀ sunny" occupies 9 bytes (Fig. 5.2). Another relevant observation would be that among the seven characters of "☀ sunny", six are from the ASCII subset and one from outside of it in the UTF-8 set. This means that even in our example, ASCII characters have a presence of 86% (ASCII% = (100/7 characters) × 6 characters = 85.7%). If these characters had been stored in UTF-32, the text "☀ sunny" would have occupied 28 bytes (7 characters × 4 bytes = 28 bytes) instead of the 9 bytes it occupies in UTF-8. That is because UTF-32 encodes all symbols in the set using 4 bytes.

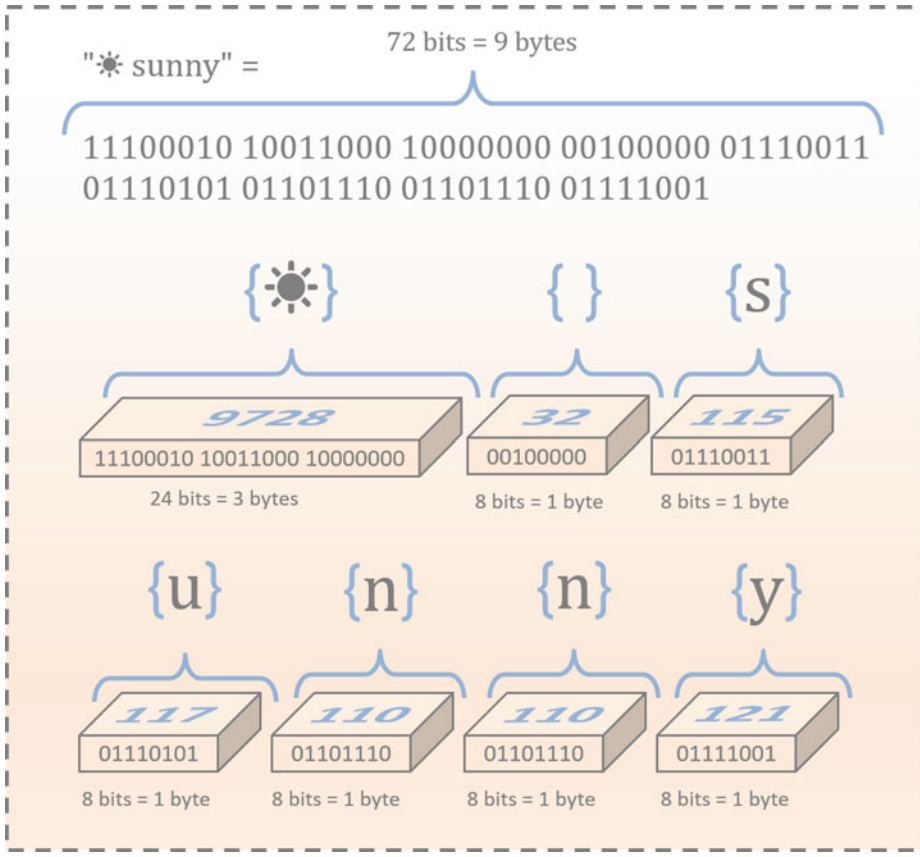


Fig. 5.2 Size of text according to UTF-8. It shows the size of text "☀ sunny" under UTF-8. This further includes the code points (the whole number associated with a symbol), the corresponding bit sequences and the actual symbols associated with these abstract representations. Note that boxes indicate 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

5.2.3 The Encoding

The encoding representation can be explained in a simplistic manner. The meaning of the term "encoded" is important to be understood in the right context, because the 8-bit block does not store the sequence of bits that make up the drawing pattern of a symbol, as one might already suspect. The bits that represent the shape of the symbols are separate pieces of bit sequences that can be displayed as white pixels for 0s and black pixels for 1s. Since the hardware of the past was limited, the purpose of the encoding process was to store information in the shortest possible manner. The role of the 8-bit blocks is to allow specific characters to be retrieved from an already existing set in memory. Thus, a sequence of characters is constructed, which makes readable text visible to humans. In other words, each 8-bit sequence in this encoding is a label for what symbols should be fetched from an existing set. In other words, each 8-bit sequence in the ASCII encoding is a label for the symbols that should be retrieved from a known set stored on the same general-purpose computer. In UTF-8, these character labels can be represented by one 8-bit sequence up to four 8-bit sequences. Realistically, the events are much more complex than described above. These characters have different fonts. Therefore, a computer has many sets of symbols with the same meaning, not just one. The encoding label for each character will be associated with a memory address that points to the set from where the string of bits that make up the symbol shape can be found and retrieved for display. Moreover, the pixels that make up the shape of a symbol are not as simple as black and white in our modern computers, and the color of each pixel is represented by large numbers that indicate a complex color. Nevertheless, the examples from above refer to how symbols are encoded in a general purpose computer. Therefore, symbols stored as characters represent a data type. One can already suspect that data types are constructs that can take many forms. However, regardless of the data type, all of these constructs are measured in bytes (Table 5.2).

5.2.4 A Hypothetical System of Reference

In order to precisely understand the reference system used today in a general-purpose computer, a narrower reference system can be imagined and discussed. A narrow frame of reference can strongly indicate both the universality and natural evolution of information encoding. Historically, the term "byte" was ambiguous as it represented units of bits of different sizes. Without "byte" standardization, one can put any length for the representative bit sequence under the pattern:

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

Data Type	Size	Description
byte	1 byte	Stores whole numbers in the range: –128 to 127.
boolean	1 bit	Stores either true or false (size not precisely defined).
char	1 byte	Stores a symbol encoded into a single character.
short	2 bytes	Stores whole numbers in the range: –32,768 to 32,767.
integer	2 or 4 bytes	Stores whole numbers between: –2,147,483,648 to 2,147,483,647.
float	4 bytes	Stores numbers up to 7 decimal digits.
double	8 bytes	Stores numbers up to 15 decimal digits.
long	8 bytes	Stores numbers in the range: –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

$$
\begin{aligned}
x\,bits &= 1\,byte = 0.001\,Kilobytes = 0.000001\,Megabytes \\
x0\,bits &= 10\,bytes = 0.01\,Kilobytes = 0.00001\,Megabytes \\
x00\,bits &= 100\,bytes = 0.1\,Kilobytes = 0.0001\,Megabytes \\
x000\,bits &= 1000\,bytes = 1\,Kilobyte = 0.001\,Megabytes \\
x0000\,bits &= 10000\,bytes = 10\,Kilobytes = 0.01\,Megabytes \\
x00000\,bits &= 100000\,bytes = 100\,Kilobytes = 0.1\,Megabytes \\
x000000\,bits &= 1000000\,bytes = 1000\,Kilobytes = 1\,Megabyte
\end{aligned}
$$

where "x" is a whole number ($x > 0$). The point of the above example is to show that "byte" has been a volatile unit of measurement. Note that the term "octet" is often used to enforce that "byte" means a unit sequence of 8-bits.

5.2.5 The Bytes of an Alien World

Let us imagine that instead of the critical characters found in ASCII, one uses a very small critical set of an alien civilization. In the universe imagined here, this alien civilization uses 5 critical symbols with which most representations and descriptions can be made. In order to optimize this text, a representation can be envisioned by answering the following question: How many bits are needed to represent the five characters from the set? As discussed above, 1 bit can encode two possibilities, either 1 or 0. Two bits may encode a total of four possibilities, namely: "00", "10", "01", "11". A 3-bit sequence can encode 8 possibilities, namely: "000", "100", "001", "110", "011", "111", "101", "010". Therefore,

the 2-bit representation has a maximum of 4 possibilities and is unable to cover all 5 characters of this hypothetical alphabet. The 3-bit representation has a maximum of 8 possibilities, which is more than enough to represent the 5 characters. Moreover, since five positions will be associated with the five characters and the other three positions will be devoid of any association, there will be room to encode three more characters. For simplicity, in this system each of the five characters from the alien alphabet would be represented by a 2D-map made of five rows of 6-bit sequences (Fig. 5.3). The map of any character would contain 30 bits (i.e. 5 rows × 6 bits). Since our alien byte is defined

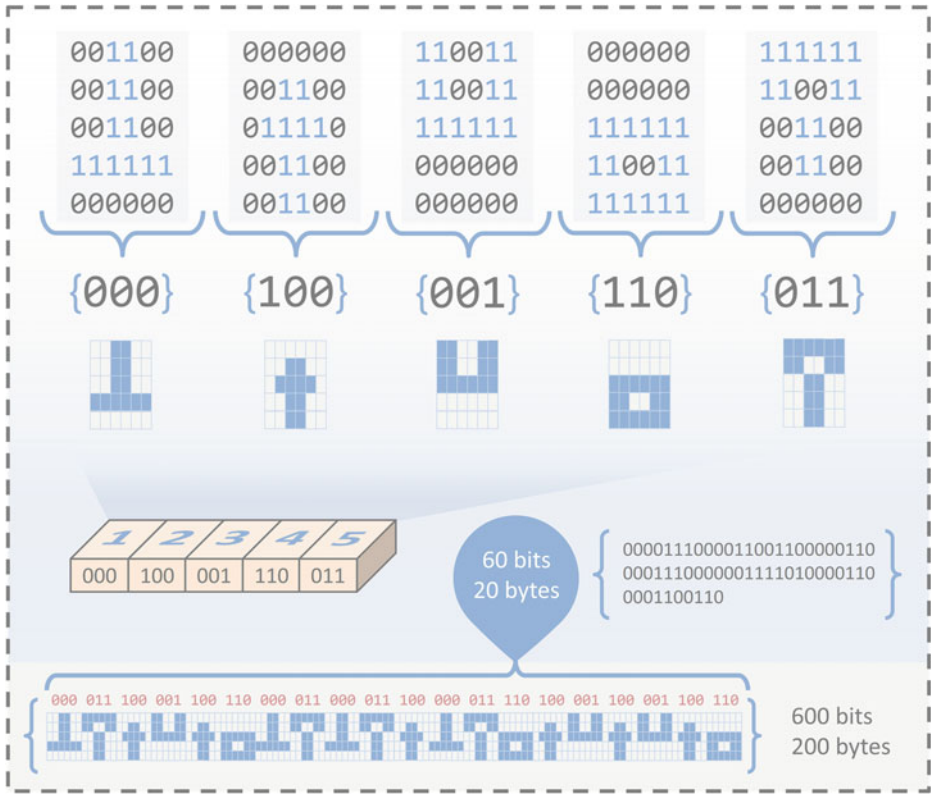


Fig. 5.3 The alien text measured in alien bytes. The top of the figure shows five hypothetical characters in a 2D formation of 5 × 6 bits. Below the representations are the 3-bit codes that can be associated with these object characters. Just below the 3-bit codes, the characters are displayed using 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

as a 3-bit sequence, one character would have 10 bytes (i.e. 30 bits/3 bits in a byte). An alien wants to write meaning using the five characters. If it writes a proposition of 20 characters then the size of the text will be 200 bytes (i.e. 20 characters × 10 bytes = 200 bytes). In other words, this can also be understood as 20 characters of the text multiplied by 30 bits per character, which equals 600 bits. The 600 bits divided by the 3 bits that make up our "alien" byte would give a result of 200 bytes for the entire text. The alien of this supposed universe realizes that 200 bytes is too much for his computer hardware and asks himself a question: How many characters can be encoded and decoded in as few bits as possible? The alien envisiones the association between characters and the 3-bit combinations. He realizes that by storing the five characters on every computer in his world, he can only send and receive sequences of 3-bit combinations that represent characters of the alphabet. If he receives a sequence of bits, he will structure the sequence in groups of 3-bits and ask a map to see which character corresponds to each 3-bit group. Likewise, when the alien sends a text, instead of the actual characters, he will send a sequence of 3-bit codes.

He looks at the map of association to see which character corresponds to which group of 3-bits. Thus, it replaces each character with a 3-bit code, from which, a sequence of bits emerges. In the case of the 20 character proposition, the mapping of characters to 3-bit combinations reduces the size of the information from 200 to 20 bytes (i.e. 90% reduction in the size of this information). In perspective, a 1 Mb file of 5 × 6 bit-maps is reduced to 100 Kb when the bit-maps are replaced by 3-bit sequences.

5.3 Data Type

The discussion of how data is represented on general purpose computers showed the significance of data types. In fact, the examples given in the previous sections provide information about one data type, namely the character (char). A basic description is necessary to see the whole picture about these constructs. Data type can be divided into *primitive data types* and *composite data types*. Primitive data types are a set of fundamental data types from which all other data types are constructed (Fig. 5.4). Among the primitive data types one can enumerate: *byte*, *short*, *integer*, *long*, *float*, *double*, *boolean* and *char*, whereas in the case of composite data types (non-primitive data types) structures such as *Arrays*, *Strings* and Classes (blueprints for *objects*) can be mentioned (Tables 5.2 and 5.3). In the case of *composite data types*, the situation becomes more complex, where for example a variable may contain additional substructures. To continue the abstract illustration from before, such a variable can be painted as a big box containing smaller boxes inside. In the majority of instances, these complex variables are called *arrays*. Often, *arrays* are known to be in fact *objects* in some modern computer languages (ex. Python,

Ruby, JavaScript and so on). For example, Ruby is a pure Object-Oriented computer language and all data types are based on classes. Thus, in Ruby and others, there are no real primitive data types, only simulations of primitives.

Thus, the whole collection of data types can sometimes be language specific and other times these are well established regardless of language (Table 5.3). For instance, in JavaScript one can distinguish three main *data types* which are used, namely: *Number*, *String*, and *Boolean*. A *Number* can be either an *integer* or a *decimal*. *Strings* on the other hand are sequences of characters enclosed in a single or double quotes, depending on the programming or scripting language used. A string variable is a variable that holds a character string (i.e. a sequence of characters).

One observation deserving attention is that a symbol may represent any shape that has a meaning. Thus, among others, symbols may represent the shape for the letters or for numbers. Last but not least, *Boolean* type represents either true or false. In practice *boolean* variables are rarely used. In other computer languages, these *data types* may differ, as it can be seen in Table 5.3. In the case of *primitive data types* vs *composite data types* we can discuss about two main points, namely: (i) Variables for *primitive data types* hold the actual value of the data. (ii) variables for *composite data types* hold only references to the values of the composite type.

5.3.1 The Curious Case of the String Data Type

String data type is one of the most recent proofs of the natural evolution of computer languages. In regard to strings, one simple question can be asked, namely: *Are all computer languages equipped with a string data type?* The majority of modern general-purpose programming and scripting languages contain by default a string data type in their core library or their additional libraries. Moreover, today a computer language without a string data type is almost inconceivable for the majority of us, and straightforward useless. Another question that arises is: *What can we do in the case of a computer language that is not equipped by default with a string data type?* The answer is: programmers can design a string data type themselves in an additional library. Not only that, but new kinds of experimental data types can be formulated. The string data type is relatively a new concept. Older programming languages lack the *string* data type. For instance, the "C" language and of course the "Assembly" language do not have a string data type. The "Assembly" language, which resides somewhere between machine code and the high level languages, lacks the concept of data types altogether. As mentioned above, the "C" language has no actual string type. Instead, it uses the *char* data type to make a *string* construct. In modern computer languages, *char* is a primitive type that is able to store only one character (2 bytes), whereas *string* is a class that makes the *string* object, which in turn may encapsulate zero or more characters. Internally, a *string* can be viewed as an ordinary *array* of *chars*. Thus, when referring to *strings* as a data type, we are doing this by extension.

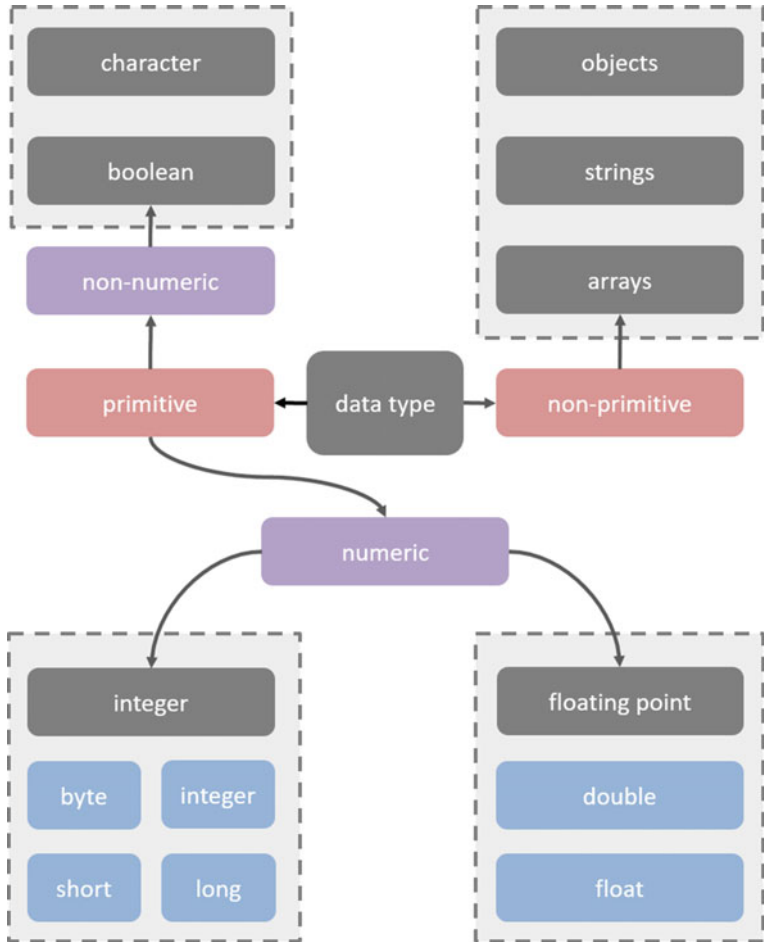


Fig. 5.4 Data Type representation. It describes the general constructs used by computer languages to represent data. The data type constructs shown here are normally divided into two, primitive data types and non-primitive data types. Primitive data types in turn are divided into two other categories, 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

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

JS	C#/C++	VB	PHP	PERL	Ruby	Java	Python
Number Booleans String Object ...	int float double bool char string long (C#) ...	Array Boolean Byte Currency Date Double Integer Long String ...	String Integer Float Boolean Array Object NULL Resource ...	Scalars Arrays of scalars Hashes ...	Numbers Strings Symbols Hashes Arrays Booleans ...	Byte Short Int Long Float Double Boolean Char String Arrays ...	str int, float complex list tuple range dict set frozenset bool bytes bytearray NoneType ...

Just by looking at the table from above, it can be intuitively said that those computer languages that have a limited number of data types show stability and maturity, while computer languages with a multitude of data types show a kind of immaturity, like these are not yet fully distilled. This is not necessarily a bad thing, and it means that the designers of those computer languages are still experimenting, and other useful constructs may emerge from them for the future of software technology. Thus, data types are constructs, which really appeared as a natural consequence of our needs to optimize programming time and the speed of software applications.

5.3.2 Experimental Constructs

However, the use of experimental constructs (ex. a new data type) is highly dangerous for those using them. The removal of an experimental construct by the provider of a computer language reflects, of course, in all designs that made use of it. Thus, external dependency on experimental constructs can sharply cut the life cycle of a software application. Moreover, experimental methods embedded in computer languages cannot be considered legacy and expectations for back compatibility are absurd since they are called "experimental". Over time, this issue was directly visible for JavaScript/CSS/HTML applications, were methods rendered absolute affected all applications that used the experimental methods up to that point. The same is true with the experimental API of an operating system. It can disappear from one update to another or from one version of the operating system to another. Therefore, methods that are deeply engraved and intertwined with the makeup of

the computer language, are the only methods that are safe to use in order to prolong the life cycle of an application.

5.4 Statements

In the normal language phrases are made of consecutive written symbols and then recognized for their meaning by educated individuals of *Homo sapiens*, and others. In high-level computer languages, the same pattern of meaning is preserved. However, computers must first be able to represent the symbols used to write meaning. At the core of any human–machine interaction, there is a critical list of symbols used for that particular purpose. This list of symbols is called: *American Standard Code for Information Interchange* (ASCII). Secondary and less critical, is the symbol extension under the *Unicode Transformation Format* (UTF) of the *Unicode Consortium.*

5.4.1 ASCII Symbols

ASCII witnessed many changes over time. It has its roots in the symbols used in the telegraph system and has an even older history, which dissolves into the most primitive signaling modes for communication. However, a stable version of ASCII became the standard in computers many decades ago. ASCII contains 256 characters of 8 bits each. ASCII characters represent the critical symbols needed to signify the basic operations of a general-purpose machine. ASCII control characters (character code 0–31) are the non-printable control codes used primarily with hardware (escape, enter, back space, and so on) and software constructs (files, data structures, and so on). ASCII printable characters (from code 32 to 127) represent letters, digits, punctuation marks, and other symbols. The extended ASCII codes start from code 128 and end at code 255. In this section, the reader also understands the numbering styles used, which is most useful at the end of this chapter, which involves array variables. Notice that one mention states 256 characters and another refers to code 255. In reality there are 256 characters, but the numbering of these symbols starts from zero instead of one. In other words, one range of numbers (1–256) refers to a total and the other range (0–255) represents the character codes or the labeling system.

5.4.2 Unicode Transformation Format

In the modern era, the range of symbols exceeds the critical 256 characters limit of ASCII. International efforts were made to include the entire set of known symbols in what we recognize today as the Unicode standard. The reason for the Unicode standard is to encode

symbols from every language on the planet, and also to encode other significant symbols with subjective or objective meaning. The *Unicode Transformation Format* (UTF) refers to several types of Unicode character encodings, including: UTF-7, UTF-8, UTF-16, and UTF-32. The numbers following the word "UTF" represent the bytes used to encode the symbols (i.e. UTF-8, means 8 bytes). Many computer languages now use UTF-8 instead of ASCII. UTF-8 encodes for much more symbols than ASCII. Moreover, UTF-8 is also compatible with ASCII, which allows ASCII characters to be interpreted using UTF-8. In other words, UTF-8 codes start with the full ASCII range (0–255). Thus, programmers using UTF-8-based computer languages will hardly notice the difference, unless they exceed the limit of 255 ASCII character codes.

5.4.3 Sentences are Made of Constructs

A statement is a complete instruction composed from consecutive ASCII characters. Statements may contain keywords, operators, constants, variables, and entire expressions. Statements can be declarative or executable. *Declaration statements* include variable naming, data types, constants, procedures. *Executable statements* include assignments, loops, branching through blocks of code, and calls to methods or functions. A statement is a section of the source code that can be evaluated by the compiler or by the interpreter depending on the case. The organization of statements is done in lines of text, which deserves a little expansion of the historical roots.

5.4.4 The Root of Behavior

All European languages use the Cyrillic or Latin alphabets written from left to right and the reading for meaning is made in the same sequence. To highlight similar genetic identity among certain sub-populations of European, small additions such as diacritics have been added over time from one country to another. Because of this beautiful inheritance, the text of computer languages has the same rules. Namely, the interpretation of sentences is done from left to right. However, exceptions exist. For instance, assignments of values are made from right to left. Furthermore, the evaluation rules for mathematical expressions have no left or right directions, and the order of operations dictates where and what is evaluated.

5.4.5 The End of the Line

Actions that a program takes are expressed in statements. Thus, please note that a *source code* is a collection of statements, just as a cookbook is a collection of phrases. How

statements are represented is important. Each statement can be composed of one or more lines. The end of each line is represented by different ASCII characters depending on the type of syntax. In some computer languages the end of the line maybe marked in several ways. For instance, in scripting languages like JavaScript, PHP or PERL, or in the case of programming languages like Java, C Sharp or C++, the end of the line always ends in a semicolon ";". This is the way in which the interpreter or the compiler understand the end of a line in their syntax. On the other hand, programming languages like Visual Basic, Ruby, or Python, designate their end of line with an invisible character. This end of the line invisible character is represented by a combination between the ASCII characters *line feed* and *cage return*. Of course, the combination of the two ASCII characters contains a history routed in the reality of an older mechanical technology of the past. As we can guess, we are talking about the typewriter in which the start of a new line was composed of two mechanical actions performed by the writer. Namely, the push of the entire paper placed on the cage of the typewriter from the left to the right of the typewriter in order to place the middle of the machine at the beginning of the row on the paper. However, this wasn't the only mechanical action needed in order to begin a new line. Thus, the writer performed another mechanical action called line feed. This action performed a push of the paper on the vertical axis to place the middle of the typewriter below the previous written sentence. This is the history of the composite character behind the "Enter" key. Of course, there are differences in between certain operating systems. What we talked about previously, was related to the way Microsoft Windows operating system uses to denote the end of the line for any text. In Unix for instance, the end of the line is noted by using the ASCII character for *line feed*.

5.4.6 Statements and Lines

In the BASIC family of computer languages, non-printable ASCII characters can be provided by calls to a built-in function called "*chr(c)*". This function receives an argument "*c*" which represents the ASCII character code. Then, on return, the function is able to provide the character in its pure form (Additional algorithm 5.1). All computer languages are equipped with in-built functions that return the ASCII characters based on the symbol code (Table 5.4). The last two columns of Table 5.4, titled "*Code to Char*" and "*Char to Code*" show either built-in functions with the role of forward and backward conversions or show the use of properties from the String object (Additional algorithm 5.1). In practice, the returned values from string objects or the in-built functions, are usually concatenated with strings. Therefore, an example is given for each computer language (Additional algorithm 5.1).

In VB, some important non-printable ASCII characters which are responsible for structuring data, have their own dedicated keywords. Such keywords are: *vbCr*, *vbLf* and

Lang.	Example
JS	`print ("Code 65 is the:'" + String.fromCharCode(65) + "' letter");` `print ("Letter A has the code:'" + 'A'.charCodeAt(0) + "'");`
C++	`cout<<"Code 65 is the:'"<<(char) 65<<"' letter\n";` `cout<<"Letter A has the code:'"<<(int) 'A'<<"'";`
C#	`Console.WriteLine("Code 65 is the:'" + (char) 65 + "' letter");` `Console.WriteLine("Letter A has the code:'" + (int) 'A' + "'");`
VB	`MsgBox "Code 65 is the:'" & Chr(65) & "' letter"` `MsgBox "Letter A has the code:'" & Asc("A") & "'"`
PHP	`echo "Code 65 is the:'" . chr(65) . "' letter";` `echo "Letter A has the code:'" . ord('A') . "'";`
PERL	`print "Code 65 is the:'" . chr(65) . "' letter";` `print "Letter A has the code:'" . ord('A') . "'";`
Ruby	`puts "Code 65 is the:'" + 65.chr + "' letter"` `puts "Letter A has the code:'" + ('A'.ord).to_s + "'"`
Java	`System.out.println("Code 65 is the:'" + (char) 65 + "' letter");` `System.out.println("Letter A has the code:'" + (int) 'A' + "'");`
Python	`print ("Code 65 is the:'" + chr(65) + "' letter")` `print ("Letter A has the code:'" + str(ord('A')) + "'")`

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

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

Lang.	LF	CR	CR & LF	Code to Char	Char to Code
JS	`"\n"`	`"\r"`	`"\r\n"`	`String.fromCharCode(a);`	`b.charCodeAt(0)`
PERL	`"\n"`	`"\r"`	`"\r\n"`	`chr($a);`	`ord($b);`
PHP	`"\n"`	`"\r"`	`"\r\n"`	`chr($a);`	`ord($b);`
C++	`"\n"`	`"\r"`	`"\r\n"`	`(char) a;`	`(int) b;`
C#	`"\n"`	`"\r"`	`"\r\n"`	`(char) a;`	`(int) b;`
Java	`"\n"`	`"\r"`	`"\r\n"`	`(char) a;`	`(int) b;`
Ruby	`"\n"`	`"\r"`	`"\r\n"`	`a.chr`	`b.ord`
Python	`"\n"`	`"\r"`	`"\r\n"`	`chr(a)`	`ord(b)`
VB	`vbLf`	`vbCr`	`vbCrLf`	`Chr(a)`	`Asc(b)`

vbCrLf. In the story above, a discussion of typewriters detailed the origin of some non-printable ASCII characters. In that historical note, the technical explanations of *carriage return* indicated that it meant *return to the beginning of the line*. Thus, the *vbCr* keyword represents a *carriage return* for print and display functions. On the other hand, line feed was described as "*go to next line*". The representative keyword is *vbLf* and it represents a *line feed* character for print and display. A concatenation between the two characters is represented by the keyword *vbCrLf* which represents a *carriage return* character combined with a *line feed* character, again both with frequent uses for print and display functions. In short, these two characters are provided by the *Enter* key each time users press it. By contrast, these two characters (*vbCrLf*) also can simulate the *Enter* key. All other computer languages from the list contain a specific and universally accepted representations of the *carriage return* and *line feed* characters. The representation for the *line feed* character is made by using two printable ASCII characters, namely "\n". In contrast, the representation for the *carriage return* character is made by another two ASCII characters, namely "\r". Other important non printable ASCII characters are represented in a similar way with the above (ex. the tab character is written as "\t"). The end of the line is OS-specific.

As mentioned before, UNIX-based systems use "\n" to specify the end of line for text, MacOS used "\n" as line delimiter, whereas DOS-based systems use "\r\n" for the same purpose. The representations for "*line feed*" and/or "*carriage return*" characters are used directly inside strings (ex. "\n" can be used as: *print* "*this\n is under*";). In the ASCII table, the representation ("\r") for *line feed* is evaluated as the ASCII character with code 13 (*Carriage Return*—CR), and "\n" is evaluated and replaced with the ASCII character under code 10 (*Line Feed*—LF). The ASCII character set should be seen as a core of basic symbols. This critical set is today a part of the Unicode Transformation Format (UTF). Many computer languages are UTF-8 compatible. Computer languages that are UTF-8 compatible can also display UTF-8 symbols in their console. This is one method to test for UTF-8 compatibility. For instance, the replacement of character code 65 in the examples from Additional algorithm 5.1, with numbers greater than 255, will display symbols outside the ASCII set. For instance, the character containing the sun symbol ("☀") is represented by UTF-8 code "9728", the character showing a cloud ("☁") is represented by UTF-8 code "9729", the UTF-8 character showing the symbol for umbrella ("☂") is known as the UTF-8 code "9730", and one character used often in this chapter, namely the filled square ("■"), is represented by UTF-8 code "9608" (Please test this by using Additional algorithm 5.1).

5.4.7 Multiple Statements and Line Continuation

Line continuation allows a statement that is too long to be split into multiple lines. Thus, the entire statement, regardless of length, is brought to the attention of a programmer on

the vertical axis instead. In general, the breaking up of a long instruction is most easily done between the operands on the right side of an assignment. This method is detailed here (Table 5.5). Statement continuation on multiple lines can be achieved in numerous ways. Line breaks of a statement also accepts other variations, such as certain *array* variables in Python, the details of which can be filled in on different lines as needed (eg. [*a,* ■*b,* ■*c*]; where the "■" signifies the *Enter* key). In PERL, the variables written inside a string value are evaluated for *string interpolation*. Specifically in PERL, strings accept newline continuation every time the "\" symbol is written (ex. "$a\■$b\■$c";). However, the above expression can also be written differently by using the *string concatenation* operator (ex. "${a}.■${b}.■${c}";). On the other hand, there are also situations where the lines are very short and the focus would be on too many lines of code.

Thus, most computer languages have symbols that can be used to add lines of code one after another, so that the focus of the programmer goes more to the right side of the screen than to the bottom of the screen. The above description is referring to multiple statements on the same line. Therefore, multiple statements can be combined on a single line. Also, a statement can continue on multiple lines. Depending on the computer language used, multiple statements may exist on a single line, only if these are separated by a particular ASCII character. In the BASIC family of computer languages, the symbol that designates the end of one line and the beginning of another is the colon (i.e. the ":" statement separator symbol). In the other computer languages from the list, the symbol that denotes the end of one line and the beginning of another, is the semicolon (i.e. the ";" statement separator symbol).

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

Lang.	Multiple statements	Line continuation
PERL	`$a;$b;$c`	`= ${a} .■${b} .■${c};`
PHP	`$a;$b;$c`	`= $a .■$b .■$c;`
JS	`a;b;c`	`= a +■b +■+ c;`
C++	`a;b;c`	`= a +■b +■+ c;`
C#	`a;b;c`	`= a +■b +■+ c;`
Java	`a;b;c`	`= a +■b +■+ c;`
Ruby	`a;b;c`	`= a +■b +■c`
Python	`a;b;c`	`= a + /■b + /■c`
VB	`a:b:c`	`= a & _■b & _■c`

5.4.8 Recommended Versus Acceptable Statements

Basic good practices for statements require some subjective classifications, such as *recommended practices* or *acceptable* versus *wrong practices*. Multiple statements on one line or a line continuation for very long statements, are *acceptable practices* only when the complexity requires it. Namely, when statements are too long, a statement continuation on multiple lines is recommended. However, when statements are short and clearly visible at average display resolutions, the statement continuation on multiple lines is acceptable, but not recommended. Also, short statements that occupy too many lines, can be placed on the same line using the appropriate delimiter symbols. To illustrate the above, a JavaScript example can be given in Additional algorithm 5.2, by following subjective rules about what practices are supposed to be.

JavaScript allows for the omission of the semicolon at the end of each line, whereas in Python the omission of the semicolon is normal. In the case of C-like computer languages that require a semicolon, it is advisable, as a good practice, to place these characters at the end of each line in order to avoid errors. More complex examples can be given for each computer language from our list (Additional algorithm 5.3). This time, only the *recommended* and *acceptable* practices are shown. Understandably, *wrong* practices can be many and are not discussed further. However, these are usually blocked from execution and then alerted by the compiler or interpreter, or even by the IDE's syntax verification modules.

The example is closely mirrored in all computer languages from the main list (Additional algorithm 5.3). It demonstrates string concatenation, the use of carriage returns, and statement continuation on multiple lines, in two forms: a recommended form and an acceptable form. The recommended form contains three statements on four lines, whereas the acceptable form contains three statements on a single line. Note that variables are used

```
//Recommended:
a = 7;
b = 3;

//Acceptable:
a = 7; b = 3;

//Wrong:
3 = b;
a =
7;
```

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

Lang.	Example	Output

JS	`// Recommended:`
	
	`a = "this is ";`
	`b = a + "JavaScript\n" +`
	`"output";`
	
	`print(b);`
		`JS Output:`
		`this is JS`
		`output`

	`// Acceptable:`
	
	`a = "this is "; b = a + "JavaScript\n" +`
	`"output"; print(b);`	

C#	`// Recommended:`
	
	`String a = "this is ";`
	`String b = a +`
	`"C#\n" +`
	`"output";`
	
	`Console.WriteLine(b);`
		`C# Output:`
		`this is C#`
		`output`

	`// Acceptable:`
	
	`String a = "this is "; String b = a + "C#\n"`
	`+ "output"; Console.WriteLine(b);`	

VB	`' Recommended:`
	
	`a = "this is "`
	`b = a & "Visual Basic 6.0" & _`
	`vbCrLf & "output"`
	
	`MsgBox b`
		`VB Output:`
		`this is VB`
		`output`

	`' Acceptable:`
	
	`a = "this is ": b = a & "Visual Basic 6.0" &`
	`vbCrLf & "output": Debug.Print b`	

PHP	`// Recommended:`
	
	`$a = "this is ";`
	`$b = $a . "PHP\n" .`
	`"output";`
	
	`echo $b;`
		`PHP Output:`
		`this is PHP`
		`output`

	`// Acceptable:`
	
	`$a = "this is "; $b = $a . "PHP\n" .`
	`"output"; echo $b;`	

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

PERL

```
# Recommended:

$a = "this is ";
$b = ${a} . "PERL\
output";

print $b;
```

```
# Acceptable:

$a = "this is ";$b = ${a} . "PERL\n" .
"output";print $b;
```

```
PERL Output:
this is PERL
output
```

Ruby

```
# Recommended:

a = "this is "
b = a + "Ruby
output"

puts "#{b}"
```

```
# Acceptable:

a = "this is "; b = a + "Ruby\n" + "output";
puts "#{b}"
```

```
Ruby Output:
this is Ruby
output
```

Java

```
// Recommended:

String a = "this is ";
String b = a +
"Java\n" +
"output";

System.out.println(b);
```

```
// Acceptable:

String a = "this is "; String b = a +
"Java\n" + "output"; System.out.println(b);
```

```
Java Output:
this is Java
output
```

Python

```
# Recommended:

a = "this is "
b = a + "Python\n" + \
"output"

print (b)

# Acceptable:

a = "this is "; b = a + "Python\n" +
"output"; print (b)
```

```
Python Output:
this is Python
output
```

Additional algorithm 5.3 (continued)

C++

```
// Recommended:

string a = "this is ";
string b = a +
"C++\n" +
"output";

cout<<b;
```

```
// Acceptable:

string a = "this is "; string b = a + "C++\n"
+ "output"; cout<<b;
```

```
C++ Output:
this is C++
output
```

Additional algorithm 5.3 (continued)

before any formal explanations are given in relation to these structures. For the moment, one can say that a variable (like *a* or *b*) is representative for a piece of data, and that simple definition is enough for this stage. In words, a letter *a* is associated with a string with the value "*this is*". Then, a letter *b* is assigned the content of letter *a* and a string with the value "*PERL\output*", where the symbol "\" from the previously mentioned string, denotes that the value of the string is continued on another line. Post-assignment, the string value from *b* is written to the output for the observer.

5.5 The Source Code

The source code is at the very top of data structures and it represents a higher-order organization of machine statements. Strict rules of organization make source code universal among programmers of a particular computer language. Source code is a list of instructions that an interpreter program will follow and a compiler program will turn into machine code. Thus, compilation can also be viewed as structure change to a lower-order. Nonetheless, in the same source code there are statements for computers and statements for people. In other words, the source code contains two types of instructions, namely machine instructions as shown above and comments whose meaning is interpreted only by the programmer. Moreover, both types of statements are organized by indentations, of which a clear description is made below.

5.5.1 Indentations

Code indentation represents the empty space in front of the computer instructions that show the structure of the source code in a nested manner. Usually, ASCII characters

encode visible symbols (shapes with a meaning). However, empty space is represented by characters that contain no symbols at all. Code indentation is the part of what we call "*good practices*". Indentation is a subjective activity in the majority of programming and scripting languages. That is, the amount of indentation it is on the eyes of the beholder, mainly the human that writes the code. Usually, two types of characters can be used. One character is the *space* character, and the other character can be the *tab* character. The amount of indentation is represented by the number of *space* characters or/and *tab* characters in front of the line. One exception to the above discussion is the Python scripting language where the indentation is mandatory. In Python, each step in the indentation shows the hierarchy of the computer instructions in the structure of the code. Moreover, these indentations help the Python interpreter to parse the code for interpretation of the instructions and their execution. Python allows for both the *space* character and the *tab* character, however, in establishing the outer indentation level; it can be one or the other. If both types of ASCII characters are used, an error will be raised by the interpreter, namely "*IndentationError: unindent does not match any outer indentation level*". In all other computer languages, indentation can even be used to hide the source code of scripts from the view of the screen when a file is opened by different text editing tools. Another found use of indentations is to deceive the eye in source code obfuscation. The first and second tactics have been widely used over time by both novice and advanced malware writers. Note that terms such as *outdent* or *unindent* are used to describe the removal of indentation, repositioning the line with a previous level of indentation towards the left edge of the window.

5.5.2 Comments

Comments are an important part of the source code. Comments are used to explain code. In some extreme cases, comments can be used to describe each individual line of code. Imagine a situation in which a software application is designed today with no comments at all, only the code. The designer of this application is able to remember the structure of this code for a couple of months. However, for longer periods of time this clarity over the code starts to fade away, as it is natural. Years later, further developments of the same application by the same software developer or by other software developers, may be difficult and time-consuming without some initial comments. Thus, comments can be used to make the code more readable by humans, and it can explain the structure of the application in a reasonable manner. Thus, comments are disregarded by the compiler or the interpreter because they are useful only to the makers of the application. One other use for comments is the prevention of execution for specific lines when testing of the code is done. For instance, one may write two different lines of code, namely two or more solutions that must be tested in sequence. Thus, instead of deleting all the lines of the other solutions, those lines can simply be temporarily converted into a comment. Depending on

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

Language	Single Line comments	Multiline comments
VB	`'  A comment in VB`	`'a ■'b ■'c`
C#	`// A comment in C#`	`/*a ■ b ■ c*/`
JavaScript	`// A comment in JavaScript`	`/*a ■ b ■ c*/`
PHP	`// A comment in PHP`	`/*a ■ b ■ c*/`
Java	`// A comment in Java`	`/*a ■ b ■ c*/`
C++	`// A comment in C++`	`/*a ■ b ■ c*/`
PERL	`#  A comment in PERL`	`=begin a ■ b ■ c =end ■=cut`
Ruby	`#  A comment in Ruby`	`=begin a ■ b ■ c =end`
Python	`#  A comment in Python`	`'''a ■ b ■ c'''`

the programming or scripting environment, a comment may begin by inserting a special character at the beginning of the line. In VB this special character is "'", whereas in PERL, Ruby and Python, the comment character is represented by "#". In JavaScript, C++, C#, PHP and Java, the comment begins with a set of two characters, namely "//" (please see Table 5.6).

The table from above shows some examples of comments in each programming and scripting language used here (Table 5.6). In order to be useful, these comments can be positioned one line above the code, or, immediately after the code on the same line. Multiline comments are also available when sophisticated comments are a necessity in the source code. In order to be able to show how to open and close comments in computer languages from our list, a one-dimensional representation method was used (Table 5.6). In these patterns, the letters *a*, *b* and *c* may represent one or more lines of text, and the "■" the symbol represents the action of pressing the *Enter* key. For instance, the unfolding of the "'*a*■*b*■*c*'" pattern for Python, yelds the meaning from Additional algorithm 5.4. Notice also that multiline comments are not allowed in VB. Lack of possibility for multiline comments has the advantage of restricting the programmer from over-commenting the source code. In VB, the '*a*■'*b*■'*c* indicates that a single quote symbol is inserted at the beginning of each new line.

Comments are also used by advanced programmers to format the entire source code as a piece of art. This type of formatting consists of an interplay between indentation and different special characters. Such formatting strategies can lead even to the development of low resolution images made from ASCII characters (ASCII art).

```
print('''
=========================================
||                                     ||
|| Python                              ||
|| multiline comment [single commas]   ||
=========================================
''')

print("""
=========================================
||                                     ||
|| Python                              ||
|| multiline comment [double commas]   ||
=========================================
""")
```

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

5.6 Conclusions

When something is born on the basis of something else, the process is called natural evolution. This is also the case in computers, where data types exist based on the fundamental constructs of biological evolution. Thus, by extension data types are in themselves natural constructs. In order to expand on the above interpretation, this chapter described the meaning of data, data types, statements and finally the meaning of source code. It was shown how data is measured and what is the meaning of the terms “bit” and “byte”. Explanations made on different encoding systems, pointed out the importance of inheritance, were the lack of encoding inheritance leads to inefficient computers and wasteful resources. Consequently, the most established data types were discussed in terms of primitive and non-primitive. The emphasis was made on non-primitive data types, as these have led to the notion of strings and other more complex data types. The second half of the chapter describes the rules for designing computer statements. ASCII was again mentioned as the set of discrete units that makeup a computer statement. With modern computers, this ASCII set is now a subset of UTF-8, which in turn was also described as an important tool of statement representation. At the end of the chapter the source code was described in the context of human-machine interaction.

Classic and Modern Variables

6

6.1 Introduction

Variables are data containers that allow instructions to be broken down into smaller, more manageable chunks. Consequently, variables are at the core of all processes. This chapter describes what variables are and how these structures can be modified to accommodate different data types. First, best practices are discussed regarding variable names. The difference between explicit and implicit variable declarations are then explained in the context of statically vs dynamically typed computer languages. Most importantly, experiments in regard to variable assignments are discussed under the umbrella of operator precedence and association, and their behavior with respect to different data types. These simple experiments are made in each of the languages provided in this work. In the second part, the chapter describes the structure of more complex variables, such as arrays. Thus, the creation of arrays, the accessibility of their elements, and the addition or deletion of elements are discussed extensively with examples. Furthermore, properties such as length, upper bound, or lower bound are also discussed at the end of the chapter with more complex implementations.

6.2 Variables

In order to understand the fundamentals, one must first learn about the concept of variable. What a variable is and how it can be dealt with, is the main concern of this entire chapter. A variable is a name associated with a piece of data, namely a reference to the data.

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/978-3-031-23277-0_6.

P. A. Gagniuc, *An Introduction to Programming Languages: Simultaneous Learning in Multiple Coding Environments*, Synthesis Lectures on Computer Science,
https://doi.org/10.1007/978-3-031-23277-0_6

Variables allow the storage or the manipulation of data in software applications. In an abstract manner, variables may be viewed as small boxes which hold a specific piece of information. In most computer languages, variables are first declared and then "loaded" with literals. What are literals?

6.2.1 Literals

The term "literal" comes from the fact that data is literally written into the program. Thus, literals are values declared in the source code and are later a part of the compiled application (for computer languages with the possibility of compilation to machine code). In other words, a literal is data stored directly in the source code rather than indirectly taken from a variable or function call. Some examples of literals of different data types are shown on each line in Additional algorithm 6.1.

Literals are fixed values and cannot be changed by the program containing them. However, literals can be copied into a variable for further use. Consequently, literals are often used to initialize variables by assigning the literal to the variable. These written values are also important for the read-only variables discussed later in this chapter. Below, there are some examples that show how variables are created and how literals of different data types are assigned to these variables above different interpreted and compiled languages (Additional algorithm 6.2).

Up to this point, the variable declarations show only small differences between the languages used here. These differences may consist in the presence or the absence of the semi colon at the end of each line, or, small differences in the way the variables are written. For instance, in PERL and PHP the variables start with the dollar sign before the name of the variable, whereas in other languages the variables contain only the name of the variable. Of course, regardless of the language used; there are some rules when it comes to the name of a variable. Such restrictions are imposed by the functionality of the compiler or interpreter, as discussed in the previous chapters.

```
42
3.14159265358979323846264
'a'
"this text"
```

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

Lang.	Example

JS	`<script>`
	
	`var x = 10;`
	`var y = 17;`
	`var color = "green";`
	`var xname = "Paul";`
	`var logic = true;`
	
	`</script>`

C#	`using System;`
	`class HelloWorld {`
	`  static void Main() {`
	
	`int x = 12;`
	`int y = 15;`
	`string color = "green";`
	`string xname = "Paul";`
	`bool logic = true;`
	
	`  }`
	`}`

VB	`Private Sub Form_Load()`
	
	`Dim x As Integer`
	`Dim y As Integer`
	`Dim color As String`
	`Dim name As String`
	`Dim logic As Boolean`
	
	`x = 12`
	`y = 15`
	`color = "green"`
	`xname = "Paul"`
	`logic = True`
	
	`End Sub`

PHP	`<?php`
	
	`$x = 12;`
	`$y = 15;`
	`$color = "green";`

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

PHP

```php
    $xname = "Paul";
    $logic = true;

?>
```

PERL

```perl
my $x = 12;
my $y = 15;
my $color = "green";
my $xname = "Paul";
my $logic = true;
```

Ruby

```ruby
x = 12;
y = 15;
color = "green";
xname = "Paul";
logic = true;
```

Java

```java
public class Main
{
    public static void main(String[] args) {

        int x = 12;
        int y = 15;
        String color = "green";
        String xname = "Paul";
        boolean logic = true;

    }
}
```

Python

```python
x = 12
y = 15
color = "green"
xname = "Paul"
logic = True
```

C++

```cpp
#include <iostream>
using namespace std;

int main()
{
    int x = 12;
    int y = 15;
    string color = "green";
    string xname = "Paul";
    bool logic = true;
}
```

Additional algorithm 6.2 (continued)

6.2.2 Naming Variables

In regard to variable names, there are the few good practices that can be mentioned. Variable names must start with a letter and not with a digit. When a variable name starts with the digit, the interpreter will detect this variable as a number which should be followed by an equal sign. Thus, it will rise an error. In order to provide an example, let us imagine an expression like: "$a = 5 + 5a$", or "$5a = 5$". For both of these statements an error will be raised. Also, the variable names are case sensitive and may contain only alpha numeric characters and underscores ("a–z", "A–Z", "0–9", and "_"). Lowercase letters and uppercase letters make the difference. For instance, a variable name like "*Paul*" is not the same as a variable name like "*paul*", the two are separate entities. Usually, variable names do not have word length restrictions. The only rule is that variable names cannot contain *space* characters or *tab* characters. *Space* characters or *tab* characters are used in specific circumstances by the interpreter in order to split the code in meaningful smaller pieces of code. Most important of all, a variable name must not be the same as a reserved keyword, such as computer instructions like "*if*", "*for*", "*print*" and so on. Reserved keywords are able to instruct the interpreter, therefore a variable with the same name as a reserved keyword will be treated by the interpreter as an instruction. All reserved keywords are placed in a specific context, otherwise the lack of context will raise an error. For instance, in context, reserved keywords will not be followed by the equal sign like variables are. As a simple recommendation, variable names should describe their purpose in the overall context of the application. Unless an algorithm is built, it is advisable to avoid single letter variables. Single letter variables allow for short and compact implementations which can be seen all at once by a programmer. Single letter variables are used in two circumstances: (i) The code represents an algorithm with local variables isolated inside a function, or, (ii) the programmer is highly experienced and explains the code and the meaning of the variables in the comments. The shorter the variable name, the more likely it is to reassign the same variable to another purpose by mistake, especially in the case of very complex applications. For instance, for one letter variables there are 24 variable names that can be used. Thus, as the length of the variable name increases, the probability to reassign a variable by mistake, of course, decreases. It goes without saying, however, it is vitally important to distinguish between the name of the variable and the value of the variable. For example, if an expression is considered, in which a variable is named "*color*" and it is equal to a few characters, namely "*green*", then *color* is the name of the variable and "*green*" is the value. Far below, there are some examples in multiple programming and scripting languages that point out those discussed above. If we still consider the abstract example in which variables are associated with boxes, then "*color*" would be the name of the box while "*green*" is what the box contains.

6.2.3 Variables: Explicit and Implicit

Perhaps all computer languages of the past were *typed*, which means that the type of variable is declared before any value is actually stored in it. Thus, explicit declared variables are dynamically reserved memory locations to which values can be stored. Some computer languages can be *untyped*, which means that once they are declared the variables don't have an explicit data type. In other words, once it is declared the variable may take the data type of the value when the assignment is made. Notes: The terms *typed* and *untyped* are actually known as, and refer to, "*strongly typed*" (typed) and "*weakly typed*" (untyped) languages. A computer language can be of both types, *strongly typed* and *weakly typed* at the same time. For example, VB6 is such a programming language, where the programmer can indicate an option at the beginning of the code (*Option Explicit*), which forces the express declarations of all variables. Otherwise, all implicit declarations are considered of the type "*Variant*". But how implicit declarations work? Variables in computer languages like JavaScript or Python do not need explicit declaration to reserve memory space. The trick here is that "explicit" declaration happens automatically when a value is assigned to a variable. In short, when some computer languages lack explicit declarations of variables, they must have an implicit declarations of variables. For example, JavaScript is such a scripting language. In JavaScript there is no possibility to specify that a particular variable represents an integer, or a string, or a real number. Moreover, the same variable can have different data types in different contexts. This simplification of JavaScript represents a colossal evolution of modern software technology. The simple fact that a programmer attaches a data type to a variable on the fly, represents one of the main optimizations in the software technology today. However, when we consider the old customs in software development, we understand the possible downsides on easily changing data types on variables. In the old days, all variables were declared at the beginning of the implementation. Thus, it was hard and time-consuming for a programmer to change the data type of a variable. In this way, the entire software application was built from the beginning on a skeleton foundation. With the advent of this optimization regarding the change of data types for variables on the fly, the good practices have become important for teams of programmers. Thus, the good practices have allowed a common ground and a common anchoring when it comes to specific software projects/applications.

6.2.4 Statically Versus Dynamically Typed Languages

Why is the explicit declaration of variables important? Why don't we go by default with implicit declarations in all computer languages? The simple answer is related to error catching and the optimization that a compiler does. When the type of variable is known in advance, the compiler can use different optimization strategies. Compiler optimization strategies refer to the way a high-level source code is translated into machine code

for higher speed and/or smaller size. In the case of implicit declarations of variables, a compiler has a lot of surprising information, information that it does not know from the beginning, which is why it cannot formulate a global strategy for optimizing the machine code. For example, as shown below in the sub-chapter "*Evaluations of expressions*", there are situations in which the implicit declaration of variables can be done only after a function provides the result or/and an expression is evaluated. Normally, for type-checking a compiler must identify the name and type of variables first and then evaluate expressions and other semantics issues. Thus, in order for a compiler to identify the type of a variable, it would have to evaluate expressions that contain variables themselves. Thus, it is not possible and here a paradoxical situation may arise. Is not by mistake that the majority of dynamically typed computer languages are in fact scripting languages. Regardless of the model of interpretation, scripting languages have no compiler to satisfy and account for. Those that have a kind of compiler, either make no optimizations because the hardware of today is highly powerful and is worth the tradeoff, or, the implementation of the compiler is able to gather information about the source code before the actual compilation process (pre-compilation). Such a pre-scan of the source code means that all implicit declarations can be translated into explicit declarations. That pre-scan implies also the evaluation of expressions to find out the type of variable (a situation discussed above). In short, the source code can be silently interpreted first with information gathering, and compiled later based on that information. Nonetheless, it should be noted that a dynamic type language allows programmers to change the type of a variable on the fly depending on what data is assigned to it (VB6, JavaScript, Python, and so on). Again, as stated above, most scripting languages have this feature as there is no compiler to do static type-checking. Static typing is verbose when compared to dynamic because it declares all variables, parameters, and return values (return values are related to functions that are discussed in the next chapter). To understand what verbose really means, one may take a look at Additional algorithm 6.3 on the VB6 section, where all variables are explicitly declared. Declarations like "*Dim x As Integer*" can be seen, which in simple words means "*make a one-**dim**ension empty space in memory of 2 bytes for a variable named **x** that awaits for data of this size*" (also please see Table 5.2 for reference). One can compare the amount of source code from VB6 with the mirrors from JavaScript, Ruby, or Python, where the source code is short due to implicit variable declarations. Implicit variable declarations are normal in many *weakly typed* computer languages, as reusability is required for optimization.

Lang.	Example	Output

JS	<script>
	
	var x = 12;
	var y = 15;
	var color = "green";
	var xname = "Paul";
	var logic = true;
	
	print(logic);
	print(x + y);
	print(color + x);
	print(x + xname);
	print(x + y + color);
	print(color + x + y);
	print(x + x / x - x * x);
	
	</script>
		JS Output:
		true
		27
		green12
		12Paul
		27green
		green1215
		-131

C#	using System;
	class HelloWorld {
	  static void Main() {
	
	  int x = 12;
	  int y = 15;
	  string color = "green";
	  string xname = "Paul";
	  bool logic = true;
	
	  Console.WriteLine(logic);
	  Console.WriteLine(x + y);
	  Console.WriteLine(color + x);
	  Console.WriteLine(x + xname);
	  Console.WriteLine(x + y + color);
	  Console.WriteLine(color + x + y);
	  Console.WriteLine(x + x / x - x * x);
	  }
	}
		C# Output:
		True
		27
		green12
		12Paul
		27green
		green1215
		-131

VB	Private Sub Form_Load()
	
	  Dim x As Integer
	  Dim y As Integer
	  Dim color As String
		VB Output:
		True
		27
		green12
		12Paul

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

```
    Dim name As String
    Dim logic As Boolean

    x = 12
    y = 15
    color = "green"
    xname = "Paul"
    logic = True

    Debug.Print (logic)
    Debug.Print (x + y)
    Debug.Print (color & x)
    Debug.Print (x & xname)
    Debug.Print (x + y & color)
    Debug.Print (color & x + y)
    Debug.Print (x + x / x - x * x)

End Sub
```

```
27green
green27
-131
```

PHP

```
<?php

    $x = 12;
    $y = 15;
    $color = "green";
    $xname = "Paul";
    $logic = true;

    print ($logic);
    print ($x + $y);
    print ($color . $x);
    print ($x . $xname);
    print ($x + $y . $color);
    print $color . ($x + $y);
    print ($x + $x / $x - $x * $x);

?>
```

```
PHP Output:
1
27
green12
12Paul
27green
Green27
-131
```

PERL

```
my $x = 12;
my $y = 15;
my $color = "green";
my $xname = "Paul";
my $logic = true;

print ($logic);
print ($x + $y);
print ($color . $x);
print ($x . $xname);
print ($x + $y . $color);
print ($color . $x + $y);
print ($x + $x / $x - $x * $x);
```

```
PERL Output:
True
27
green12
12Paul
27green
15
-131
```

Additional algorithm 6.3 (continued)

Ruby

```ruby
x = 12;
y = 15;
color = "green";
xname = "Paul";
logic = true;

puts logic;
puts (x + y);
puts (color + x.to_s);
puts (x.to_s + xname);
puts (x + y).to_s + color;
puts color + (x + y).to_s;
puts (x + x / x - x * x);
```

```
Ruby Output:
true
27
green12
12Paul
27green
green27
-131
```

Java

```java
public class Main
{
    public static void main(String[] args) {

        int x = 12;
        int y = 15;
        String color = "green";
        String xname = "Paul";
        boolean logic = true;

        System.out.println(logic);
        System.out.println(x + y);
        System.out.println(color + x);
        System.out.println(x + xname);
        System.out.println(x + y + color);
        System.out.println(color + x + y);
        System.out.println(x + x / x - x * x);

    }
}
```

```
Java Output:
true
27
green12
12Paul
27green
green1215
-131
```

Python

```python
x = 12
y = 15
color = "green"
xname = "Paul"
logic = True

print (logic)
print (x + y)
print (color + str(x))
print (str(x) + xname)
print (str(x + y) + color)
print (color + str(x + y))
print (x + x / x - x * x)
```

```
Python Output:
True
27
green12
12Paul
27green
green27
-131
```

Additional algorithm 6.3 (continued)

C++

```
#include <iostream>
using namespace std;

int main()
{
    int x = 12;
    int y = 15;
    string color = "green";
    string xname = "Paul";
    bool logic = true;

    cout<<logic<<"\n";
    cout<<(x + y)<<"\n";
    cout<<(color + to_string(x))<<"\n";
    cout<<(to_string(x) + xname)<<"\n";
    cout<<(to_string(x + y) + color)<<"\n";
    cout<<(color + to_string(x + y))<<"\n";
    cout<<(x + x / x - x * x)<<"\n";
}
```

C++ Output:

```
1
27
green12
12Paul
27green
green27
-131
```

Additional algorithm 6.3 (continued)

6.3 Evaluations of Expressions

Much of the field of mathematics is represented two-dimensionally on paper (ex. matrices, exponentiation, summations and so on). In the past, issues obviously existed with the representation of mathematical expressions right from the beginning of the modern era of computers and prior to it. Nevertheless, ultimately these one-dimensional representations often provided more value and practicality than previously expected. Some discussions about the evaluations of one-dimensional expressions may indeed seem trivial, until they are not trivial anymore. How do interpreters/compilers make the evaluations of expressions? and why is this important? In short, this is especially important for computer languages that have an implicit assignment model, because these are the most affected. Expressions may dictate the type of variable by a kind of induction. Namely, the result of the evaluation establishes the type of variable by assignment. For instance, an expression which adds two integers, will be evaluated to an integer (ex. $1 + 1 = 2$). This result is of course expected by any of us. Let us consider a variable name "a". The assignment of the result of the expression to variable a, dictates what type the a variable can be (ex. $a = 1 + 1 = 2$; thus: $a = 2$). Therefore, variable a is an integer type because the result of the evaluation dictated so. On the other hand, an expression which adds a string and a number (again just an integer for simplicity), will evaluate to a string (ex. "Paul" $+ 1 =$ "Paul1"). The assignment of the result of the expression to variable a will convert it into a string variable (ex. $a =$ "Paul" $+ 1 =$ "Paul1"; thus: $a =$ "Paul1"). This is why the assignment of an entire expression to a new variable is extremely important and deserves attention.

The above observations are true for JavaScript. Computer languages use different operator symbols for mathematical expressions. Moreover, some operator symbols may have multiple meanings. For instance, the plus symbol in JavaScript is an arithmetic operator and a concatenation operator, whereas in other languages the two operations have different representative symbols. For these reasons the result of the evaluation may differ from one computer language to another. Therefore, the behavior of the evaluations discussed above in the main text, can be inspected by using new examples for each programming or scripting language used in this work.

The first thing to be noticed, is that for the interpreter the order of the variables does matter. For instance, in Java, JavaScript, and C Sharp (also noted as C#), the "+" operator is able to add different data types on the fly. However, the order in which the content of different data types are added together imposes different results. If a string value is declared first, followed by the "+" operator, and a number value is declared after the "+" operator, then the result will be another string. Vice versa, if a number value is declared first, followed by the "+" operator, and the string value is declared after the "+" operator, the result will be the same, namely a string value. The situation becomes more interesting when a string value is followed by the "+" operator and then by a mathematical expression. In such a case, the result will be a string value because the evaluation made by the interpreter is done from the left to the right of this composite expression. Thus, since the first term is a string value, anything that is added one by one to this string value; will result in another string value. A different behavior is expected when a mathematical expression is found in front of a string value. Namely, the evaluator will encounter the mathematical expression first (from left to right) and the string value after. The output will encompass the result of the mathematical expression, followed by the string value. Thus, order matters. In other programming and scripting languages the "+" operator is not allowed in this manner. For instance, Python, Ruby, Perl, PHP or VB do not allow for the use of the "+" operator in this manner. To be more precise, Python and Ruby use the "+" operator to add two strings together, namely to concatenate two strings. In these two environments string values cannot be added directly to numbers by using the "+" operator. Number data types first require a conversion to strings by the use of special dedicated methods (see below). Languages such as PERL, PHP and VB, require dedicated operators in order to correctly evaluate a composite expression.

6.3.1 Details by Language

A few point-by-point observations can be made here. As expected, in the case of Java the "+" operator works in the same manner has it works in JavaScript. In C#, the use of the "+" operator between a *Boolean* value and a number will of course fail to evaluate (*Operator "+" cannot be applied to operands of type "bool" and "int"*). In Visual

Basic, the "+" operator can be used only for mathematical expressions whereas the addition/concatenation of a string to a number is possible through the use of another operator, namely "&". Note that in PHP the *Boolean* variables are shown as "1" for true and "0" for false. In PHP, string values cannot be concatenated with integers by using the same "+" operator as it is done in JavaScript. Instead, for concatenation the "." character is used as a string operator. In PHP, a string value added to a mathematical expression results in error (ex. "$color. $x + $y;"). In order to avoid the error, the mathematical expression is separated from the concatenation by the use of parentheses (ex. "$color. ($x + $y);"). Thus, the PHP example found in Additional algorithm 6.3, shows that a concatenation of the string with the result (an integer) of the mathematical expression will provide a valid result. The "*print*" instruction followed by the expression "$color. ($x + $y);", yields "Green27". Moreover, the plus operator in Visual Basic and PHP cannot be used as it is done in JavaScript or C#. For concatenation of strings, Visual Basic uses the "&" character as an operator ("Debug.Print (color & x + y)"), whereas PHP uses the "." character as an operator ("print $color. ($x + $y);"). In PERL, the concatenation between a string and any other data type is made by using the "." character as an operator, just like in PHP. Thus, concatenation of two strings will result into another string. Also, the concatenation between a string and a number will also result into a string. In PERL, concatenations that use the "+" operator instead of the "." operator, lead to the error free removal of the string value from the result. For instance, when the "+" operator is used for concatenation between string values and number values, the result is "true2712122727", as it can be seen in Additional algorithm 6.4.

In the result of Additional algorithm 6.4, one may observe that all string values are missing from the output. In other words, a string followed by the "+" operator and then followed by a number value, will output the number value. Vice versa is also true. For

```
print ($logic);           #true
print ($x + $y);          #27
print ($color + $x);      #12
print ($x + $xname);      #12
print ($x + $y + $color); #27
print ($color + $x + $y); #27
print ($color + $x + $x / $x - $x * $x); #-131
print ($color . $x + $x / $x - $x * $x); #-143
```

```
PERL Output:
true2712122727
```

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

instance, in the context of the above example, "print ($color + $x);" will output "12", and a switch of the terms provides "print ($color + $x);", which also outputs "12". Notice however, that in PERL, the use of the "+" operator for concatenations between strings and mathematical expressions, leads to an output in which the string is disregarded and the result of the mathematical expression is preserved and displayed (i.e. "print ($x + $y + $color);" will output "27" and not "27green"). Each programming language or scripting language contains different helpers. These helpers are the internal functions, or the built in functions, which may be considered as some black boxes that receive data and output data. In the case of data type converters, these languages contain special built in functions that are able to switch between data types. In Ruby, there is no implicit conversion of an integer into a string by using the "+" operator. Thus, in order to allow for concatenation, the number is converted to string on the fly by using the ".to_s" method. The same situation can be encountered in Python where numbers must be converted to strings prior to the concatenation process. In Python, a number can be converted into a string on the fly by using the "str()" method. Without the conversion of the number values to string values, the "+" operator cannot automatically join a string and a number by default. Thus, without additional help, strings can be added only to strings. This is true for both Python and Ruby. Note that the last line of code in each computer language from Additional algorithm 6.3, tests the operator precedence and association discussed far above (ex. "$x + x/x - x \times x$"). PERL particularities in regard to different operations sparkled supplemental experimentation in Additional algorithm 6.4. Although the rules and the results are known from the previous chapters, the operator precedence and association example is tested in PERL with a string value and two different operators, namely the addition operator and the concatenation operator (i.e. "$color. $x + $x/$x − $x * $x"). The string variable (i.e. "$color") is positioned first from left to right and the math expression is added to it, either by using the "+" operator or the "." operator. Each of the two operators provide different results in the evaluation of the expression, namely "−131" for the addition operator and "−143" for the concatenation operator. These results should be of note in case of debugging perl scripts.

6.4 Constants

Constants are read-only variables that store immutable (fixed) values called literals. In other words, constants can be viewed as variables with post-definition restrictions. In the source code of any computer language one would like to have keywords that indicate a constant value. Such keywords can set a variable to a specific value once, providing the certainty that the value of the variable never changes. Usually, when the computer language does not allow read-only variables due to various reasons, a constant must be written with upper-case letters so that the programmer knows, by convention, that the

variable is actually a constant. Thus, a good practice is to adhere to the established consensus, in which variables with all upper-case names should be treated as constants. This upper-case notation tells programmers that a variable contains a constant, but this does not prevent the assignment of other values to that variable. Computer languages such as Ruby or Python subscribe to those previously described. The other languages in the list use the "const" keyword to declare a constant. Java uses the keyword "final" pointing out the fact that the assignment for a variable is final and cannot be changed. Often times, constants can store more than simple literals (i.e. "a", 1 or 3.1415, and so on). Computer languages like Javascript can declare constants that store whole objects.

For example, a statement like "const x = ['a', 'b', 'c'];" will be valid, where *x* is a constant whose value cannot be changed (please see the arrays subchapter). As described above, static typing has the advantage of making types immutable and the disadvantage of being too rigid at the same time. Static typing means that variables can store only data of a certain type (i.e. a data type). In computer languages such as C++, C#, Java or VB6, the variable declaration is accompanied by a data type declaration. The same is true for constants, the data type declaration can indicate integers, floating point numbers, characters, strings or boolean values (Additional algorithm 6.5). In C++, C#, Java or VB6, the data type of constants must be declared before the assignment. For instance, the declaration of a constant *x* in the form "*const string x = 'anything';*" will store the string "anything" without the possibility of changing the value of *x*. The same can be done for an integer; for example "*const int x = 10;*", where *x* cannot be changed after it has been declared.

Lang.	Constants	Variable
VB	`Const X As Double = 3.1415926`	`x = 3.14159265358979323846264`
Ruby	`X = 3.14159265358979323846264`	`x = 3.14159265358979323846264`
Python	`X = 3.14159265358979323846264`	`x = 3.14159265358979323846264`
PERL	`use constant X => 3.14159265;`	`$x = 3.14159265358979323846;`
PHP	`define("X", 3.1415926535897);`	`$x = 3.14159265358979323846;`
JS	`const X = 3.1415926535897932;`	`var x = 3.141592653589793238;`
C#	`const double X = 3.141592653;`	`double x = 3.141592653589793;`
Java	`final double X = 3.141592653;`	`double x = 3.141592653589793;`
C++	`const float a = 3.1415926535;`	`float b = 3.1415926535897932;`

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

6.5 Classes and Objects

This subchapter briefly describes the meaning of classes and what these are in relation to the objects. Why is it important to describe the meaning of classes and objects at this stage? Many computer languages that are either pure object oriented or only partially object oriented, can use objects to represent variables. Thus, in such cases a variable goes from its classic meaning of a simple memory address to a more complex meaning of reference to an object. For example, in object oriented computer languages like Ruby, everything is an object. Also, objects must have the ability to interact with each other. In Ruby, objects interact by exchanging messages. If one wishes to find out the length of an array, it sends the "length" message to an instance of an array object. That array object calculates the number of elements present and returns that number to the caller. The above explanation is very different from the classical case, where data was found at the memory address associated with the name of the variable. Thus, although this book is not focused on the subject of classes, understanding objects is important because these structures are used internally to represent variables in many computer languages. Note that object-oriented programming is a paradigm and is not crucial to computers, as non-object-oriented languages make software just as efficient.

6.5.1 About Design Patterns

Classes and objects are often concepts of object-oriented computer languages. A class is a template (or a pattern) from which objects are created. Thus, a class does not occupy memory, only its objects (except perhaps in interpreted languages which use memory, such as, for example, the classloaders from Java). Because these structures are templates, or models, the term "design patterns" refers to the most useful templates that can fit different situations. In other words, object-oriented design patterns describe templates for reusable solutions to frequently encountered context-specific problems. Design patterns also take into account the relationships and interactions between classes and, consequently, between their objects. To exaggerate an association we can say that a class is much like photolithography, where the presence or absence of light can draw shapes on the surface of a photosensitive material. In short, an object is an instance of a class (however, in some computer languages, such as Javascript, objects can be created without using classes). The object is stored in a specific memory location. A reference is a variable that points to the memory location of the variables and methods of the object. Thus, variable names that represent objects are in fact references to objects.

6.5.1.1 Constructors and Destructors

In software engineering, the term "constructor" is often used to point out the construction of an object based on a class. When the terms "constructor" and "destructor" are used,

most of the time the reference is made in regard to objects. For the creation of an object, the term constructor indicates that an object is instantiated (materialization), or an object is brought into existence, so to speak. When this object is no longer required, the term "destructor" is used, which indicates the deletion of the object and the release of the memory occupied by it.

6.6 Arrays

Arrays are the *Alfa & Omega* of all imperative algorithms written in high-level languages. An array is a compound data type that stores numbered pieces of data. Each numbered datum it is called an element of the array and the indexed assigned to it is called an index. The elements of an array may be of any type. In modern languages, an array can even store elements of different types. For instance, inside an array, one element may store numbers and another element may store strings. This is true for the scripting language JavaScript. In older computer languages, arrays were of one data type. To circumvent that, programmers declared the arrays as string data type. In this manner, the programmer was able to store any type of data camouflaged as a string. When data was needed from the array, the first step before anything else was the conversion of the strings into numbers, of course if that was the case. This interesting approach of simulating a modern programming or scripting language by using an older one; was done for years in Visual Basic or in Visual Basic for Applications (Excel) and continues to be done in the same manner today. Of note, the methodology by which numbers are stored as strings and later converted back to numbers when needed, is an approach highly practical even for modern programming or scripting languages. However, for beginners the method is error-prone. The best practice for beginners and even teams of programmers is to follow the universally agreed types throughout the source code without any conversions. Array variables can be used for implementations that require multiple dimensions in their methodology. First, some examples for one-dimensional arrays are given in full below, and later these cases extend to multidimensional array variables.

6.6.1 Creating an Empty Array

To better understand arrays, one needs to observe some examples in different computer languages (Additional algorithm 6.6). As discussed above, a classic variable is represented by a memory location. Because an array normally represents a group of variables, a classic array variable is represented by a sequential allocation of memory. Currently, sequential allocation of memory for arrays is true mainly for compiled computer languages that are not fully object-oriented (ex. C++, VB). For example, arrays in javascript are not

Lang.	Method 1	Method 2
VB	`Dim A() as String`	none
PERL	`@A = ();`	none
Python	`A = []`	none
Ruby	`A = []`	`A = Array.new`
PHP	`$A = [];`	`$A = array();`
JS	`var A = [];`	`var A = new Array();`
C++	`string A[0];`	`string *A = new string[0];`
C#	`string[] A;`	`string[] A = new string[0];`
Java	`String[] A;`	`String[] A = new String[0];`

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

represented by sequential memory allocations. Javascript arrays are objects with enumerable property names and certain important methods, such as ".length", which returns the number of elements in the array.

Depending on the computer language used, there are two ways to create an array, either by using array square parentheses (i.e. "var A = [1, 2, 3];") or by instantiating a constructor function (i.e. "A = new Array(1, 2, 3);"). In purely object-oriented languages (Javascript, Ruby, and so on) everything is represented by objects, including variables. In Ruby, even digits are treated as objects with proprietes. Thus, in such computer languages there is no difference between the array literal and the array constructor, because array is a class that is instantiated in both cases, and the result is the array object.

6.6.1.1 Using the Array Constructor

What is the point of the constructor ? Classically, in Java-like computer languages a class must be instantiated by using the keyword "new". The point of the array constructor is to remind the programmer that the array variable is in fact an object. It is pointless in modern times, but still is a reminder. The use of the array constructors takes longer to write and lacks performance when compared to array literal declarations. However, it is able to create an array with a certain number of empty elements. For instance, passing an argument to the function like "A = new Array(10)", will create an array object with 10 empty elements. In other words, the statement "new Array(10);" creates the object and the reference to it is assigned to the variable named "A".

6.6.1.2 Using Array Literals

In all computer languages, literals are fixed values, that is, values that are written into the source code. Thus, the following types of literals are regularly used: integer literals (ex. "12"), double literals (ex. "4.1234"), string literals (ex. "anything"), and so on. Array literals include the previously mentioned literals. The performance advantage of the array literal is that it provides array compilation when the script is loaded. Note that literal notation becomes the good practice standard in regard to arrays. All examples that follow, use array literals.

6.6.2 Creating an Array with Values

Examples were discussed that showed the ways in which an empty array variable can be declared (Additional algorithm 6.6). Up to this point it is understood that an array is a special variable that can hold an array of values. The following examples show how the array variables are declared without any value, then show the method by which the literals are loaded into the array elements (Additional algorithm 6.7). In a first instance, two array variables, *A* and *B*, are declared empty. Then, to show the use of two types of literals, variable *A* is loaded with three string literals (i.e. "a", "b" and "c") and variable *B* is loaded with integer literals (i.e. 1, 2, 3). The individual elements of each array variable are displayed in the output for visualisation. The elements of array variables are numbered from zero to *n*. Notice that an element of an array variable is accessed with the help of a whole number (the index).

Different peculiarities are also shown for languages such as PERL or Ruby. In PERL, there are many ways to declare an array; either by using the "qw" keyword for string values that are delimited by the space character, or by declaring ranges of numeric or letter values. For computer languages such as C# and Java, the constructor declaration is also shown in the comments (Additional algorithm 6.7). An important aspect to remember in the case of computer languages that are untyped, is the fact that an array variable can hold the reference for several types of data (after all, in those computer languages the array is an object). However, in typed computer languages, an array variable can hold values of only one type.

6.6.3 Adding Elements

The next step shows a series of examples by which elements can be added to an array. Here an empty array variable with the name "A" is declared (Additional algorithm 6.8). Then new elements are generated to which different literal values are assigned, according to the model A[index of the element] = literal. As in the previous example, after constructing the array variable, the individual elements are displayed in the output for visualization,

Lang.	Example	Output

JS	`var A = [];`
	`var B = [];`
	
	`A = ["a", "b", "c"];`
	`B = [1, 2, 3];`
	
	`print(A[0] + A[1] + A[2]);`
	`print(B[0] + B[1] + B[2]);`
		`JS Output:`
		`abc`
		`6`

C#	`using System;`
	`class HelloWorld {`
	`  static void Main() {`
	
	`//string[] A = new string[] {"a", "b", "c"};`
	`//int[] B = new int[] {1, 2, 3};`
	
	`    string[] A = {"a", "b", "c"};`
	`    int[] B = {1, 2, 3};`
	
	`    Console.WriteLine(A[0] + A[1] + A[2]);`
	`    Console.WriteLine(B[0] + B[1] + B[2]);`
	`  }`
	`}`
		`C# Output:`
		`abc`
		`6`

VB	`A = Array("a", "b", "c")`
	`B = Array(1, 2, 3)`
	
	`Debug.Print A(0) & A(1) & A(2)`
	`Debug.Print B(0) + B(1) + B(2)`
		`VB Output:`
		`abc`
		`6`

PHP	`$A = [];`
	`$B = [];`
	
	`$A = ["a", "b", "c"];`
	`$B = [1, 2, 3];`
	
	`print($A[0] . $A[1] . $A[2]);`
	`print($B[0] + $B[1] + $B[2]);`
		`PHP Output:`
		`abc6`

PERL	`@A = ();`
	`@B = ();`
	
	`@A = ("a", "b", "c");`
	`@B = (1, 2, 3);`
		`PERL Output:`
		`abc6`

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

```
print($A[0] . $A[1] . $A[2]);
print($B[0] + $B[1] + $B[2]);

# @A = ("a", "b", "c");
# @B = (1, 2, 3);
# @C = (1, 2, 'c');
# @D = qw/a b c/;
# @E = (4..10);
# @F = (c..p);
```

Ruby

```
A = ["a", "b", "c"]
B = [1, 2, 3]

puts A[0] + A[1] + A[2]
puts B[0] + B[1] + B[2]

#A = Array.new(3, "a")
#puts "#{A}"
```

```
Ruby Output:
abc
6
```

Java

```
public class Main
{
    public static void main(String[] args) {

//String[] A = new String[] {"a", "b", "c"};
//int[] B = new int[] {1, 2, 3};

      String[] A = {"a", "b", "c"};
      int[] B = {1, 2, 3};

      System.out.println(A[0] + A[1] + A[2]);
      System.out.println(B[0] + B[1] + B[2]);
    }
}
```

```
Java Output:
abc
6
```

Python

```
A = ["a", "b", "c"]
B = [1, 2, 3]

print (A[0] + A[1] + A[2])
print (B[0] + B[1] + B[2])
```

```
Python Output:
abc
6
```

C++

```
#include <iostream>
using namespace std;

int main()
{
    string A[] = {"a", "b", "c"};
    int B[] = {1, 2, 3};

    cout<<(A[0] + A[1] + A[2]);
    cout<<(B[0] + B[1] + B[2]);

    return 0;
}
```

```
C++ Output:
abc6
```

Additional algorithm 6.7 (continued)

where the expected result is "abc". We note that in certain cases the number of empty elements must be determined prior to any assignment of literals. For example, in C++, C#, Java, VB6 and Python, the number of elements is predefined, while in other computer languages the array elements can be created at the time of assignment (Additional algorithm 6.8).

Here, one can take Javascript as an example for array elements that can be created at assignment time. In the past, to add new empty elements to an array, one could simply assign a value to it by using a constructor. A statement like "var A = new Array(10);" is an example, where 10 indicates the number of empty array elements. However, one can just as well declare an empty array (i.e. "var A = [];") and insert the number of elements with the help of an additional line (i.e. "A.length = 10;"). The value 10, as discussed above, indicates the number of empty elements in the new array. So the total number of elements is 10, that is: 0, 1, 2, … 9. Next, let us assume that a value (eg. 42) is added to this variable at the element with the index 100 (i.e. "A[100] = 42;"). The question that arises would be: how many elements does the array variable now have? Through experimentation, it can be observed that the number of elements of the array variable is automatically extended up to 101 (i.e. 0,1,2, … 100), where only element 100 will have a value, and the other elements will be empty.

6.6.4 Accessing Array Elements

Thus, array elements are accessed by using the parentheses (ex. "()" or the "[]" operators) and the index of the individual elements (ex. "A[the index]" or "A(the index)", and so on). This approach can list any value from the array. In the following example, an array literal declaration is used in which three string values are declared. Then the value from the element with index 1 (i.e. in Javascript: "var x = A[1];") is assigned to a variable *x* and the value from the element with index 2 (i.e. in Javascript: "var y = A[2];") is assigned to a variable *y*. The two variables *x* and *y* are then displayed in the output. Since the array elements contain string values, consequently the *x* and *y* values also contain the same type and the concatenation operator is used to display them. Thus, the expected result in the output is "bc". In some computer languages this operator is either represented by the "+" sign or by the "." sign.

Some interesting features are commented on in the Ruby computer language. In Ruby, an array can be filled directly with successive values. For example, the statement "A = Array(5.0.9)" will create an array with 5 elements (i.e. 5,6,7,8,9). Besides the classical method shown in Additional algorithm 6.9, in Ruby a value can be extracted from an element by using a statement such as "x = A.at(2)", where number 2 represents the index of the element in the array. Thus, the value from the element with index 2 will be provided, namely number 7 (i.e. index 0 is 5, index 1 is 6, index 2 is 7, index 3 is 8, index 4 is 9).

Lang.	Example	Output

JS	`var A = [];`
	
	`A[0] = "a";`
	`A[1] = "b";`
	`A[2] = "c";`
	
	`print(A[0] + A[1] + A[2]);`
		`JS Output:`
		`abc`

C#	`using System;`
	`class HelloWorld {`
	`  static void Main() {`
	
	`  string[] A = new string[3];`
	
	`  A[0] = "a";`
	`  A[1] = "b";`
	`  A[2] = "c";`
	
	`  Console.WriteLine(A[0] + A[1] + A[2]);`
	`  }`
	`}`
		`C# Output:`
		`abc`

VB	`Dim A(0 To 3) As String`
	
	`A(0) = "a"`
	`A(1) = "b"`
	`A(2) = "c"`
	
	`MsgBox A(0) & A(1) & A(2)`
		`VB Output:`
		`abc`

PHP	`$A = [];`
	
	`$A[0] = "a";`
	`$A[1] = "b";`
	`$A[2] = "c";`
	
	`echo $A[0] . $A[1] . $A[2];`
		`PHP Output:`
		`abc`

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

PERL

```
@A = [];

$A[0] = "a";
$A[1] = "b";
$A[2] = "c";

print $A[0] . $A[1] . $A[2];
```

```
PERL Output:
abc
```

Ruby

```
A = []

A[0] = "a"
A[1] = "b"
A[2] = "c"

puts A[0] + A[1] + A[2]
```

```
Ruby Output:
abc
```

Java

```
public class Main
{
    public static void main(String[] args) {

        String[] A = new String[3];

        A[0] = "a";
        A[1] = "b";
        A[2] = "c";

        System.out.println(A[0] + A[1] + A[2]);
    }
}
```

```
Java Output:
abc
```

Python

```
A = [0]*3

A[0] = "a"
A[1] = "b"
A[2] = "c"

print (A[0] + A[1] + A[2])
```

```
Python Output:
abc
```

C++

```
#include <iostream>
using namespace std;

int main()
{
    string A[3];

    A[0] = "a";
    A[1] = "b";
    A[2] = "c";

    cout<<(A[0] + A[1] + A[2]);

    return 0;
}
```

```
C++ Output:
abc
```

Additional algorithm 6.8 (continued)

Lang.	Example	Output

JS	`A = ["a", "b", "c"];`
	
	`var x = A[1];`
	`var y = A[2];`
	
	`print(x + y);`
		JS Output:
		bc

C#	`using System;`
	`class HelloWorld {`
	`  static void Main() {`
	
	`  string[] A = {"a", "b", "c"};`
	
	`  string x = A[1];`
	`  string y = A[2];`
	
	`  Console.WriteLine(x + y);`
	
	`  }`
	`}`
		C# Output:
		bc

VB	`A = Array("a", "b", "c")`
	
	`x = A(1)`
	`y = A(2)`
	
	`Debug.Print x & y`
		VB Output:
		bc

PHP	`$A = ["a", "b", "c"];`
	
	`$x = $A[1];`
	`$y = $A[2];`
	
	`print($x . $y);`
		PHP Output:
		bc

PERL	`my @A = ("a", "b", "c");`
	
	`my $x = $A[1];`
		PERL Output:
		bc

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

```
my $y = $A[2];

print($x . $y);
```

Ruby

```
A = ["a", "b", "c"]

x = A[1]
y = A[2]

puts x + y

# A = Array(0..9)
# x = A.at(3)
# puts "#{x}"
```

```
Ruby Output:
bc
```

Java

```
public class Main
{
    public static void main(String[] args) {

      String[] A = {"a", "b", "c"};

      String x = A[1];
      String y = A[2];

      System.out.println(x + y);
    }
}
```

```
Java Output:
bc
```

Python

```
A = ["a", "b", "c"]

x = A[1]
y = A[2]

print (x + y)
```

```
Python Output:
bc
```

C++

```
#include <iostream>
using namespace std;

int main()
{
    string A[] = {"a", "b", "c"};

    string x = A[1];
    string y = A[2];

    cout<<(x + y);

    return 0;
}
```

```
C++ Output:
bc
```

Additional algorithm 6.9 (continued)

6.6.5 Changing Values in Array Elements

Next, the following experiment shows how the values inside the elements of an array can be changed (Additional algorithm 6.10). In a first instance, an array literal declaration named *A* is used, where string literals are directly written for each individual element (i.e. *A* = [“a”, “b”, “c”]; where index 0 is “ a”, index 1 is “b” and index 2 is “c”). Then, the value from the array element with index 1 is assigned to a new variable *x*. The *x* variable is a temporary variable because it has the role of holding a value for a later step. The element with index 0 in array *A* is assigned with a string literal, namely the value “d”. Up to this point, the only change to the elements of array *A* is the replacement of the value “a” with a value “d” (i.e. the situation up to the current point is [“d”, “b”, “c”]).

Next, the string value from the element with index 2 of array *A* is assigned to the element with index 1 of array *A* (i.e. the situation up to the current point is [“d”, “c”, “c”]). In the last step, the value from the temporary variable called *x* is assigned to the element with index 2 of the array *A*. Variable *x* held a copy of the initial value from the element with index 1 of the array variable *A* (i.e. the situation up to the current point is [“d”, “c”, “b”]). The content of each element of the array variable *A* is displayed in the output for visualization of the result (Additional algorithm 6.10).

6.6.6 Array Length

In the next step, a series of examples are presented by which the number of elements in an array can be found (Additional algorithm 6.11). Here, there are two possibilities. For computer languages that are less object-oriented, this information can be found by calling an internal function that returns the number of elements in an array variable. For object-oriented computer languages, an array variable is an object, and information related to the number of elements in the array is given by a method of the array object. As in the previous example, an array literal is declared with three values, “a”, “b”, and “c”. Then, a variable *x* is declared and a value is assigned to it. This value can be either returned by a method of the object or it can be returned by an in-built function. Then, this value in *x* is displayed for visualization. One thing to remember when it comes to arrays, is that these composite variables contain a lower bound and an upper bound, namely the place from which they begin the indexing in the first element and the place where the indexing ends, that is, the last element from the array (Fig. 6.1). In many computer languages the lower bound is set to zero.

In object-oriented computer languages, the number of elements inside the array is specified by the length method of the array object, which represents the upper bound, while the lower bound is set to zero. In other less object-oriented computer languages, the lower bound situation is different. To find the index of the last element in the array, an upper bound function is usually available. The same may be true for the first element in the

Lang.	Example	Output

JS	`A = ["a", "b", "c"];`
	
	`var x = A[1];`
	
	`A[0] = "d";`
	`A[1] = A[2];`
	`A[2] = x;`
	
	`print(A[0] + A[1] + A[2]);`
		JS Output:
		dcb

C#	`using System;`
	`class HelloWorld {`
	`  static void Main() {`
	
	`    string[] A = {"a", "b", "c"};`
	
	`    string x = A[1];`
	
	`    A[0] = "d";`
	`    A[1] = A[2];`
	`    A[2] = x;`
	
	`    Console.WriteLine(A[0] + A[1] + A[2]);`
	
	`  }`
	`}`
		C# Output:
		dcb

VB	`A = Array("a", "b", "c")`
	
	`x = A(1)`
	
	`A(0) = "d"`
	`A(1) = A(2)`
	`A(2) = x`
	
	`Debug.Print A(0) & A(1) & A(2)`
		VB Output:
		dcb

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

PHP

```php
$A = ["a", "b", "c"];

$x = $A[1];

$A[0] = "d";
$A[1] = $A[2];
$A[2] = $x;

print($A[0] . $A[1] . $A[2]);
```

```
PHP Output:
dcb
```

PERL

```perl
my @A = ("a", "b", "c");

my $x = $A[1];

$A[0] = "d";
$A[1] = $A[2];
$A[2] = $x;

print($A[0] . $A[1] . $A[2]);
```

```
PERL Output:
dcb
```

Ruby

```ruby
A = ["a", "b", "c"]

x = A[1]

A[0] = "d"
A[1] = A[2]
A[2] = x

puts (A[0] + A[1] + A[2])
```

```
Ruby Output:
dcb
```

Java

```java
public class Main
{
    public static void main(String[] args) {

      String[] A = {"a", "b", "c"};

      String x = A[1];

      A[0] = "d";
      A[1] = A[2];
      A[2] = x;

      System.out.println(A[0] + A[1] + A[2]);
    }
}
```

```
Java Output:
dcb
```

Additional algorithm 6.10 (continued)

Python

```
A = ["a", "b", "c"]

x = A[1]

A[0] = "d"
A[1] = A[2]
A[2] = x

print (A[0] + A[1] + A[2])
```

```
Python Output:
dcb
```

C++

```
#include <iostream>
using namespace std;

int main()
{
    string A[] = {"a", "b", "c"};
    string x;

    x = A[1];

    A[0] = "d";
    A[1] = A[2];
    A[2] = x;

    cout<<(A[0] + A[1] + A[2]);

    return 0;
}
```

```
C++ Output:
dcb
```

Additional algorithm 6.10 (continued)

array, i.e. there may be a lower bound function. Note that the comments in the VB6/VBA example show a particularly important version that is rarely found as a possibility in other computer languages. Namely, an array variable is declared from an arbitrary index to another arbitrary index. This means that in VB6/VBA, array variables can be defined to start at any index, not just at index 0.

For instance, VB6 may have the following valid statement for an array: "*Dim A(−4 To −1) As String*". If these array bounds are calculated and set by an algorithm, then VB is equipped with two built-in functions that return both the lower bound and the upper bound of the array. The total length of the array as the meaning is observed in object-oriented computer languages (i.e. the number of elements in the array) can be calculated in the case of VB. The total number of elements in the array is the result of a subtraction between the value returned by the *UBound* function and the value returned by the *LBound* function (i.e. "total = UBound(A) – LBound(A)").

Lang.	Example	Output

JS	`A = ["a", "b", "c"];`
	
	`var x = A.length;`
	
	`print(x);`
		`JS Output:`
		`3`

C#	`using System;`
	`class HelloWorld {`
	`  static void Main() {`
	
	`  string[] A = {"a", "b", "c"};`
	
	`  int x = A.Length;`
	
	`  Console.WriteLine(x);`
	
	`  }`
	`}`
		`C# Output:`
		`3`

VB	`A = Array("a", "b", "c")`
	
	`x = Ubound(A)`
	
	`Debug.Print x`
	
	`'Dim A(4 To 7) As String`
	
	`'A(4) = "a"`
	`'A(5) = "b"`
	`'A(6) = "c"`
	`'A(7) = "d"`
	
	`'x = LBound(A)`
	`'y = UBound(A)`
	
	`'Debug.Print x & " ... " & y`
		`VB Output:`
		`2`

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

PHP

```php
$A = ["a", "b", "c"];

$x = count($A);

print($x);
```

```
PHP Output:
3
```

PERL

```perl
my @A = ("a", "b", "c");

$x = @A;

print($x);
```

```
PERL Output:
3
```

Ruby

```ruby
A = ["a", "b", "c"]

x = A.size

puts (x)
```

```
Ruby Output:
3
```

Java

```java
public class Main
{
    public static void main(String[] args) {

        String[] A = {"a", "b", "c"};

        int x = A.length;

        System.out.println(x);
    }
}
```

```
Java Output:
3
```

Python

```python
A = ["a", "b", "c"]

x = len(A)

print(x)
```

```
Python Output:
3
```

C++

```cpp
#include <iostream>
using namespace std;

int main()
{
    string A[] = {"a", "b", "c"};

    int x = sizeof(A) / sizeof(string);

    cout<<x;

    return 0;
}
```

```
C++ Output:
3
```

Additional algorithm 6.11 (continued)

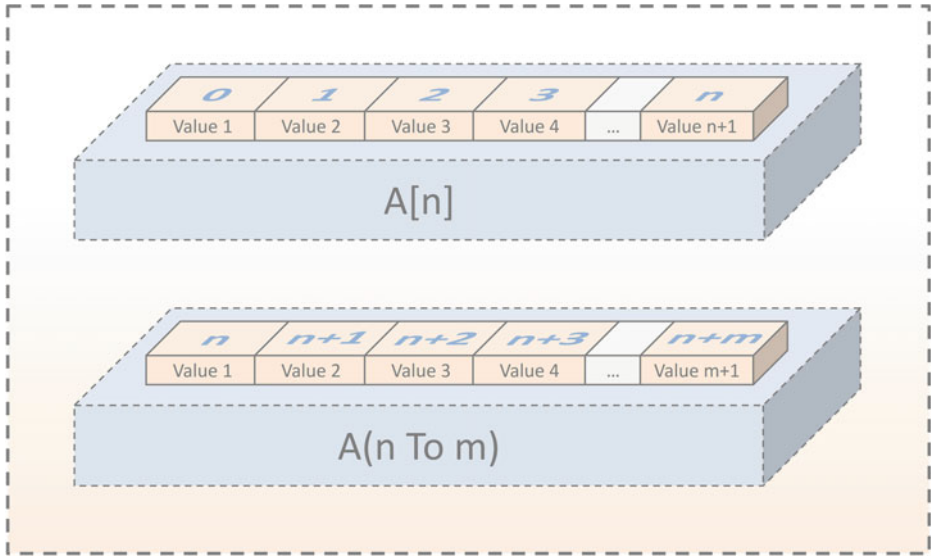


Fig. 6.1 One-dimensional array variables. It presents two different representations of array variables. The first approach from above shows how the lower bound starts from zero (0 … n). Notice that the total number of elements in the array is $n + 1$. This is the case with many modern computer languages. 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 … $n + m$; $m > n$). Notice that the total number of elements in the array is $m + 1$

6.6.7 Nested Arrays

In some object oriented computer languages the array variables can hold other array variables in their elements. This can be done in a nested manner where elements of array variables can hold other array variables and so on. For example, object-oriented computer languages such as Javascript, Ruby, or Python can easily support nested arrays because these structures are objects that hold references to other objects (Additional algorithm 6.12). In classical computer languages the array variable is represented by a sequential assignment of memory and the process of using nested arrays in not that simple. Some computer languages lack support for nested arrays.

In the example, seven array variables are used, namely: *A, B, C, D, E, F* and *G*. Array variables *A, B,* and *C* are loaded with string literals. Next, array variables *D, E* and *F* are loaded with different combinations of arrays *A, B* and *C*. Finally, an array variable *G* is loaded with the array variables *D, E* and *F*. Thus, here there are three levels of nested arrays. The levels can continue if needed. The content of the first element of an array of each nested level is shown in the output for visualization.

Lang.	Example	Output

JS	`var A = ["a", "b", "c"];`
	`var B = ["d", "e", "f"];`
	`var C = ["g", "h", "i"];`
	
	`var D = [A, B, C];`
	`var E = [B, C, A];`
	`var F = [C, B, A];`
	
	`var G = [D, E, F];`
	
	`print (A[0]);`
	`print (D[0]);`
	`print (G[0]);`
		`JS Output:`
		`a`
		`a,b,c`
		`a,b,c,d,e,f,g,h,i`

Ruby	`A = ["a", "b", "c"]`
	`B = ["d", "e", "f"]`
	`C = ["g", "h", "i"]`
	
	`D = [A, B, C]`
	`E = [B, C, A]`
	`F = [C, B, A]`
	
	`G = [D, E, F]`
	
	`print (A[0])`
	`print (D[0])`
	`print (G[0])`
		`Ruby Output:`
		`a`
		
		`["a", "b", "c"]`
		
		`[["a", "b", "c"],`
		`["d", "e", "f"],`
		`["g", "h", "i"]]`

Python	`A = ["a", "b", "c"]`
	`B = ["d", "e", "f"]`
	`C = ["g", "h", "i"]`
	
	`D = [A, B, C]`
	`E = [B, C, A]`
	`F = [C, B, A]`
	
	`G = [D, E, F]`
	
	`print (A[0])`
	`print (D[0])`
	`print (G[0])`
		`Python Output:`
		`a`
		
		`['a', 'b', 'c']`
		
		`[['a', 'b', 'c'],`
		`['d', 'e', 'f'],`
		`['g', 'h', 'i']]`

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

6.6.8 Multidimensional Arrays

Up to this point, an array variable with a single dimension has been discussed. Most of the time, one-dimensional array variables describe vectors. These one-dimensional composite variables can also hold char type elements that form strings (eg. in the computer language C, this is how strings are constructed), or even elements that hold strings, and in some cases references to other objects. In Additional algorithm 6.13, a number of examples are shown for each individual computer language. In many object-oriented computer languages, such as Javascript, PHP, PERL, Ruby or Python, array variables with multiple dimensions can be declared in a nested array and can contain literals of multiple data types. In the case of traditional computer languages, these array variables can also have an unlimited number of dimensions, but only one data type per array. This is the case for C++, C#, Java or VB6.

In Additional algorithm 6.13, some examples are given for Java, namely one example showing the formulation of a two-dimensional array, another example for the formulation of a three-dimensional array (Fig. 6.2). The last Java example shows a declaration of an array variable with a fixed number of dimensions and elements per dimension. Once the array variables have been declared in this manner, individual elements can be loaded with values using the known index range for each dimension.

Note that examples shown for C# are the same as in Java. In VB, three separate cases are shown, some of which resemble the Java and C# examples. The first example shows how an array variable can be declared with a fixed number of dimensions right from the start, then it shows how the individual elements can be loaded with values by using the index for each dimension. Moreover, while in other languages there is only an upper bound, in VB each dimension has a lower bound and an upper bound. A second example for VB shows how the lower bound and upper bound can be found in multiple array dimensions. Precisely, this example shows the statements made for a one-dimensional array, then a two-dimensional array, and then a three-dimensional array. The statement model can be extended for more than three dimensions for any of the mentioned computer languages. The last example for VB shows the possibility of array resizing in the case of two-dimensional array variables; intuitively showing how this statement model can be extended to multiple dimensions.

6.7 Conclusions

Different structures were discussed, from simple variables that can be read and written, to read-only variables, known as constants. In a first phase, these simple variables were described and the values written expressly in the source code were presented under the terminology of literals. Good practices were also the focus of this chapter, especially with

Lang.	Example	Output

JS	`//2D`
	
	`var A = [`
	`  ["a", 88, 146],`
	`  ["b", 34, 124],`
	`  ["c", 96, 564],`
	`  [100, 12, "d"],`
	`];`
	
	`print(A[1][2]);`
		JS Output:
		124
		g,26,884

	`//3D`
	`var A = [`
	`  [`
	`  ["a", 88, 146],`
	`  ["b", 34, 124],`
	`  ["c", 96, 564],`
	`  [100, 12, "d"],`
	`  ],`
	`  [`
	`  ["e", 48, 996],`
	`  ["f", 34, 554],`
	`  ["g", 26, 884],`
	`  [111, 92, "h"],`
	`  ]`
	`  ];`
	
	`print(A[1][2]);`	

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

PHP

```php
$A = [
   ["a", 88, 146],
   ["b", 34, 124],
   ["c", 96, 564],
   [100, 12, "d"],
];

print($A[1][2]);
```

```
PHP Output:
124
```

PERL

```perl
my @A = (
   ["a", 88, 146],
   ["b", 34, 124],
   ["c", 96, 564],
   [100, 12, "d"],
);

print($A[1][2]);
```

```
PERL Output:
124
```

Ruby

```ruby
A = [["a", 88, 146],
     ["b", 34, 124],
     ["c", 96, 564],
     [100, 12, "d"]]

print(A[1][2])
```

```
Ruby Output:
124
```

Python

```python
A = [["a", 88, 146],
     ["b", 34, 124],
     ["c", 96, 564],
     [100, 12, "d"]]

print(A[1][2])
```

```
Python Output:
124
```

Java

```java
public class Main
{
    public static void main(String[] args) {

       int[][] A = new int[][] {{1,2},
                                {3,4},
                                {5,6},
                                {7,8}};

       System.out.println(A[0][1]);
```

```
Java Output:
2
5
6
```

Additional algorithm 6.13 (continued)

```
        int[][][] B = new int[][][] {{
                                      {1,2,3},
                                      {4,5,6}},
                                      {{7,8,9},
                                      {10,11,12}}
                                      };

        System.out.println(B[0][1][1]);
```

```
        int[][] C = new int[2][3];

        C[0][0] = 5;
        C[0][1] = 6;
        C[0][2] = 5;

        C[1][0] = 5;
        C[1][1] = 5;
        C[1][2] = 5;

        System.out.println(C[0][1]);
    }
}
```

C#

```
using System;
class HelloWorld {
  static void Main() {

    int[, ] A = new int[, ] {{1,2},
                             {3,4},
                             {5,6},
                             {7,8}};

    Console.WriteLine(A[0,1]);
```

```
    int[,, ] B = new int[,, ] {
                               {{1,2,3},
                                {4,5,6}},
                               {{7,8,9},
                                {10,11,12}}
```

```
    int[,] C = new int[2,3];
```

```
C# Output:
2
5
6
```

Additional algorithm 6.13 (continued)

```
    C[0,0] = 5;
    C[0,1] = 6;
    C[0,2] = 5;

    C[1,0] = 5;
    C[1,1] = 5;
    C[1,2] = 5;

    Console.WriteLine(C[0,1]);

  }
}
```

C++

```
#include <iostream>
using namespace std;

int main()
{
    int A[][2] = {{1,2},
                  {3,4},
                  {5,6},
                  {7,8}};

    cout<<A[0][1];
```

```
    int B[][2][3] = {
                    {{1,2,3},
                     {4,5,6}},
                    {{7,8,9},
                     {10,11,12}}
                    };

    cout<<"\n"<<B[0][1][1];
```

```
    int C[2][3];

    C[0][0] = 5;
    C[0][1] = 6;
    C[0][2] = 5;

    C[1][0] = 5;
    C[1][1] = 5;
    C[1][2] = 5;

    cout<<"\n"<<C[0][1];

    return 0;
}
```

```
Java Output:
2
5
6
```

Additional algorithm 6.13 (continued)

VB

```
Private Sub Form_Load()

    Dim C(0 To 2, 0 To 3) As Integer

    C(0, 0) = 5
    C(0, 1) = 6
    C(0, 2) = 5

    C(1, 0) = 5
    C(1, 1) = 5
    C(1, 2) = 5

    Debug.Print C(0, 1)

End Sub
```

```
Private Sub Form_Load()

    Debug.Print ("1D -------")

    '1D
    Dim A(2 To 7)

    Debug.Print UBound(A)
    Debug.Print LBound(A)

    '2D
    Dim B(2 To 7, 3 To 9)

    Debug.Print ("2D -------")

    'Debug.Print UBound(B)
    Debug.Print UBound(B, 1)
    Debug.Print LBound(B, 1)

    'Debug.Print LBound(B)
    Debug.Print UBound(B, 2)
    Debug.Print LBound(B, 2)

    '3D
    Dim C(2 To 7, 3 To 9, 8 To 15)

    Debug.Print ("3D -------")

    Debug.Print UBound(C)
    Debug.Print LBound(C)

    Debug.Print UBound(C, 1)
    Debug.Print LBound(C, 1)
```

```
VB Output:
 6
1D -------
 7
 2
2D -------
 7
 2
 9
 3
3D -------
 7
 2
 7
 2
 9
 3
 15
 8
Before--
 1
 2
-------
 1
 1
-------
After--
 1
 2
-------
 1
 3
-------
```

Additional algorithm 6.13 (continued)

```
    Debug.Print UBound(C, 2)
    Debug.Print LBound(C, 2)

    Debug.Print UBound(C, 3)
    Debug.Print LBound(C, 3)

End Sub


Private Sub Form_Load()

    Dim n, m As Integer

    n = 2
    m = 1

    Dim A() As Variant
    ReDim A(1 To n, 1 To m)

    Debug.Print ("Before--")
    Debug.Print LBound(A, 1)
    Debug.Print UBound(A, 1)
    Debug.Print ("-------")
    Debug.Print LBound(A, 2)
    Debug.Print UBound(A, 2)
    Debug.Print ("-------")

    n = n + 1
    m = m + 1

    ReDim Preserve A(1 To m, 1 To n)

    Debug.Print ("After--")
    Debug.Print LBound(A, 1)
    Debug.Print UBound(A, 1)
    Debug.Print ("-------")
    Debug.Print LBound(A, 2)
    Debug.Print UBound(A, 2)
    Debug.Print ("-------")

End Sub
```

Additional algorithm 6.13 (continued)

reference to the way variables are named and declared. The explicit or implicit declaration was explained and some notions related to statically vs dynamically typed computer languages were discussed. To make a connection between operator precedence, operator association and variables, some expressions and different operators were experimented with for each language in the list. Because of the way variables are handled internally in some computer languages, the notion of a class and the notion of an object have been briefly discussed. This was especially important for computer languages that internally,

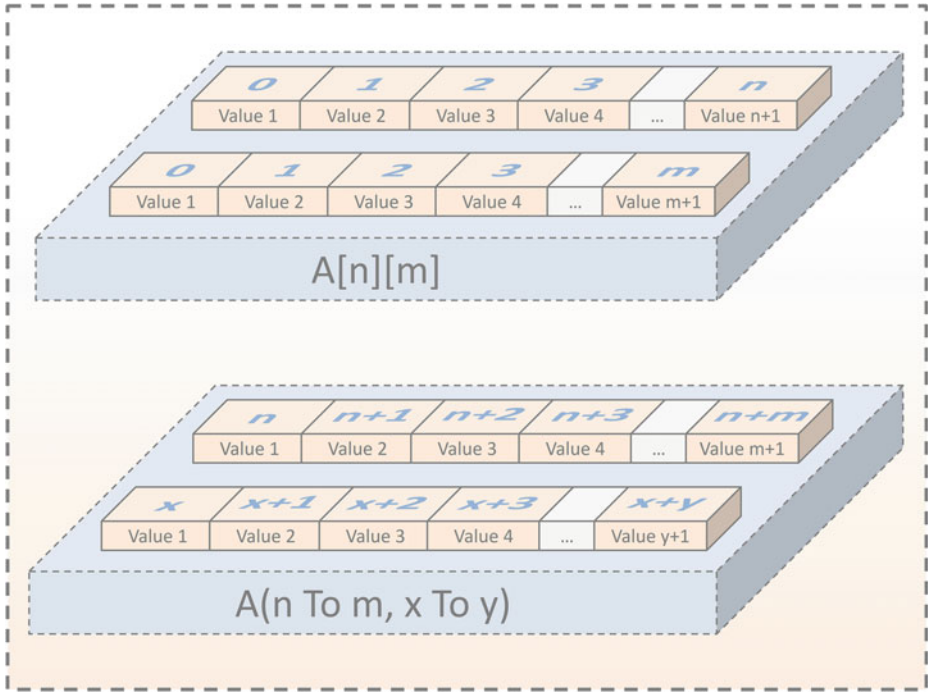


Fig. 6.2 Multi-dimensional arrays. It shows two diagrams that represent array variables with two dimensions. The first diagram shows a lower bound that starts at zero for both dimensions, and the second (bottom) diagram representing an array variable with an arbitrary lower bound position for each dimension. 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

have variables represented by objects, compared to other computer languages whose variables are represented by an association between a name and a memory address. Towards the end of the chapter, the array variables are discussed. Namely, the discussions focused on how one-dimensional array variables are represented in various computer languages (i.e. as objects or as successive memory locations). Numerous examples have been given that show how to declare an array variable, how to create new elements in this type of variable, and other specific operations. Moreover, nested arrays were presented as well as the way in which multidimensional array variables are declared.

Control Structures

7

7.1 Introduction

Decisions made on the results of an abstract rule-based organization of information are the foundations of everything. For humans, decisions represent a conclusion or judgment reached after a logical consideration. In computing, these rule-based organizations and decisions involve control structures and are fully deterministic in our frame of reference because they are made in a closed system, without any interference. Some of these control structures require specific statements, such as conditional statements, which are exemplified in the first part of the chapter. In the second part of the chapter, different types of repeat-loops are discussed. Among the control structures that involve repeat-loops, while-loops and for-loops are presented in detail by extending the previous examples shown in other chapters. Two types of while-repeat-loop structures are discussed, namely the do-loop and the while-loop. Direction of iteration in these structures is exemplified by the use of either some unary operators or the assignment operators. Moreover, the chapter shows how to build nested loop structures for traversing array variables with multiple dimensions. At the very end of the chapter, a strategy of traversing multidimensional array variables is shown by using a single loop. Note that the solutions are interpreted in several computer languages.

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/978-3-031-23277-0_7.

P. A. Gagniuc, *An Introduction to Programming Languages: Simultaneous Learning in Multiple Coding Environments*, Synthesis Lectures on Computer Science,
https://doi.org/10.1007/978-3-031-23277-0_7

7.2 Conditional Statements

In short, the "if-then" statement allows programmers to make a decision based on previous values. An "if-then" statement is used whenever a "fork" is reached in the implementation. Conditions can be written in several ways. Please consider three variables *a*, *b*, and *c*. The target in this example is a different value for variable *c*, depending on the values from variables *a* and *b*. The content of variables a and *b* is known here because it is declared, but, within an implementation, the content of variables *a* and *b* can be uncertain depending on previous factors, external to the implementation. Most of the time a condition can be written as follows:

```
If a < b Then c = 4 else c = 1
```

Which is the same as:

```
c = 1
if a < b then c = 4
```

If the relationship between *a* and *b* is true, the first condition sets the value of variable *c*. In the second example, the condition only changes the value of variable *c*. Below are some examples in different computer languages. Three variables are declared, namely: $a = 1$, $b = 2$ and $c = 3$ (Additional algorithm 7.1). Then, variable *c* is further incremented only if the value in variable *a* is lower than the value in variable *b*, otherwise the value in variable *c* is decremented. The result stored in *c* is then printed in output. Since $a = 1$ and $b = 2$, the condition will allow the increase of the value stored in variable *c*, from the value 3 to the final value 4.

Where applicable, the use of the aggregate assignment incrementation (i.e. "+=") or the aggregate assignment decrementation (i.e. "−=") can be observed. In computer languages where the aggregate assignments are not permitted, there is a replacement of these aggregate assignments with expressions (i.e. "c = c + 1"). A simple demonstration of the conditions using simple integer variables was proposed above (Additional algorithm 7.2). These examples can be extended to more complex variables such as array variables. As previously discussed with array variables, their elements can be accessed by using an index (integer).

Let us consider a variable array *A*, with three elements. The first element contains value 1, the second element contains value 2 and the third element contains value 3 (i.e. $A = [1, 2, 3]$). Then, the value in the last element of the array is incremented only if a condition confirms that the value in the first element of the array (i.e. "A[0]") is smaller than the second element (i.e. "A[1]"). The value from the last element of the array is then displayed in the output. The aggregate assignment incrementation (i.e. "+=") or the aggregate assignment decrementation (i.e. "−=") are used in computer languages where the aggregate assignments are permitted. Note that instead of the aggregate assignment operators

Lang.	Example	Output

JS	`a = 1;`
	`b = 2;`
	`c = 3;`
	
	`if(a < b){c += 1;} else {c -= 1;}`
	
	`print("c=" + c);`
		`JS Output:`
		`c=4`

C#	`using System;`
	`class HelloWorld {`
	`  static void Main() {`
	
	`  int a = 1;`
	`  int b = 2;`
	`  int c = 3;`
	
	`  if(a < b){c += 1;} else {c -= 1;}`
	
	`  Console.WriteLine("c=" + c);`
	`  }`
	`}`
		`C# Output:`
		`c=4`

VB	`Dim a, b, c As Integer`
	
	`a = 1`
	`b = 2`
	`c = 3`
	
	`If a < b Then c = c + 1 Else c = c - 1`
	
	`Debug.Print c`
		`VB Output:`
		`c=4`

PHP	`$a = 1;`
	`$b = 2;`
	`$c = 3;`
	
	`if($a < $b){$c += 1;} else {$c -= 1;}`
	
	`echo "c=" . $c;`
		`PHP Output:`
		`c=4`

PERL	`my $a = 1;`
	`my $b = 2;`
	`my $c = 3;`
	
	`if($a < $b){$c += 1;} else {$c -= 1;}`
	
	`print "$/c=" . $c;`
		`PERL Output:`
		`c=4`

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

Ruby

```ruby
a = 1
b = 2
c = 3

if a < b then c += 1 else c -= 1 end

puts "c=" + c.to_s
```

```
Ruby Output:
c=4
```

Java

```java
public class Main
{
    public static void main(String[] args) {

        int a = 1;
        int b = 2;
        int c = 3;

        if(a < b){c += 1;} else {c -= 1;}

        System.out.println("c=" + c);
    }
}
```

```
Java Output:
c=4
```

Python

```python
a = 1
b = 2
c = 3

if a < b: c += 1
else: c -= 1

print ("c=" + str(c))
```

```
Python Output:
c=4
```

C++

```cpp
int main()
{
    int a = 1;
    int b = 2;
    int c = 3;

    if(a < b){c += 1;} else {c -= 1;}

    cout<<"c="<<c;

    return 0;
}
```

```
C++ Output:
c=4
```

Additional algorithm 7.1 (continued)

Lang.	Example	Output

JS	`var A = [1, 2, 3];`
	
	`if(A[0] < A[1]){A[2] += 1;}`
	
	`print("A[2]=" + A[2]);`
		JS Output:
		A[2]=4

	`var A = [];`
	
	`A[0] = 1;`
	`A[1] = 2;`
	`A[2] = 3;`
	
	`if(A[0] < A[1]){A[2] += 1;}`
	
	`print("A[2]=" + A[2]);`	

C#	`using System;`
	`class HelloWorld {`
	`  static void Main() {`
	
	`  int[] A = {1, 2, 3};`
	
	`  if(A[0] < A[1]){A[2] += 1;}`
	
	`  Console.WriteLine("A[2]=" + A[2]);`
	`  }`
	`}`
		C# Output:
		A[2]=4

VB	`A = Array(1, 2, 3)`
	
	`If A(0) < A(1) Then A(2) = A(2) + 1`
	
	`Debug.Print "A(2)=" & A(2)`
		VB Output:
		A(2)=4

PHP	`$A = [1, 2, 3];`
	
	`if($A[0] < $A[1]){$A[2] += 1;}`
	
	`echo "A[2]=" . $A[2];`
		PHP Output:
		A[2]=4

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

PERL

```
my @A = (1, 2, 3);

if($A[0] < $A[1]){$A[2] += 1;}

print "$/A[2]=" . $A[2];
```

```
PERL Output:
$A[2]=4
```

Ruby

```
A = [1, 2, 3]

if A[0] < A[1] then A[2] += 1 end

puts "A[2]=" + A[2].to_s
```

```
Ruby Output:
A[2]=4
```

Java

```
public class Main
{
    public static void main(String[] args) {

        int[] A = {1, 2, 3};

        if(A[0] < A[1]){A[2] += 1;}

        System.out.println("A[2]=" + A[2]);
    }
}
```

```
Java Output:
A[2]=4
```

Python

```
A = [1, 2, 3];

if A[0] < A[1]: A[2] += 1

print ("A[2]=" + str(A[2]))
```

```
Python Output:
A[2]=4
```

C++

```
#include <iostream>
using namespace std;

int main()
{
    int A[] = {1, 2, 3};

    if(A[0] < A[1]){A[2] += 1;}

    cout<<"A[2]="<<A[2];

    return 0;
}
```

```
C++ Output:
A[2]=4
```

Additional algorithm 7.2 (continued)

(i.e. "+="; "−="), the operators for increment and decrement can be used directly (i.e. "++"; "−", namely "A[2]++" instead of "A[2]+= 1" and "A[2]–" instead of "A[2]−= 1"). Note that in the case of Javascript there are two examples. An example that uses an array literal as in all cases from Additional algorithm 7.2, and another example where the array is declared empty and the elements are step by step associated with values. The rest of the examples use the array literal and assume that the second method from Javascript can be deduced from Additional algorithm 6.8.

7.3 Repeat Loops

Up to this point, values have been assigned to variables line by line. The question that arises is: How to perform an algorithmic distribution of values in variables? This process is done using repeat loops. Loops are a group of statements that repeat until a specified condition is met. Repeat loops are very powerful programming methods, especially for imperative computer languages. Two main types of repeat loop structures can be used: while loops and for loops. However, these are not the only possibilities for a loop manipulation of values. For instance, "goto" statements (i.e. conditional jumps) and conditions, or recursion of functions and conditions, manly do the same thing (but are not popular among programmers for various reasons). The next chapter discusses recursion with clear examples.

7.3.1 The While Loop

The "while loop" is used to execute a block of code while a certain condition is true. Please consider a variable *i* with the value zero, which is incremented inside a while loop until a condition indicates the upper bound for the contents of variable *i*. We can take some examples in javascript, which can be extrapolated to other computer languages. For instance, the following example can be written:

```
let i = 0;
while (i <= 5) {
    print(i);
    i++;
}

let i = 0;
do {
  print(i);
  i++;
} while (i <= 5);
```

Where in both cases the result is a sequence of values from 0 to 5. Notice a second type of statement, namely the "do—while" statement, which has exactly the same meaning as the first. Just as well, the value in variable i can be equal to ten (i.e. $i = 10$), and the white loop can do a decrement of i, where the condition for breaking the loop is for the value in i to become zero or less than zero. For instance, the following example can be written:

```
let i = 5;
while (i >= 0) {
    print(i);
    i--;
}

let i = 5;
do {
  print(i);
  i--;
} while (i >= 0);
```

Above, the result is a sequence of values from 5 to 0. Below multiple examples can be observed for all computer languages from our list (Additional algorithm 7.3). These examples are given only for the incremental version of the above example to conserve paper space. Of course, please notice the replacement of the increment operator with a statement when the computer language does not allow it (i.e. i++; $i = i + 1$;).

In older computer languages that maintained a low-level connection to the assembly language style (the non-structured programming paradigm), the "goto" keyword for absolute jump statements still exists. In structured imperative computer languages of today, keywords for absolute jumps like "goto", no longer exist as an option. The "goto" statement allows the move of execution thread to an arbitrary line (this is why it is called absolute). Thus, instead of "white loop" structures, conditional jumps can be used instead. A conditional jump is represented by an absolute jump and a condition. For instance, in the computer language Visual Basic 6.0, which has the option for absolute jumps, the "while loop" can be replaced by:

```
//first goto example
Private Sub Form_Load()

    Dim i As Integer
    i = 0
1:
    Debug.Print i
    i = i + 1
    If i > 5 Then GoTo 2
    GoTo 1
2:

End Sub
```

<table>
<tr><th>Lang.</th><th>Example</th><th>Output</th></tr>
<tr><td>JS</td><td>

```
let i = 0;

while (i < 5) {
  print("i = " + i);
  i++;
}
```

</td><td rowspan="2">

```
JS Output:
i = 0
i = 1
i = 2
i = 3
i = 4
```

</td></tr>
<tr><td>

```
let i = 0;

do {
  print("i = " + i);
  i++;
}
while (i < 5);
```

</td></tr>
<tr><td>C#</td><td>

```
using System;
class HelloWorld {
  static void Main() {

        int i = 0;

        while (i < 5) {
          Console.WriteLine("i = " + i);
          i++;
        }
  }
}
```

</td><td rowspan="2">

```
C# Output:
i = 0
i = 1
i = 2
i = 3
i = 4
```

</td></tr>
<tr><td>

```
using System;
class HelloWorld {
  static void Main() {

        int i = 0;

        do {
          Console.WriteLine("i = " + i);
          i++;
        }
        while (i < 5);
  }
}
```

</td></tr>
<tr><td>VB</td><td>

```
Dim i As Integer
i = 0
```

</td><td>

```
VB Output:
i = 0
```

</td></tr>
</table>

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

```
Do While i < 5

    Debug.Print "i = " & i
    i = i + 1

Loop
```

```
i = 1
i = 2
i = 3
i = 4
```

```
Dim i As Integer
i = 0

Do
    Debug.Print "i = " & i
    i = i + 1

Loop Until i >= 5
```

PHP

```
$i = 0;

while ($i < 5) {
  echo "\n i = " . $i;
  $i++;
}
```

```
PHP Output:
i = 0
i = 1
i = 2
i = 3
i = 4
```

```
$i = 0;

do {
  echo "\n i = " . $i;
  $i++;
}
while ($i < 5);
```

PERL

```
my $i = 0;

while ($i < 5) {
  print "\n i = " . $i;
  $i++;
}
```

```
PERL Output:
i = 0
i = 1
i = 2
i = 3
i = 4
```

```
my $i = 0;

do {
  print "\n i = " . $i;
  $i++;
}
while ($i < 5);
```

Additional algorithm 7.3 (continued)

Ruby

```ruby
i = 0

while i < 5
    puts "i = " + i.to_s
    i += 1
end
```

```ruby
i = 0

loop do

    puts "i = " + i.to_s
    i += 1

    if i >= 5 then break end
end
```

```
Ruby Output:
i = 0
i = 1
i = 2
i = 3
i = 4
```

Java

```java
public class Main
{
    public static void main(String[] args) {

        int i = 0;

        while (i < 5) {
          System.out.println("i = " + i);
          i++;
        }
    }
}
```

```java
public class Main
{
    public static void main(String[] args) {

        int i = 0;

        do {
          System.out.println("i = " + i);
          i++;
        }
        while (i < 5);
    }
}
```

```
Java Output:
i = 0
i = 1
i = 2
i = 3
i = 4
```

Python

```python
i = 0

while i < 5:
  print('i = ' + str(i))
  i += 1
```

```
Python Output:
i = 0
i = 1
i = 2
```

Additional algorithm 7.3 (continued)

```
i = 3
i = 4
```

C++

```
#include <iostream>
using namespace std;

int main()
{
    int i = 0;

    while (i < 5) {
        cout<<"\n i = "<<i;
        i++;
    }

    return 0;
}
```

```
C++ Output:
i = 0
i = 1
i = 2
i = 3
i = 4
```

```
#include <iostream>
using namespace std;

int main()
{
    int i = 0;

    do {
        cout<<"\n i = "<<i;
        i++;
    }
    while (i < 5);

    return 0;
}
```

Additional algorithm 7.3 (continued)

```
//second goto example
Private Sub Form_Load()

    Dim i As Integer
    i = 0
1:
    Debug.Print i
    i = i + 1
    If i <= 5 Then GoTo 1
2:

End Sub
```

In the first example from above, two lines that are labeled “1:” and “2:”, make up the sections of this implementation. Between line 1 and 2 a variable *i* is incremented and

a condition checks if the value of the *i* variable exceeds a certain limit (i.e. 5). If the condition is false, i.e. *i* is less than the limit, then it does not trigger any other statements. However, the line after this condition makes an absolute jump back to line 1, were variable *i* is again incremented. This cycle continues until the condition is true, which results in an absolute jump to line 2, where the execution ends. Of course, this structure can be greatly simplified by reformulating the condition. In the second example above, the condition triggers an absolute jump to line 1, only if *i* is less than or equal to a certain limit, namely value 5. Otherwise, if nothing triggers, the execution is terminated naturally by continuing to line 2. This second solution is of course the elegant one. Additionally, repeated loops can also be achieved recursively using functions, however, these structures will be discussed separately. Coming back to the while loop statements, they can be extremely important for looping through array elements using the value from a variable as an index. Thus, to enable more efficient software designs, iterative loops are ideally suited for working with arrays. The examples from Additional algorithm 7.4 show two types of statements that allow an array traversal.

The examples above use array literals with one character in each element of the array (Additional algorithm 7.4). A traversal of the array elements is done with the help of a "while loop" and simultaneously the values of each element are added to a string variable (i.e. *t*), which is then printed in the output for inspection. It can be seen how in this condition of the while loop the upper limit is represented by the number of elements in the array. Note that depending on the computer language, the number of elements is given either by the object method or by a specialized function.

7.3.2 The For Loop

The structure of the "while loop" was first described. However, the second structure of the repeat loop is by far the most used loop in all computer languages. This structure is the "for loop". The "for loop" is used when the number of steps is known, and a counter is needed for some variables inside a block of code. The counter is initialized before the loop begins in most computer languages and is tested after each iteration to see if it is below the target value. To understand the for loop strategy, some examples can be of great value. Below are a number of examples in several computer languages that explain the for loop on simple integer variables (Additional algorithm 7.5). The simplest example would be the output of the value from a counter variable. As an example, a "for loop" from the C-like family of computer languages can be shown:

```
for (i = 1; i <= 10; i++){
    print(i);
}
```

Lang.	Example	Output

JS	`A = ["a", "b", "c", "d", "e", "f", "g"];`
	
	`let i = 0;`
	`let t = '';`
	
	`while (i < A.length) {`
	
	`    t += "\n i[" + i + "]=" + A[i];`
	`    i++;`
	`}`
	
	`print(t);`
		JS Output:
		i[0]=a
		i[1]=b
		i[2]=c
		i[3]=d
		i[4]=e
		i[5]=f
		i[6]=g

	`A = ["a", "b", "c", "d", "e", "f", "g"];`
	
	`let i = 0;`
	`let t = '';`
	
	`do {`
	`    t += "\n i[" + i + "]=" + A[i];`
	`    i++;`
	`}`
	`while (i < A.length);`
	
	`print(t);`	

C#	`using System;`
	`class HelloWorld {`
	`  static void Main() {`
	
	`    string[] A = {"a", "b", "c", "d", "e"};`
	
	`    int i = 0;`
	`    string t = "";`
	
	`    while (i < A.Length) {`
	`        t += "\n i[" + i + "]=" + A[i];`
		C# Output:
		i[0]=a
		i[1]=b
		i[2]=c
		i[3]=d
		i[4]=e

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

```
        i++;
    }

    Console.WriteLine(t);
  }
}
```

```
using System;
class HelloWorld {
  static void Main() {

    string[] A = {"a", "b", "c", "d", "e"};

    int i = 0;
    string t = "";

    do {
        t += "\n i[" + i + "]=" + A[i];
        i++;
    }
    while (i < A.Length);

    Console.WriteLine(t);
  }
}
```

VB

```
Dim i As Integer

A = Array("a", "b", "c", "d", "e", "f", "g")

i = 0

Do While i <= UBound(A)

    t = t & vbCrLf & "i[" & i & "]=" & A(i)
    i = i + 1

Loop

Debug.Print t
```

```
Dim i As Integer

A = Array("a", "b", "c", "d", "e", "f", "g")

i = 0
```

```
VB Output:
i[0]=a
i[1]=b
i[2]=c
i[3]=d
i[4]=e
i[5]=f
i[6]=g
```

Additional algorithm 7.4 (continued)

```
Do
    t = t & vbCrLf & "i[" & i & "]=" & A(i)
    i = i + 1

Loop Until i > UBound(A)

Debug.Print t
```

PHP

```
$A = ["a", "b", "c", "d", "e", "f", "g"];

$i = 0;
$t = '';

while ($i < count($A)) {

    $t .= "\n i[" . $i . "]=" . $A[$i];
    $i++;
}

echo $t;
```

```
PHP Output:
i[0]=a
i[1]=b
i[2]=c
i[3]=d
i[4]=e
i[5]=f
i[6]=g
```

```
$A = ["a", "b", "c", "d", "e", "f", "g"];

$i = 0;
$t = '';

do {
    $t .= "\n i[" . $i . "]=" . $A[$i];
    $i++;
}
while ($i < count($A));

echo $t;
```

PERL

```
my @A = ("a", "b", "c", "d", "e", "f", "g");

my $i = 0;
my $t = '';

while ($i < scalar(@A)) {

    $t .= "\n i[" . $i . "]=" . $A[$i];
    $i++;
}

print $t;
```

```
PERL Output:
i[0]=a
i[1]=b
i[2]=c
i[3]=d
i[4]=e
i[5]=f
i[6]=g
```

Additional algorithm 7.4 (continued)

```
my @A = ("a", "b", "c", "d", "e", "f", "g");

my $i = 0;
my $t = '';

do {
    $t .= "\n i[" . $i . "]=" . $A[$i];
    $i++;
} while ($i < scalar(@A));

print $t;
```

Ruby

```
A = ["a", "b", "c", "d", "e", "f", "g"]

i = 0
t = ''

while i < A.size
    t += "\n i[" + i.to_s + "]=" + A[i].to_s
    i += 1
end

puts t
```

```
Ruby Output:
i[0]=a
i[1]=b
i[2]=c
i[3]=d
i[4]=e
i[5]=f
i[6]=g
```

```
A = ["a", "b", "c", "d", "e", "f", "g"]

i = 0
t = ''

loop do
    t += "\n i[" + i.to_s + "]=" + A[i].to_s
    i += 1

    if i >= A.size then break end
end

puts t
```

Java

```
public class Main
{
    public static void main(String[] args) {

        String[] A = {"a", "b", "c", "d", "e"};

        int i = 0;
        String t = "";

        while (i < A.length) {
              t += "\n i[" + i + "]=" + A[i];
              i++;
        }
```

```
Java Output:
```

Additional algorithm 7.4 (continued)

```
        System.out.println(t);
    }
}
```

```
public class Main
{
    public static void main(String[] args) {

        String[] A = {"a", "b", "c", "d", "e"};

        int i = 0;
        String t = "";

        do {
            t += "\n i[" + i + "]=" + A[i];
            i++;
        }
        while (i < A.length);

        System.out.println(t);
    }
}
```

Python

```
A = ["a", "b", "c", "d", "e", "f", "g"]

i = 0
t = ''

while i <= len(A)-1:
  t += "\n i[" + str(i) + "]=" + str(A[i])
  i += 1

print(t)
```

```
Python Output:
i[0]=a
i[1]=b
i[2]=c
i[3]=d
i[4]=e
i[5]=f
i[6]=g
```

C++

```
#include <iostream>
using namespace std;

int main()
{
    string A[] = {"a", "b", "c", "d", "e"};
    int n = sizeof(A) / sizeof A[0];

    int i = 0;
    string t = "";

    while (i < n) {
        t += "\n i[" + to_string(i);
        t += "]=" + A[i];
        i++;
    }

    cout<<t;
    return 0;
}
```

```
C++ Output:
i[0]=a
i[1]=b
i[2]=c
i[3]=d
i[4]=e
```

Additional algorithm 7.4 (continued)

```
#include <iostream>
using namespace std;

int main()
{
    string A[] = {"a", "b", "c", "d", "e"};
    int n = sizeof(A) / sizeof A[0];

    int i = 0;
    string t = "";

    do {
        t += "\n i[" + to_string(i);
        t += "]=" + A[i];
        i++;
    }
    while (i < n);

    cout<<t;
    return 0;
}
```

Additional algorithm 7.4 (continued)

where the "for loop" from above, prints the value of i at each iteration, from zero to ten (i.e. 1 … 10). The first declaration in the round parentheses represents (i.e. $i = 1$) the initialization part of the counter variable i. The second statement indicates a condition that verifies the upper limit for the number of iterations (i.e. $i <= 10$). The third statement updates the counter at each iteration (i.e. i++). In this specific case, the third statement uses the increment operator to indicate that one is added to the value of the counter variable at each iteration. In order to show the "direction" of the iteration, another example can use the decrement operator:

```
for (i = 10; i >= 1; i--){
    print(i);
}
```

where the "for loop" prints the value of i at each iteration, from ten to one (i.e. 10 … 1). In this example the i variable is initialized with the upper bound value, namely the number ten (i.e. $i = 10$). The condition is then modified to check the lower bound of i (i.e. $i >= 1$), as the decrement operator subtracts one from the counter value at each iteration (i.e. i–). Multiple examples can be observed in the syntax of all computer languages from our list (Additional algorithm 7.5). In order to conserve paper space, these examples are given only for the incremental version of the "for loop". Note: When a computer language does not allow the increment operator, an assignment statement is used instead (i.e. i++; $i = i + 1$;). In the examples from Additional algorithm 7.5, two variables a and b are

Lang.	Example	Output

JS	`let a = 5;`
	`let b = 0;`
	
	`for (let i = 0; i < a; i++) {`
	`  b += i;`
	`}`
	
	`print(b);`
		`JS Output:`
		`10`

C#	`using System;`
	`class HelloWorld {`
	`  static void Main() {`
	
	`  int a = 5;`
	`  int b = 0;`
	
	`  for (int i = 0; i < a; i++) {`
	`  b += i;`
	`  }`
	
	`  Console.WriteLine(b);`
	`  }`
	`}`
		`C# Output:`
		`10`

VB	`Private Sub Form_Load()`
	
	`  A = 4`
	`  b = 0`
	
	`  For i = 0 To A`
	`  b = b + i`
	`  Next i`
	
	`  Debug.Print (b)`
	
	`End Sub`
	
	`Private Sub Form_Load()`
	
	`  Dim A(4 To 8) As Integer`
	
	`  A(4) = 1`
	`  A(5) = 2`
	`  A(6) = 3`
	`  A(7) = 4`
		`VB Output:`
		`10`
		
		`A[0]=1`
		`A[1]=2`
		`A[2]=3`
		`A[3]=4`
		`A[4]=5`
		
		`A[4]=1`
		`A[5]=2`
		`A[6]=3`
		`A[7]=4`
		`A[8]=5`
		
		`A[0]=0`
		`A[1]=0`
		`A[2]=1`
		`A[3]=2`
		`A[4]=3`
		`A[5]=0`

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

```
    A(8) = 5

    u = UBound(A)
    l = LBound(A)

    For n = 0 To (u - l)
        t = t & "A[" & n & "]=" & A(l + n)
        t = t & vbCrLf
    Next n

    Debug.Print t

End Sub
```

```
A[6]=0
A[7]=0
A[8]=0
```

```
Private Sub Form_Load()

    Dim A(4 To 8) As Integer

    A(4) = 1
    A(5) = 2
    A(6) = 3
    A(7) = 4
    A(8) = 5

    For n = LBound(A) To UBound(A)
        t = t & "A[" & n & "]=" & A(n)
        t = t & vbCrLf
    Next n

    Debug.Print t

End Sub
```

```
Private Sub Form_Load()

    Dim A(0 To 8) As Integer

    A(2) = 1
    A(3) = 2
    A(4) = 3

    For n = 0 To UBound(A)
        t = t & "A[" & n & "]=" & A(n)
        t = t & vbCrLf
    Next n

    Debug.Print t

End Sub
```

Additional algorithm 7.5 (continued)

PHP

```php
$a = 5;
$b = 0;

for ($i = 0; $i < $a; $i++) {
  $b += $i;
}

echo $b;
```

```
PHP Output:
10
```

PERL

```perl
my $a = 5;
my $b = 0;

for (my $i = 0; $i < $a; $i++) {
  $b += $i;
}

print $b;
```

```
PERL Output:
10
```

Ruby

```ruby
a = 4
b = 0

for i in 0..a
   b += i
end

puts b
```

```
Ruby Output:
10
```

Java

```java
public class Main
{
    public static void main(String[] args) {

        int a = 5;
        int b = 0;

        for (int i = 0; i < a; i++) {
          b += i;
        }

        System.out.println(b);
    }
}
```

```
Java Output:
10
```

Python

```python
a = 5
b = 0
```

```
Python Output:
10
```

Additional algorithm 7.5 (continued)

```
for i in range(0, a):
   b += i

print (b)
```

C++

```
#include <iostream>
using namespace std;

int main()
{
    int a = 5;
    int b = 0;

    for (int i = 0; i < a; i++) {
        b += i;
    }

    cout<<b;
    return 0;
}
```

```
C++ Output:
10
```

Additional algorithm 7.5 (continued)

initialized with integer values (i.e. $a = 5$; $b = 0$). Then a "for loop" uses a variable i as a counter, where the upper limit of the loop is dictated by the a variable. Inside the loop, the value of the i variable is added to the value from variable b at each iteration (i.e. $b = b + i$; namely for $b = 0 + 0$, $b = 0 + 1$, $b = 1 + 2$, $b = 3 + 3$, $b = 6 + 4$, were the condition for the end of the loop is true; $i < a$). In Additional algorithm 7.5, three more examples are given as a bonus for VB6 to show the useful and interesting way of arrays and their bounds in the context of a "for loop".

The "while loops" and "for" loops" are essential to all algorithm designs. However, the "while" types of cycles are both a blessing and a course. For instance, in the majority of previous computer languages, "do-while loops" or "while loops" may enter an infinite loop when their condition points out to infinity, freezing the computer in the process. These type of events are characteristic for beginners or professionals who rarely use "do-while loops" or "while loops". The "for loops" are straightforward and intuitive, however, even within "for loop" structures some problems can occur. For instance, a programmer may encounter difficulties when it comes to the starting point and the ending point of the counter. Most often than not, these issues appear when the counter is used in dealing with the lower bound and the upper bound of arrays. These common issues appear regardless of experience in the world of software development. Computer languages such as Python or JavaScript contain special instructions in the case of "for loops" that work with array objects. Such a "for loop" is written as "*For x in object*", which will retrieve each element one by one from the object in an ordered manner. This type of "for loop" statement helps the programmer to write a traversal strategy without special considerations for the upper

or lower bound of the array. In the following series of examples, "for loops" are used for the classical traversal over the elements of an array variable (Additional algorithm 7.6).

Consider an array variable *A* with five elements, which store different string values. A "for loop" structure is used to traverse the elements of the array variable *A*, with an addition of the content of each element to a variable *t*. In other words, these examples have the role of listing the values inside the elements of an array variable. Note that element numbering in an array usually starts at zero. In some computer languages, such as VB, the numbering of the elements of an array can begin arbitrarily as long as the lower bound is less than the upper bound (Additional algorithm 6.13).

7.3.3 Nested Loops

A nest is a concave object made of dry straw, in which the eggs of birds lie. This formation contains two layers, the nest itself and the egg. For reference, among the man-made objects that have multiple layers in this way are Matryoshka dolls where the dolls are nested inside each other. Another association related to nesting is the onion structure. However, in software engineering, the term "nesting" refers to information or processes organized in layers located within each other. Up to this point, "for loop" structures have been used either for manipulating simple variables or for manipulating one-dimensional composite variables, such as arrays. But how can multidimensional arrays be traversed? An intuitive approach would be to embed a "for loop" inside another "for loop" in the case of two-dimensional arrays, or to use three "for-loop" structures inside each other in the case of a three-dimensional array, and so on. The example below deals with the traversal of an array variable with two dimensions (Additional algorithm 7.7).

In short, an array literal with two dimensions contains both string values and integer values. Two nested "for loop" structures are defined, where a variable counter *i* is used for the first dimension and a variable counter *j* is used for the second dimension. Inside the two loops, the counter variables *i* and *j* are used to traverse the elements of the array, and the individual values from these elements are added to a variable *t*. The content of variable *t* is then printed to the output for inspection. Please notice that in both "for loops" the conditions stop the counter depending on the size of each dimension.

7.3.4 Multidimensional Traversal by One For-Loop

It is probably clear that the number of dimensions of an array is covered by a for loop each. But is such a direct association between a "for loop" structure and a dimension the only solution for traversing multiple dimensions inside an array variable? The short answer is no. This is because at a low level, array elements are stored sequentially in a linear fashion, and the low-level representation can of course be implemented at a high

Lang.	Example	Output

JS	`const A = ["a", "b", "c", "d", "e"];`
	
	`let t = "";`
	
	`for (let i = 0; i < A.length; i++) {`
	`  t += "\n A[" + i + "]=" + A[i];`
	`}`
	
	`print(t);`
		`JS Output:`
		`A[0]=a`
		`A[1]=b`
		`A[2]=c`
		`A[3]=d`
		`A[4]=e`

C#	`using System;`
	`class HelloWorld {`
	`  static void Main() {`
	
	`  string[] A = {"a", "b", "c", "d", "e"};`
	`  string t = "";`
	
	`  for (int i = 0; i < A.Length; i++) {`
	`  t += "\n A[" + i + "]=" + A[i];`
	`  }`
	
	`  Console.WriteLine(t);`
	`  }`
	`}`
		`C# Output:`
		`A[0]=a`
		`A[1]=b`
		`A[2]=c`
		`A[3]=d`
		`A[4]=e`

VB	`Private Sub Form_Load()`
	
	`  A = Array("a", "b", "c", "d", "e")`
	
	`  i = 0`
	`  t = ""`
	
	`  For i = 0 To UBound(A)`
	`  t = t & "A(" & i & ")=" & A(i) & vbCrLf`
	`  Next i`
	
	`  Debug.Print (t)`
	
	`End Sub`
		`VB Output:`
		`A(0)=a`
		`A(1)=b`
		`A(2)=c`
		`A(3)=d`
		`A(4)=e`

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

PHP

```php
$A = ["a", "b", "c", "d", "e"];

$t = "";

for ($i = 0; $i < count($A); $i++) {
  $t .= "\n A[" . $i . "]=" . $A[$i];
}

echo($t);
```

```
PHP Output:
A[0]=a
A[1]=b
A[2]=c
A[3]=d
A[4]=e
```

PERL

```perl
my @A = ("a", "b", "c", "d", "e");
my $t = "";

for (my $i = 0; $i < scalar(@A); $i++) {
  $t .= "\n A[" . $i . "]=" . $A[$i];
}

print $t;
```

```
PERL Output:
A[0]=a
A[1]=b
A[2]=c
A[3]=d
A[4]=e
```

Ruby

```ruby
A = ["a", "b", "c", "d", "e"]

i = 0
t = ''

for i in 0..(A.size-1)
    t += "\n A[" + i.to_s + "]=" + A[i].to_s
    i += 1
end

puts t
```

```
Ruby Output:
A[0]=a
A[1]=b
A[2]=c
A[3]=d
A[4]=e
```

Java

```java
public class Main
{
    public static void main(String[] args) {

        String[] A = {"a", "b", "c", "d", "e"};
        String t = "";

        for (int i = 0; i < A.length; i++) {
          t += "\n A[" + i + "]=" + A[i];
        }

        System.out.println(t);
    }
}
```

```
Java Output:
A[0]=a
A[1]=b
A[2]=c
A[3]=d
A[4]=e
```

Additional algorithm 7.6 (continued)

Python

```
A = ["a", "b", "c", "d", "e"]

i = 0
t = ''

for i in range(0, len(A)):
    t += "\n A[" + str(i) + "]=" + str(A[i])

print (t)
```

```
Python Output:
A[0]=a
A[1]=b
A[2]=c
A[3]=d
A[4]=e
```

C++

```
#include <iostream>
using namespace std;

int main()
{
    string A[] = {"a","b","c","d","e"};
    string t = "";

    int n = sizeof(A) / sizeof A[0];

    for (int i = 0; i < n; i++) {

        t += "A["+to_string(i)+"]=";
        t += A[i]+"\n";
    }

    cout<<t;

    return 0;
}
```

```
Python Output:
A[0]=a
A[1]=b
A[2]=c
A[3]=d
A[4]=e
```

Additional algorithm 7.6 (continued)

level. To better understand this description related to sequentiality, a question can be asked: how can a matrix be represented in one dimension? The answer to this question drives the implementation. Let us imagine a normal 3×3 matrix A with some random values:

$$A = \begin{pmatrix} 0\ 1\ 0 \\ 1\ 1\ 0 \\ 0\ 1\ 1 \end{pmatrix}$$

The above matrix (A) clearly shows two dimensions, however, these two dimensions can be broken to one dimension. In fact, this is how information is stored in memory at a low level. For instance, matrix A can be converted into sequence A, such as:

$$A = (010110011)$$

Lang.	Example	Output

JS	`var A = [`
	`  ["a", 88, 146],`
	`  ["b", 34, 124],`
	`  ["c", 96, 564],`
	`  [100, 12, "d"],`
	`];`
	
	`let t = "";`
	
	`for (let i = 0; i < A.length; i++) {`
	`  for (let j = 0; j < A[i].length; j++) {`
	
	`  t += "\n A["+i+"]["+j+"]=" + A[i][j];`
	
	`  }`
	`}`
	
	`print(t);`
		`JS Output:`
		`A[0][0]=a`
		`A[0][1]=88`
		`A[0][2]=146`
		`A[1][0]=b`
		`A[1][1]=34`
		`A[1][2]=124`
		`A[2][0]=c`
		`A[2][1]=96`
		`A[2][2]=564`
		`A[3][0]=100`
		`A[3][1]=12`
		`A[3][2]=d`

PHP	`$A = [`
	`  ["a", 88, 146],`
	`  ["b", 34, 124],`
	`  ["c", 96, 564],`
	`  [100, 12, "d"],`
	`];`
	
	`$t = "";`
	
	`for ($i = 0; $i < count($A); $i++) {`
	`  for ($j = 0; $j < count($A[$i]); $j++) {`
	
	`  $t .= "\n A[".$i."][".$j."]=".$A[$i][$j];`
	
	`  }`
	`}`
	
	`echo $t;`
		`PHP Output:`
		`A[0][0]=a`
		`A[0][1]=88`
		`A[0][2]=146`
		`A[1][0]=b`
		`A[1][1]=34`
		`A[1][2]=124`
		`A[2][0]=c`
		`A[2][1]=96`
		`A[2][2]=564`
		`A[3][0]=100`
		`A[3][1]=12`
		`A[3][2]=d`

PERL	`my @A = (` `  ["a", 88, 146],`	`PERL Output:` `A[0][0]=a`

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

```
   ["b", 34, 124],
   ["c", 96, 564],
   [100, 12, "d"],
);

my $r = $#A;
my $c = $#{$A[0]}+1;

for($i=0; $i<=$r; $i++)
{
  for($j=0; $j<$c; $j++)
  {
    $t .= "\n A[".$i."][".$j."]=". $A[$i][$j];
  }
}

print $t;
```

```
A[0][1]=88
A[0][2]=146
A[1][0]=b
A[1][1]=34
A[1][2]=124
A[2][0]=c
A[2][1]=96
A[2][2]=564
A[3][0]=100
A[3][1]=12
A[3][2]=d
```

Ruby

```
A = [
    ["a", 88, 146],
    ["b", 34, 124],
    ["c", 96, 564],
    [100, 12, "d"]
    ]

t = ""

for i in 0..(A.size-1)
    for j in 0..(A[i].size-1)
        t += "\n A[" + i.to_s + "]["
        t += j.to_s + "]=" + A[i][j].to_s
    end
end

puts t
```

```
Ruby Output:
A[0][0]=a
A[0][1]=88
A[0][2]=146
A[1][0]=b
A[1][1]=34
A[1][2]=124
A[2][0]=c
A[2][1]=96
A[2][2]=564
A[3][0]=100
A[3][1]=12
A[3][2]=d
```

Python

```
A = [["a", 88, 146],
     ["b", 34, 124],
     ["c", 96, 564],
     [100, 12, "d"]]

t = ""

for i in range(0, len(A)):
    for j in range(0, len(A[i])):
        t += "\n A[" + str(i) + "]["
        t += str(j) + "]=" + str(A[i][j])

print(t)
```

```
Python Output:
A[0][0]=a
A[0][1]=88
A[0][2]=146
A[1][0]=b
A[1][1]=34
A[1][2]=124
A[2][0]=c
A[2][1]=96
A[2][2]=564
A[3][0]=100
A[3][1]=12
A[3][2]=d
```

Additional algorithm 7.7 (continued)

Java

```
public class Main
{
    public static void main(String[] args) {

    String[][] A = new String[][] {
                                      {"a","b"},
                                      {"c","d"},
                                      {"e","f"},
                                      {"g","h"}
                                    };
    String t = "";

    int x = A.length;
    int y = A[0].length;

    for (int i = 0; i < x; i++) {
        for (int j = 0; j < y; j++) {

          t += "\n A["+i+"]["+j+"]="+A[i][j];

        }
    }
        System.out.println(t);
    }
}
```

```
Java Output:
A[0][0]=a
A[0][1]=b
A[1][0]=c
A[1][1]=d
A[2][0]=e
A[2][1]=f
A[3][0]=g
A[3][1]=h
```

C#

```
using System;
class HelloWorld {
  static void Main() {

    string[, ] A = new string[, ] {
                                      {"a","b"},
                                      {"c","d"},
                                      {"e","f"},
                                      {"g","h"}
                                    };
    string t = "";

    int x = A.GetLength(0);
    int y = A.GetLength(1);

    for (int i = 0; i < x; i++) {
        for (int j = 0; j < y; j++) {

            t += "\n A["+i+","+j+"]=" + A[i,j];
        }
    }

    Console.WriteLine(t);
  }
}
```

```
C# Output:
A[0,0]=a
A[0,1]=b
A[1,0]=c
A[1,1]=d
A[2,0]=e
A[2,1]=f
A[3,0]=g
A[3,1]=h
```

Additional algorithm 7.7 (continued)

C++

```cpp
#include <iostream>
using namespace std;

int main()
{
    string A[][2] = {
                     {"a","b"},
                     {"c","d"},
                     {"e","f"},
                     {"g","h"},
                     };
    string t = "";

    int n =  sizeof(A) / sizeof A[0];
    int m = sizeof A[0] / sizeof(string);

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {

            t += "A["+to_string(i);
            t += "]["+to_string(j);
            t += "]="+A[i][j]+"\n";
        }
    }
    cout<<t;

    return 0;
}
```

C++ Output:

```
A[0][0]=a
A[0][1]=b
A[1][0]=c
A[1][1]=d
A[2][0]=e
A[2][1]=f
A[3][0]=g
A[3][1]=h
```

VB

```
Private Sub Form_Load()

    Dim A(0 To 1, 0 To 2) As String
    Dim i As Integer
    Dim t As String

    A(0, 0) = "a"
    A(0, 1) = "b"
    A(0, 2) = "c"
    A(1, 0) = "d"
    A(1, 1) = "e"
    A(1, 2) = "f"

    For i = LBound(A, 1) To UBound(A, 1)
        For j = LBound(A, 2) To UBound(A, 2)
            t = t & "A(" & i & "," & j & ")="
            t = t & A(i, j) & vbCrLf
        Next j
    Next i

    Debug.Print (t)

End Sub
```

VB Output:

```
A(0,0)=a
A(0,1)=b
A(0,2)=c
A(1,0)=d
A(1,1)=e
A(1,2)=f
```

Additional algorithm 7.7 (continued)

A space between groups of three can be inserted for a better visualization, namely "010 110 011". Thus, a matrix is represented by a one-dimensional sequence of numbers and a rule for the reconstruction of the two dimensions. In a left-to-right traversal of the one-dimensional sequence, this rule indicates a new row every time a multiple of three is reached. Thus, based on this rule, one can implement a "for loop" that traverses a two-dimensional array A. The question now is: How can multiples of n be found? One may recall in an earlier chapter the modulo operator and the examples related to sticks, which are exactly what is required for this idea. In this approach, the importance of the modulo operator is paramount (please see Chap. 4). What is needed from this approach are the correct values for each counter (i.e. i and j) in order to be able to traverse the two-dimensional array, as if there are two for loop structures. Examples of two-dimensional arrays that are traversed in a single for loop can be found in Additional algorithm 7.8. As in the previous example, array literals with two dimensions are used. The difference that can be observed is the use of a single for loop for traversing the bi-dimensional array.

Initially, a variable n and a variable m will hold the length of the two dimensions of the array A, and the variables i and j will be initialized to zero. To cover all the elements of an array in a single for-loop, the first dimension and the second dimension are multiplied (i.e. $n \times m$) and the result indicates the upper limit at which the for-loop structure stops. The counter variable in this loop is denoted by v. The value from the v variable is then used to deduce the i and j coordinates of matrix A. Inside the loop, the dimension j is calculated using the modulo operator:

```
// abstract formulation

j = v % m
if j = 0 then i = i + 1

// step by step

j = 0 % 3 = 0; i = 0 + 1
j = 1 % 3 = 1
j = 2 % 3 = 2
j = 3 % 3 = 0; i = 1 + 1
j = 4 % 3 = 1
j = 5 % 3 = 2
j = 6 % 3 = 0; i = 2 + 1
```

The modulo operator provides the remainder from a division. Thus, every time the dimension m will be a multiple of v, the value of j will be zero and the value of i will be incremented. Thus, for matrix A, index j can take values from 0 to 2 (i.e. 3 columns), and i can take values from 0 to 3 (i.e. 4 rows). Note, however, that this approach does not provide any real optimization, as the single for-loop takes the same number of steps as nested for-loops that have the same purpose. This methodology can be applied to any number of dimensions. For example, the implosion of three dimensions to a single dimension is done by expanding the number of rules. Examples of three-dimensional arrays that are traversed by using a single for-loop, can be found in Additional algorithm 7.9. In this

Lang.	Example	Output

JS	`var A = [`
	`  ["a", 88, 146],`
	`  ["b", 34, 124],`
	`  ["c", 96, 564],`
	`  [100, 12, "d"],`
	`  ];`
	
	`let t = "";`
	`let n = A. length;  // rows`
	`let m = A[0].length;  // columns`
	
	`let i = 0;`
	`let j = 0;`
	
	`for (let v = 0; v < n*m; v++) {`
	
	`  j = v % m;`
	
	`  if(v!=0 && j == 0){i++;}`
	
	`  t += v + " A["+i+"]["+j+"]=";`
	`  t += A[i][j] + "\n";`
	`}`
	
	`print(t);`
		`JS Output:`
		`0 A[0][0]=a`
		`1 A[0][1]=88`
		`2 A[0][2]=146`
		`3 A[1][0]=b`
		`4 A[1][1]=34`
		`5 A[1][2]=124`
		`6 A[2][0]=c`
		`7 A[2][1]=96`
		`8 A[2][2]=564`
		`9 A[3][0]=100`
		`10 A[3][1]=12`
		`11 A[3][2]=d`

C#	`using System;`
	`class HelloWorld {`
	`  static void Main() {`
		`C# Output:`
		`0 A[0,0]=a`

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

```
    string[, ] A = new string[, ] {
                                     {"a","b"},
                                     {"c","d"},
                                     {"e","f"},
                                     {"g","h"}
                                   };

    string t = "";

    int n = A.GetLength(0);
    int m = A.GetLength(1);
    int i = 0;
    int j = 0;

    for (int v = 0; v < n*m; v++) {

        j = v % m;
        if(v!=0 && j == 0){i++;}

        t += v + " A["+i+","+j+"]=";
        t += A[i,j] + "\n";
    }

    Console.WriteLine(t);
  }
}
```

```
 2 A[1,0]=c
 3 A[1,1]=d
 4 A[2,0]=e
 5 A[2,1]=f
 6 A[3,0]=g
 7 A[3,1]=h
```

VB

```
Private Sub Form_Load()

    Dim A(0 To 1, 0 To 2) As String
    Dim i As Integer
    Dim t As String

    A(0, 0) = "a"
    A(0, 1) = "b"
    A(0, 2) = "c"
    A(1, 0) = "d"
    A(1, 1) = "e"
    A(1, 2) = "f"

    ' rows
    n = UBound(A, 1) - LBound(A, 1) + 1
    ' columns
    m = UBound(A, 2) - LBound(A, 2) + 1

    i = 0
    j = 0

    For v = 0 To (n * m) - 1

        j = v Mod m

        If (v <> 0 And j = 0) Then i = i + 1
```

```
VB Output:
0 A(0,0)=a
1 A(0,1)=b
2 A(0,2)=c
3 A(1,0)=d
4 A(1,1)=e
5 A(1,2)=f
```

Additional algorithm 7.8 (continued)

```
        t = t & v & " A(" & i & "," & j & ")="
        t = t & A(i, j) & vbCrLf

    Next v

    Debug.Print (t)

End Sub
```

PHP

```
$A = [
   ["a", 88, 146],
   ["b", 34, 124],
   ["c", 96, 564],
   [100, 12, "d"],
     ];

$t = "";
$n = count($A);
$m = count($A[0]);

$i = 0;
$j = 0;

for ($v = 0; $v < $n*$m; $v++) {

    $j = $v % $m;

    if($v!=0 && $j == 0){$i++;}

    $t .= $v . " A[".$i."][".$j."]=";
    $t .= $A[$i][$j] . "\n";
}

echo $t;
```

```
PHP Output:
0 A[0][0]=a
1 A[0][1]=88
2 A[0][2]=146
3 A[1][0]=b
4 A[1][1]=34
5 A[1][2]=124
6 A[2][0]=c
7 A[2][1]=96
8 A[2][2]=564
9 A[3][0]=100
10 A[3][1]=12
11 A[3][2]=d
```

PERL

```
my @A = (
    ["a", 88, 146],
    ["b", 34, 124],
    ["c", 96, 564],
    [100, 12, "d"],
        );

my $t = "";
my $n = $#A + 1;       # rows
my $m = $#{$A[0]} + 1; # columns

my $i = 0;
my $j = 0;

for ($v = 0; $v < $n*$m; $v++) {
```

```
PERL Output:
0 A[0][0]=a
1 A[0][1]=88
2 A[0][2]=146
3 A[1][0]=b
4 A[1][1]=34
5 A[1][2]=124
6 A[2][0]=c
7 A[2][1]=96
8 A[2][2]=564
9 A[3][0]=100
10 A[3][1]=12
11 A[3][2]=d
```

Additional algorithm 7.8 (continued)

```
    $j = $v % $m;

    if($v != 0 && $j == 0){$i++;}

    $t .= $v . " A[".$i."][".$j."]=";
    $t .= $A[$i][$j] . "\n";
}

print $t;
```

Ruby

```
A = [
    ["a", 88, 146],
    ["b", 34, 124],
    ["c", 96, 564],
    [100, 12, "d"],
    ]

t = ""
n = (A.size)     # rows
m = (A[0].size)  # columns

i = 0
j = 0

for v in 0..(n*m)-1

    j = v % m

    if(v != 0 and j == 0)
        i = i + 1
    end

    t += v.to_s + " A[" + i.to_s + "]["
    t += j.to_s + "]=" + A[i][j].to_s + "\n"
end

puts(t)
```

```
Ruby Output:
0 A[0][0]=a
1 A[0][1]=88
2 A[0][2]=146
3 A[1][0]=b
4 A[1][1]=34
5 A[1][2]=124
6 A[2][0]=c
7 A[2][1]=96
8 A[2][2]=564
9 A[3][0]=100
10 A[3][1]=12
11 A[3][2]=d
```

Java

```
public class Main
{
    public static void main(String[] args) {

        String[][] A = new String[][] {
                                        {"a","b"},
                                        {"c","d"},
                                        {"e","f"},
                                        {"g","h"}
                                        };
```

```
Java Output:
0 A[0,0]=a
1 A[0,1]=b
2 A[1,0]=c
3 A[1,1]=d
4 A[2,0]=e
5 A[2,1]=f
6 A[3,0]=g
7 A[3,1]=h
```

Additional algorithm 7.8 (continued)

```
        String t = "";

        int n = A.length;
        int m = A[0].length;

        int i = 0;
        int j = 0;

        for (int v = 0; v < n*m; v++) {

            j = v % m;
            if(v!=0 && j == 0){i++;}

            t += v + " A["+i+","+j+"]=";
            t += A[i][j] + "\n";
        }

        System.out.println(t);
    }
}
```

Python

```
A = [["a", 88, 146],
     ["b", 34, 124],
     ["c", 96, 564],
     [100, 12, "d"]]

t = ""
n = len(A)     # rows
m = len(A[0])  # columns

i = 0
j = 0

for v in range(0, n*m):
    j = v % m
    if(v != 0 and j == 0):
        i = i + 1
    t += str(v) + " A[" + str(i) + "]["
    t += str(j) + "]=" + str(A[i][j]) + "\n"

print(t)
```

```
Python Output:
0 A[0][0]=a
1 A[0][1]=88
2 A[0][2]=146
3 A[1][0]=b
4 A[1][1]=34
5 A[1][2]=124
6 A[2][0]=c
7 A[2][1]=96
8 A[2][2]=564
9 A[3][0]=100
10 A[3][1]=12
11 A[3][2]=d
```

C++

```
#include <iostream>
using namespace std;

int main()
{
    string A[][2] = {
                    {"a","b"},
                    {"c","d"},
                    {"e","f"},
                    {"g","h"},
                    };
```

```
C++ Output:
0 A[0][0]=a
1 A[0][1]=b
2 A[1][0]=c
3 A[1][1]=d
4 A[2][0]=e
5 A[2][1]=f
6 A[3][0]=g
7 A[3][1]=h
```

Additional algorithm 7.8 (continued)

```
    string t = "";

    int n =  sizeof(A) / sizeof A[0];
    int m = sizeof A[0] / sizeof(string);

    int i = 0;
    int j = 0;

    for (int v = 0; v < n*m; v++) {

        j = v % m;
        if(v!=0 && j == 0){i++;}

        t += to_string(v);
        t += " A["+to_string(i)+"][";
        t += to_string(j)+"]=";
        t += A[i][j] + "\n";
    }
    cout<<t;

    return 0;
}
```

Additional algorithm 7.8 (continued)

example an array A with three dimensions is first declared. Next, variables s, m and n store the length of each dimension of array A.

Variable s represents the number of matrices, variable n stores the number of rows of a matrix and finally variable n stores the number of columns of a matrix. A variable q stores the result of the multiplication between the three dimensions and represents the upper bound of the main for-loop (i.e. $q = n \times m \times s$). The index variables i, j, and d are declared and set to zero. The for-loop is then set to iterate from zero up to q. The main counter of the for-loop is then used to calculate all the other index variables such as i, j, k and d. Notice that a variable k becomes zero when the result of m times n is a multiple of v. That means, k is zero once a matrix was completely traversed. Thus, for the third dimension of the array, a condition increments a variable d each time variable k equals zero. Also, variable i is set to zero in the same condition. Notice that in this example, the first index (i.e. d) represents the matrix, the second index (i.e. i) represents the row of a matrix and the last index (i.e. j) of the array represents the columns of the matrix.

7.4 Conclusions

In order to design more useful and life like machines in the future, we instill our values and our rules into them. For this endeavor, our foundations have become the foundations of machines, whether these foundations are represented by absolute universal laws or not. Thus, machines are a copy of us. Some of the rules that adhere to these foundations are

JS

```
var A = [
        [
   ["a", 55, 146],
   ["b", 34, 124],
   ["c", 96, 564],
   [100, 12, "d"],
        ],
        [
   ["e", 88, 146],
   ["f", 34, 124],
   ["g", 96, 564],
   [100, 12, "h"],
        ],
        [
   ["i", 88, 146],
   ["j", 34, 124],
   ["k", 96, 564],
   [100, 12, "k"],
        ],
        [
   ["m", 88, 146],
   ["n", 34, 124],
   ["o", 96, 564],
   [100, 12, "p"],
        ],
```

```
JS Output:
0 A[0][0][0]=a
1 A[0][0][1]=55
2 A[0][0][2]=146
3 A[0][1][0]=b
4 A[0][1][1]=34
5 A[0][1][2]=124
6 A[0][2][0]=c
7 A[0][2][1]=96
8 A[0][2][2]=564
9 A[0][3][0]=100
10 A[0][3][1]=12
11 A[0][3][2]=d
12 A[1][0][0]=f
13 A[1][0][1]=88
14 A[1][0][2]=146
15 A[1][1][0]=b
16 A[1][1][1]=34
17 A[1][1][2]=124
18 A[1][2][0]=c
19 A[1][2][1]=96
20 A[1][2][2]=564
21 A[1][3][0]=100
22 A[1][3][1]=12
```

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

```
        [
   ["q", 88, 146],
   ["r", 34, 124],
   ["s", 96, 564],
   [100, 12, "t"],


        ]
        ];

let t = "";

let s = A.length;        // 5 matrices
let m = A[0].length;     // 4 rows
                         // 3 columns

let i = 0;
let j = 0;
let d = 0;
let k = 0;

let q = n * m * s;

for (let v = 0; v < q; v++){

   k = v % (m*n);
   j = v % n;

   if(v!=0 && j == 0){i++;}
   if(v!=0 && k == 0){i = 0; d++;}

   t += v + " A["+d+"]["+i+"]["+j+"]=";
   t += A[d][i][j] + "\n";
}

print(t);
```

```
23 A[1][3][2]=d
24 A[2][0][0]=x
25 A[2][0][1]=88
26 A[2][0][2]=146
27 A[2][1][0]=b


28 A[2][1][1]=34
29 A[2][1][2]=124
30 A[2][2][0]=c
31 A[2][2][1]=96
32 A[2][2][2]=564


35 A[2][3][2]=d
36 A[3][0][0]=f
37 A[3][0][1]=88
38 A[3][0][2]=146
39 A[3][1][0]=b
.
.
.
48 A[4][0][0]=x
49 A[4][0][1]=88
50 A[4][0][2]=146
51 A[4][1][0]=b
52 A[4][1][1]=34
53 A[4][1][2]=124
54 A[4][2][0]=c
55 A[4][2][1]=96
56 A[4][2][2]=564
57 A[4][3][0]=100
58 A[4][3][1]=12
59 A[4][3][2]=d
```

Additional algorithm 7.9 (continued)

control structures, which guide our own behavior. In the case of machines, two basic types of control structures were presented: the conditional statement and the repeat-loop statement. In case of repetitive loops, three types have been identified namely do-while-loop, while-loop and for-loop. Each control structure is shown to be capable of handling a block of expressions. The iteration direction of a repeat loop was also described. In addition, the use of nested loops was presented in the context of multidimensional array variables. The end of the chapter presented a method for looping through multidimensional array variables by using the modulo operator in conjunction with a single for-loop structure. Note that all expression blocks were written differently between the computer languages presented here, however, the main design was preserved in a mirror-like manner.

Functions

8

8.1 Introduction

Functions are constructs intended to eliminate redundancy or pseudo-redundancies. These block structures show the significance of input-output principles in software development. The notion of functions is fully described in context using both abstract explanations and concrete examples. Parameters are presented as the input, namely as means of one-way communication with the function, whereas the return value is presented as the output of a function. In different computer languages, the keyword for representing a function is different, which is why these keywords are discussed on a case-by-case basis. Two types of function calls are shown, namely cascading calls and recursive calls or a mixture of both. Examples are given for cascading function calls, and then for the recursive process, where it is shown how repeated loops can be replaced by a function that calls itself. In this chapter, functions provide an opportunity to discuss global and local variables and their meaning in context. Next, some examples of pure and impure functions, discussed in the first chapters, are presented here by way of example.

8.2 Defining Functions

Functions are blocks of instructions that perform redundant tasks. Usually, functions are used either to eliminate redundancy, or to structure the code into subproblems. The question that arises now is: How can such functions be defined? In all computer languages, the function statement consists of the function keyword followed by the function name,

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/978-3-031-23277-0_8.

P. A. Gagniuc, *An Introduction to Programming Languages: Simultaneous Learning in Multiple Coding Environments*, Synthesis Lectures on Computer Science,
https://doi.org/10.1007/978-3-031-23277-0_8

a comma-separated list of parameter names, usually in parentheses, and the statements that contain the function body. For the compiler or interpreter to be compatible with the syntax used, the definition must be language-specific. For instance, below are some examples in several computer languages that show the most common way to define a function (Additional algorithm 8.1). Some of the computer languages define a function by using the keyword "function" (i.e. VB, Javascript or PHP). That is, because the meaning of the function comes from the world of mathematics. In recent decades, the word function was replaced with other words having in fact the same meaning. For example, in Python and Ruby the keyword for such a "black box" is called "def". In other languages like Perl, the keyword for defining a function is "sub", and in computer languages like C# or Java, the keyword for defining a function is "static". Note that in object oriented computer languages, functions are also called methods, and have the added role of object communication. In C++, a function simply begins with a data type keyword and the name of the function, without any special keyword specifying that the block of code represents a function. In some computer languages like C++, C#, Java or VB6, the datatype returned by the function is specified. In C# and Java, the "static" keyword can be followed by another keyword that signifies the datatype returned by the function. In case the function is not meant to return anything, the keyword for a datatype is replaced with the keyword "void", as "void" means that the function (method) has no return value. In VB, this datatype is specified by a keyword at the end of the statement. In cases that a function is not meant to return anything but simply execute some instructions, the datatype keyword is not written. Depending on the syntax, the body of the function can be closed using different strategies. For example, C-like computer languages enclose the function body in curly braces. On the other hand, other computer languages like Ruby or VB prefer to end the body of a function with a keyword, namely "end" or "end function". Moreover, Python even disregards any termination and uses the indentation to signify the hierarchical order of the code. Just before the termination of the body, a function usually contains a return keyword followed by a variable or an entire expression, depending on the case. The syntax of C-like computer languages is more closely related to the meaning found in mathematics, while non-C-like syntaxes such as Python or Visual Basic are more closely related to what we call pseudo-code, or more precisely, a syntax that is closer to the natural language. Both classes of syntax discussed above are of great value to industry, education, and research. These two main types of syntax allow a wider range of people to participate and be productive in different fields of science and industry. Those people from exact sciences such as mathematics, physics, or chemistry will prefer the C-like syntax because of the striking similarities to mathematical notation, while those from other fields will probably prefer that syntax which is closer to the natural language. Again, both the C-like and non-C-like (i.e. VB-like) syntaxes have equal value for the progress of our civilization, as natural language and mathematics have a lot in common.

8.2.1 Simple Arguments

A function can hold pieces of code with various meanings. For example, a function may contain a simple equation or an entire algorithm (or even big parts of entire applications). A parameter is a variable in the declaration of the function, whereas an argument is the

Lang.	Example	Output

JS	`a = 10;`
	`b = compute(a);`
	
	`print(b);`
	
	`function compute(x)`
	`{`
	`  return x + x / x - x * x;`
	`}`
		`JS Output:`
		`-89`

C#	`using System;`
	`class HelloWorld {`
	
	`  static int compute(int x)`
	`  {`
	`  return x + x / x - x * x;`
	`  }`
	
	`  static void Main(string[] args)`
	`  {`
	`  int a = 10;`
	`  int b = compute(a);`
	
	`  Console.WriteLine(b);`
	`  }`
	`}`
		`C# Output:`
		`-89`

VB	`Private Sub Form_Load()`
	`  a = 10`
	`  b = compute(a)`
	`  Debug.Print b`
	`End Sub`
	
	`Function compute(x) As Integer`
	`  compute = x + x / x - x * x`
	`End Function`
		`VB Output:`
		`-89`

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

PHP

```php
$a = 10;
$b = compute($a);
echo $b;

function compute($x)
{
    return $x + $x / $x - $x * $x;
}
```

```
PHP Output:
-89
```

PERL

```perl
my $a = 10;
my $b = compute($a);
print $b;

sub compute
{
    my ($x) = @_;
    return $x + $x / $x - $x * $x;
}
```

```
PERL Output:
-89
```

Ruby

```ruby
def compute(x)
    return x + x / x - x * x
end

a = 10
b = compute(a)
puts b
```

```
Ruby Output:
-89
```

Java

```java
public class Main
{
    public static void main(String[] args) {

        int a = 10;
        int b = compute(a);

        System.out.println(b);
    }

    static int compute(int x)
    {
        return x + x / x - x * x;
    }
}
```

```
Java Output:
-89
```

Python

```python
def compute(x):
    return x + x / x - x * x
```

```
Python Output:
-89.0
```

Additional algorithm 8.1 (continued)

```
a = 10
b = compute(a)
print(b)
```

C++

```
int compute(int x)
{
    return x + x / x - x * x;
}

int main()
{
    int a = 10;
    int b = compute(a);

    cout<<b;

    return 0;
}
```

```
C++ Output:
-89.0
```

Additional algorithm 8.1 (continued)

value of the variable that is passed to the function. In all cases the function arguments are the inputs to the code blocks inside the function. The result of these code blocks is the function output that is returned to the caller (the main line that uses the function is known as the caller). Nonetheless, the simplest function, which is also the most useless, is a function that returns the value of the argument directly to the output. As described above, a function incorporates three main parts, namely the function name, the input or parameter, and the output. A useful function that can be declared is a function that receives an argument and immediately returns the result based on a computation that uses the value of the argument. Functions can be called by reference, by value, or by name if the function contains no parameters. A "parameter" is a variable identifier provided as input to a function (i.e., a placeholder in the function declaration), while an "argument" is a value passed as an input to the function when the call is made. The argument can be passed to the function either by value or by reference. In a "call by value" the argument is copied from one memory location to a different memory location associated with the parameter variable of the function. Any changes to the parameter variable will be local to the function. However, in a "call by reference" the memory address of the variable used by the caller, is passed to the parameter variable of the function (i.e., reference to a variable). Thus, any changes made to the parameter variable inside the function, are reflected in the variable used by the caller. In other words, changes to the reference arguments will propagate to the original value. In such cases, a "call by reference" modifies the data at the location of a variable used by the caller, which in turn can make the return of the function optional compared to the "call by value" approach. Thus, a "call by reference" uses less memory compared to a "call by value", increases the speed of an application, and is used

in hardware-intensive implementations. Conversely, compared to a "call by reference", a "call by value" uses more memory and the application speed is slower because of it, but it is safe and very methodical (easy to debug). However, Additional algorithm 8.1 shows the simplest use of a single-parameter function. Two variables *a* and *b* are declared.

The first variable, namely *a*, is assigned with an integer literal. The second variable, variable name *b*, is assigned with a value that is returned by a function called "compute". The call to the *compute* function sends the argument to the *x* variable. Thus, the value of the parameter is taken from the value found in variable *a*, which is then passed to the body of the function for further computations. Inside the body of the function, the value of *x* is used for a trivial calculation of a mathematical expression (i.e. $x + x/x - x \times x$), previously discussed in the subsection operator precedence and association. Once this expression is evaluated, the result is returned directly to the caller. The caller then assigns the new result to variable *b*, which is then printed to the output for inspection. Note: Unlike other computer languages from the list, in Perl the parameters are declared inside the body of the function.

8.2.2 Complex Arguments

Philosophically, each software application can be viewed as a universe separate from our own, with a beginning and an end. Thus, functions in turn can always be viewed either as pocket universes of the main universe, namely the application, or basically as subcodes that can always be called repeatedly. To better understand the importance of such blocks of statements, a more complex example shows a function with two parameters, namely a parameter that is represented by a string variable and a parameter that is represented by an array variable that contains string literals (Additional algorithm 8.2). In Additional algorithm 8.2, a function lists the values from the elements of an array variable. First, string literals are assigned to the elements of an array *A*. Also, a string variable *t* is set to empty for later use as an accumulator for different string values. Then a variable *b* is declared and the value returned by a function is assigned to it. The function called "compute" takes two arguments, namely the variable *t* and the array variable *A*.

Secondly, inside the function, a for loop is initialized with an integer variable *i* that spans from zero to the upper bound of the array *A*. Inside the for loop, a string containing the values from element *i* of the array *A* is added to the string variable *t* on each iteration. Once the for loop completes the iteration process, the content of the string variable *t* is returned to the caller. Thus, the value returned by the function is assigned to variable *b*. The content of variable *b* is then printed to the output for inspection.

Lang.	Example	Output

JS	`const a = ["a", "b", "c", "d", "e"];`
	`let t = "";`
	`let b = compute(t, a);`
	`print(b);`
	
	`function compute(t, a)`
	`{`
	`  for (let i = 0; i < a.length; i++) {`
	`  t += "\n a[" + i + "]=" + a[i];`
	`  }`
	`  return t;`
	`}`
		JS Output:
		a[0]=a
		a[1]=b
		a[2]=c
		a[3]=d
		a[4]=e

C#	`using System;`
	`class HelloWorld {`
	
	`  static string compute(string t, string []`
	`a)`
	`  {`
	`  for (int i = 0; i < a.Length; i++) {`
	`  t += "\n a[" + i + "]=" + a[i];`
	`  }`
	
	`  return t;`
	`  }`
	
	`  static void Main(string[] args)`
	`  {`
	`  string[] a = {"a", "b", "c", "d", "e"};`
	`  string t = "";`
	`  string b = compute(t, a);`
	
	`  Console.WriteLine(b);`
	`  }`
	`}`
		C# Output:
		a[0]=a
		a[1]=b
		a[2]=c
		a[3]=d
		a[4]=e

VB	`Private Sub Form_Load()`
	
	`  a = Array("a", "b", "c", "d", "e")`
	`  t = ""`
	`  b = compute(t, a)`
	`  Debug.Print b`
	
	`End Sub`
		VB Output:
		a[0]=a
		a[1]=b
		a[2]=c
		a[3]=d
		a[4]=e

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

```
Function compute(ByRef t, ByRef a) As String

    For i = 0 To UBound(a)
        t = t & "a[" & i & "]=" & a(i)
        t = t & vbCrLf
    Next i

    compute = t

End Function
```

PHP

```
$a = ["a", "b", "c", "d", "e"];
$t = "";
$b = compute($t, $a);
echo $b;

function compute($t, $a)
{
    for ($i = 0; $i < count($a); $i++) {
        $t .= "\n a[" . $i . "]=" . $a[$i];
    }
    return $t;
}
```

```
PHP Output:
a[0]=a
a[1]=b
a[2]=c
a[3]=d
a[4]=e
```

PERL

```
my @a = ("a", "b", "c", "d", "e");
my $t = "";
my $b = compute($t, $a);
print $b;

sub compute
{
    my ($t, $a) = @_;

    for ($i = 0; $i < scalar(@a); $i++) {
        $t .= "\n a[" . $i . "]=" . $a[$i];
    }
    return $t;
}
```

```
PERL Output:
a[0]=a
a[1]=b
a[2]=c
a[3]=d
a[4]=e
```

Ruby

```
def compute(t, a = [])
    for i in 0..(a.size-1)
        t += "\n a[" + i.to_s + "]="
        t += a[i].to_s
    end
    return t
end

a = ["a", "b", "c", "d", "e"]
t = ''
```

```
Ruby Output:
a[0]=a
a[1]=b
a[2]=c
a[3]=d
a[4]=e
```

Additional algorithm 8.2 (continued)

```
b = compute(t, a)
puts b
```

Java

```
public class Main
{
    public static void main(String[] args) {

        String[] a = {"a", "b", "c", "d", "e"};
        String t = "";
        String b = compute(t, a);

        System.out.println(b);
    }

  static String compute(String t, String [] a)
    {
        for (int i = 0; i < a.length; i++) {
            t += "\n a[" + i + "]=" + a[i];
        }

        return t;
    }
}
```

```
Java Output:
a[0]=a
a[1]=b
a[2]=c
a[3]=d
a[4]=e
```

Python

```
def compute(t, a = []):
    for i in range(0, len(a)):
        t += "\n a[" + str(i) + "]="
        t += str(a[i])
    return t

a = ["a", "b", "c", "d", "e"]
t = ''
b = compute(t, a)
print(b)
```

```
Python Output:
a[0]=a
a[1]=b
a[2]=c
a[3]=d
a[4]=e
```

C++

```
#include <iostream>
using namespace std;
//using std::to_string;

string compute(string t, string a[], int l)
{
    for (int i = 0; i < l; i++) {
        t += "\n a[" + to_string(i) +
        "]=" + a[i];
    }
    return t;
}


int main()
```

```
C++ Output:
a[0]=a
a[1]=b
a[2]=c
a[3]=d
a[4]=e
```

Additional algorithm 8.2 (continued)

```
{
    string a[5] = {"a", "b", "c", "d", "e"};
    string t = "";

    int l = sizeof(a) / sizeof(string);
    string b = compute(t, a, l);

    cout<<b;

    return 0;
}
```

Additional algorithm 8.2 (continued)

8.2.3 Nested Function Calls

As previously described in the case of nested loops, in software engineering, the term nesting refers to information or processes organized in layers located within each other. Only a few of the meanings of the term "nested" can be mentioned here, such as: nested levels of parentheses in arithmetic expressions, nested data structures (i.e. records, objects or classes), and nested blocks of imperative source code such as nested if-clauses or repeat-loop clauses. To simply call a function and directly assign the return value to a variable is a straightforward statement (Additional algorithm 8.2). However, what if the same function needs to be called multiple times with the return value as the argument for the next call? An iterative method can be used where the function is called a certain number of times. In such cases, an additional new variable must be declared and used both as storage for the previous return value and as an argument to the current call. Let us consider the following example:

```
// VB out-of-context formulation

a = 0

For i = 0 To UBound(a)
    a = compute(a)
Next i
```

Where variable *a* and the entire repeat loop can be replaced with nested function calls. In Additional algorithm 8.3 an example shows the idea of nested function calls. The example involves two variables, namely *a* and *b*. An integer literal is assigned to the variable *a*, while the results from a chain of calls to a function *c* are assigned to variable *b*. A call to function *c* sends one argument to variable *x*. Thus the value of the parameter is taken from the value found in variable *a*, which is then passed to the body of the function. Like before, inside the body of function *c*, the argument is used for a

mathematical expression, namely $x + x/x - x \times x$. Once this expression is evaluated, the result is returned directly to the caller. The caller then assigns the result as an argument for the next call. Once this chain of calls ends, the return value is assigned to variable b.

In the next step, the negative sign of the result is changed to positive by simply attaching the minus sign in front of the b variable. Thus, the positive value from b is then

Lang.	Example	Output

JS	`a = 3;`
	`b = c(c(c(c(a))));`
	`b = -b;`
	
	`print(b);`
	
	`function c(x)`
	`{return x + x / x - x * x;}`
		JS Output:
		756029

C#	`using System;`
	`class HelloWorld {`
	`  static void Main() {`
	
	`  int a = 3;`
	`  int b = c(c(c(c(a))));`
	`  b = -b;`
	
	`  Console.WriteLine(b);`
	`  }`
	
	`  static int c(int x)`
	`  {return x + x / x - x * x;}`
	
	`}`
		C# Output:
		756029

VB	`Private Sub Form_Load()`
	`  a = 3`
	`  b = c(c(c(c(a))))`
	`  b = -b`
	`  Debug.Print b`
	`End Sub`
	
	`Function c(x) As Double`
		VB Output:
		756029

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

```
    c = x + x / x - x * x
End Function
```

PHP

```
$a = 3;
$b = c(c(c(c($a))));
$b = -$b;

echo $b;

function c($x)
{return $x + $x / $x - $x * $x;}
```

```
PHP Output:
756029
```

PERL

```
my $a = 3;
my $b = c(c(c(c($a))));
$b = -$b;

print $b;

sub c{
    my ($x) = @_;
    return $x + $x / $x - $x * $x;
}
```

```
PERL Output:
756029
```

Ruby

```
def c(x)
    return x + x / x - x * x
end

a = 3
b = c(c(c(c(a))))
b = -b

puts b
```

```
Ruby Output:
756029
```

Java

```
public class Main
{
    public static void main(String[] args) {

        int a = 3;
        int b = c(c(c(c(a))));
        b = -b;

        System.out.println(b);
    }

    static int c(int x)
    {return x + x / x - x * x;}
}
```

```
Java Output:
756029
```

Additional algorithm 8.3 (continued)

Python

```
def c(x):
    return x + x / x - x * x

a = 3
b = c(c(c(c(a))))
b = -b

print(b)
```

```
Python Output:
756029.0
```

C++

```
#include <iostream>
using namespace std;

int c(int x)
{
    return x + x / x - x * x;
}


int main()
{
    int a = 3;
    int b = c(c(c(c(a))));
    b = -b;

    cout<<b;

    return 0;
}
```

```
C++ Output:
756029
```

Additional algorithm 8.3 (continued)

assigned to the same variable, which is then printed to the output for inspection. Notes: The point of the minus sign in this example is to show that numeric values from inside variables can be manipulated in this manner. Please notice the initial seed, namely the integer value stored in variable *a*. Experimentation will show that a call to function *c* using variable *a* as an argument, will return -5 as the result. However, once function *c* is declared as an argument to another call to function *c*, the returned value will be -29 (i.e. "c(c(a))"). For three calls to function *c* that pass the returned value as an argument for the next call, the result will show number -869 (i.e. "c(c(c(a)))"). In the case of four nested calls, the result is -756029 (i.e. "c(c(c(c(a))))").

8.2.4 Chained Function Calls

In Additional algorithm 8.4, it is shown how functions can in turn call other functions. This time the integer literals 1 through 5 are declared for an array "*a*". An integer variable *t* is declared and set to zero and then used as an accumulator. Next, a variable *b* is declared and set with the return value provided by a function called *c*1. The result from variable *b* is then printed to the output for inspection. However, what happens beyond the *c*1 function is the main demonstration of this example. Please note that function *c1* calls a function *c*2, and function *c*2 calls a function *c*3 and so on. Most importantly here, notice how complex arguments are passed from one function to another (i.e. the variable *t* and the array *a*). These arguments can of course be used at any level in this chain of function calls. In this specific case, the arguments are passed to the last level and used there to calculate the return values. Nevertheless, at each link in this chain, something new is added to the returned value in order to show some possibilities to the reader. Once function *c*1 calls function *c*2, function *c*2 returns a value made of an integer and whatever a function *c*3 returns. In turn, function *c*3 returns whatever a function *c*4 provides, plus an integer number. Notice that one function added a value before the return value and the other function added a value after the return value. Next, the *c*3 function demonstrates that it can declare a local variable inside the body of the function with an assigned integer value (i.e. 1).

The return value from function *c*3 is the contents of variable *s* plus the result provided by a function *c*4. In turn, function *c*4 returns the value provided by function *c5* plus the same value returned by the same function *c*5. This demonstrates the ability to return values from different types of functions inside other functions. Function *c*5 is the end of the chain in this example. The *c*5 function takes the two arguments passed along the chain and uses their values to process some information. Namely, function *c*5 takes the value from each element of array *a* and makes a sum that is progressively accumulated in variable *t*. At the end of the for-loop, variable *t* is returned to the caller, namely function *c*4, and function *c*4 returns its own value to function *c*3, and function *c*3 returns its value to function *c*2, which returns its value to function *c*1, which finally in turn assigns its returned value to variable *b*.

8.2.5 Relative Positioning of Functions

The main program is where execution begins. Depending on the computer language used, the main program is represented by the code that is outside the functions, or the code that resides in a special block of code called main. Probably at this stage, one of the most important observations that can be made in this set of computer languages will be the position of functions relative to the main program. One can notice some particularities in languages like C++, or Python and Ruby, and the others from the set. In C++, the order of the functions must be visible to the compiler in their entirety. For example, the call

Lang.	Example	Output

JS	`const a = [1, 2, 3, 4, 5];`
	`let t = 0;`
	`let b = c1(t, a);`
	`print(b);`
	
	`function c1(t, a){`
	`  return 5 + c2(t, a);`
	`}`
	
	`function c2(t, a){`
	`  return c3(t, a) + 5;`
	`}`
	
	`function c3(t, a){`
	`  let s = 1;`
	`  return s + c4(t, a);`
	`}`
	
	`function c4(t, a){`
	`  return c5(t, a) + c5(t, a);`
	`}`
	
	`function c5(t, a){`
	
	`  for (let i = 0; i < a.length; i++){`
	`  t += a[i];`
		`JS Output:`
		`41`

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 $c1$, which pass their values to other functions in a chain that ends in a function $c5$. 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 $c5$ 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 $c4$, which adds some other value to the this response. In turn, function $c4$ returnes the value to function $c3$, until it reaches the path to function $c1$, 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 $c5$. 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

```
    }

    return t;
}
```

C#

```
using System;
class HelloWorld {

    static void Main(string[] args)
    {
        int[] a = {1, 2, 3, 4, 5};
        int t = 0;
        int b = c1(t, a);

        Console.WriteLine(b);
    }

    static int c1(int t, int[] a){
        return 5 + c2(t, a);
    }

    static int c2(int t, int[] a){
        return c3(t, a) + 5;
    }

    static int c3(int t, int[] a){
        int s = 1;
        return s + c4(t, a);
    }

    static int c4(int t, int[] a){
        return c5(t, a) + c5(t, a);
    }

    static int c5(int t, int[] a){

        for (int i = 0; i < a.Length; i++) {
            t += a[i];
        }

        return t;
    }
}
```

```
C# Output:
41
```

VB

```
Private Sub Form_Load()

    Dim t, b As Integer

    a = Array(1, 2, 3, 4, 5)
    t = 0
    b = c1(t, a)
```

```
VB Output:
41
```

Additional algorithm 8.4 (continued)

```
    Debug.Print b

End Sub

Function c1(ByRef t, ByRef a) As Integer
    c1 = 5 + c2(t, a)
End Function

Function c2(ByRef t, ByRef a) As Integer
    c2 = c3(t, a) + 5
End Function

Function c3(ByRef t, ByRef a) As Integer
    Dim s As Integer
    s = 1
    c3 = s + c4(t, a)
End Function

Function c4(ByRef t, ByRef a) As Integer
    c4 = c5(t, a) + c5(t, a)
End Function

Function c5(ByRef t, ByRef a) As Integer

    For i = 0 To UBound(a) - 1
        t = t + a(i)
    Next i

    c5 = t

End Function
```

PHP

```
$a = [1, 2, 3, 4, 5];
$t = 0;
$b = c1($t, $a);
echo $b;

function c1($t, $a){
    return 5 +c2($t, $a);
}

function c2($t, $a){
    return c3($t, $a) + 5;
}

function c3($t, $a){
    $s = 1;
    return $s + c4($t, $a);
}

function c4($t, $a){
    return c5($t, $a) + c5($t, $a);
```

```
PHP Output:
41
```

Additional algorithm 8.4 (continued)

```
}

function c5($t, $a)
{
    for ($i = 0; $i < count($a); $i++) {
        $t += $a[$i];
    }
    return $t;
}
```

PERL

```
my @a = (1, 2, 3, 4, 5);
my $t = 0;
my $b = c1($t, $a);
print $b;

sub c1{
    my ($t, $a) = @_;
    return 5 + c2($t, $a);
}

sub c2{
    my ($t, $a) = @_;
    return c3($t, $a) + 5;
}

sub c3{
    my ($t, $a) = @_;
    my $s = 1;
    return $s + c4($t, $a);
}

sub c4{
    my ($t, $a) = @_;
    return c5($t, $a) + c5($t, $a);
}

sub c5{
    my ($t, $a) = @_;

    for ($i = 0; $i < scalar(@a); $i++) {
        $t += $a[$i];
    }
    return $t;
}
```

```
PERL Output:
41
```

Ruby

```
def c1(t, a = [])
    return 5 + c2(t, a)
end

def c2(t, a = [])
    return c3(t, a) + 5
```

```
Ruby Output:
41
```

Additional algorithm 8.4 (continued)

```
end

def c3(t, a = [])
    s = 1
    return s + c4(t, a)
end

def c4(t, a = [])
    return c5(t, a) + c5(t, a)
end

def c5(t, a = [])
    for i in 0..(a.size-1)
        t += a[i]
    end
    return t
end

a = [1, 2, 3, 4, 5]
t = 0
b = c1(t, a)
puts b
```

Java

```
public class Main
{
    public static void main(String[] args) {

        int[] a = {1, 2, 3, 4, 5};
        int t = 0;
        int b = c1(t, a);

        System.out.println(b);
    }

    static int c1(int t, int[] a){
        return 5 + c2(t, a);
    }

    static int c2(int t, int[] a){
        return c3(t, a) + 5;
    }

    static int c3(int t, int[] a){
        int s = 1;
        return s + c4(t, a);
    }

    static int c4(int t, int[] a){
        return c5(t, a) + c5(t, a);
    }

    static int c5(int t, int[] a){
```

```
Java Output:
41
```

Additional algorithm 8.4 (continued)

```
        for (int i = 0; i < a.length; i++) {
            t += a[i];
        }

        return t;
    }
}
```

Python

```
def c1(t, a = []):
    return 5 + c2(t, a)

def c2(t, a = []):
    return c3(t, a) + 5

def c3(t, a = []):
    s = 1
    return s + c4(t, a)

def c4(t, a = []):
    return c5(t, a) + c5(t, a)

def c5(t, a = []):
    for i in range(0, len(a)):
        t += a[i]
    return t

a = [1, 2, 3, 4, 5]
t = 0
b = c1(t, a)
print(b)
```

```
Python Output:
41
```

C++

```
#include <iostream>
using namespace std;

// order of functions matters !

int c5(int t, int a[]){

    int l = t;
    t = 0;

    for (int i = 0; i < l; i++) {
        t += a[i];
    }

    return t;
}

int c4(int t, int a[]){
    return c5(t, a) + c5(t, a);
```

```
C++ Output:
41
```

Additional algorithm 8.4 (continued)

```
}

int c3(int t, int a[]){
    int s = 1;
    return s + c4(t, a);
}

int c2(int t, int a[]){
    return c3(t, a) + 5;
}

int c1(int t, int a[]){
    return 5 + c2(t, a);
}


int main()
{
    int a[] = {1, 2, 3, 4, 5};
    int t = sizeof(a) / sizeof(int);
    int b = c1(t, a);

    cout<<b;

    return 0;
}
```

Additional algorithm 8.4 (continued)

of $c1$ eventually leads to the call of function $c5$. Thus, the C++ compiler must load all functions into memory for this chain to be triggered. An order like: $c1$, $c2$, $c3$, $c4$, $c5$ will result in an error at runtime, as function $c1$ is executed before the other functions are loaded into memory. However, an order like $c5$, $c4$, $c3$, $c2$, $c1$, allows the compiler to load all of the functions into memory before function $c1$ is executed. In Ruby and Python, the position of the functions relative to the main program is also important, but not as important as it is in C++. Ruby and Python implementations result in errors if functions are declared after the main program. Notice that in Python and Ruby all functions are declared before the main program, however, the order of their declarations is irrelevant, as it seems that all functions are loaded into memory before any calls are made. This is in contrast to C++, where calls are made early. In the other computer languages from the list, it can be seen that the order in which the functions are declared relative to the main program is irrelevant. In Additional algorithm 8.4, where permitted, functions are intentionally positioned after the main program to emphasize this observation. This means that in computer languages like Javascript, PHP, PERL, C#, Java and VB, execution begins only after all functions are known to the interpreter or compiler.

8.2.6 Recursive Calls

Recursion refers to self-calls or a chain of calls that result in a self-call. A simple function that returns a call to itself will probably throw an error if the execution is supervised, otherwise the execution will point to infinity and lead to high CPU usage and even hang the operating system (OS). However, there is an elegance when it comes to recursivity. The increment of an integer variable and a simple condition statement inside such functions can stop the recursive process as desired. Simple mistakes in the implementation of recursive calls, such as wrong condition statements, can lead to problems as described above. For this reason, recursive calls are less popular than processes like repeated loops.

```
// abstract formulation

function do_it() {

    // do stuff

    if(condition) {
        // stop self-calling
        // return result
    } else {
        do_it();
    }
}
```

Intuitively, when we think of repeat loops, the word "push" may come to mind, while recursive functions may be associated with the word "pull". In Additional algorithm 8.5, an example of recursive function is presented. A function called "for-loop" is designed, and the name of the function points out the similarity of this process with the repeat-loop structures. The "for-loop" function is able to receive three arguments, by using a parameter *a* as a counter, a parameter *b* used as the upper limit of the recursion process, and another parameter *r* that is able to store the results.

In the main program, the return value of the for-loop function is assigned to variable *a*, that is printed in the output for inspection. Once the for-loop function is called, the thread of execution goes to the interior of the function, were the first non-empty line increments the value of integer variable *a*. Next, a comment section indicates a region in which different variations of computations can be inserted. In this specific case of the section, an integer literal is added to the value stored in variable *r*. Next, a condition verifies if the value from the counter (variable *a*) is higher or equal to the value from variable *b*. If the condition is true, that means the recursion process has reached the limit indicated by the argument (i.e. variable *b*), and the result found in variable *r* is then returned to the caller. Otherwise, the condition executes another statement which returns a call to the same function, this time using the updated values for variable *b* and *r*. Notice that the result indicates a value of 35 because 5 is added to the value found in variable *r* exactly 7 times (or *b* times). Self-calling is not the only type of recursion possible. Among other

methods, recursion can also be made by a kind of induction, namely a circular chain of function calls. For instance, such an example is shown below only in Javascript for demonstration:

```
a = for_loop(0, 7, 0);
print(a);

// recursion by induction
function for_loop(a, b, r)
{
    return add(a, b, r);
}

function add(a, b, r)
{
    return by_induction(a, b, r);
}

function by_induction(a, b, r){

    a++;
    r += 5;

    if(a>=b){
        return r;
    } else {
        return for_loop(a, b, r);
    }
}
```

Of course, this circular setup can take different paths to closure, not necessarily a sequence of calls in a simple chain. Nevertheless, in the example from above, three functions call each other in sequence in a circular manner. The return value of a function called “for_loop” is assigned to a variable that is then printed to the output for inspection. Once the “for_loop” function is called, the return value of this function is further requested from another function called “add”. In turn, function “add” returns whatever a function called “by_induction” returns. Inside the body of “by_induction” function one can see a classical recursion strategy. However, in the cases where the condition is false, the return value is requested from the “for-loop” instead of a self-call. Thus, the recursion works as in the previous examples, however, the call to self is done through intermediary functions. Note that the assignment to variable *a* can be done from any of the three functions, as the result will be the same (i.e. 35).

Lang. | Example | Output

JS

```
a = for_loop(0, 7, 0);
print(a);

// replacement for repeat loops
function for_loop(a, b, r){

    a++;
    // do stuff from
    r += 5;
    // to here

    if(a>=b){
        return r;
    } else {
        return for_loop(a, b, r);
    }
}
```

```
JS Output:
35
```

C#

```
using System;
class HelloWorld {

    // replacement for repeat loops
    static int for_loop(int a, int b, int r){

        a++;
        // do stuff from
        r += 5;
        // to here

        if(a>=b){
            return r;
        } else {
            return for_loop(a, b, r);
        }
    }
```

```
C# Output:
35
```

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

```
    static void Main() {
        int a = for_loop(0, 7, 0);
        Console.WriteLine(a);
    }
}
```

VB

```
Private Sub Form_Load()
    Dim a As Integer
    a = for_loop(0, 7, 0)
    Debug.Print a
End Sub


' replacement for repeat loops
Function for_loop(a, b, r) As Integer

    a = a + 1
    ' do stuff from
    r = r + 5
    ' to here

    If (a >= b) Then
        for_loop = r
    Else
        for_loop = for_loop(a, b, r)
    End If

End Function
```

```
VB Output:
35
```

PHP

```
$a = for_loop(0, 7, 0);
echo($a);

// replacement for repeat loops
function for_loop($a, $b, $r){

    $a++;
    // do stuff from
    $r += 5;
    // to here

    if($a>=$b){
        return $r;
    } else {
        return for_loop($a, $b, $r);
    }
}
```

```
PHP Output:
35
```

Additional algorithm 8.5 (continued)

PERL

```perl
$a = for_loop(0, 7, 0);
print $a;

# replacement for repeat loops
sub for_loop{
    my ($a, $b, $r) = @_;

    $a++;
    # do stuff from
    $r += 5;
    # to here

    if($a>=$b){
        return $r;
    } else {
        return for_loop($a, $b, $r);
    }
}
```

```
PERL Output:
35
```

Ruby

```ruby
# replacement for repeat loops
def for_loop(a, b, r)
    a = a + 1
    # do stuff from
    r += 5
    # to here

    if(a>=b)
        return r
    else
        return for_loop(a, b, r)
    end
end



a = for_loop(0, 7, 0)
puts(a)
```

```
Ruby Output:
35
```

Java

```java
public class Main
{
    // replacement for repeat loops
    static int for_loop(int a, int b, int r){

        a++;
        // do stuff from
        r += 5;
        // to here
```

```
Java Output:
35
```

Additional algorithm 8.5 (continued)

```
        if(a>=b){
            return r;
        } else {
            return for_loop(a, b, r);
        }
    }

    public static void main(String[] args) {
        int a = for_loop(0, 7, 0);
        System.out.println(a);
    }
}
```

Python

```
# replacement for repeat loops
def for_loop(a, b, r):
    a = a + 1
    # do stuff from
    r += 5
    # to here

    if(a>=b):
        return r
    else:
        return for_loop(a, b, r)


a = for_loop(0, 7, 0)
print(a)
```

```
Python Output:
35
```

C++

```
#include <iostream>
using namespace std;

// replacement for repeat loops
int for_loop(int a, int b, int r){

    a++;
    // do stuff from
    r += 5;
    // to here

    if(a>=b){
        return r;
    } else {
        return for_loop(a, b, r);
    }
}


int main()
{
    int a = for_loop(0, 7, 0);
    cout<<a;
    return 0;
}
```

```
C++ Output:
35
```

Additional algorithm 8.5 (continued)

8.2.7 Global Versus Local Variables

The main difference between global and local variables is that global variables can be accessed globally in the entire program, whereas local variables can be accessed only within the function in which they are defined. The main advantage of local variables when compared to global variables is the immediate release of memory once their pocket of code (i.e. functions, subs or procedures) finishes execution (additionally, local variables have the ability to constrain errors when debugging is made). Global variables may remain active in memory until the program is unloaded. For example, in Javascript, a declaration of a global variable is made at the beginning of the main block, outside of any function, and uses the keyword "var" in front of the variable name. In C++, Visual Basic 6.0 and VBA, constants and global variables are declared first, outside any block of code. In C# and Java, both constants and global like variables (class variable) are declared specifically as such by using keywords such as "public" in front of the name of the variable, whereas constants are declared by using the keyword "const". In the case of PHP, constants have a special type of declaration, while a global variable is just another variable positioned outside of any code block. In other computer languages, such as Python, Ruby, or Perl, constants do not have a special type of declaration. Instead, good practices are applied in these computer languages, such as upper case letters for the name of constants. In computer languages that lack the ability to declare constants, functions can be used as constants. For example, instead of "*THIS_CONSTANT = 5;*", which can be easily modified by mistake, a function named "*THIS_CONSTANT(){return 5;}*" is more difficult to modify by mistake. Such a function can be used exactly like a constant. For example, in the expression "$x = x + THIS\_CONSTANT;$", a programmer will not feel the difference between the name of the variable and the name of the function. The example from Additional algorithm 8.6 shows both the meaning of global vs local variables and the meaning of constants.

In the main program, a constant (i.e. *a*) is declared and set to the value of *PI*. Next, a global variable *b* is declared and a value is assigned to it. The point of the example is a call to a function "compute" that is able to receive arguments by directly taking the values from the global variable *b*. The value from global variable *b* is seen from the interior of the function and is further assigned to a new local variable *x*, which is then used for the computation of the return value. Next, the return value is assigned in the main program to a variable *c*, which together with the read-only variable *a*, is then printed in the output for inspection.

Lang.	Example	Output

JS	`const a = 3.1415; // constant`
	`var b = 11;  // global variable`
	
	`b = compute();`
	`print(b + "\n" + a);`
	
	`function compute(){`
	`  let x = b;`
	`  return x + x / x - x * x;`
	`}`
		`JS Output:`
		`-109`
		`3.1415`

C#	`using System;`
	`class HelloWorld {`
	
	`  // constant`
	`  public const double a = 3.1415;`
	`  // global like (class variable)`
	`  public static int b;`
	
	`  static int compute()`
	`  {`
	`  int x = b;`
	`  return x + x / x - x * x;`
	`  }`
	
	`  static void Main(string[] args)`
	`  {`
	`  b = 11;`
	`  int c = compute();`
	`  Console.WriteLine(c + "\n" + a);`
	`  }`
	`}`
		`C# Output:`
		`-109`
		`3.1415`

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

VB

```vb
Const a = 3.1415  'constant
Dim b As Integer  'global variable

Private Sub Form_Load()
    b = 11
    c = compute()
    Debug.Print c & vbCrLf & a
End Sub

Function compute() As Integer
    x = b
    compute = x + x / x - x * x
End Function
```

```
VB Output:
-109
3.1415
```

PHP

```php
define("a", 3.1415); // constant
$b = 11;             // global variable

$c = compute();
echo $c . "\n" . a;

function compute()
{
    global $b;
    $x = $b;
    return $x + $x / $x - $x * $x;
}
```

```
PHP Output:
-109
3.1415
```

PERL

```perl
use constant a => 3.1415; # constant
$b = 11;                  # global variable

my $c = compute($b);
print $c . "\n" . a;

sub compute
{
    my $x = $b;
    return $x + $x / $x - $x * $x;
}
```

```
PERL Output:
-109
3.1415
```

Ruby

```ruby
def compute()
    x = $b
    return x + x / x - x * x
end

A = 3.1415 # constant
$b = 11    # global variable

c = compute()

puts "#{c}\n#{A}";
```

```
Ruby Output:
-109
3.1415
```

Java

```java
public class Main
{
    // constant
```

```
Java Output:
-109
3.1415
```

Additional algorithm 8.6 (continued)

```
    public static final double a = 3.1415;
    // global like (class variable)
    public static int b;

    static int compute()
    {
        int x = b;
        return x + x / x - x * x;
    }

    public static void main(String[] args) {

        b = 11;
        int c = compute();
        System.out.println(c + "\n" + a);
    }
}
```

Python

```
def compute():
    x = b
    return x + x / x - x * x

A = 3.1415 # constant
b = 11     # global variable

c = compute()

print(str(c) + "\n" + str(A));
```

```
Python Output:
-109
3.1415
```

C++

```
#include <iostream>
using namespace std;

const float a = 3.141592; // constant
int b = 11;               // global variable

int compute(){
    int x = b;
    return x + x / x - x * x;
}

//const float a = 3.141592;
//int b = 11;

int main()
{
    b = compute();
    cout<<(to_string(b) + "\n" + to_string(a));
    return 0;
}
```

```
C++ Output:
-109
3.141592
```

Additional algorithm 8.6 (continued)

8.2.8 Functions: Pure and Impure

Earlier in this work, some discussion mentioned the notion of pure functions and impure functions. This differentiation is important because functional programming is based on pure functions. Thus, some examples of pure and impure functions are given here (Additional algorithm 8.7). In pure functions, the same inputs lead to the same outputs. These types of functions have no side effects, meaning there is no change to the attributes of the program that reside outside of the function. In contrast, side effects change the state of the program from the inside of a function. For example, a function is impure if it changes the value of variables that are outside the function (ex. global variables). An integer value (i.e. 10) is assigned to a global variable named *a*. Variable *a* is used as an argument for two functions: A function called "pure" that returns the result from the evaluation of a mathematical expression, and another function called "impure" that assigns a new value to a global variable *a* (Additional algorithm 8.7).

Both functions receive the value from variable *a*, which is further used by a local variable *x*. Notice that in the first call the result returned by the "pure" function and by the "impure" function is −89. However, on a second call, the result is different, namely −109. The reason is the modification made by the impure function to the global variable *a* in the prior call. Since global variable *a* is an argument for the "impure" function prior to the modification of the value in variable *a*, the result is not affected in the first call but only in the second call. Notice that subsequent calls to the "pure" function will provide the same result of -109. However, the result changes every time the "impure" function was called in the previous call.

8.2.9 Function Versus Procedure

Some terms are deeply connected to the historical perspective. Initially, the main concern in the science of computers was the elimination of redundancies in order to preserve hardware resources. Thus, blocks of code that were frequently executed were organized in subprograms, or subroutines, or procedures. Such structures are made of blocks of instructions that can be called when needed. A call to a procedure is made without any arguments, and no return values are expected from it. Thus, the thread of executions simply moves from the main program to the subroutine instructions and back. Functions on the other hand, are parameterized blocks of instructions, that can receive arguments when they are called and may provide useful return values. Both functions and procedures are pieces of code that can be called from different places, either from the main program or from other procedures and functions. Functions may behave like procedures if they take no arguments. In other words, a function receives inputs for its piece of code and it can return an output. In contrast, a procedure lacks any inputs for their piece of code and it has no return output. In recent decades, the notion of procedure started to fade away in some computer languages, as functions can be used as procedures. A caller that uses a function

that has no arguments and no usable return values, is in fact a masked procedure. A series of examples regarding functions and procedures are given in Additional algorithm 8.8.

A global variable *a* is declared and an integer value is assigned to it. A variable *b* is also declared and a value returned by a function "f" is assigned to it. Function "f" receives an argument and returns the result of the evaluation of a mathematical expression. Next

Lang.	Example	Output

JS	`a = 10;`
	
	`b = pure(a);`
	`print(b + " & " + a);`
	
	`c = inpure(a);`
	`print(c + " & " + a);`
	
	`d = inpure(a);`
	`print(d + " & " + a);`
	
	`function pure(x){`
	`  return x + x / x - x * x;`
	`}`
	
	`function inpure(x){`
	`  a = 11;`
	`  return x + x / x - x * x;`
	`}`
		`JS Output:`
		`-89 & 10`
		`-89 & 11`
		`-109 & 11`

C#	`using System;`
	`class HelloWorld {`
	
	`  public static int a;`
	
	`  static void Main() {`
	
	`  a = 10;`
	
	`  int b = pure(a);`
	`  Console.WriteLine(b + " & " + a);`
		`C# Output:`
		`-89 & 10`
		`-89 & 11`
		`-109 & 11`

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

```
        int c = inpure(a);
        Console.WriteLine(c + " & " + a);

        int d = pure(a);
        Console.WriteLine(d + " & " + a);
    }

    static int pure(int x){
        return x + x / x - x * x;
    }

    static int inpure(int x){
        a = 11;
        return x + x / x - x * x;
    }
}
```

VB

```
Dim a, b As Integer

Private Sub Form_Load()

    a = 10
    b = pure(a)
    Debug.Print b & " & " & a

    c = inpure(a)
    Debug.Print c & " & " & a

    d = pure(a)
    Debug.Print d & " & " & a

End Sub
Function pure(x) As Integer
    pure = x + x / x - x * x
End Function

Function inpure(x) As Integer
    a = 11
    inpure = x + x / x - x * x
End Function
```

```
VB Output:
-89 & 10
-109 & 11
-109 & 11
```

PHP

```
$a = 10;

$b = pure($a);
echo "$b & $a\n";

$c = inpure($a);
echo "$c & $a\n";

$d = inpure($a);
echo "$d & $a";

function pure($x){
    return $x + $x / $x - $x * $x;
}
```

```
PHP Output:
-89 & 10
-89 & 11
-109 & 11
```

Additional algorithm 8.7 (continued)

```
function inpure($x){
    global $a;
    $a = 11;
    return $x + $x / $x - $x * $x;
}
```

PERL

```
$a = 10;

my $b = pure($a);
print "$b & $a\n";

my $c = inpure($a);
print "$c & $a\n";

my $d = inpure($a);
print "$d & $a";

sub pure{
    my ($x) = @_;
    return $x + $x / $x - $x * $x;
}

sub inpure{
    my ($x) = @_;
    $a = 11;
    return $x + $x / $x - $x * $x;
}
```

```
PERL Output:
-89 & 10
-89 & 11
-109 & 11
```

Ruby

```
def pure(x)
    return x + x / x - x * x
end

def inpure(x)
    $a = 11
    return x + x / x - x * x
end

$a = 10

b = pure($a)
puts "#{b} & #{$a}";

c = inpure($a)
puts "#{c} & #{$a}";

d = pure($a)
puts "#{d} & #{$a}";
```

```
Ruby Output:
-89 & 10
-89 & 11
-109 & 11
```

Additional algorithm 8.7 (continued)

Java

```java
public class Main
{
    public static int a;

    public static void main(String[] args) {

        a = 10;

        int b = pure(a);
        System.out.println(b + " & " + a);

        int c = inpure(a);
        System.out.println(c + " & " + a);

        int d = pure(a);
        System.out.println(d + " & " + a);
    }

    static int pure(int x){
        return x + x / x - x * x;
    }

    static int inpure(int x){
        a = 11;
        return x + x / x - x * x;
    }
}
```

Java Output:

```
-89 & 10
-89 & 11
-109 & 11
```

Python

```python
def pure(x):
    return x + x / x - x * x

def inpure(x):
    global a
    a = 11
    return x + x / x - x * x

a = 10

b = pure(a)
print(str(b) + " & " + str(a))

c = inpure(a)
print(str(c) + " & " + str(a))

d = pure(a)
print(str(d) + " & " + str(a))
```

Python Output:

```
-89.0 & 10
-89.0 & 11
-109.0 & 11
```

Additional algorithm 8.7 (continued)

C++

```
#include <iostream>
using namespace std;

int a = 10;

int pure(int x){
    return x + x / x - x * x;
}

int inpure(int x){
    a = 11;
    return x + x / x - x * x;
}

int main()
{
    int b = pure(a);
    cout<<b<<" & "<<a<<"\n";

    int c = inpure(a);
    cout<<c<<" & "<<a<<"\n";

    int d = pure(a);
    cout<<d<<" & "<<a;

    return 0;
}
```

C++ Output:

```
-89 & 10
-89 & 11
-109 & 11
```

Additional algorithm 8.7 (continued)

the result is printed in the output for inspection. A call is made to procedure or function "p", which receives no arguments and returns no values. The block of code from "p" is made from two lines. In the first line the result of the subtraction of integer 11 from the content of variable *a*, is assigned to a local integer variable *x*. In the second line of the block of code from "p" the result of a mathematical expression is assigned to global variable *b*, and the thread of execution is returned to the caller. Once the procedure "p" returns the thread of execution, the content of variable *b* is again printed into the output for inspection. Please note that in C# and Java, procedures have the "void" declaration to indicate that the return value does not exist. In PHP and Python, global variables must be declared inside functions or procedures to make their values visible to their code blocks.

8.2.10 Built-In Functions

Any computer language is equipped by default with a series of internal functions designed to help the programmer in case of basic processing. Many of these functions have been

Lang.	Example	Output

JS	`a = 16;`
	
	`b = f(a); print(b);`
	`p(); print(b);`
	
	`function f(x){`
	`  return x + x / x - x * x;`
	`}`
	
	`function p(){`
	`  let x = a - 11;`
	`  b = x + x / x - x * x;`
	`}`
		`JS Output:`
		`-239`
		`-19`

C#	`using System;`
	`class HelloWorld {`
	
	`  public static int b;`
	`  public static int a;`
	
	`  static void Main() {`
	
	`  a = 16;`
	`  b = f(a);`
	
	`  Console.WriteLine(b);`
	
	`  p();`
	`  Console.WriteLine(b);`
	`  }`
	
	`  static int f(int x){`
	`  return x + x / x - x * x;`
	`  }`
		`C# Output:`
		`-239`
		`-19`

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

```
    static void p(){
        int x = a - 11;
        b = x + x / x - x * x;
    }
}
```

VB

```
Dim a, b As Integer

Private Sub Form_Load()
    a = 16
    b = f(a)
    Debug.Print b
    p
    Debug.Print b
End Sub

Function f(x)
    f = x + x / x - x * x
End Function

Sub p()
    x = a - 11
    b = x + x / x - x * x
End Sub
```

```
VB Output:
-239
-19
```

PHP

```
$a = 16;
$b = f($a);
echo $b . "\n";
p(); echo $b;

function f($x){
    return $x + $x / $x - $x * $x;
}

function p(){
    global $a;
    global $b;
    $x = $a - 11;
    $b = $x + $x / $x - $x * $x;
}
```

```
PHP Output:
-239
-19
```

PERL

```
$a = 16;
$b = f($a);
print $b . "\n";
p(); print $b;

sub f{
    my ($x) = @_;
    return $x + $x / $x - $x * $x;
}
```

```
PERL Output:
-239
-19
```

Additional algorithm 8.8 (continued)

```
sub p{
    $x = $a - 11;
    $b = $x + $x / $x - $x * $x;
}
```

Ruby

```
def f(x)
    return x + x / x - x * x
end

def p()
    x = $a - 11
    $b = x + x / x - x * x
end

$a = 16
$b = f($a); puts $b
p(); puts $b
```

```
Ruby Output:
-239
-19
```

Java

```
public class Main
{
    public static int b;
    public static int a;

    public static void main(String[] args) {

        a = 16;
        b = f(a);

        System.out.println(b);

        p();

        System.out.println(b);
    }

    static int f(int x){
        return x + x / x - x * x;
    }

    static void p(){
        int x = a - 11;
        b = x + x / x - x * x;
    }
}
```

```
Java Output:
-239
-19
```

Python

```
def f(x):
    return x + x / x - x * x
```

```
Python Output:
-239
-19
```

Additional algorithm 8.8 (continued)

```
def p():
    global a
    global b
    x = a - 11
    b = x + x / x - x * x

a = 16
b = f(a); print(b)
p(); print(b)
```

C++

```
#include <iostream>
using namespace std;

int a = 16;
int b = 0;

int f(int x){
    return x + x / x - x * x;
}

void p(){
    int x = a - 11;
    b = x + x / x - x * x;
}

int main()
{
    b = f(a); cout<<b;
    p(); cout<<b;
    return 0;
}
```

C++ Output:

```
-239
-19
```

Additional algorithm 8.8 (continued)

used in the examples given throughout the chapters. For example, the functions for conversion of an integer to a string, the functions for detecting the number of characters of a string, or the detection of the number of elements inside an array, and so on. The example below demonstrates a strategy for detecting a string inside another string by using built-in functions, such as "replace", "split", "join", "len" or their synonyms in different computer languages. The approach is based on the length difference between two strings. Namely, the difference in length between an original string and the length of the same string after replacing some characters of interest. Notes: Another strategy that is demonstrated here is function chaining, where the value returned by one function is directly an argument to the next function, without the need to use intermediate variables. Function chaining is grouping function calls in one single line using dot notation (i.e. $a = \mathrm{f1}(x).\mathrm{f2}(y).\mathrm{f3}(z)$). In normal conditions, a search for a substring is done by using a specialized function that is able to return the position of the target on the main string. However, in many

cases the specific position of the target is not important. The detection of a target on the main string may be more important than an exact location. The presence or absence of the target string value is sufficient for triggering a condition statement in one direction or another. The example from Additional algorithm 8.9 starts by declaring two variables, namely *a* and *q*. A string is assigned to variable *a* and another string is assigned to variable *q*. Variable *a* holds the source string. Variable *q* holds the target string that must be searched inside variable *a*.

Next, a variable *b* holds the length of the original string found in variable *a*, whereas another variable *c* holds the length of a version of the string from variable *a*, that lacks any instance of *q*. Two main strategies for replacing substrings are shown here. Both of them use built-in functions. The first approach is the split of the string from variable *a*, based on the delimiter found in *q*. That will return an unnamed array. The array is then converted back to a string using the "join" function. The "join" function allows one argument, namely the option to insert a delimiter between the values found in the elements of an array when the string is constructed. However, in order to obtain an "unpolluted" string, the argument for the "join" function is void. Next, the string provided by the "join" function is passed to the "length" function (or method), which in turn returns the number of characters inside the string. This strategy can be observed in computer languages like Javascript and PERL. Notice that the "split", "join" and "length" functions are chained to one another and the final result is assigned to variable *c*. The second approach is the "replace" function. In the string stored in variable *a*, all *q* encounters are replaced with nothing. Next, the string provided by the "replace" function is an argument for the "length" function. The "length" function in turn provides an integer value that specifies the number of characters in the string. Notice again that the two functions are chained together and the final result is assigned to variable *c*. Thus, in order to detect the presence of *q* in *a*, the values of the two variables *b* and *c* are compared. If the two variables contain the same value, then it means that characters of *q* were not present inside the string found in variable *b*. Otherwise, different values stored in variables *b* and *c*, show that characters of *q* are present inside the string found in variables *b*. Note again that functions are called methods in object oriented computer languages. However here the term function is used nonetheless.

8.3 Conclusions

Functions are used whenever the repetition of a block of instructions is necessary. Functions exist as a solution to redundancy, namely a solution to avoid writing the same code over and over again. Therefore, functions are pieces of code that are executed repeatedly in redundant tasks. These pieces of codes called functions, are able to receive certain inputs which in turn lead to specific outputs. The input values that are passed to a function are known as arguments. One way to imagine a function by association is to think of

Lang.	Example	Output

JS	`a = "*******%%**********%%******";`
	`q = "%%";`
	
	`b = a.length;`
	`c = a.split(q).join("").length;`
	
	`if(c < b){print("a contains q");}`
		JS Output:
		a contains q

C#	`using System;`
	`class HelloWorld {`
	`  static void Main() {`
	
	`    string a = "*******%%**********%%******";`
	`    string q = "%%";`
	
	`    int b = a.Length;`
	`    int c = a.Replace(q, "").Length;`
	
	`    if(c < b){`
	`        Console.WriteLine("a contains q");`
	`            }`
	`  }`
	`}`
		C# Output:
		a contains q

VB	`Private Sub Form_Load()`
	`    a = "*******%%**********%%******"`
	`    q = "%%"`
	
	`    b = Len(a)`
	`    c = Len(Replace(a, q, ""))`
	
	`    If (c < b) Then`
	`        Debug.Print "a contains q"`
	`    End If`
	`End Sub`
		VB Output:
		a contains q

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

PHP

```php
$a = "*******%%**********%%******";
$q = "%%";

$b = strlen($a);
$c = strlen(str_replace($q, "", $a));

if($c < $b){echo "a contains q";}
```

```
PHP Output:
a contains q
```

PERL

```perl
$a = "*******%%**********%%******";
$q = "%%";

$b = length($a);

$c = length(join "", split($q, $a));

if($c < $b){print "a contains q";}
```

```
PERL Output:
a contains q
```

Ruby

```ruby
a = "*******%%**********%%******"
q = "%%"

b = a.length
c = a.gsub(q,'').length

if(c != b) then puts("a contains q") end
```

```
Ruby Output:
a contains q
```

Java

```java
public class Main
{
    public static void main(String[] args) {

    String a = "*******%%**********%%******";
    String q = "%%";

    int b = a.length();
    int c = a.replace(q, "").length();

    if(c < b){
        System.out.println("a contains q");
    }

    }
}
```

```
Java Output:
a contains q
```

Python

```python
a = "*******%%**********%%******"
q = "%%"
```

```
Python Output:
a contains q
```

Additional algorithm 8.9 (continued)

```
b = len(a)
c = len(a.replace(q, ""))

if(c < b): print("a contains q")
```

C++

```
#include <iostream>
#include <regex>
#include <string>
using namespace std;

int main()
{
    string a = "*******%%**********%%******";
    string q = "%%";

    int b = a.size();
    int c = (regex_replace(a,
    regex(q), "")).size();

    if(c < b){cout<<"a contains q";}

    return 0;
}
```

```
C++ Output:
a contains q
```

Additional algorithm 8.9 (continued)

the function as a "black box" that performs an operation, that is, it does something when you put something inside. The reason for which a function can be viewed as a "black box" is related to the software community. The software community, worldwide, contains a set of programmers. Out of this set of programmers, a good percentage of them provide coding solutions to other programmers in the community. Thus, a complex function made in the past by a programmer can always be used as is by both novice and advanced software developers without having to fully understand the contents of that function. The only requirement in such cases is for another programmer to understand that this black box called 'the function' may receive a series of arguments and in turn it can provide a specific output format.

Implementations and Experiments

9

9.1 Introduction

The similarity of programming paradigms between different computer languages leads to a steep learning curve for new computer languages, which mainly involves knowledge of syntaxes. For this to be true, prior knowledge of at least one computer language is required. However, regardless of the computer language used, knowledge of algorithms can be a useful tool for performing various automation operations regardless of the computer language used. This chapter begins with descriptions of a series of recursive functions where different types of arguments and return values are demonstrated. Examples include recursive functions that: (1) repeat a character *n* times to form a longer string value, (2) perform a sum of integers between zero and *n*, (3) compute the factorial of an integer *n*, (4) make number sequence generators, (5) compute the Fibonacci sequence, and (6) sum all the integers found over an array variable. In the second part of the chapter, a scanner is presented as a means to traverse a range of numbers in order to detect their distribution when a mathematical expression is used. In the last part, the chapter describes the implementation of a new method called *Spectral Forecast*. In this example, the use of string values as number sequences is thoroughly explored. The example highlights the use of the split function and array variables in the context of input values seen in data science, namely data stored as string values in various file types.

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/978-3-031-23277-0_9.

P. A. Gagniuc, *An Introduction to Programming Languages: Simultaneous Learning in Multiple Coding Environments*, Synthesis Lectures on Computer Science,
https://doi.org/10.1007/978-3-031-23277-0_9

9.2 Recursion Experiments

Recursion can be an optimal way to program a computer in certain situations because it eliminates the redundant steps normally performed by a chain of calls. In the example from Additional algorithm 9.1, a series of implementations are shown. A total of six implementations of recursive functions are presented, each with different particularities that may help the reader understand at least some of the variations that are possible. Note that only the following examples are given in this subsection: (1) an example of recursive function that provides a string value of a certain length, which is made by a repetition of a specific character *n* times. (2) A recursive function that provides the sum of a range starting from zero to *n*. (3) A recursive function that provides the result for *n* factorial. (4) A recursive function that returns a sequence of numbers based on a specified mathematical expression. (5) Another recursive function that returns the *Fibonacci* sequence, and finally, (6) a recursive function that calculates a sum based on the integers found above an array variable.

9.2.1 Repeat String *n* Times

The first example presents a recursive function "x" that is able to return a string of a certain length made from a specific ASCII character. Function "x" takes three arguments. Namely, an argument (i.e. parameter *c*) containing the string character used by the recursive function to make the return string value, a second argument (i.e. parameter *s*) set to an empty string value that is used as an accumulator for the forming string, and a third argument (i.e. parameter *n*) that is used as an upper limit of the recursion. Inside function "x", the character stored in variable *c* is added by aggregate assignment (or by simple assignment) to whatever string value is already present inside variable *s*. Next, a condition verifies if the number of characters in variable *s* is higher or equal to the upper limit indicated in variable *n*. If the condition is false, the "x" function makes a call to itself by using the updated values for variable *s*. If the condition is true, function "x" returns the value of variable *s*. Once the value is returned, it is immediately assigned to variable *a*, which is further printed into the output for inspection. Notes: The first parameter also takes groups of characters (entire strings).

9.2.2 Sum from 0 to *n*

The second example makes a sum of all numbers in the range 0 and *n*. This is perhaps the simplest possible recursive function as it takes one argument and it contains one condition statement. Inside the "sum" function a condition verifies if the value of variable *n* is less or equal to 1. If the condition is false, then nothing is triggered and the thread

of execution continues to the next line, where the returned value is the value of variable *n* plus the return value of a call to itself by using *n* minus one as an argument. However,

Lang.	Example	Output

JS

```
a = x("*", "", 10);
print("Repeat:\n[" + a + "]");

b = sum(23);
print("Sum:[" + b + "]");

c = factorial(10);
print("Factorial:\n[" + c + "]");

d = sequence(5, [], 0, 5);
print("A sequence:\n[" + d + "]");

e = fibonacci(2, [0, 1, 2], 5);
print("Fibonacci:\n[" + e + "]");

q = [1, 3, 3, 4, 5, 9];
f = sum_array(q.length - 1, q, 0);
print ("Sum array:[" + f + "]");


// repeat string n times
function x(c, s, n){

    s += c;

    if(s.length>=n){
        return s;
    } else {
        return x(c, s, n);
    }
}

// sum from 0 to n
function sum(n){
    if (n <= 1) {return n;}
    return n + sum(n - 1);
}
```

```
JS Output:
Repeat:
[**********]
Sum:[276]
Factorial:
[3628800]
A sequence:
[5,7,11,19,35]
Fibonacci:
[0,1,2,3,5,8]
Sum array:[25]
```

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

```
// factorial from 0 to n
function factorial(n){
    if (n <= 1) {
        return n;
    } else {
        return factorial(n - 1) * n;
    }
}


// just a sequence
function sequence(n, m, i, t){

    m[i] = n;
    i++;

    if (i >= t) {
        return m;
    } else {
        return sequence((n-1)+(n-2), m, i,
t);
    }
}

// fibonacci
function fibonacci(n, m, t){

    n++;
    m[n] = m[n-1] + m[n-2];

    if (n >= t) {
        return m;
    } else {
        return fibonacci(n, m, t);
    }
}

// sum all from array
function sum_array(n, q, r){

    r += q[n];

    if (n <= 0) {
        return r;
    } else {
         return sum_array(n - 1, q, r);
     }
 }
```

Additional algorithm 9.1 (continued)

C#

```
using System;
class HelloWorld {
  static void Main() {

    string a = x("*", "", 10);
    Console.WriteLine("Repeat:\n[" + a +
"]");

    int b = sum(23);
    Console.WriteLine("Sum:[" + b + "]");

    int c = factorial(10);
    Console.WriteLine(
           "Factorial:\n[" + c + "]"
                     );

    int[] d = sequence(5, new int[5], 0, 5);

    Console.WriteLine(
              "A sequence:\n[{0}",
              string.Join(",", d) + "]"
                     );

   int[] e = fibonacci(2,
             new int[3] {1,2,3}, 5);
   Console.WriteLine(
             "Fibonacci:\n[{0}",
             string.Join(",", e) + "]"
                    );

    int[] q = {1, 3, 3, 4, 5, 9};

    int f = sum_array(q.Length - 1, q, 0);
    Console.WriteLine(
           "Sum array:[" + f + "]"
                     );
  }

    // repeat string n times
    static string x(string c, string s,
                    int n){
        s += c;

        if(s.Length>=n){
            return s;
        } else {
            return x(c, s, n);
        }
```

```
C# Output:
Repeat:
[**********]
Sum:[276]
Factorial:
[3628800]
A sequence:
[5,7,11,19,35]
Fibonacci:
[1,2,3,5,8,13]
Sum array:[25]
```

Additional algorithm 9.1 (continued)

```
    }


    // sum from 0 to n
    static int sum(int n){
        if (n <= 1) {return n;}
        return n + sum(n - 1);
    }


    // factorial from 0 to n
    static int factorial(int n){
        if (n <= 1) {
            return n;
        } else {
            return factorial(n - 1) * n;
        }
    }


     // just a sequence
     static int[] sequence(int n, int[] m,
                           int i, int t){
        m[i] = n;
        i++;

        if (i >= t) {
          return m;
        } else {
          return sequence((n-1)+(n-2),
                           m, i, t);
        }
    }


     // fibonacci
     static int[] fibonacci(int n, int[] m,
                            int t){
        n++;
        Array.Resize(ref m, n+1);
        m[n] = m[n-1] + m[n-2];

        if (n >= t) {
            return m;
        } else {
            return fibonacci(n, m, t);
        }
    }


    // sum all from array
    static int sum_array(int n, int[] q,
                         int r){
```

Additional algorithm 9.1 (continued)

```
        r += q[n];

        if (n <= 0) {
            return r;
        } else {
            return sum_array(n - 1, q, r);
        }
    }
}
```

VB

```
Private Sub Form_Load()

    ' repeat string n times
    a = x("*", "", 10)
    Debug.Print ("Repeat:" & _
    vbCrLf & "[" & a & "]")

    ' sum from 0 to n
    b = sum(23)
    Debug.Print ("Sum:[" & b & "]")

    ' factorial from 0 to n
    c = factorial(10)
    Debug.Print ("Factorial:[" & _
    vbCrLf & "[" & c & "]")

    ' just a sequence
    Dim m(0 To 4) As Integer
    Dim d() As Integer
    d = sequence(5, m, 0, 5)
    For i = 0 To UBound(d)
        t = t & d(i) & ","
    Next i
    Debug.Print ("A sequence:[" & _
    vbCrLf & "[" & t & "]")

    ' fibonacci
    Dim e() As Integer
    Dim h(0 To 5) As Integer

    h(0) = 0
    h(1) = 1
    h(2) = 2

    t = Empty
    e = fibonacci(2, h, 5)
    For i = 0 To UBound(e)
        t = t & e(i) & ","
    Next i
    Debug.Print ("Fibonacci:[" & _
    vbCrLf & "[" & t & "]")
```

```
VB Output:
Repeat:
[**********]
Sum:[276]
Factorial:[
[3628800]
A sequence:[
[5,7,11,19,35,]
Fibonacci:[
[0,1,2,3,5,8,]
Sum array:[25]
```

Additional algorithm 9.1 (continued)

```
    ' sum all from array
    q = Array(1, 3, 3, 4, 5, 9)
    f = sum_array(UBound(q), q, 0)
    Debug.Print ("Sum array:[" & f & "]")

End Sub


' repeat string n times
Function x(c, s, n)

    s = s + c

    If (Len(s) >= n) Then
        x = s
    Else
        x = x(c, s, n)
    End If

End Function


' sum from 0 to n
Function sum(n)
    If (n <= 1) Then
        sum = n
    Else
        sum = n + sum(n - 1)
    End If
End Function


' factorial from 0 to n
Function factorial(n)
    If (n <= 1) Then
        factorial = n
    Else
        factorial = factorial(n - 1) * n
    End If
End Function


' just a sequence
Function sequence(n, ByRef m, i, t)

    m(i) = n
    i = i + 1

    If (i >= t) Then
        sequence = m
    Else
        sequence = sequence( _
        (n - 1) + (n - 2), m, i, t)
```

Additional algorithm 9.1 (continued)

```
    End If

End Function


' fibonacci
Function fibonacci(n, ByRef m, t)

    n = n + 1
    m(n) = m(n - 1) + m(n - 2)

    If (n >= t) Then
        fibonacci = m
    Else
        fibonacci = fibonacci(n, m, t)
    End If

End Function


' sum all from array
Function sum_array(n, q, r)

    r = r + q(n)

    If (n <= 0) Then
        sum_array = r
    Else
        sum_array = sum_array(n - 1, q, r)
    End If

End Function
```

PHP

```
// repeat string n times
$a = x("*", "", 10);
echo("Repeat:\n[" . $a . "]\n");

// sum from 0 to n
$b = sum(23);
echo("Sum:[" . $b . "]\n");

// factorial from 0 to n
$c = factorial(10);
echo("Factorial:\n[" . $c . "]\n");

// just a sequence
$s=""; $d = sequence(5,  [], 0, 5);
foreach($d as $i){$s .= $i . ",";}
echo("A sequence:\n[" . $s . "]\n");

// fibonacci
$s=""; $e = fibonacci(2, [0, 1, 2], 5);
```

```
PHP Output:
Repeat:
[**********]
Sum:[276]
Factorial:
[3628800]
A sequence:
[5,7,11,19,35,]
Fibonacci:
[0,1,2,3,5,8,]
Sum array:[25]
```

Additional algorithm 9.1 (continued)

```php
foreach($e as $i){$s .= $i . ",";}
echo("Fibonacci:\n[" . $s . "]\n");

// sum all from array
$q = [1, 3, 3, 4, 5, 9];
$f = sum_array(count($q) - 1, $q, 0);
echo ("Sum array:[" . $f . "]");


// repeat string n times
function x($c, $s, $n){

    $s .= $c;

    if(strlen($s) >= $n){
        return $s;
    } else {
        return x($c, $s, $n);
    }
}


// sum from 0 to n
function sum($n){
    if ($n <= 1) {return $n;}
    return $n + sum($n - 1);
}


// factorial from 0 to n
function factorial($n){
    if ($n <= 1) {
        return $n;
    } else {
        return factorial($n - 1) * $n;
    }
}


// just a sequence
function sequence($n, $m, $i, $t){

    $m[$i] = $n;
    $i++;

    if ($i >= $t) {
        return $m;
    } else {
    return sequence(($n-1)+($n-2),
                        $m, $i, $t);
    }
}
```

Additional algorithm 9.1 (continued)

```
// fibonacci
function fibonacci($n, $m, $t){

    $n++;
    $m[$n] = $m[$n-1] + $m[$n-2];

    if ($n >= $t) {
        return $m;
    } else {
        return fibonacci($n, $m, $t);
    }
}


// sum all from array
function sum_array($n, $q, $r){

    $r += $q[$n];

    if ($n <= 0) {
        return $r;
    } else {
        return sum_array($n - 1, $q, $r);
    }
}
```

PERL

```
my $a = x("*", "", 10);
print "Repeat:\n[" . $a . "]\n";

# sum from 0 to n
my $b = sum(23);
print "Sum:[" . $b . "]\n";

# factorial from 0 to n
my $c = factorial(10);
print "Factorial:\n[" . $c . "]\n";

# just a sequence
my @d = sequence(5, [], 0, 5);
print "A sequence:\n[" . join(",", @d) .
"]\n";

# fibonacci
my @e = fibonacci(3, (1,2,3), 5);
print "Fibonacci:\n[" . join(",", @e) .
"]\n";

# sum all from array
@q = (1, 3, 3, 4, 5, 9);
$f = sum_array($#q+1, $q, 0);
print "Sum array:[" . $f . "]";
```

```
PERL Output:
Repeat:
[**********]
Sum:[276]
Factorial:
[3628800]
A sequence:
[5,7,11,19,35]
Fibonacci:
[1,2,3,5,8]
Sum array:[25]
```

Additional algorithm 9.1 (continued)

```perl
# repeat string n times
sub x{

    my ($c, $s, $n) = @_;

    $s .= $c;

    if(length($s) >= $n){
        return $s;
    } else {
        return x($c, $s, $n);
    }
}


# sum from 0 to n
sub sum($n){
    my ($n) = @_;
    if ($n <= 1) {return $n;}
    return $n + sum($n - 1);
}


# factorial from 0 to n
sub factorial($n){
    my ($n) = @_;
    if ($n <= 1) {
        return $n;
    } else {
        return factorial($n - 1) * $n;
    }
}


# just a sequence
sub sequence{

    my ($n, $m, $i, $t) = @_;

    $m[$i] = $n;
    $i++;

    if ($i >= $t) {
        return @m;
    } else {
        return sequence(
               ($n-1)+($n-2), $m, $i, $t
                       );
    }
}
```

Additional algorithm 9.1 (continued)

```perl
# fibonacci
sub fibonacci{

    my ($n, @m, $t) = @_;

    $n++;
    $m[$n] = $m[$n-1] + $m[$n-2];

    if ($n >= $t) {
        return @m;
    } else {
        return fibonacci($n, @m, $t);
    }
}


# sum all from array
sub sum_array{
    my ($n, $q, $r) = @_;

    $r += $q[$n];

    if ($n <= 0) {
        return $r;
    } else {
        return sum_array($n - 1, $q, $r);
    }
}
```

Ruby

```ruby
# repeat string n times
def x(c, s, n)

    s += c

    if(s.size()>=n)
        return s
    else
        return x(c, s, n)
    end
end


# sum from 0 to n
def sum(n)
    if (n <= 1)
        return n
    end
    return n + sum(n - 1)
end
```

```
Ruby Output:
Repeat:
[**********]
Sum:[276]
Factorial:
[3628800]
A sequence:
[5,7,11,19,35]
Fibonacci:
[0,1,2,3,5,8]
Sum array:[25]
```

Additional algorithm 9.1 (continued)

```
# factorial from 0 to n
def factorial(n)
    if (n <= 1)
        return n
    else
        return factorial(n - 1) * n
    end
end


# just a sequence
def sequence(n, m, i, t)

    m[i] = n
    i = i + 1

    if (i >= t)
        return m
    else
        return sequence((n-1)+(n-2), m, i, t)
    end
end


# fibonacci
def fibonacci(n, m, t)

    n = n + 1
    m[n] = m[n-1] + m[n-2]

    if (n >= t)
        return m
    else
        return fibonacci(n, m, t)
    end
end


# sum all from array
def sum_array(n, q, r)

    r = r + q[n]

    if (n <= 0)
        return r
    else
        return sum_array(n - 1, q, r)
    end
end


# repeat string n times
a = x("*", "", 10)
```

Additional algorithm 9.1 (continued)

```
puts("Repeat:\n[" + a + "]")

# sum from 0 to n
b = sum(23)
puts("Sum:[" + b.to_s + "]")

# factorial from 0 to n
c = factorial(10)
puts("Factorial:\n[" + c.to_s + "]")

# just a sequence
d = sequence(5, [0]*5, 0, 5)
puts("A sequence:\n" + d.to_s)

# fibonacci
t = [0]*6
t[1] = 1; t[2] = 2
e = fibonacci(2, t, 5)
puts("Fibonacci:\n" + e.to_s)

# sum all from array
q = [1, 3, 3, 4, 5, 9]
f = sum_array(q.size() - 1, q, 0)
puts("Sum array:[" + f.to_s + "]")
```

Java

```
public class Main
{
    public static void main(String[] args) {

    // repeat string n times
    String a = x("*", "", 10);
    System.out.println("Repeat:\n[" +
                    a + "]"
                      );

    // sum from 0 to n
    int b = sum(23);
    System.out.println("Sum:[" + b + "]");

    // factorial from 0 to n
    int c = factorial(10);
    System.out.println(
          "Factorial:\n[" + c + "]"
                      );

    // just a sequence
    int[] d = sequence(5, new int[5], 0, 5);
    String l = "";
    for(int i = 0; i < d.length; i++){
        l += d[i] + ",";
        }
```

```
Java Output:
Repeat:
[**********]
Sum:[276]
Factorial:
[3628800]
A sequence:
[5,7,11,19,35,]
Fibonacci:
[1,2,3,5,8,13,]
Sum array:[25]
```

Additional algorithm 9.1 (continued)

```
        System.out.println(
               "A sequence:\n[" + l + "]"
                           );

        // fibonacci
        l = "";
        int[] e = fibonacci(2,
                  new int[]{1,2,3,0,0,0}, 5);
        for(int i = 0; i < e.length; i++){
            l += e[i] + ",";
            }

        System.out.println(
                "Fibonacci:\n[" + l + "]"
                           );

        // sum all from array
        int[] q = {1, 3, 3, 4, 5, 9};
        int f = sum_array(q.length - 1, q, 0);
        System.out.println(
               "Sum array:[" + f + "]"
                           );
        }


        // repeat string n times
        static String x(String c, String s,
                                    int n){
            s += c;

            if(s.length() >= n){
                return s;
            } else {
                return x(c, s, n);
            }
        }


        // sum from 0 to n
        static int sum(int n){
            if (n <= 1) {return n;}
            return n + sum(n - 1);
        }


        // factorial from 0 to n
        static int factorial(int n){
            if (n <= 1) {
                return n;
            } else {
                return factorial(n - 1) * n;
            }
        }
```

Additional algorithm 9.1 (continued)

```
    // just a sequence
    static int[] sequence(int n, int[] m,
                          int i, int t){
        m[i] = n;
        i++;

        if (i >= t) {
            return m;
        } else {
            return sequence((n-1)+(n-2),
                            m, i, t);
        }
    }


  // fibonacci
  static int[] fibonacci(int n, int[] m,
                         int t){
        n++;
        m[n] = m[n-1] + m[n-2];

        if (n >= t) {
            return m;
        } else {
            return fibonacci(n, m, t);
        }
    }


    // sum all from array
    static int sum_array(int n, int[] q,
                         int r){
        r += q[n];

        if (n <= 0) {
            return r;
        } else {
            return sum_array(n - 1, q, r);
        }
    }
}
```

Python

```
# repeat string n times
def x(c, s, n):

    s += c

    if(len(s)>=n):
        return s
    else:
```

```
Python Output:
Repeat:
[**********]
Sum:[276]
Factorial:
[3628800]
A sequence:
[5,7,11,19,35]
```

Additional algorithm 9.1 (continued)

```
        return x(c, s, n)


# sum from 0 to n
def sum(n):
    if (n <= 1): return n
    return n + sum(n - 1)


# factorial from 0 to n
def factorial(n):
    if (n <= 1):
        return n
    else:
        return factorial(n - 1) * n


# just a sequence
def sequence(n, m, i, t):

    m[i] = n
    i = i + 1

    if (i >= t):
        return m
    else:
        return sequence((n-1)+(n-2), m, i, t)


# fibonacci
def fibonacci(n, m, t):

    n = n + 1
    m[n] = m[n-1] + m[n-2]

    if (n >= t):
        return m
    else:
        return fibonacci(n, m, t)


# sum all from array
def sum_array(n, q, r):

    r = r + q[n]

    if (n <= 0):
        return r
    else:
        return sum_array(n - 1, q, r)


# repeat string n times
```

```
Fibonacci:
[0,1,2,3,5,8]
Sum array:[25]
```

Additional algorithm 9.1 (continued)

```
a = x("*", "", 10)
print("Repeat:\n[" + a + "]")

# sum from 0 to n
b = sum(23)
print("Sum:[" + str(b) + "]")

# factorial from 0 to n
c = factorial(10)
print("Factorial:\n[" + str(c) + "]")

# just a sequence
d = sequence(5, [0]*5, 0, 5)
print("A sequence:\n" + str(d))

# fibonacci
t = [0]*6
t[1] = 1; t[2] = 2
e = fibonacci(2, t, 5)
print("Fibonacci:\n" + str(e))

# sum all from array
q = [1, 3, 3, 4, 5, 9]
f = sum_array(len(q) - 1, q, 0)
print("Sum array:[" + str(f) + "]")
```

C++

```
#include <iostream>
#include <string>
using namespace std;

// repeat string n times
string x(string c, string s, int n){

    s += c;

    if(s.size()>=n){
        return s;
    } else {
        return x(c, s, n);
    }
}


// sum from 0 to n
int sum(int n){
    if (n <= 1) {return n;}
    return n + sum(n - 1);
}


// factorial from 0 to n
int factorial(int n){
```

```
C++ Output:
Repeat:
[**********]
Sum:[276]
Factorial:
[3628800]
A sequence:
[5,7,11,19,35,]
Fibonacci:
[0,1,2,3,5,8,13,]
Sum array:[25]
```

Additional algorithm 9.1 (continued)

```
    if (n <= 1) {
        return n;
    } else {
        return factorial(n - 1) * n;
    }
}


// just a sequence
int* sequence(int n, int m[], int i, int t){

    m[i] = n;
    i++;

    if (i >= t) {
        return m;
    } else {
        return sequence((n-1)+(n-2), m, i,
t);
    }
}


// fibonacci
int* fibonacci(int n, int m[], int t){

    n++;
    m[n] = m[n-1] + m[n-2];

    if (n >= t) {
        return m;
    } else {
        return fibonacci(n, m, t);
    }
}


// sum all from array
int sum_array(int n, int q[], int r){

    r += q[n];

    if (n <= 0) {
        return r;
    } else {
        return sum_array(n - 1, q, r);
    }
}


int main()
{
    // repeat string n times
```

Additional algorithm 9.1 (continued)

```
    string a = x("*", "", 10);
    cout<<("Repeat:\n[" + a + "]\n");

    // sum from 0 to n
    int b = sum(23);
    cout<<("Sum:[" + to_string(b) + "]");

    // factorial from 0 to n
    int c = factorial(10);
    cout<<("\nFactorial:\n[" +
    to_string(c) + "]");

    // just a sequence
    int t = 5;
    int* d; //pointer to hold address
    d = sequence(5, new int[t], 0, t);
    //address of m

    cout<<"\nA sequence:\n[";
    for(int i=0; i<t; i++)
    {cout<<d[i]<<",";}
    //d[i] is equivalent to *(d+i)
    cout<<"]";

    // fibonacci
    t = 7; int* e;
    //e pointer to hold address
    e = fibonacci(2, new int[8]{0,1,2}, t);

    cout<<"\nFibonacci:\n[";
    for(int i=0; i<t; i++)
    {cout<<*(e+i)<<",";}
    //e[i] is equivalent to *(e+i)
    cout<<"]";

    // sum all from array
    int q[] = {1, 3, 3, 4, 5, 9};
    int l = sizeof(q) / sizeof(int);

    int f = sum_array(l - 1, q, 0);
    cout<<("\nSum array:[" +
    to_string(f) + "]");

    return 0;
}
```

Additional algorithm 9.1 (continued)

if the condition is true, than the value of variable *n* is returned to the caller and is then assigned to variable *b*. Variable *b* is then printed into the output for inspection.

9.2.3 Factorial from 0 to *n*

The third example shows how to calculate the factorial in the range 1 to *n*. A function "factorial" takes one argument (i.e. parameter *n*) that represents the upper limit of the range of iterations. Inside the function, a condition verifies if *n* is less or equal to 1. If the condition is false then a return value is given from a multiplication of *n* with the value coming from a call to itself, in which the argument is *n* minus one. If the condition is true, than the return value is *n*. Then, the return value is assigned to a variable *c* that is printed in the output for inspection.

9.2.4 Simple Sequence Generator

The fourth example shows a function that generates a sequence of numbers. Thus, the "sequence" function takes a total of four arguments and is able to return an array variable. The argument for the first parameter *n* represents the seed number from which the sequence starts (in this case number five). The second parameter *m* represents an array variable. The third parameter is a counter variable *i*, and the fourth parameter *t*, takes the role of the upper limit for the number of recursions. Inside the function new elements are added to the array *m* by using the counter *i*, namely the variable *i*. Thus, at each recursion the value of *n* is assigned to an element *i* of variable *m*. Next, the value of variable *i* is incremented and a condition is stated. The condition verifies if the value from the counter is higher or equal to the value from variable *t*.

If the condition is false then the return value is taken from a call made to self, namely the "sequence" function that uses the updated argument values for *n*, *i* and *m*. If the condition is true, then function "sequence" returns the value of array variable *m*. In turn, the returned value is assigned to variable *d* which is further printed into the output. Note that the rule of getting the sequence (i.e. $((n - 1) + (n - 2))$, can be any mathematical expression that is suitable for this approach.

9.2.5 Fibonacci Sequence

The fifth example shows a recursive function that provides the fibonacci sequence up to a predefined position. A function "fibonacci" is able to take three arguments, namely an argument for *n* wich represents the counter for the recursion, an array variable were the results are stored (i.e. *m*), and another argument for *t* that specifies the upper limit of the recursion. In this function the first non-empty line increments the counter variable *n*. In the second line, the value for the element *n* of array *m* is calculated as the value from the previous element of *m* plus the value from the element *n* – 1 of *m*. Next, a condition verifies if counter *n* reached the upper limit indicated by variable *t*. In case the condition stands false, the recursion continues by a call to self, with updated values for the counter *n* and the array *m*. If the condition becomes true, namely the value stored in variable *n* is higher or equal than the value stored in variable *t*, then the function returns the array variable *m*. On the caller side, the returned value is assigned to variable *e*, which is then printed in the output.

9.2.6 Sum All Integers from Array

The last example in Additional algorithm 9.1, shows a recursive approach on how to sum all integers found in the elements of an array. Thus, an array *q* is declared in the main program by using integer literals. Next, the returned value from a function "sum array" is assigned to a variable *f*, which in turn is printed to the output for inspection. Function "sum array" takes three arguments. The argument for the first parameter *n* initially represents the total number of elements in array *q*. The second parameter is the array variable and the third parameter (i.e. *r*) is an integer variable used as an accumulator for the sum of the values from the elements of array *q*. Inside function "sum array", the integer value from the element *n* of array *q* is added to the integer value from the accumulator variable *r*. Next, a condition checks if *n* is less or equal to zero. If the condition is false, the return value is further requested by a call to self. However, if the condition is true, the sum from variable *r* is returned to the caller, were it is further assigned to variable *f*, as specified above.

9.3 Interval Scanning

Earlier in the above chapters, especially in the chapter describing functions, a mathematical expression has been used to demonstrate the principles behind functions, namely: $x + x/x - x \times x$. In this mathematical expression, different integers used for variable x provided specific integer values in the output. However, what if a distribution is required for a specific range of integers? A scanner may come in handy when such a distribution is needed for any mathematical expression. A function named "distribution" takes two arguments as the range of the input, namely the start integer and the stop integer. Thus, the function is able to provide an output integer for each integer in the input of that specified range. Initially, the return value of function "distribution" is assigned to variable *a*, which is further printed into the output for inspection. Once the thread of execution reaches the inside of the "distribution" function, the computations are made for each integer between the integer value declared for variable *start* (i.e. 3) and the integer value declared in variable *stop* (i.e. 21). The Additional algorithm 9.2 shows two versions of the "distribution" function. One version that uses a string variable to store the results as a string value, and another version that uses an array variable to store the results as integers.

Inside the first version, a variable *t* is declared and set to an empty string. Next, a for-loop statement traverses the range of the input, beginning from the integer value stored in the *start* variable up to the integer value stored in the *stop* variable. Inside the for-loop, the value returned by a "compute" function is added by aggregate assignment (i.e. "+=") to variable *t*. Once the for-loop finishes the iterations, the content of variable *t* is returned back to the caller, where it is further assigned to a variable *a*, as specified above. Inside the second version of the "distribution" function, an array variable *b* is set to empty. Next, a for-loop statement traverses the range between zero and a calculated value resulting from a subtraction of *start* from *stop* (i.e. *start—stop*). Inside the for-loop, the return value of function "compute" is assigned to the element *i* of array *b*. At the end of the iteration cycle, the array variable *b* is returned to the caller, where it is assigned in turn to a variable that is used to print the results in the output for inspection. Notes: In both versions, function "compute" takes an argument for *x* and returns the result of the evaluation of a mathematical expression. In the first version of the "compute" function the value of the *i* variable is used as an argument. In the second example, the argument used is the value of *i* plus the integer value from the *start* variable. That is, because the for-loop in the second example starts from zero to satisfy the positions of the elements in the array. Also of importance are the differences between computer languages when printing array data. In VB, Java and C++, the contents of the elements from array variables are usually taken by a repeat-loop that traverses the array. In C#, some advantages of split and join cascades can be used to display array data as strings without the additional help

Lang.	Example	Output

JS	`let a = distribution(3, 21);`
	`print(a);`
	
	`function distribution(start, stop){`
	
	`    let t = "";`
	
	`    for (let i = start; i < stop; i++) {`
	`        t += compute(i) + "\n";`
	`    }`
	
	`    return t;`
	`}`
	
	`function compute(x){`
	`    return x + x / x - x * x;`
	`}`
		JS Output:
		-5
		-11
		-19
		-29
		-41
		-55
		-71
		-89
		-109
		-131
		-155
		-181
		-209
		-239
		-271
		-305
		-341
		-379

	`let a = distribution(3, 21);`
	`print(a);`
	
	`function distribution(start, stop){`
	
	`    let b = [];`
	
	`    for (let i = 0; i < stop - start; i++) {`
	`        b[i] = compute(i + start);`
	`    }`
	
	`    return b;`
	`}`
	`function compute(x){`
	`    return x + x / x - x * x;`
	`}`	

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

C#

```
using System;
class HelloWorld {

  static void Main() {
    string a = distribution(3, 21);
    Console.WriteLine(a);
  }


  static string distribution(
                          int start, int stop){

    string t = "";

    for (int i = start; i < stop; i++) {
      t += compute(i) + "\n";
    }

    return t;
  }


  static int compute(int x){
    return x + x / x - x * x;
  }
}
```

```
C# Output:
-5
-11
-19
-29
-41
-55
-71
-89
-109
-131
-155
-181
-209
-239
-271
-305
-341
-379
```

```
using System;
class HelloWorld {

  static void Main() {
    int[] a = distribution(3, 21);
    Console.WriteLine(string.Join("\n", a));
  }


  static int[] distribution(
                          int start, int stop){

    int[] b = new int[stop - start];

    for (int i = start; i < stop; i++) {
      b[i - start] = compute(i);
    }

    return b;
  }
```

Additional algorithm 9.2 (continued)

```
  static int compute(int x){
    return x + x / x - x * x;
  }
}
```

VB

```
Private Sub Form_Load()
    a = distribution(3, 21)
    Debug.Print a
End Sub

Function distribution(stt, stp)

    t = ""

    For i = stt To stp
        t = t & compute(i) & vbCrLf
    Next i

    distribution = t

End Function


Function compute(x)
    compute = x + x / x - x * x
End Function
```

```
VB Output:
-5
-11
-19
-29
-41
-55
-71
-89
-109
-131
-155
-181
-209
-239
-271
-305
-341
-379
```

```
Private Sub Form_Load()

    Dim a() As Double
    a = distribution(3, 20)
    For i = LBound(a) To UBound(a)
        Debug.Print a(i)
    Next i

End Sub

Function distribution(st, sp) As Double()

    Dim b() As Double
    ReDim b(st To sp) As Double

    For i = 0 To sp - st
        b(i + st) = compute(i + st)
    Next i

    distribution = b

End Function
```

Additional algorithm 9.2 (continued)

```
Function compute(x) As Integer
    compute = x + x / x - x * x
End Function
```

PHP

```
$a = distribution(3, 21);
echo ($a);

function distribution($start, $stop){

    $t = "";

    for ($i = $start; $i < $stop; $i++) {
        $t .= compute($i) . "\n";
    }

    return $t;
}


function compute($x){
    return $x + $x / $x - $x * $x;
}
```

```
PHP Output:
-5
-11
-19
-29
-41
-55
-71
-89
-109
-131
-155
-181
-209
-239
-271
-305
-341
-379
```

```
$a = distribution(3, 21);
foreach($a as $i){echo $i . "\n";}

function distribution($start, $stop){

    $b = [];

    for ($i = 0; $i < $stop - $start; $i++) {
        $b[$i] = compute($i + $start);
    }

    return $b;
}


function compute($x){
    return $x + $x / $x - $x * $x;
}
```

PERL

```
my $a = distribution(3, 21);
print $a;

sub distribution{

    my ($start, $stop) = @_;
```

```
PERL Output:
-5
-11
-19
-29
```

Additional algorithm 9.2 (continued)

```
    $t = "";

    for ($i = $start; $i < $stop; $i++) {
        $t .= compute($i) . "\n";
    }

    return $t;
}


sub compute{
    my ($x) = @_;
    return $x + $x / $x - $x * $x;
}
```

```
-41
-55
-71
-89
-109
-131
-155
-181
-209
-239
-271
-305
-341
-379
```

```
my @a = distribution(3, 21);
print join("\n", @a);

sub distribution{

    my ($start, $stop) = @_;
    @b = [];

    for ($i = 0; $i < $stop - $start; $i++) {
        $b[$i] = compute($i + $start);
    }

    return @b;
}


sub compute{
    my ($x) = @_;
    return $x + $x / $x - $x * $x;
}
```

Ruby

```
def distribution(start, stop)

    t = ""
    for i in start..(stop-1)
        t += compute(i).to_s + "\n"
    end

    return t
end

def compute(x)
    return x + x / x - x * x
end

a = distribution(3, 21)
puts(a)
```

```
Ruby Output:
-5
-11
-19
-29
-41
-55
-71
-89
-109
-131
-155
-181
-209
-239
-271
```

Additional algorithm 9.2 (continued)

```
def distribution(start, stop)

    b = [stop - start]
    for i in 0..(stop - start - 1)
        b[i] = compute(i + start)
    end

    return b
end

def compute(x)
    return x + x / x - x * x
end

a = distribution(3, 21)
puts(a)
```

```
-305
-341
-379
```

java

```
public class Main
{
    public static void main(String[] args) {

        String a = distribution(3, 21);
        System.out.println(a);
    }


  static String distribution(
                          int start, int stop){

    String t = "";

    for (int i = start; i < stop; i++) {
      t += compute(i) + "\n";
    }

    return t;
  }


  static int compute(int x){
    return x + x / x - x * x;
  }
}
```

```
Java Output:
-5
-11
-19
-29
-41
-55
-71
-89
-109
-131
-155
-181
-209
-239
-271
-305
-341
-379
```

```
public class Main
{
    public static void main(String[] args) {

        int[] a = distribution(3, 21);
        String l = "";

        for(int i = 0; i < a.length; i++){
            l += a[i] + "\n";
        }
```

Additional algorithm 9.2 (continued)

```
        System.out.println(l);
    }


    static int[] distribution(
                        int start, int stop){

        int[] b = new int[stop - start];

        for (int i = start; i < stop; i++) {
            b[i - start] = compute(i);
        }

        return b;
    }


    static int compute(int x){
        return x + x / x - x * x;
    }
}
```

Python

```
def distribution(start, stop):

    t = ""
    for i in range(start, stop):
        t += str(compute(i)) + "\n"

    return t


def compute(x):
    return x + x / x - x * x

a = distribution(3, 21)
print(a)
```

```
def distribution(start, stop):

    b = [0]*(stop - start)

    for i in range(0, stop - start):
        b[i] = compute(i + start)

    return b


def compute(x):
    return x + x / x - x * x

a = distribution(3, 21)
print(a)
```

```
Python Output:
-5.0
-11.0
-19.0
-29.0
-41.0
-55.0
-71.0
-89.0
-109.0
-131.0
-155.0
-181.0
-209.0
-239.0
-271.0
-305.0
-341.0
-379.0
```

Additional algorithm 9.2 (continued)

C++

```cpp
#include <iostream>
using namespace std;

int compute(int x){
    return x + x / x - x * x;
}


string distribution(int start, int stop){

    string t = "";

    for (int i = start; i < stop; i++) {
        t += to_string(compute(i)) + "\n";
    }

    return t;
}


int main()
{
    string a = distribution(3, 21);
    cout<<a;

    return 0;
}
```

```
C++ Output:
-5
-11
-19
-29
-41
-55
-71
-89
-109
-131
-155
-181
-209
-239
-271
-305
-341
-379
```

```cpp
#include <iostream>
using namespace std;

int compute(int x){
    return x + x / x - x * x;
}

int* distribution(int start, int stop)
{
    int* b = new int[stop - start];

    for (int i = 0; i < stop - start; i++) {
        b[i] = compute(i + start);
    }

    return b;
}

int main()
{
    int* a = distribution(3, 21);

    for(int i=0; i<21-3; i++)
        cout<<a[i]<<"\n";

    // delete allocated memory
    delete[] a;
    return 0;
}
```

Additional algorithm 9.2 (continued)

```
    return b;
}

int main()
{
    int* a = distribution(3, 21);

    for(int i=0; i<21-3; i++)
        cout<<a[i]<<"\n";

    // delete allocated memory
    delete[] a;
    return 0;
}
```

Additional algorithm 9.2 (continued)

of a for-loop that traverses the array elements. In the other languages from out list, array variables can be printed just like any other primitive variables, without any additional use of statements.

9.4 Spectral Forecast

Spectral Forecast is a new method of prediction. The following example uses a new mathematical equation to process signals. The spectral forecast equation is part of this method and can be used for computations on multidimensional objects such as number sequences (1-dimensional), matrices (2-dimensional), tensor-like structures (n-dimensional), and so on. Here, a form of the *Spectral Forecast* equation is adapted for one-dimension, namely a sequence of numbers. This version of *Spectral Forecast* computes a mixed signal (i.e. M) between two other signals (i.e. A and B). Such a mix made by Spectral Forecast can be considered a special type of interpolation that is able to formulate a signal with dynamic characteristics (namely M). That is, the signal that is generated (M) may be closer in shape to one of the two signals (A or B) based on an index called distance (d). The index d takes values between zero and the maximum number found above the two discrete signals A and B. Thus, an index d of zero will provide a signal exactly like signal B, whereas an index d of maximum (whatever the maximum value is), will provide a signal M exactly like signal A. Thus, as d gets closer or farther from the maximum value, signal M will resemble more and more one of the signals and less of the other. The formulation of the spectral forecast is shown below:

$$M_{id} = \left[\left(\frac{d}{\text{Max}(A_i)}\right) \times A_i\right] + \left[\left(\frac{(\text{Max}(d) - d)}{\text{Max}(B_i)}\right) \times B_i\right]$$

where Max(A_i) is the maximum value found above signal A and Max(B_i) is the maximum value found above the components of signal B. Components A_i and B_i also represent the homologous values of the two signals at position i. Last but not least, Max(d) represents the maximum value between Max(A_i) and Max(B_i). Therefore the value of d can take values between 0 and Max(d). The above expression translates into C-like computer languages as seen below:

```
tmp = ((d / maxA) * tA[i]) + (((max - d) / maxB) * tB[i]);
```

Where *tA* and *tB* are the array variables that hold the consecutive values of the two signals. In case of iterations, the *tmp* variable holds the value for the third signal at position *i*. Note that the maximum distance is the maximum value found over the two signals. The example from Additional algorithm 9.3 shows an implementation of the spectral forecast equation. Here, the example is shown on two signals A and B, both represented by sequences of numbers. Thus, string literals representing number sequences of the signals, are assigned to variables A and B. Next, two empty array variables are declared, namely *tA* and *tB*. The string value assigned to *A* is then split based on a comma delimiter into new elements of array *tA*. The string value assigned to *B* is also split based on a comma delimiter (i.e. ‘,’), into new elements of array *tB*. Depending on the computer language used, the arrays that store the numbers as strings are either converted to float in computer languages like C#, Java, Python and Ruby, or, these values dictate the datatype of the variables without any sort of conversion, as is the case of Javascript, VB6/VBA, PHP and PERL. That is, if the string value represents a number, then the split function will automatically assign the value to the elements of an array as a float or double. Next, the maximum and minimum values are detected in both array variables *tA* and *tB*. This detection can be done in two ways. One approach is to use the “MAX” or “MIN” built-in functions that return the highest or lowest value from the array elements. In the case of *Spectral Forecast*, the maximum value among both number sequences is required. Thus, the maximum value found among the elements of *tA* will be assigned to variable *maxA*, and the maximum value among the elements of *tB* will be assigned to a variable *maxB*. In a second step, the “MAX” function will be called again, to return the highest value between *maxA* and *maxB*. When these built-in functions are not available, the maximum value among the elements of the two array variables *tA* and *tB* is computed by using a for-loop with conditional statements, where the maximum value above the elements is alwais assigned to a variable *max* (Additional algorithm 9.3).

Next, an integer variable *d* is declared and set to an arbitrary value, namely 33. In the last stage, a for-loop is declared from zero to an integer value that represents the number of elements of any of the two array variables (as they contain the same number of elements). Inside the for-loop, the result of the *Spectral Forecast* equation is assigned to a variable *v*. The content of variable *v* is added to an accumulator variable *M*. For convenience, a

Lang.	Example

JS

```
// Spectral forecast for signals in Javascript

var A ='10.3,23.4,44.8,63.2,44.1,35.1,46.5,62.6,50.4';
var B ='18.8,43.1,52.2,45.5,46.8,46.6,67.9,66.3,70.4';
var M ='';

var tA = [];
var tB = [];

var tA = A.split(',');
var tB = B.split(',');

var maxA = Math.max.apply(null, tA);
var maxB = Math.max.apply(null, tB);

var max = Math.max(maxA, maxB)

var d = 33;

for(var i=0; i < tA.length; i++) {
    var v = ((d/maxA)*tA[i])+(((max-d)/maxB)*tB[i]);
    M += v.toFixed(2);
    if(i < tA.length-1){M += ','}
}

print('Signal A:'+A);
```

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

```
print('Max(A[i]):'+maxA);

print('Signal M:'+M);

print('Signal B:'+B);
print('Max(B[i]):'+maxB);
```

C#

```
// Spectral forecast for signals in C#

using System;

public class SpectralForecast
{
    public static void Main(string[] args)
    {
        string A = "10.3,23.4,44.8,63.2,44.1,35.1,46.5,62.6,50.4";
        string B = "18.8,43.1,52.2,45.5,46.8,46.6,67.9,66.3,70.4";

        string[] nA = A.Split(',');
        string[] nB = B.Split(',');

        string M = "";
        double maxA = 0;
        double maxB = 0;
        double d = 33;
        double v = 0;

        double[] tA = new double[nA.Length];
        double[] tB = new double[nB.Length];

        for(int i=0; i < nA.Length; i++)
        {
            tA[i] = double.Parse(nA[i]);
            tB[i] = double.Parse(nB[i]);
            if (tA[i] > maxA){maxA = tA[i];}
            if (tB[i] > maxB){maxB = tB[i];}
        }

        double maxAB = Math.Max(maxA, maxB);

        for(int i=0; i < tA.Length; i++) {
            v = ((d/maxA)*tA[i])+(((maxAB-d)/maxB)*tB[i]);
            M += Math.Round(v, 2);
            if(i < tA.Length-1){M += ',';}
        }

        Console.WriteLine (M);
    }
}
```

Additional algorithm 9.3 (continued)

VB

```vb
' Spectral forecast for signals in VB

Sub main()

    Dim tA() As String
    Dim tB() As String

    A = "10.3,23.4,44.8,63.2,44.1,35.1,46.5,62.6,50.4"
    B = "18.8,43.1,52.2,45.5,46.8,46.6,67.9,66.3,70.4"
    M = ""

    tA = Split(A, ",")
    tB = Split(B, ",")

    For i = 0 To UBound(tA)
        If (tA(i) > maxA) Then maxA = tA(i)
        If (tB(i) > maxB) Then maxB = tB(i)
        If (maxA > Max) Then Max = maxA
        If (maxB > Max) Then Max = maxB
    Next i

    d = 33

    For i = 0 To UBound(tA)
        v = ((d / maxA) * tA(i)) + (((Max - d) / maxB) * tB(i))
        M = M & Round(v, 2)
        If (i < UBound(tA) - 1) Then M = M & ","
    Next i

    s = s & "Signal A: " & A & vbCrLf
    s = s & "Max(A[i]):" & maxA & vbCrLf

    s = s & "Signal M: " & M & vbCrLf

    s = s & "Signal B: " & B & vbCrLf
    s = s & "Max(B[i]):" & maxB & vbCrLf

    MsgBox s

End Sub
```

PHP

```php
// Spectral forecast for signals in PHP

$A ='10.3,23.4,44.8,63.2,44.1,35.1,46.5,62.6,50.4';
$B ='18.8,43.1,52.2,45.5,46.8,46.6,67.9,66.3,70.4';
$M ='';

$tA = [];
$tB = [];

$d = 33;

$tA = explode(",", $A);
```

Additional algorithm 9.3 (continued)

```
$tB = explode(",", $B);

$maxA = Max($tA);
$maxB = Max($tB);
$max = Max($maxA, $maxB);

for($i=0; $i < count($tA); $i++) {
    $v = (($d/$maxA)*$tA[$i])+(((($max-$d)/$maxB)*$tB[$i]);
    $M .= strval(round($v,2));
    if($i < count($tA)-1){$M .= ',';}
}

echo $M;
```

PERL

```
# Spectral forecast for signals in Perl

my $A = '10.3,23.4,44.8,63.2,44.1,35.1,46.5,62.6,50.4';
my $B = '18.8,43.1,52.2,45.5,46.8,46.6,67.9,66.3,70.4';
my $M = "";

my $d = 33;
my $v = 0;

my @tA = split(',', $A);
my @tB = split(',', $B);

use List::Util qw(min max);
my $maxA = max @tA;
my $maxB = max @tB;
my $maxAB = ($maxA, $maxB)[$maxA < $maxB];

for(my $i = 0; $i <= $#tA; $i++){
    $v = (($d/$maxA)*$tA[$i])+(((($maxAB-$d)/$maxB)*$tB[$i]);
    $M .= sprintf("%.2f", $v);
    if($i < $#tA){$M .= ',';}
}

print($M);
```

Ruby

```
# Spectral forecast for signals in Ruby

A = "10.3,23.4,44.8,63.2,44.1,35.1,46.5,62.6,50.4"
B = "18.8,43.1,52.2,45.5,46.8,46.6,67.9,66.3,70.4"
m = ""

tA = A.split(",")
tB = B.split(",")

maxA = tA.max.to_f
maxB = tB.max.to_f
maxAB = [maxA, maxB].max
```

Additional algorithm 9.3 (continued)

```
d = 33
v = 0

for i in (0...tA.length)
    tmpA = tA[i].to_f
    tmpB = tB[i].to_f
    v = ((d / maxA) * tmpA) + (((maxAB - d) / maxB) * tmpB)
    m += "#{v.round(2)}"
    if i < tA.length then m += "," end
end

puts "#{m}"
```

Java

```
// Spectral forecast for signals in Java

public class Main
{
    public static void main(String[] args) {

        String A = "10.3,23.4,44.8,63.2,44.1,35.1,46.5,62.6,50.4";
        String B = "18.8,43.1,52.2,45.5,46.8,46.6,67.9,66.3,70.4";
        String M = "";

        String[] tA = A.split(",");
        String[] tB = B.split(",");

        float tmpA;
        float tmpB;

        float v = 0;
        float d = 33;
        float maxA = 0;
        float maxB = 0;
        float maxAB = 0;

        for (int i = 0; i < tA.length; i++)
        {
            tmpA = Float.parseFloat(tA[i]);
            tmpB = Float.parseFloat(tB[i]);

            if (tmpA > maxA){maxA = tmpA;}
            if (tmpB > maxB){maxB = tmpB;}
            if (maxA > maxAB){maxAB = maxA;}
            if (maxB > maxAB){maxAB = maxB;}
        }

        for (int i=0; i < tA.length; i++) {

            tmpA = Float.parseFloat(tA[i]);
            tmpB = Float.parseFloat(tB[i]);
```

Additional algorithm 9.3 (continued)

```
            v = ((d/maxA)*tmpA)+(((maxAB-d)/maxB)*tmpB);
            M += Math.round(v*100)/100.0;
            if(i < tA.length-1){M += ",";}
        }

        System.out.println(M);
    }
}
```

Python

```
# Spectral forecast for signals in Python

A = "10.3,23.4,44.8,63.2,44.1,35.1,46.5,62.6,50.4"
B = "18.8,43.1,52.2,45.5,46.8,46.6,67.9,66.3,70.4"
M = ""

tA = list(map(float, A.split(",")))
tB = list(map(float, B.split(",")))

maxA = max(tA)
maxB = max(tB)
maxAB = max(maxA, maxB)

d = 33

for i in range(0, len(tA)):
    v = ((d / maxA) * tA[i]) + (((maxAB - d) / maxB) * tB[i])
    M += str(round(v,2))
    if i < (len(tA) - 1): M += ","

print(M)
```

C++

```
// Spectral forecast for signals in C++

#include <iostream>
#include <vector>
#include <sstream>
using namespace std;

void tokenize(string const &str,
const char l, vector<string> &o)
{
    stringstream g(str);
    string s;
    while (getline(g, s, l))
    {o.push_back(s);}
}

int main()
{
    string A = "10.3,23.4,44.8,63.2,44.1,35.1,46.5,62.6,50.4";
    string B = "18.8,43.1,52.2,45.5,46.8,46.6,67.9,66.3,70.4";
```

Additional algorithm 9.3 (continued)

```
    const char l = ',';

    vector<string> tA;
    vector<string> tB;

    tokenize(A, l, tA);
    tokenize(B, l, tB);

    string M = "";
    float maxA = 0;
    float maxB = 0;
    float maxAB = 0;
    float d = 33;
    float v = 0;

    int n = tA.size();

    for(int i=0; i < n; i++)
    {
        if (stof(tA[i]) > maxA){maxA = stof(tA[i]);}
        if (stof(tB[i]) > maxB){maxB = stof(tB[i]);}
        if (maxA > maxB){maxAB = maxA;} else {maxAB = maxB;}
    }

    for(int i=0; i < n; i++)
    {
        v = ((d/maxA)*stof(tA[i]))+(((maxAB-d)/maxB)*stof(tB[i]));
        M += to_string(v).substr(0,5);
        if(i < n-1){M += ',';}
    }

    cout<<M;

    return 0;
}
```

Additional algorithm 9.3 (continued)

conditional statement checks if variable i had reached the maximum number of elements of the array tA. If the condition is false, a delimiter is added to in between the values that are accumulated inside the string variable M. In contrast, if the condition is true then a delimiter is not added at the end of the string accumulated in M. Thus, the purpose of the condition is to avoid the addition of a delimiter character at the end of the result stored in M. Finally, once the iterations end, the three main strings representing the signals, are printed in the output for inspection. For the examples given in Additional algorithm 9.3, the result stored in variable M is:

"15.37, 35.12, 51.12, 57.17, 47.89, 43.08, 60.35, 67.91, 63.72"

This example demonstrates the use of string values as number sequences, both for the input and for the output. Another version that can be implemented is the use of array variables both for the two signals (*A* and *B*) from the input but also for the signal from the output (*M*). Since such an implementation can store numeric values directly, the implementation can be shorter and much easier to understand. However, it will not be presented here, but it is left as an exercise for the reader. Note that the numeric data is usually stored as string sequences in practice, like for instance everything related to data science is usually stored in text files (ex. ".CSV", ".TXT" and so on). Therefore, the example shown in Additional algorithm 9.3 may be more useful than the array version of it.

9.5 Conclusions

The first part of the chapter has shown examples of recursive functions. For example, the following recursive methods were discussed: A recursive function for repeating a character *n* times to construct a string of a given length, or a function that sums integers from zero to *n*, or a function that preforms the factorial for an integer *n*. Furthermore, other more complex examples have included a function that generated a sequence of numbers based on various rules, or another recursive function that provided the Fibonacci sequence. Finally, a recursive function that was able to sum all the integers stored in the elements of an array variable. Next, the use of a scanner was discussed in the context of a range of integers, for which a distribution was calculated using a mathematical expression. Towards the end of the chapter, a new method called Spectral Forecast was described. The implementation of the *Spectral Forecast* equation used sequences of numbers as string values to show the possibility of using numerical data found in text-based files. This final chapter concludes the examples made in our set of computer languages, namely: C++, C#, Java, Javascript, Python, PHP, PERL, Ruby and VB.

References

1. P.A. Gagniuc, *Algorithms in Bioinformatics: Theory and Implementation, Hoboken* (Wiley & Sons, USA, New Jersey, 2021)
2. N. Lahav, S. Nir, A.C. Elitzur, The emergence of life on earth. Prog. Biophys. Mol. Biol. **75**(1–2), 75–120 (2001)
3. H.J.C. Ii, Prebiotic chemistry: what we know, what we don't. Evol. Educ. Outreach **5**(1), 342–360 (2012)
4. K.A. Dill, L. Agozzino, Driving forces in the origins of life. Open Biol. **11**(1), 200324 (2021)
5. R. Etxepare, A. Irurtzun, Gravettian hand stencils as sign language formatives. Philos. Trans. R. Soc. Lond B Biol. Sci. **376**(1824), 20200205 (2021)
6. F. Facchin, Symbolism in prehistoric man. Coll. Antropol. **24**(2), 541–553 (2000)
7. M. Daly, M.I. Wilson, Human evolutionary psychology and animal behavior. Anim. Behav. **57**(3), 509–519 (1999)
8. C. Everett, The sounds of prehistoric speech. Phil. Trans. R. Soc. B **376**(1), 20200195 (2021)
9. I. Cross, E.C. Blake, The acoustic and auditory contexts of human behavior. Curr. Anthropol. **56**(1), 81–103 (2015)
10. L. Progovac, A. Benítez-Burraco, Reconstructing prehistoric languages. Phil. Trans. R. Soc. B **376**(1), 20200187 (2021)
11. A. Salah, B. Lepri, F. Pianesi, A. Pentland, Human behavior understanding for inducing behavioral change: application perspectives, in *Human Behavior Understanding. HBU 2011. Lecture Notes in Computer Science*, vol. 7065 ed. by A.A. Salah, B. Lepri (Springer, Berlin, 2011)
12. J.P. Nelson, Mythic forecasts: researcher portrayals of extraterrestrial life discovery. Int. J. Astrobiol. **19**(1), 16–24 (2020)
13. C. Cuskley, Alien forms for alien language: investigating novel form spaces in cultural evolution. Palgrave Commun. **5**(1), 87 (2019)
14. T. Koetsier, On the prehistory of programmable machines: musical automata, looms, calculators. Mech. Mach. Theory **5**(36), 589–603 (2001)
15. S. Olson, Mythical androids and ancient automatons. Science **362**(6410), 39 (2018)
16. R.C. Pooley, Automatons or english teachers? Engl. J. **50**(3), 168–173 (1961)

P. A. Gagniuc, *An Introduction to Programming Languages: Simultaneous Learning in Multiple Coding Environments*, Synthesis Lectures on Computer Science,
https://doi.org/10.1007/978-3-031-23277-0

17. L. Albaugh, J. McCann, L. Yao, S.E. Hudson, Enabling personal computational handweaving with a low-cost jacquard loom, in *CHI Conference on Human Factors in Computing Systems (CHI'21), May 8–13, 2021, Yokohama, Japan* (ACM, New York, 2021)
18. H.W. Nelson, *Jacquard Machines: Instruction Paper*. (American School of Correspondence, Chicago, Illinois, USA, 1869)
19. D.M. Fryer, J.C. Marshall, The motives of Jacques de Vaucanson. Technol. Cult. **2**(20), 257 (1979)
20. T.F. Bell, *Jacquard Weaving and Designing*. (Longmans, Green, New York, USA, 1895), pp.1–371
21. F.G. Heath, Origins of the binary code. Sci. Am. **227**(2), 76–83 (1972)
22. A.G. Bromley, Charles Babbage's analytical engine, 1838. Ann. Hist. Comput. **4**(3), 196–217 (1982)
23. R. Rojas, Konrad Zuse's legacy: the architecture of the Z1 and Z3. IEEE Ann. Hist. Comput. **19**(2), 5–16 (1997)
24. R.C. Lyndon, The Zuse computer. Math. Tables Other Aids Comput. **2**(20), 354–359 (1947)
25. J. Gilbey, Biography: the ABC of computing. Nature **468**(1), 760–761 (2010)
26. H.H. Goldstine, A. Goldstine, The electronic numerical integrator and computer (ENIAC). Math. Tables Other Aids Comput. **2**(15), 97–110 (1946)
27. J.A. Fleming, Thermionic valves. Sci. Mon. **20**(5), 530–534 (1925)
28. C.D. Martin, ENIAC: press conference that shook the world. IEEE Technol. Soc. Mag. **14**(1), 3–10 (1995)
29. W. Shockley, The theory of *p-n* junctions in semiconductors and *p-n* junction transistors. Bell Syst. Tech. J. Inst. Electr. Electron. Eng. (IEEE) **28**(3), 435–489 (1949)
30. W. Shockley, G.L. Pearson, J.R. Haynes, Hole injection in germanium-quantitative studies and filamentary transistors. Bell Syst. Tech. J. Inst. Electr. Electron. Eng. (IEEE) **28**(2), 344–366 (1949)
31. D.G. Fink, Transistors versus vacuum tubes. Proc. IRE **44**(4), 479–482 (1956)
32. D.P. Anderson, Biographies: Tom Kilburn: a pioneer of computer design. IEEE Ann. Hist. Comput. **31**(2), 82–86 (2009)
33. M.M. Irvine, Early digital computers at bell telephone laboratories. IEEE Ann. Hist. Comput. **23**(3), 22–42 (2001)
34. E.G. Andrews, The Bell computer, model VI. Electr. Eng. **68**(9), 751–756 (1949)
35. A.N. Saxena, Monolithic concept and the inventions of integrated circuits by Kilby and Noyce, in *Nano Science and Technology Institute Annual Conference, Santa Clara Convention Center, California, USA, May 20–24*, 2007
36. B. Hevly, T.R. Reid, The chip: how two Americans invented the microchip and launched a revolution. Technol. Cult. **27**(4), 873 (1986)
37. R.W. Bemer, A proposal for character code compatibility. Commun. ACM **3**(2), 71–72 (1960)
38. G.S. Robinson, C. Cargill, History and impact of computer standards. Computer **29**(10), 79–85 (1996)
39. K. MacDonald, F. Scherjon, E. van Veen, W. Roebroeks, Middle pleistocene fire use: the first signal of widespread cultural diffusion in human evolution. PNAS **118**(31), e2101108118 (2021)
40. A. Molcosean, The dimensions of the personality of stephen the great reflected in history textbooks in the Republic of Moldova (1990–2013). Ann. Putna **XVI**(1), 247–274 (2020)
41. L. Cotovanu, An epitrachelion with the portrait of Stephen the great discovered in the collection of the Byzantine and Christian Museum from Athens. Ann. Putna **XV**(1), 135–148 (2019)

42. O. Cristea, For a critical edition of Stephen the Great's letter to the Christendom. Remarks on the list of Ottoman commanders. Ann. Putna **XV**(1), 167–188 (2019)
43. C.H. Sterling, *Military Communications: From Ancient Times to the 21st Century*, Illustrated edn. (Santa Barbara, California, ABC-CLIO, 2007)
44. L. Cîmpeanu, Considerations on the artillery of voivode Stephen the great. Ann. Putna **XIV**(1), 321–334 (2018)
45. A.A. Lumsdaine, Mass communication experiments in wartime and thereafter. Soc. Psychol. Q. **47**(2), 198–206 (1984)
46. J.D. Peters, *Speaking into the Air: A History of the Idea of Communication* (University of Chicago Press, USA, 1999)
47. H. Nyquist, Certain factors affecting telegraph speed. Trans. Am. Inst. Electr. Eng. **XLIII**(1), 412–422 (1924)
48. P.N. Das, Telegraph codes and their uses. IETE J. Educ. **1**(1), 7–14 (1959)
49. B. Gold, Machine recognition of hand-sent morse code. IRE Trans. Inf. Theory **5**(1), 17–24 (1959)
50. D.M. Clawson, A.F. Healy, K.A. Ericsson, L.E. Bourne Jr., Retention and transfer of morse code reception skill by novices: part-whole training. J. Exp. Psychol. Appl. **7**(2), 129–142
51. I.E. Sutherland, Sketchpad: a man-machine graphical communication system. Technical Report No. 574 (University of Cambridge Computer Laboratory, 2003)
52. I.E. Sutherland, SketchPad: a man-machine graphical communication system. AFIPS Conf. Proc. **23**(1), 323–328 (1963)
53. J.E. Timothy, Sketchpad III, three dimensional graphical communication with a digital computer. Massachusetts Institute of Technology, Department of Mechanical Engineering, 1963
54. D.C. Engelbart, Augmenting human intellect: a conceptual framework. SRI Summary report AFOSR-3223, prepared for: Director of Information Sciences, Air Force Office of Scientific Research, Washington 25, DC. Contract AF 49(638)-1024. SRI Project No. 3578 (AUGMENT, 3906), 1962
55. V.G. Cerf, Augmented intelligence. IEEE Internet Comput. **17**(5), 96–96 (2013)
56. P. Atkinson, The best laid plans of mice and men: the computer mouse in the history of computing. Des. Issues **23**(3), 46–61 (2007)
57. C.P. Thacker, E.M. McCreight, B.W. Lampson, R.F. Sproull, D.R. Boggs, *Alto: A Personal Computer*. (Xerox Corporation, Palo Alto, California, 1979)
58. A.C. Kay, The early history of smalltalk. ACM SIGPLAN Not. **28**(3), 69–95 (1993)
59. A. Goldberg, A. Kay, *Smalltalk-72 Instruction Manual*. (The Learning Research Group, Xerox Palo Alto Hesearch Center, Palo Alto, California, 1976)
60. D.C. Smith, C. Irby, R. Kimball, E. Harslem, The Star user interface: an overview, in *AFIPS'82: Proceedings of the June 7–10, 1982, National Computer Conference, New York, NY, United States* (Association for Computing Machinery, 1982), pp. 515–528
61. W.L. Bewley, T.L. Roberts, D. Schroit, W.L. Verplank, Human factors testing in the design of Xerox's 8010 "Star" office workstation, in *CHI'83: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, USA, 1983)
62. L.W. Friedman, From Babbage to Babel and beyond: a brief history of programming languages. Comput. Lang. **17**(1), 1–17 (1992)
63. T.J. Bergin, A history of the history of programming languages. Commun. ACM **50**(5), 69–74 (2007)
64. N. Stern, HOPL: history of programming languages conference. Ann. Hist. Comput. **1**(1), 68–71 (1979)
65. D. Knuth, L.T. Pardo, *The Early Development of Programming Languages*. (Stanford University, Department Of Computer Science, ADA032123, Palo Alto, California, 1976)

66. W. Giloi, Konrad Zuse's Plankalkul: the first high-level, non von Neumann programming language. IEEE Ann. Hist. Comput. **19**(2), 17–24 (1997)
67. R.A. Brooker, D. Morris, An assembly program for a phrase structure language. Comput. J. **3**(3), 168–174 (1960)
68. A.D. Booth, K.H.V. Britten, *Coding for ARC*. (The Institute for Advanced Study, Princeton, New Jersey, USA, 1947)
69. A.D. Booth, K.H.V. Britten, Principles and progress in the construction of high-speed digital computers. Q. J. Mech. Appl. Math. **2**(2), 182–197 (1949)
70. A. Booth, K. Britten, *General Considerations in the Design of an All Purpose Electronic Digital Computer*, 2nd edn. (Unpublished Report, Birkbeck College, London, 1947)
71. B. Clarke, G.E. Felton, The computer journal. Pegasus Autocode **1**(4), 192–195 (1959)
72. R.A. Brooker, Further autocode facilities for the Manchester (Mercury) computer. Comput. J. **1**(3), 124–127 (1958)
73. J. Backus, The history of FORTRAN I, II and III. Ann. Hist. Comput. **1**(1), 21–37 (1979)
74. J.W. Backus, F.L. Bauer, J. Green, C. Katz, J. McCarthy, A.J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J.H. Wegstein, A. van Wijngaarden, M. Woodger, Report on the algorithmic language ALGOL 60. Numer. Math. **2**(1), 106–136 (1960)
75. K Zuse, Über den Plankalkül. Inf. Technol. **1**(1–4), 68–71 (1959)
76. K. Zuse, Über den Allgemeinen Plankalkül als Mittel zur Formulierung schematisch-kombinativer Aufgaben. Arch. Math **1**(1), 441–449 (1948)
77. R.M. Paine, Automatic coding for business applications. Comput. J. **3**(3), 144–149 (1960)
78. L.A. Lombardi, B. Raphael, *LISP as the Language for An Incremental Computer*. (Massachusetts Institute of Technology, Cambridge, Massachusetts, 1964)
79. T. Dwyer, C. Len, E. Zielinski, V. Salko, M. Critchfield, M. Seaton, *A Primer for the NEWBASIC/CATALYST System.* (University of Pittsburgh, Pittsburgh, Pennsylvania, 1970)
80. N. Wirth, The programming language pascal. Acta Inform. **1**(1), 35–63 (1971)
81. D.M. Ritchie, The development of the C language, in *Second History of Programming Languages Conference, Cambridge, Massachusetts* (Association for Computing Machinery, 1993)
82. D. Chamberlin, Early history of SQL. IEEE Ann. Hist. Comput. **34**(4), 78–82 (2012)
83. C. Moler, in *AFIPS'80: Proceedings of the May 19–22, 1980, National Computer Conference, Design of An Interactive Matrix Calculator, New York, NY, United States* (Association for Computing Machinery, 1980), pp. 363–368
84. B. Stroustrup, The C++ programming language: reference manual: computing science. Technical Report, No. 108 (AT&T Bell Laboratories, New Jersey, 1984)
85. B.J. Cox, *Object Oriented Programming: An Evolutionary Approach.* (Addison-Wesley Longman Publishing Co., Inc., Boston, USA, 1986)
86. L. Wall, T. Christiansen, J. Orwant, *Programming Perl*, Third edn. (O'Reilly Media, Sebastopol, California, 2000)
87. C. Pratchett, M. Wright, CGI/Perl Cookbook. Assem. Autom. **19**(1), 78–78 (1999)
88. C.V. Hall, K. Hammond, S.L.P. Jones, P.L. Wadler, Type classes in Haskell. ACM Trans. Program. Lang. Syst. **18**(2), 109–138 (1996)
89. J. Launchbury, S.L.P. Jones, State in Haskell. LISP Symb. Comput. **8**(1), 293–341 (1995)
90. C. Bishop-Clark, Comparing understanding of programming design concepts using visual basic and traditional basic. J. Educ. Comput. Res. **18**(1), 37–47 (1998)
91. E. Coburn, *Visual Basic Made Easy*. (PWS Publishing Company, Boston, Massachusetts, 1995)
92. D. Schneider, *An Introduction to Programming using Visual Basic*. (Prentice Hall, Englewood Cliffs, New Jersey, 1995)
93. D. Zak, *Programming with Microsoft Visual Basic 2.0/3.0 for Windows*. (Course Technology, Cambridge, Massachusetts, 1995)

94. P. Dubois, T.-Y. Yang, Extending python with fortran. Comput. Sci. Eng. **1**(5), 66–73 (1999)
95. R. Ihaka, R. Gentleman, R: a language for data analysis and graphics. J. Comput. Graph. Stat. **5**(3), 299–314 (1996)
96. S. Hadjerrouit, Java as first programming language: a critical evaluation. ACM SIGCSE Bull. **30**(2), 43–47 (1998)
97. A.V. Royappa, The PHP web application server. J. Comput. Sci. Coll. **15**(3), 201–211 (2000)
98. B. Baas, Ruby in the CS curriculum. J. Comput. Sci. Coll. **17**(5), 95–103 (2002)
99. W. Buchanan, JavaScript, in *Mastering the Internet* (London, Palgrave, 1997), p. 155–178
100. D.S. Hennen, S. Ramachandran, S.A. Mamrak, The Object-JavaScript language. Softw. Pract. Exp. **30**(14), 1571–1585 (2000)
101. B.W. Benson Jr., JavaScript. ACM SIGPLAN Not. **34**(4), 25–27 (1999)
102. E.O. Cerqueira, R.J. Poppi, Using dynamic data exchange to exchange information between Visual Basic and Matlab: application to a diode array spectrophotomete. TrAC Trends Anal. Chem. **15**(10), 500–503 (1996)
103. D.I. Schneider, *An Introduction to Programming Using Visual Basic 6.0.* (Prentice Hall, Upper Saddle River, New Jersey, USA, 1999)
104. P. McKenna, N. Seeve-McKenna, Hyperlanguage: to boldly go…? Lang. Learn. J. **6**(1), 71–72 (1992)
105. F.P. Brooks, *The Mythical Man-Month : Essays on Software Engineering.* (Addison-Wesley Pub. Co, 1975)
106. M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, *Refactoring: improving the design of existing code* (Addison-Wesley, Boston, 1999)
107. M. de Jonge, Developing product lines with third-party components. Electron. Notes Theor. Comput. Sci. **238**(5), 63–80 (2009)
108. B. Klatt, Z. Durdik, H. Koziolek, K. Krogmann, J. Stammel, R. Weiss, Identify impacts of evolving third party components on long-living software systems, in *16th European Conference on Software Maintenance and Reengineering* (Szeged, Hungary, 2012), pp. 461–464
109. T. Remencius, A. Sillitti, G. Succi, Assessment of software developed by a third-party: a case study and comparison. Inf. Sci. 328(1), 237–249 (2016)
110. W.T. Councill, Third-party testing and the quality of software components. IEEE Softw. **16**(4), 55–57 (1999)
111. L.L. Larson, Third party software. Intern. Audit. **52**(2), 44–48 (1995)
112. D. Cotroneo, R. Natella, R. Pietrantuono, S. Russo, A survey of software aging and rejuvenation studies. ACM J. Emerg. Technol. Comput. Syst. (JETC) **10**(1), 1–34 (2014)
113. M. Jakobsson, M.K. Reiter, Discouraging software piracy using software aging, in *ACM Workshop on Digital Rights Management* (Springer, Berlin, 2001), pp. 1–12
114. A. Bovenzi, D. Cotroneo, R. Pietrantuono, S. Russo, On the aging effects due to concurrency bugs: a case study on MySQL, in *23rd International Symposium on Software Reliability Engineering* (IEEE, Dallas, Texas, 2012), pp. 211–220
115. G. Booch, *Object Oriented Design with Applications.* (Benjamin/Cummings Publishing Company, 1991)
116. P.M. Duvall, *Continuous Integration: Improving Software Quality and Reducing Risk.* (Addison-Wesley, 2007)
117. B. Lennon, Program text, programming style, programmer labor: some further comments on comments. Cult. Polit. 14(3), 372–394 (2018)
118. P. Gagniuc, C. Ionescu-Tirgoviste, Gene promoters show chromosome-specificity and reveal chromosome territories in humans. BMC Genomics **14**, 278 (2013)
119. P.A. Gagniuc, Markov chains: from theory to implementation and experimentation. (Wiley & Sons, Hoboken, NJ, USA, 2017)

120. P.A. Gagniuc, C. Ionescu-Tirgoviste, E. Gagniuc, M. Militaru, L.C. Nwabudike, B.I. Pavaloiu, A. Vasilăţeanu, N. Goga, G. Drăgoi, I. Popescu, S. Dima, Spectral forecast: a general purpose prediction model as an alternative to classical neural networks. Chaos **30**, 033119–033126 (2020)
121. A.E. Osbourn, B. Field, Operons. Cell. Mol. Life Sci. **66**(23), 3755–3775 (2009)
122. X. Yang, J. Coulombe-Huntington, S. Kang, G.M. Sheynkman, T. Hao, A. Richardson, S. Sun, F. Yang, Y.A. Shen, R.R. Murray, K. Spirohn et al., Widespread expansion of protein interaction capabilities by alternative splicing. Cell **164**(4), 805–817 (2016)
123. P. Gagniuc, C. Ionescu-Tirgoviste, Eukaryotic genomes may exhibit up to 10 generic classes of gene promoters. BMC Genomics **13**, 512 (2012)
124. P. Gagniuc et al., A sensitive method for detecting dinucleotide islands and clusters through depth analysis. Rom. J. Diabetes Nutr. Metab. Dis. 18(2), 165–70 (2011)