Programming a computer is a simple as mastering the A, B, Cs. Once one has learnt the primary Abstractions, Basics, and Concepts of modern programming, he can master any scholarly subject based upon this skill.

- Art

Programming Fundamentals

Let Us Talk

Arturo Koruna De Laine

Acknowledgements

Many thanks are due. Firstly, the author, a man who is a Protestant Christian Theist and who accepts that all of His children, regardless of their personal faith walk, are precious in His Eyes, thanks El-Olam for his Everlasting Grace, Love, and Mercy. The Joy and Peace which this life has shown in knowing Him is beyond expressing in a trillion tongues over the course of millions of millennia. Secondly, the author thanks the great instructors which he has had over the years while attending primary and secondary school among the students of a fully integrated school district in the Heartland of the United States. It was while riding the school bus from his light-blue collar suburban neighborhood, where he was one of the few minorities, that he met a “dear” friend, a member of the social majority, who indirectly inspired some of his uncompensated his work with Sun Microsystems in the early-1990s. She also ultimately was the reason that he attended Vanderbilt during the 1988-1989 academic year, a school experience that was rife with tragedy and triumph. He thanks her for many “happy” moments and a chuckle that still resonates in his ears. In that his personal social network birthed some of his “novel” visions in computing for which others received credit, he thanks this world’s engineers who sculpted products like “universal” virtual machine language platforms and “simpler” type-setting systems in the form of modern markup languages. In particular, the author thanks “extended family” who was in executive leadership at Sun and the likes of Bill Joy, James Gosling, and the professional artist who took his “crude” Duke caricature, a tear-drop doodle, that looked more like a “chicken” and made it what it is this day. In many ways, you have made this life a “Josephine” dream. And, he does not seek any credit or accolades, seeing that others took very simple ideas and crafted truly complex marvels of modern computing. Plus, his ego does not require it.

Copyright & License

Copyright

License

Prologue

“Truth is stranger than fiction, but it is because Fiction is obliged to stick to possibilities; Truth isn't.”

— Mark Twain, Following the Equator: A Journey Around the World

It is the case that sand tablets were a means of reckoning among ancient mathematicians. These men would write their figures and expressions upon “etch-a-sketch”-like devices. And, erase them as needed.

Computing devices advanced over the years in various cultures. Some of these included the use of an abacus. Other early devices for computation also were in use which pre-date the harnessing of electricity. In fact, the game Mancala is based upon a “computation” that East African herdsman used for monitoring their livestock based upon the natural grazing patterns of the cattle, goats, or sheep. As such, a Mancala board is a “special-purpose” computing device much like a Sun Microsystems or Silicon Graphics workstation from the early-1990s. Another category of machine were the mechanical timepieces which arose. Others were the programmable looms for producing Jacquard fabric. And, finally the famous mechanical calculating tools designed by Charles Babbage and theoretically programmed by Ada de Lovelace in the 1800s.

The term “computer” historically has meant any person who “computes”. Before the age of electronic computing devices, men who were mathematicians would be employed for the purpose of producing tables of mathematical data, such as a list of the first ten thousand prime numbers, the sines, cosines, and tangents for a range of angles measurements, or the logarithms for the first thousand whole numbers in a given base. So, humans were and still are “computers”. And, the artificial tools we use these days and that we commonly call “computers” simply aid us in the computations which we do.

It is the case that the first working electronic computer was built at Iowa State University during the Second World War. It is called the Atanasoff-Berry Computer and was a special-purpose machine that could perform only certain mathematical function. Yet, this ABC machine was quite an advance. The next major advances in computing came with the development of the transistor in the 1950s and semiconductors a decade or so later. These advances let mankind build smaller computing devices.

Yet, we must communicate with the tools we use. Otherwise, they will not know our desires or the tasks that we must accomplish or how they might help us realize them. There are numerous language generations and types. The first generation of languages was each computer platforms machine language which is described using zeroes and ones. These were not easily read or remembered by humans. Then, a novel innovation, assembly language arose. These languages associated a simple mnemonic like ADD, SUB, MULT, DIV, STO, and BNZ with these strings of zeroes and ones. Which were commands. Those mnemonic stand for add, subtract, multiply, divide, store, and branch not zero. The third generation of languages used more “human-readable” commands and statements like “readln”, “println”, and “if”. Its fourth generation would read text-based specifications which were like blueprints and automatically generated third generation programs. This generation was somewhat short-lived but is still present this day. The fifth generation was birthed at Vanderbilt University on the floor of a super single dorm room in its Quadrangle during the Spring of 1989. Yet, this origination story is not consistent with the “widely-known” and accepted “official” story. This generation uses virtual machines platforms and an object-oriented concerns partitioning approach. Concerns partitioning is a fancy form of problem-

solving and object-orientation is an modelling approach and a reasoning technique for doing such. What the future holds for sixth, seventh, and eighth generations is unforeseen.

All computer languages involve issuing a computer commands. As such, many of these languages are called command-based or "imperative" languages. An imperative is a fancy language arts term for a command. They are synonymous terms. An imperative sentence in English would be "Bring me my favorite book from the library." In this sentence, the subject "You" is understood. In other words, it is "You, bring me my favorite book from the library." When we write computer programs, we are issuing commands which the computer must follow. For instance, "print('Hello, World!')" should be interpreted and read as "Computer, please print the phrase 'Hello, World!' on the terminal." And, ultimately all languages issue commands. But, the nature of how one describes those commands when working with a computer varies. Some languages use descriptions of a program's logic called schemas such as Prolog. These are known as logic programming languages. Others simply provide a declaration of what the program's result should be. These are known as declarative languages. And, hypertext markup language, if you are familiar with It, has some declarative features. And, other less common language types exist. In this short text, we will explore how one might program a computer conceptually.

Abecedarianism

It's... supercalifragilisticexpialidocious!
Even though the sound of it is something quite atrocious
If you say it loud enough, you'll always sound precocious:
Supercalifragilisticexpialidocious!

- **Mary Poppins**

"A" is for abstractions. Can we all say "abstractions"? Its root term is "abstract". With "abstraction" comes "intellectual power". A certain insurmountable strength exists in simplicity. Also does manifold complexity. Consider the work of great artists: paintings, tapestries, sculpted works, statues, poems, novels, psalms, and the rest. Often many intricate layers of symbols, meanings, and moral lessons exist in these timeless treasures. Yet, one will tell you that even the most exacting and detail-oriented art forms such as pointillism are fundamentally abstract.

Abstraction is the skill of seeing the "big" picture and commonalities among apparent disparate, dissimilar, and uniquely different principles. The "greatest" scientific concepts and axioms (rules) are pure abstractions. For instance,

- For every action, an equal and opposite action exists. (a fundamental law of physics)
- Opposite charges attract. Like ones repel. (a fundamental law of electromagnetism)
- If one can measure the size of a "grain of sand" and then count the number of such grains on the seashore, he could know the "exact" size of it. (the "ancient" observation that produced calculus by Newton and Leibnitz)
- All objects are composed of properties and behaviors. (a fundamental axiom of modern object theory)

Notice how the fundamental axiom of modern object theory finds commonality among all objects which exist whether, physical or logical.

Among the traditional trade of construction engineering, architects are the ones who shape the overall design of the building. They draft the "big" picture. They do not specify the exact position of electrical outlets or plumbing fixtures. This detail is provided by the designer who describes subunits of the building in a greater level of detail. Since the advent of the discipline of software architecture and its first formal international conference in 1989, software developers have been studying "older" more "mature" forms of engineering looking for insight on how we should structure a software creation process. And, as a computer programmer, one must work both at an abstract and detailed level. In fact, a principle development style known as top-down design underlies the software creation process. With this design approach, one crafts an abstraction that describes the entire system. Then, he decomposes this one idea producing many sub-ideas. Each sub-idea has a greater level of detail and might be composed of further sub-ideas. This iterative (repeated) process produces

an idea hierarchy with the most general ones near the top and the most specific and concrete ones forming the base of the structure.

So, “abstraction” is simple in that it represents “universal” ideas. Yet, it is “complex”, since each one holds many more “concrete” and “specific” ideas. An abstraction is much like the image produced by a kaleidoscope that one would simply describe a “pure beauty”. Well, what is that? It is the myriad mixture of colors and hues plus shapes and forms cast by the kaleidoscope’s screen when light illuminates it. Each vibrant portion, sub-portion, shape, and sub-shape cast from the screen are the varying levels of individual “concretions” forming the abstraction, “pure beauty”. Each of these concrete forms are “simple” in that they are “smaller”, “singular” parts of the larger picture, their arrangement, and relative positioning. Although it follows a “regular” pattern, it is “seemingly” complex. So, it can be said that the relationship between abstraction and concretion is “simply-complex” and “complexly-simple”.

“B” is for basics. Can everyone say, “basics”? The root term for “basic” is “base” whose synonyms are “foundation” or “fundamental”. A house must be built upon a “base” or “foundation” with a cornerstone. The cornerstone for reasoning about any academic topic is “abstraction”. This supports the “fundamental” building blocks which form the “foundation” of our “house of learning”. Many beautiful and awe-inspiring works in art, music, and literature are based upon the notion of “fundamentals”. The artisan’s color wheel is formed from three primitive colors: red, yellow, and blue; from which we derive our set of secondary composite colors: orange, green, and violet. And, in the Hand of the Maker, these produce the Sign of a Promise which is drawn as a spectrum of red, orange, yellow, green, blue, indigo, and violet in the sky after the worst of rainstorms. The greatest of symphonies is the artistic composition and arrangement of fundamental chords. Basics are the building blocks of “pure and true” beauty. They help us “keep life simple” whether drifting on the clouds of abstractions or mired in the morass of many concretions. We can navigate each well when we focus upon the “basics”.

The observation that “all objects are the composition of zero or more properties and behaviors” is such an exercise in “basic” reasoning. It is a very basic, simple, and seemingly obvious idea which implies that the abstractions commonly known as a property and a behavior are both “primitive”. It should be mentioned that this word is “primitive” and not “primative”, although the terms are related in more ways than they sound. Properties and behaviors are “primitive” abstractions which are fundamental building blocks for objects, much like the prime numbers, those only divisible by one and itself, are the building blocks for composite numbers. For instance, the composite five hundred and thirteen is built from the prime numbers three and nineteen. A composite is something which is “composed” from smaller more fundamental units. And, as stated earlier, the product development process known as “top-down” decomposition determines the individual units which compose each “top-level” abstraction in varying levels of detail.

When programming, at times we work with the high-level abstractions; at others, we work with the low-level highly concrete notions which approach the primitive ideas that form the basis for the higher tiers of reasoning. At this juncture, one might say, “it seems that we

are building a structure with a “foundation” on both the top and bottom. And, this is quite “true”, although seemingly paradoxical. As said, the A, B, C’s of reasoning are an exercise in simple complexity and complex simplicity. A successful reasoner must simultaneously hold notions in his mind which seemingly conflict when discussing topics in abstract terms. This is a “limitation” of natural language. The conflicts resolve themselves with the introduction of a greater level of detail and specification. Which comes when one provides a more concrete description of the idea. Which alleviates this “pseudo” cognitive dissonance. This psychological condition of dissonance arises when one holds conflicting ideas in his mind.

“C” is for concepts. Let us all say “concepts”. This term is synonymous with idea, notion, or thought. A singular concept is much like the whole number one. For those familiar with mathematical reasoning, it represents an “identity” of sorts and certain “idempotence”. When combined with itself under certain operations, it remains the same; under others, it starts aggregating and produces a greater quantity. Many concepts exist in computing and other academic disciplines. Some of these are abstract; others are concrete. Yet, all are seemingly “basic” once mastered. So, it could be said that the “C” representing concepts also is synonymous with “common-sense”.

Concepts are powerful, seeing that they are fundamentally beliefs and thoughts. What one believes and thinks ultimately shapes who he is? It was said by the occidental philosopher and mathematician, Rene Descartes, who is credited with producing our modern Cartesian graph, “I think therefore I am”. So, thought is a fundamental of existence. The Christian author, James Allen, wrote the text As a Man Thinketh. This was inspired by the Hebrew proverb 23:7, “As a man thinketh, so is he [in his heart].” Lao Tzu, a oriental philosopher, is credited with first stating, “Watch your thoughts, they become your words; watch your words, they become your actions; watch your actions, they become your habits; watch your habits, they become your character; watch your character, it becomes your destiny.” So, if one desires the profession of an software engineer or a computer scientist, he must first think like one. A famous and free text on programming the popular modern “high-level” computer language Python by Allen B. Downey is called Python : Think Like a Computer Scientist.

So, the ideas, notions, and thoughts which permeate one’s mind form the basis of concepts which shape a person’s future.

Input -Process -Output

A computer program, in essence, is a three-step sequence: input, process, and output. Basically, every program follows this pattern. So, the first step in writing a program is identifying any input "sources". These sources might be a command-line or graphical user interface on the computer's console, a file, a folder, or a subfolder within its filesystem's hierarchy, a table or tables in a local or remote database, a primary (server) process found on a network such as a web server, or any other repository of data. After identifying that source of input, one must request data from it. This also is known as the process of extraction in a extract, transform, and load (ETL) program. Once one obtains the input data, he must modify it, possibly combining it in various ways with other input and internal data. This is known as the process "phase" and the transform step in a ETL program. It relies upon the use of "pre-defined" and "user-defined" subprograms that alter the input and instructions that control the flow of the process step. The goal of this step is taking the input and preparing it for output. Once the input is "ready", it is placed on an output channel and received by the output "sink". This output step is called the "load" step when working in an data-focused ETL environment.

This intermingling of the terms input, process, and output with extract, transform, and load is not so the reader becomes confused or overwhelmed. It is so the author might show the level of commonality between areas of computing when one thinks abstractly. Yes, this is a technical discipline; however, one should not argue over technicalities. And, we should always seek common threads of reasoning when discussing any subject. If one applies for an engineering role, he mostly likely will not be asked if he understands the paradigm of input, process, and output. Yet. he might be asked if he has an understanding of ETL programming. In which case, he confidently can say, "The sequence of steps found in ETL programs: extract, transform, and load; parallel those found in every computer program. These are input, process, and output. So, if one can write a standard computer program, he has transferrable skills which he can use in an data-centric ETL environment." You will not find this "answer" in the text Ace the Technical Interview by Michael F. Rothstein. Yet, it is spot on. So, understanding this simple sequence of input, process, and output (IPO) opens up this field of endeavor, "computing".

Throughout the 1970s and early-1980s, the work done by the modern software engineer was known as "data processing". Data was processed as it was received by programs and then shipped out the door in a new form. It is said that computers accept "raw" data and process it, ultimately producing "meaningful" information. The modern engineering models and formalism of software development met some resistance from the rank and file "data processors" of yesteryear. Many asked, "Are all of these rules, these regulations, and this regimentation necessary? It is seemingly rigamarole." And, it should be said that, when it is done properly, it produces more robust, reliable, and resilient software products.

Ultimately, we are solving problems by modeling real world processes when we program computers. So, one of the fundamental activities in computing is problem-solving. When engaging in this primitive task, we must identify those things with which we must be concerned when building a solution. From above, we can see that, when we start, we must concern ourselves with three things: input, process, and output. These are the most abstract

of a single module's (program's) concerns. As we structure our programs, we further refine theses "concerns" and classify "sub-concerns". This iterative (repeated) refinement process continues until we create a sketch of the most concrete of the concerns within the program. This is a procedure known as concern partitioning or separation. The pattern in which we place these concerns is called the task of concern organization. Once we have gathered a list of this collection of concerns, we must sketch out the routine's (program's) high-level architecture of input, process, and output. Then, we fill in the next level of detail within each of these parts. We repeat this until we have fleshed out the entire program at the lowest level of detail. An entire field of scientific endeavor is called concerns research. In essence, it is the study of problem-solving which is something we have all been doing since grammar school.

Showing the "universality" of this three-step procedure known as a computer program, we will now look at something called a read, evaluate, and process loop (REPL). Although it might not be immediately apparent, this parallels the IPO pattern. Which we have already learnt. The read and input steps are synonymous. The evaluate and process step of a REPL are simply the process phase of an IPO. It is the case that the output step of a REPL is just "understood". In other words, it is implicit. And, built in this special-purpose IPO process is a flow of control function, the L (loop) which simply repeats the process until all of the input is exhausted.

Over the course of this short reading on the IPO architecture of programs, we have seen a few different terms for a program. This might produce some confusion since a program can hold programs within itself. These are called subprograms. And, since a subprogram is simply a program on another level, one might wonder at times which program is the topic of discussion, the outer or inner one. So, we rely on numerous synonyms when discussing a program. A short list of these in English are process, module, concern, routine, function, behavior, operation, method, and others. Each of these has a similar "sub-type". So, one might realize that a IPO program is also a process with a "P" subprocess.

And, one might say that a program is a function with subfunctions composing it. And it is the case that a computer itself is simply a "hard-wired" function. So, it is a function that takes other function's (programs) as input and produces the results of their processing as output. So, a computer is a general-purpose function which can solve any problem that is "computable". The theory of computing concerns itself with the question of what is computable. Many computer scientists who specialize in theory feel that classes of problems exist which cannot be computed by any program. And, this is what the field generally holds as "true". Yet, a "rare" few theorists feel that it is the case that one can compute a solution for any valid problem statement. Although, it might be the case that mankind has not yet discovered such a solution. A basic belief undergirds this. A "rare" few mathematicians believe that all valid problem statements are resolvable with elementary finite arithmetic. Which is using the four fundamental operations of addition, subtraction, multiplication, and division on the set of positive and negative whole numbers. It is the case that this author aligns himself with these "oddballs" who think all is computable.

This general-purpose function which describes a computer also follows the IPO architecture. And, it is “best” described with using the REPL design since it repeats itself until it processes all of the commands in a program. The “R-step” in the basic computer cycle is called “fetch”. This step gets the next instruction. The “E-step” in this cycle is called “decode”. And, it translates the instruction. The “P-step” is known as the “execute” phase. It performs the instruction. Finally, the implied “output” step is called “store”. And, this process loops (L) back on itself until all of the imperatives within the program listing have been done. This four-step process of fetch, decode, execute, and store (FDES) is a fundamental concept in the subfield of computer architecture.

The Nature of Process

The processing portion of a program is built from three primitive building blocks. These processing structures are sequence, branching, and iteration (repetition or looping). The former of these represents a linear “top-down” processing of program statements. The latter pair of these uses logical conditions with either a true or false value for processing the statements in a non-linear order. With these three fundamentals one can perform any feasible computation. At least, this is what is accepted as true by modern computer scientist.

Many different command-based (imperative) languages have similar terms for the building blocks which alter the sequential flow of these instructions. The “if” statement organizes instructions in one or more collections of commands for optional programming. Given that the logical condition associated with one of the collections, also called a compound statement, is true, the block of instructions associated with it will be performed. So, the logical condition represents a “guard” or “sentinel” which controls whether or not those statements are accessible. Another type of statement which selects sections of code for processing based upon a logical criterium is a “switch” or “evaluate”. These are much like an “if” but can process more than one compound statement on occasion. The above statements, if, switch, and evaluate, are all forms of conditional branching. A couple of basic styles of statements exist for repeating collections of instructions. The first is a “definite” loop. It repeats itself a fixed and finite number of times. An example of this is the “for” loop which is controlled by an internal counter. The second type is a “indefinite” loop. It repeats itself until a condition is false. This type of loop might process zero or more times depending upon whether the logical guard is placed at the top or bottom of the compound statement that represents its body. The former is called a “pretest” loop. An example of this is a “while-do” statement. The latter is called a “post-test” loop. Which called a “do-while” statement. All of these statements are predefined in imperative languages.

These built-in instructions often come with numerous other prepackaged commands composed of them. On occasion, some of the pre-defined instructions which are available must be linked in one’s program by using statements such as “import” or “include” and the name of the instruction bundle that holds them. And, if one of the prebuilt commands is not sufficient, all imperative languages lets one craft his own as a composition of the pre-built ones using special language constructs. Process, although detailed, is simple.

The Nature of *put

Next, let us discuss input and output, the *puts. Input is a “data” source. While, output is seen as a “data” sink. In terms of computing, source and sinks of data are treated like “files” for processing purposes. So, whether one is requesting data from an input channel or placing information on an output channel, those origination and destination points are interpreted as files. So, your keyboard which is a source of information is seen as a file. The network connection which delivers a requested webpage is treated like one. The local or Internet-based database which answers questions about the data that it holds when your computer asks them also is read and written like a file. These *puts sources and sinks also have other names like channels and streams. This is consistent with the idea of data or information flow.

How does one process a file? Well, first he must open it for access. On occasion, it might be necessary that he specify how he will access that file. The most common options are sequential or random access. With sequential access, he reads the data one item at a time. With random access, he describes a position in the file from which he desires data. Then, he requests it. And, after processing the needed data from any file, one must close it. Four basic tasks exist for processing the units of data in a file. These are create, read, update, and delete (CRUD). One can create new data plus read, update, or delete some that already exist.

Input and output work with data. Internally, computers represent this as a binary number. Which is composed of only zeroes and ones. This simplifies the electrical design of these devices. And, one can perform a simple conversion between the binary and decimal form of these numbers in a computer’s memory. And, these binary signals can be associated with various types of data. So, the same number might be interpreted differently based upon the type associated with its value. At times, the number ninety-seven might be just that. At other times, it might be a lower case “a”. It also might be interpreted as “true” in a logical condition. And, it also could represent an entirely different number if it were seen as a “floating-point” value.

Data can also have structure. It can be a collection of individual items arranged in a linear or non-linear fashion. These non-linear structures might represent a regular shape like a circle or tree with a uniform branching pattern or this might represent a irregular shape like a network that looks like a spider’s web. And, any shape that the mind can imagine could be constructed for holding data. Yet, most are modelled after common everyday real-world structures like a stack of cards, a service line at the bank, or the mighty Oak in your backyard. These structures each have a protocol for accessing the data which they hold. For example, when one works with a stack, he can “push” data on the stack or “pop” it off. The data enters and exits with a last-in first-out (LIFO) protocol. So, when inputting and outputting data, some might come from such a structure. These are called abstract data types (ADTs) since they are logical abstractions of real-world structures.

The Process of Process

We have already learnt that a program itself is a process with subprocesses. Yet, the formal development of software is an engineered process. It is iterative and optimizing. This means that we truly never finish writing any software product and that it truly never reaches perfection. We follow the first cycle of development with many cycles of enhancement. These steps of modification for the purpose of improvement or customization is the maintenance phase. Each cycle of the development process has the same four basic abstract phases: planning, design, implementation, and testing (PDIT). After the first round of these are completed the product is deployed for use and maintenance. The planning phase has three sub-phases: feature suggestion, requirements gathering, and specification. During feature suggestion a visionary describes the characteristics that the final software should have. After this activity, a single engineer or team gathers the requirements needed for producing software with these features. After completing this task, a group writes a specification that describes what the software will do. The design step has a couple of sub-steps: architectural and detailed design. The architect drafts the "big" picture of the software system which is one large IPO-structure with many sources of input, many subprocesses, and many outputs. Then, a designer provides the finer more concrete details of each part of the "big" picture. This might have many levels of detail. Ultimately, the output of this step should be a detailed design which implementation engineers can transliterate producing a system description in one or more high-level computing languages. Transliteration is a very exacting "word for word" translation. During the implementation step, each engineer produces one or more units of functionality drawn directly from the parts of the detailed design. Once this is done, testing begins. Testing has a couple of parts, internal verification and external validation. During verification each unit of functionality is reviewed based upon test cases drawn from the requirements document. This will ensure that each unit does what it should individually. This is called "unit" testing. Then, one integrates these units forming subsystems and systems testing each. This is called "integration" testing. Since verification test the system's functioning on various levels, it is also known as functional testing. After the quality assurance team is satisfied that the system is "correct", it will validate it with the client by demonstrating its functioning. Once the client accepts it, it enters use. Validation is also called acceptance testing.

It should be said that this process is formally controlled by a project manager. This person uses varying tools for estimating the time for completion and adjusting the project schedule when necessary. It only should be said that the activities which form a project are inherently non-linear based upon their interdependences. Each activity itself can be seen as an individual subproject composed of smaller tasks. The project manager uses PERT and GANNT charts for describing this list of project activities. This tool helps him discover the project's "critical path" which determines the minimum amount of time that it will require. If any task on this path is delayed, then the entire project possibly could run over its allotted time.

Seeing that the act of software development is itself a process, which is synonymous with program, one might ask this "simple" question. Could one write a program that accepts a list of features as input and produces software as output? The answer is "yes"! And, it has been done. This was the novel and new approach for software development during the fourth

generation of computing languages in the 1980s. The use of these languages was not widespread or commonplace. But, they were present in “real” world computing and more than simply a research topic. The author had the pleasure of working in a information systems shop at the local housing authority during his first college internship. This team used a product for report writing called the Professional Application Creation Environment (PACE). It ran on a Wang computer. Which is a former platform that was provided by one of the leading computer companies of its day. PACE was a fourth-generation language (4GL) that accepted a specification of a program and then generated COBOL procedures from it. The COmmon Business Oriented Language (COBOL) was the primary computing language used for business processing through the 1960s, 1970s, and 1980s. It is still used this day in 2021 and has evolved possessing all of the tools found in modern languages. COBOL and languages like it are command-based and use human readable imperatives. These also are known as third generation languages (3GLs). 4GLs were displaced by “universal” command-based languages that ran on virtual machines. Although these are 3GL-style languages, the use of a “freely available” virtual machine platforms resulted in widespread adoption and proliferation of these languages. And, in a sense they are a fifth generation of languages (5GL). Also, these 5GLs support a concern partitioning approach (problem-solving and system modeling technique) known as object-orientation.

Although 4GLs are not as popular these days, it might be the case that the next generational step in computing would use a virtual machine as a platform for interpreting the specification of a procedure that is at a more abstract level of representation than the imperatives of a program. In other words, one would provide a processor the blueprint for a program. And, it would automatically run.

As far as software, it should be said that it has a general abstract model. This general model has a number of ways in which it might be viewed or described. The most abstract of these is the feature set. The next view is the list of requirements. And, these can be viewed as a specification. A specification might be viewed as an architectural and detailed design. And finally, the last and most exacting view of this model is the implementation in a computing language.

A, B, C.... And, so on...

In the late-1980s, the abstractions, concepts, and basics of computing provided the world with a new problem-solving approach. This novel means for organizing program concerns is called object-orientation. The Pascal language, which was the most common first computing language learnt, taught that all of the related data items should be in a single location in a program. And, it also taught the related behaviors should be in the same place. However, it did not teach that related data and behaviors necessarily be in the same module or describe how modules might share internal data and behavior in regimented and formal ways. Without fully describing object-orientation, it is simply an extension of the structured programming techniques which were popular in the late-1970s and 1980s. It imposes more structure in a way that simplifies how one reasons about organizing a computer program. Since computer programs model the real world, these computing objects which are simply an abstract type of data are a generalization of the characteristics of everyday things. And, all objects have a couple of basic parts, their properties and their behaviors. And, objects in the natural world inherit characteristics from their predecessors.

A new model of computation also was envisioned at the time by the originator of the concepts surrounding object-orientation. This a novel form of concern partitioning known as action-orientation. Which currently is erroneously defined in the world of computing. The “original” concept of “action-orientation” was built around the notion of an “actionable” data type. Since these new objects would have “built-in” behaviors, they could be placed in structures much like the elaborate dominos displays showcased frequently on television in the 1970s. After invoking an initial behavior in the first object like tipping the first domino, one would unleash cascades of action, as each object programmatically invoked behaviors in the following object. This sequence of action might occur in concurrent streams at times. Hence, such a concern partitioning approach would allow for natural parallelism which expedites processing. It also was envisioned that some primitive arrangements of this form of ADT would be created from which others would be composed and that these structures would have, at a minimum, a primary, secondary, and tertiary structure like a folded protein.

So, from the a, b, cs of computing one can form the words of thought that fuel the computing marvels of the future.

Let Us Talk

We all like bragging at times about who we know, what we have done, and what we know. This writer knows the Author and Finisher of his faith, El-Olam, the Lord Everlasting. This greying academic, who earnt his first PhD at UHK in the Fall of 1988, has learnt the he should ever trust, depend, and lean upon Him. And, that is not always easy; but it is better than the alternatives this world offers. For, he knows that one day in His Courts is better than ten thousand elsewhere. For it should be said that the UHK is the University of Hard Knocks and not the one in Hong Kong.

El-Roi, the Lord that Sees and Shepherds us, has placed this writer in some positions in life where he can do much good for the international community; and, he has saved him from the drama that comes with accolades, awards, and the limelight. This a blessing of locality, resting in the right place at the right time. This location has been spiritual, spatiotemporal, social, and subjective. Notably, a social relationship permitted having an informal role in the development of some this generation's "most significant" computing innovations, as a visionary. Some of these have netted Turing Awards. Unfortunately, man's failings, including my own, have brought about trouble and turmoil on this earth. This is simple the nature of our lives in this fallen world. It is bittersweet. Only, if we would accept that the Kingdom of God is among us and in us, we would fare much better.

Some would say that the claims of "positive" influence are simply fanciful babbling. Yet, had not this author's brook been babbling for some time, many would not have the comfortable work roles that they have with the ample salaries that they do, at this time. His PhD from UHK did not allow such. And, we must all remember that He has done many mighty miracles during our time on this earth. Let Him work in your life, The very words that flow from His Lips are lifegiving.

Epilogue

What does the future hold for computing devices and programming languages? What does a child's imagination hold? For it was in a place called Camelot in the early 1970s, that a small Moorish child with woolen hair and large eyes which were a hazelly mixture of caramel, orange, and green eyes gazed at a small white thirteen inch television on an chest of drawers made of particle board, He loved Saturday morning cartoons like many children. Nearly as much as he treasured watching Jean's Storytime on Sunday mornings which taught simple Biblical lessons.

Yet, it was one Saturday morning while he was watching a program hosted by a fellow named, Pete Gannett, he believes that the dream of producing his own computing language arose. The show was sponsored by Kellogg's corporation who makes great cereal and food products that has been a staple in American households for decades. It showed the host who was a computer programmer and described his work. That small child with a crown of wool prayed that he might some day create his own computing language that the whole world used ad started its day with like the cups of morning coffee that his parents often had. He hoped every engineer would use it and thought that he should give a name of something that is in every workplace, he thought possibly, "Watercooler". Since this was a congregation area for many employees when they relaxed. However, he settled upon something associated with coffee since most adults in the workplace: engineers, managers, bus drivers, custodians, and etcetera start their day with such. And, he needed a name. Coffee was so mundane. He thought that he needed something with "pizzaz". So, he settled on a "JAVA". He prayed that it would be the world's new COBOL. And, that prayer was answered through a series of events, some beautiful and some incredibly painful and frightening, that placed in that dorm room during the Spring of 1989 and in a computer class which he took on a whim as a freshman elective. It was after the graduate assistant who was the lecturer for the course challenged the class that he sat down and started writing concerning some topic in computing. Simply brainstorming, he remembered the Saturday morning programming, his dreams, and his prayers. Then, he started writing and shared these visions with others, including "friends" who worked in executive management at Sun Microsystems. The rest is history. His question for you is, "What does your future hold?" "How grandiose are your dreams?" "Are they sought so He might gain glory?" Are your dreams so big that some might label you slightly "crazy" or "insane" if they heard them? Then, they are not big enough. Because, anyone which was should be met with the reply, "Are you out of your cotton-picking mind?" "Have you gone plum-crazy?" "Are you nuts?" "You must be daft!" Yet, Josephine dreams and dreamers have shaped this world. And, it is said,

> **[Mat 19:26 KJV] 26 But Jesus beheld [them], and said unto them, With men this is impossible; but with God all things are possible.**
> **[Mar 10:27 KJV] 27 And Jesus looking upon them saith, With men [it is] impossible, but not with God: for with God all things are possible.**
> **[Luk 1:37 KJV] 37 For with God nothing shall be impossible.**
> **[Luk 18:27 KJV] 27 And he said, The things which are impossible with men are possible with God.**

So, what does the future hold for computing? What are the limits of a child's dreams and the love in his heart that El Roi see and the prayers the Jehovah-Jireh answers. They are as boundless as Eternity and His Love for His Children. Red or Yellow, Black or White, we are all precious in His Sight!

Impossible is nothing.
- **Muhammed Ali**