Laws of Computing, Version 2.0

C. H. Ting
http://www.offete.com/files/LawsofComputing.htm

edited for easy print Juergen Pintaske

Summary

Can you prove that Forth is superior than other programming languages? This is an effort in laying down fundamental laws in computing to prove that Forth is the only right language. Here are the laws of computing as I perceive them:

1. *A computable function is represented by a linear sequence of instructions with occasional branching and looping.*
2. *There is no unique solution to a computable function.*
3. *A simpler function is a better function . The simplest function is the best function.*
4. *Functions should be constructed in modules and structures.*
5. *Correctness of a function can only be proven by recursively factoring into smaller modules whose correctness can be proven*

1. Introduction

"Computer Science" is a self contradicting term because the studies of computers have not been organized in the scientific methodology as in most of the branches of science like algebra, geometry, thermodynamics, classical mechanics, quantum mechanics, etc. A fully developed branch of science generally assumes a set well organized principles or laws, from which all the observable consequences can be derived by mathematics and logic. Take thermodynamics, for example, there are three laws:

First Law of Thermodynamics:
In a closed system, energy is conserved.

Second Law of Thermodynamics:
In a closed system, entropy is always increasing.

Third Law of Thermodynamics:
The entropy of a pure, crystalline substance is 0 at the absolute zero of temperature..

From these three laws, all phenomena relating to heat, energy, mechanical work, and chemical reactions can be understood in a logical and unambiguous fashion.

Is there a 'science' of computer? There are so many different computers, computer architecture, languages, operating systems, software development tools, and applications, it is impossible to summarize "computer science" in simple and general laws to describe our understanding of computers and the computational processes. However, there are fundamental elements of computing and programming which can be stated as laws of computing. Here is my attempt to summarize our experiences in computation using electronic computers into three laws of computing.

As laws in other branches of science, these laws of computing cannot be proven from other principles, but are derived from observations on ways computers are built and used. The validity of these laws can only be confirmed by that the consequences derived from these laws match actual observations.

2. Laws of Computing

2.1 The First Law of Computing

A computable function is represented by a linear sequence of instructions with occasional branching and looping.

This is very apparent to people programming in assembly. However, it is equally true for all procedural programming languages, in which 'statements' are executed in sequence with occasional branching and looping. This First Law of Computer is simply an alternate way to describe the Turing Machine which maintains that any well behaved problem can be solved by a step by step process, applying appropriate transition rules on input data at each step.

To explain this law more clearly, I have to make the following observations:

1. A computer is a device which is designed to execute a sequence of the instructions serially. Instructions are available to modify the executing sequence by branching, conditional branching, looping, subroutine call and return.

2. A special purpose computer can be built with an instruction set tailored to the problem it is intended to solve. Ultimately, a computer can be built with only one instruction, which is the application itself. However, this type of computers are expensive to build and impossible to be used to solve any problem other than the one and only application intended.

3. A general purpose computer always have a finite and limited set of instructions so that it can be built economically.

4. Computable functions are always much more complicated than what the instructions can do individually. Therefore, a computable function must be factored so that it can be accomplished by executing a sequence of instructions.

5. A general purpose computer must contain the following components:

Memory:	Including registers, stacks, RAM, ROM, and mass storage like tape, disk, etc.
ALU:	Make arithmetic or logic operations on data stored in memory
I/O Channels:	Communicating with devices outside of the computer
CPU:	Mechanism to fetch instructions from memory and executing them serially with occasional branching and looping.

6. The instruction set of a general purpose computer includes:

Memory operator:	fetch, store, push, pop, move
Arithmetic operator:	add, subtract, shift, rotate, (multiply, divide)
Logic operator:	and, or, xor, complement
I/O operator:	input, output
Flow control:	call, conditional call, return, branch, conditional branch, loop

7. Such an instruction set is sufficient for all computable functions. Computers differ by details in their instruction sets, the bit width of words in memory, and the addressable memory space, etc. Nevertheless, with an instruction set including a minimum number of instructions in the above list, all computable functions can be represented as sequences of instructions. One example is a hand held calculator using a 4-bit microprocessor to calculate transcendental functions in floating point numbers.

8. Subroutine call and return are the most powerful instructions in a computer. They allow sub functions to be encapsulated into small modules which can be used to build more complicated functions. Subroutines allow the instruction set to be extended, and tailored to specific applications. A last-in-first-out (LIFO) stack is necessary to support nested subroutine calls and returns.

Subroutines are made even more powerful by feeding them lists of parameters. However, it is difficult to make subroutines to accept arbitrary lists of parameter. Often complicated compilers are necessary to handle the parameter lists.

9. High level languages were developed to allow the computers to be programmed at a more abstract level with syntax and semantics closer to an application. However, high level languages require complicated compilers which can translate high level statements into machine executable instructions. The compiler insulates users from the computer.

A high level language generally prescribe a unique set of pseudo instructions, and the syntax rules on how the pseudo instructions are combined so that the pseudo instructions can be executed in a unique sequence to produce desired and predictable results. No matter what the set of pseudo instructions and the syntax rules, computable functions are still programmed as a linear sequence of instructions which may take the shape of statements, with occasional branching and looping. Statements in a high level language are directives to the compiler to generate sequences of instructions which eventually will be executed by the computer sequentially.

Programming is thus the process of discovering a sequence of instructions which will generate the desired functionality. Computing is the process of executing this sequence of instruction to realize the function. The challenge to the programmers is to discover the sequence soon enough and that the sequence can be executed

on a real computer with resources limited by the constraints that the final product will bring back economic rewards to the organization funding the efforts, so that further programming efforts can be supported.

Only discovering one sequence of instructions to product the functionality is not enough. One has to be concerned with the questions that the sequence is optimized giving the constraints, and that the sequence is a correct one. The following laws deal with these problems.

2.2 The Second Law of Computing

There is no unique solution to a computable function.

There are many different ways to solve the same problem on the same computer with exactly the same configuration, because the instructions in an instruction set are not orthogonal, i. e., different instructions have overlapping functionality. For example, a bit can be tested for zero or for one, the results of an ALU operation can be tested for zero, positive, negative, or overflow, addition can be done with subtraction, a loop can be terminated at the beginning, in the middle or at the end, etc., etc.

At the most elementary level in the electronic logic circuitry, the NOR gate can be replaced by NAND gates according to the famous de Morgan theorem. Thus the possible ways to arrive at a solution to a logic problem using electronic circuits could be at least 2**N, where N is the number of NAND/NOR gates involved in the circuit. There is no unique way to implement any of the very basic logic functions at the gate level. On the top of this ambiguous and amorphous foundation, we add different CPU architecture, different instruction set designs, different computer configurations, different operating systems, different programming languages. It is no wonder that we end up in a situation where confusion reigns supreme.

To a single function, there are as many solutions as programmers assigned to solve it. Even one programmer can arrive at many different solutions at different time. This is the 'polymorphism' of programming. This is what prevents computer science to become an exacting science rather than an art or simply black magic.

If there is no unique solution to a computable function, can we determine which of the solutions are better than the others, or, which one is the best?

Here comes the problem of software quality, bench marking, and software matrix. To determine which program is better than another, we have to decide the criteria by which the performance of programs are measured, if it can be measured at all.

Here are some of the criteria generally used:

Compactness
Speed
Functionality
Ease of programming
Ease of maintenance
Ease of use

Since computers are getting bigger and faster all the time, the dominating criteria of software profitability have been time to market, ease of use, and functionality. The other factors, which are factors important in software quality receive less attention, and sometime completely ignored in commercial software development. Nevertheless, these are issues which must be dealt with in computer science, if there is such a thing.

The second law of thermodynamics states that in a spontaneous process, entropy always increases. A practical example is that gas always fills the space it is allowed to expand into. Compressing the gas back into a smaller volume requires the reduction of entropy. It takes extra energy from outside of the system, and does not happen spontaneously. It seems that software obeys the same law. It fills the memory space given to it. It takes determine efforts to make it fit into a tight space.

The most important factor in determine the software quality is simplicity. All the other desirable factors are its derivatives. I like to state it as the third law of computing:

2.3 The Third Law of Computing:

A simpler program is a better program. The simplest program is the best program.

In all the various branches of engineering, people are trained to detect and to avoid complexity, because complexity increases the probability of products failure. Simpler products are easier to manufacture, causing less problems in assembling and in use, and improves the reliability, and reduces the efforts of maintenance. Software is the only product in which complexity is made to equal value and usefulness. Mechanical engineers would be horrified at the thought of building an automobile with 1 million parts. Software systems consists of 1 million instructions are becoming common commodity. Programmers are hopeless optimists. It is so easy to add a few lines of code to a program. The memory is cheap. The CPU is doubling its capability every 18 months. When mega bytes of memory become available, programmers rush in to fill it up to the rim. The bulkier software runs slower, under the burden of the added but rarely used new features. The software survives only because of the scalability of silicon: that the CPU becomes faster and the memory becomes bigger rescue the software bloated with deficiency.

Simplicity is the best indicator of the quality of a program. As Chuck Moore commented:

One principle that guided the evolution of Forth and continues to guide its application, is bluntly:

Keep It Simple.

A simple solution has elegance. It is the result of exacting effort to understand the real problem and is recognized by its compelling sense of rightness. I stress this point because it contradicts the conventional view that power increases with complexity. Simplicity provides confidence, reliability, compactness, and speed."

To achieve simplicity, one has to spend time reworking the code. It takes time to discover the redundancies in the code and remove them. It takes time to re-partition the modules so that modules are re-used more and application code is reduced.

Lau-Tze wrote an interesting paragraph in his Tao-Te Ching:

"To gain knowledge, add a little every day. To gain Tao (wisdom), remove a little every day. You keep on removing things, until there is no artifact left. When there is no artifact left, nothing will prevent you from accomplishing everything."

What he meant was that the nature of things is simple. People make things more complicated. You have to be able to remove the artifacts to discovery the true nature of things. With the true understanding of things, you will be more productive and accomplishing more. Lau-Tze could have been a great programmer, had he lived today.

Computer itself is simple because it can only do simple things. People made it complicated by throwing layers and layers of software like assemblers, compilers, languages, operating systems , and applications on it. Programmers and users are enslaved by these layers of complexity. Unless you can peel back these layers and getting back to the computer itself and mold it to your own liking, you are condemned into the fiefdom.

The Third Law of Computing has an interesting correlation with the Third Law of Thermodynamics. Entropy is a relative quantity, associated with state changes in a system. To compute entropy of an arbitrary system, we need an absolute point of reference. This point of reference is perfect crystal at the temperature of absolute zero, which is impossible to reach experimentally. At this idealized state, entropy is zero because the whole system exists in a single state without ambiguity.

Similarly, the simplest program is most likely an idealized objective never reachable in reality. However, we should not loose sight of this goal. The simpler a program gets, the closer it is to perfection and its performance will be improved accordingly. Complexity and the associated inefficiency are sins against civilization and humanity, because they waste precious resources needed by the civilization to sustain. One machine cycle does not seem to worth much, but it becomes significant when it is multiplied by the number of machines executing it and the number of time it is executed.

2.4 The Fourth Law of Computing:

Functions should be constructed in modules and structures.

Modules are instructions grouped together as functional units. A module can be grouped together with other modules to form larger and more powerful modules. Modules can be nested, but may not overlap with one another. A module has only one entry point and only one exit point. Ideally, modules should be testable individually. However, language tools and operating systems may prevent modules to be independently testable. A structure is an instruction group inside a module which again has only one entry point and one exit point. A structure does not exit by itself, it is a part of a module and cannot be accessed from outside of the module. A structure can be viewed as the extension of machine instructions, and are often defined as pseudo instructions or macros. Hereafter, in the following discussion we will use instructions and structures interchangeably.

In the 60's, 'structured programming' was discovered and it was hailed as a revolution in programming. It was shown that it is possible to write any program using only three structures:

1. Sequence structure: Sequence of structures are executed in the sequential order as written.

2. Branch structure: IF-THEN-ELSE structure ; the true clause and the false clauses may contain any of the three structures.

3. Loop structure: A loop control mechanism such as DO-WHILE or DO-UNTIL. The repeated clause may contains any of the three structures.

Using only these three structures, it is possible to program any computable function. The source code is greatly simplified and can be read from top to bottom without ever branching to other places. The GOTO instruction can be completely eliminated. Programs are accordingly much easier to read and understand, with improved productivity and reliability.

Here, out of an infinite ways to program a computable function, a sound and useful methodology was discovered. Adhering to this methodology, we can produce better programs which are simpler, more efficient, more reliable, easier to understand, and more likely to be correct.

Partitioning a function into smaller modules is still an art form. Simply making modules smaller does not necessarily reduce the complexity of the function. Modules should be defined according to reusability. If the same sequence of instructions are used more than once, they are potential candidates for modularization. Reusable modules reduce program complexity, and often reduce the physical size of the program.

The other principle promoted in structured programming was step-wise refinement, that a function should be stated in steps at a high and abstract level at first. Each step is later refined to a lower but still abstract level. This process of refinement continues until the steps can be represented by machine instructions or elementary operations provided by a high level language. The implication of this principle is the top-down programming methodology, that a program should be written from the top level, and gradually reduced to lower levels.

My opinion is that the top-down methodology is appropriate only in the design phase. The fallacy of this technique applying to actual programming is the assumption that the programmer understand the function completely at the beginning, and that the modules evolved during the refining stages are correctly defined and partitioned. This is never the case in reality, as the programmer cannot be expected to know everything about the function at the beginning. The modules cannot be verified before all the underlying code are completed. Unexpected problems will propagate back from the low levels to the high level.

The top-down methodology has an analog in project planning, the Water Fall Model. The product development cycle is divided into stages: specification, design, production, release, and maintenance. This model is useful when the product is well understood, like a house or an automobile, which have been built for years, and we have all the necessary information on how to produce them. This model has a very deep root in the western civilization, which assumes that there is a beginning of everything and there is and end. Things progress from the beginning to the end in a straight line.

This is rarely the case in software, which must evolve in a rapidly changing environment. Nothing starts in a vacuum. A new product is conceived because deficiency in an old product or in the changing needs of customers. An old product does not die, but evolves into other products. New ideas and new technologies are incorporated in the development process. Specifications are influence by design and market acceptance. Designs are changed because of new requirements. A better model is a circle in which specification, design, manufacturing, marketing, and maintenance are iterated repeatedly. Products are spinning off this circle. If the product is successful, profits may be plowed back into the circle, making the circle bigger or running faster. If the products do not generate enough profit to sustain the circle, the circle dwindles down and may disappear completely.

The cyclic model is more attune to the Eastern thinking and is much closer to reality. The industrial revolution has only 200 years of operational experience, much too short to support the water fall model. The cyclic model has 5000 years of recorded data base.

Programming is also a cyclic and iterative process. The implementation and the design have to be reconciled repeatedly, as the implementation may demand changes in the design, and the design may be changed due to constraints in resources and pure external pressures from marketing and the customer base. The top-down model is of great importance in program design and module partition. However, it can not be held rigidly because one cannot anticipate all the possible problems at the beginning of a programming project. Hence I do not think the stepwise refinement can be qualified as a fundamental law of computing.

2.5 The Fifth Law of Computing

Correctness of a function can only be proven by recursively factoring into smaller modules whose correctness can be proven

In a large program, it is generally impossible to prove that the program is correct, because the number of possible pathways in a program grows exponentially with the number of branches and the number of variables in the program. To prove the correctness of a program, all the possible pathways must be traversed and shown to behaved correctly through the entire ranges of values in all variables. It is mechanically impossible to explore all the possibilities, and hence one cannot prove that the program will behave correctly under all circumstances.

As structured programming was discovered, it was also found that if program modules are restricted to have only one entry point and one exit point, these modules then become structures which can be string together to become larger structures. Structured in this fashion, it becomes possible to prove the correctness of a program.

To prove the correctness of a program, we have to follow the following recursive steps:

1. A program must be factored into a finite set of modules.
2. Each module must be proven to be correct.
3. As there are only finite number of modules, and the pathways among the modules in the program is finite, the program can be proven to be correct by traversing through all the pathways.

Now, proving the correctness of a program is reduced to proving the correctness of all the modules inside this program. The correctness of each module can be proven by the same process as that for a program. This process must be applied to all the modules recursively until all the modules are reduced to those which are composed of machine instructions of the computer used to execute the program and the associated modules. We have to assume that the machine instructions behave correctly at all time and under all circumstances. From this level, the correctness of all the modules can be proven, and hence that of the main program.

The key in proving the correctness of a program is that the program must be factored into non-interacting modules which must be factored further into smaller modules until the modules at the lowest level can be represented by machine instructions or primitive functions whose correctness can be assumed without doubt. The complexity within a module is thus reduced to a manageable level and all the possible pathways inside this module can be completely traversed.

The fifth law of computing gives us hope that we still have a fighting chance to overcome the problem of software complexity and to produce reliable software products. However, to achieve this goal, we have to be equipped with the right tools and the right mentality in approaching software problems. Simplicity and modularization, these are the attributes one has to insist on in all phases of the software development in order to succeed.

3. Forth Perspective

3.1 Corollary to the First Law of Computing

All computable functions can be built from a set of pseudo instructions . The set of pseudo instructions are built from primitive Forth instructions and other existing pseudo instructions.

Forth is the simplest high level language in existence. It is best considered a Virtual Computer which can be implemented on any computer with a reasonable set of machine instructions. A Forth Virtual Computer are modeled very closely after a general purpose computer, having the following primitive Forth instructions:

Memory instructions:	**C@,**	**C!,**	**@,**	**!**			
Arithmetic instructions:	**+,**	**-,**	***,**	**/,**	**MOD**		
Logic instructions:	**AND,**	**OR,**	**XOR,**	**INVERT**			
Stack instructions:	**DUP,**	**SWAP,**	**OVER,**	**DROP,**	**>R,**	**R@,**	**R>**
I/O instructions:	**IN,**	**OUT**					
Control instructions:	**IF,**	**ELSE,**	**THEN,**	**BEGIN,**	**WHILE,**		
	REPEAT,	**AGAIN,**	**DO,**	**LOOP,**	**EXECUTE,**	**EXIT**	
Defining instructions:	**:,**	**;,**	**CONSTANT,**	**VARIABLE**			

Forth is more than a high level language, because in addition to the primitive instruction set and the syntax rules on how to combine the instructions to perform more complicated pseudo instructions, it also contains an interpreter to execute the pseudo instruction sequences interactively, and a compiler which allows the user to define new pseudo instructions to replace lists of existing primitive and pseudo instructions.

Thus Forth becomes an extensible high level language to which a user can add new instructions to deal more effectively the application at hand.

Forth also provides a very unique capability in that a new pseudo instruction can be constructed using a list of native machine instructions. In most Forth system, an assembler is included so that the user can use assembly mnemonics to specify the machine instruction sequences without leaving the Forth system. It gives the user the freedom to optimize the pseudo instruction set for a specific target computer, and to utilize fully the resources incorporated in the target system otherwise inaccessible to the user.

User can define new pseudo instructions which replace lists (linear sequences) of existing primitive and pseudo instructions. The pseudo instructions added by the user can be used to construct newer and more powerful pseudo instructions. Replacing lists of instructions by new instructions move the solution of applications to a higher and more abstract level, which simplifies programming and compresses the code.

The advantages of Forth are:

a. The Forth Virtual Computer is portable across a wide spectrum of computers and microprocessors.

b. Programming in Forth is to construct new instructions from existing instructions in a modular and progressive manner. In the end, a set of fully customized instructions becomes the solution to an application.

c. A new instruction replaces a sequence of existing instructions and is an abstraction of lower level functions. The process of abstraction simplifies the programming and results in compact object code.

d. New instructions can be compiled as a named list of existing instructions. The compiler is very simple. New instructions can be executed and tested interactively, thus greatly eases the programming and debugging of substantial applications.

3.2 Corollary to the Third Law of Computing

Forth is the simplest programming language and it produces the smallest code.

Forth excels in compactness, speed, and ease of programming, because it is based on the principle that new functions are defined to replace linear sequences of existing functions. Once defined, new functions can be used to build newer functions, just like the old, existing functions. Because functions pass parameters through a push-down stack, parameter lists are not necessary between a called function and the calling function. This unique property allows functions in Forth to be interpreted and to be compiled very easily. Compiled Forth functions become lists of execution addresses or execution tokens, which are both very compact and can be executed almost as fast as functions coded entirely in machine code.

If speed, compactness, and ease of programming are the most important criteria to measure software quality, Forth will be ranked the highest among all the languages and programming tools.

Stacks were invented to facilitate subroutine calling and returning. However, most high level languages manage to use only one stack to store return addresses and to pass parameter. The overloaded stack makes it difficult to

use and expensive to do subroutine calls. It also requires complicated compilers to work out the calling sequence of parameters. High level languages using parameter lists in subroutine calls must compile the subroutines first and later link the subroutines with a main program for execution. Subroutines by themselves are not entities that can exist by themselves. In contrast, modules in Forth reside in memory and can be executed and compiled interactively.

Passing parameters on a second stack greatly simplifies the language structure and accelerates the calling and returning processes. The expense in memory for a subroutine call is reduced to a single address of reference. The address of reference is much more powerful than macros and subroutines, and it is most appropriately thought as a pseudo machine instruction. These pseudo machine instructions can be concatenated together forming lists. The lists can be named to become new and more powerful pseudo instructions.

Naming a list of pseudo instructions and making it a new pseudo instruction is the most fundamental and the most powerful concept in Forth as a high level programming language. It generally leads to the simplest solution to a problem in the shortest period of time.

3.3 Corollary to the Fourth Law of Computing

Forth is modular and structured

Forth is a structured programming language before structured programming was invented. When he was told the advancement of computing technology in 1970, his response was: "I thought that is good programming practice." Forth was the first programming language without a GOTO instruction..

Forth excels in compactness, speed, and ease of programming, because it is based on the principle that new pseudo instructions are defined to replace linear sequences of existing primitive and pseudo instructions. Once defined, new pseudo instructions can be used to build newer and more powerful pseudo instruction at a higher abstraction level. Because the pseudo instructions pass parameters through a push-down stack, parameter lists are not necessary between a called function and the calling function. This unique property allows instructions in Forth to be interpreted and to be compiled very easily. Compiled Forth instructions replace lists of execution addresses or execution tokens. The compiled pseudo instructions are very compact and can be executed very fast if the host computer can handle two stacks comfortably.

Forth pseudo instructions are true modules. They reside in memory and can be executed and compiled interactively. When compiled into a list, the pseudo instructions can be viewed as structures, having one entry point and one exit point, and are grouped into a list to be executed sequentially. A pseudo instruction is the abstraction of a sequence of structures. Forth also provides a set of compiler directives (immediate words in Forth terminology) which can arrange groups of pseudo instructions to form branch structures and loop structures in the list.

If speed, compactness, and ease of programming are the most important criteria to measure software quality, Forth will be ranked the highest among all the languages and programming tools.

3.4 Corollary to the Fifth Law of Computing

It is possible to prove that a Forth program is correct.

Forth lends itself ideally to be proven correct, because Forth functions are perfectly modularized, and new functions are constructed by a linear list of existing, ideally proven correct, functional modules. The way Forth functions are defined and tested from the bottom up assures that new functions are built correctly and tested thoroughly before proceeding onto the next level of functional constructions. Forth programs are thus easier to develop and debug, and thus tend to behave correctly at an earlier stage of program development.

Because Forth is based on independently executable modules which can be tested interactively and thoroughly, a Forth program can be proven to be correct much easier than program produced by other programming languages and other programming techniques.

It is also possible to prove the correctness of a program written in other languages, if the program is constructed from modules whose correctness can be proven. However, if the modules cannot be executed and tested directly, one would have to write a test program for every module used in the system. Writing the test programs can be a very costly endeavor. In Forth, the modules are interactively executable and compilable. Thus the programming and testing can be carried out simultaneously and interactively. This unique combination of modularity and interactiveness makes Forth the most efficient programming technology.

4. Conclusion

There is a very interesting analogy between the laws of thermodynamics and the laws of computing as I discussed above. The first law of thermodynamics states the obvious: in a closed system, energy is conserved. The first law of computing states also the obvious: all computable functions are represented by sequences of instructions. It shows the nature of a general purpose computer, that complicated functions are always reduced to the primitive instructions executable by the computer. However, the execution sequences can be represented in different ways, just like energy takes different forms like heat, kinetic energy, potential energy, etc.

The second law of computing is not obvious, just like the second law of thermodynamics. Entropy is not a quantity which lends itself to easy measurement and interpretation. It is defined in terms of energy transferred between subsystems in contact with each other and at different temperatures. However, once entropy is defined precisely, we have a accurate indicator which allows us to predict in which direction a spontaneous process will proceed. Entropy is the quantity which measured randomness. As natural, spontaneous processes go, entropy is always increasing.

The second law of computing states that there are infinite number of ways to implement a computable function. Leaving it to programmers, they will come up with infinite number of ways to fill the memory space. Murphy's Law and Parkinson's Law have their roots here. It will take determined and persistent efforts to remove redundancies, inefficiencies, and irrelevancies in these programs to arrive at the best solution to a problem. This is where professionals make their distinction against the amateurs.

The third law of thermodynamics is even more abstract. It gives a reference point to the abstract quantity of entropy, that entropy is minimal only at absolute zero of temperature only for a perfectly ordered system as in a pure crystal. As it is impossible to reach absolute zero, entropy can never be zero. Perfect crystals at absolute zero. That's like a perfect program without a bug. We all know that it is practically impossible to build a program without bugs. Nevertheless, if we simplify a program by one step, it is one step closer to a perfect program.

The fourth law of computing gives us hope to approach a perfect program by modularization. Modules isolate program parts and reduce their interactions. The end results are a great demagnification of complexity and the reduction of the infinite possibility to go wrong to a finite and manageable path ways so that we can attack the problem efficiently with limited manpower and resources.

The fifth law of computing is the trumpet of victory, proving once for all that a correct program is possible if we adopt the methodology of modularization and take the pains to traverse the modules recursively. We can be assured that a correct program can be approached as close as we care, if we make an effort to modularize the program and recursively prove the correctness of all the modules.

The victory is possible only with Forth, because it has the attributes necessary for the fifth law of computing to succeed. A Forth program is constructed in independently testable modules, which are combined into bigger structures in which correctness can then be proven. In other languages, this recursive process is too expensive and too time consuming to be practical. The very tight code-test-edit loop in Forth gives Forth programmer the luxury to traverse his system up and down many times to prove its correctness.

####

10 December 2013