The Nature of Computing[†]

Bienvenido Vélez, Ph.D.
Assistant Professor of Computer Science
University of Puerto Rico Mayagüez
bvelez@acm.org

Introduction

Two and a half years ago I joined the Department of Electrical and Computer Engineering faculty at the University of Puerto Rico – Mayagüez (UPRM) just months after completing my Doctor of Philosophy degree in Computer Science and Engineering. I envisioned that, having the only school of Engineering within the UPR system, the UPRM was the only place in our island with the potential to become a world recognized institution in my discipline. This vision was founded on the unique and critical symbiosis that, I believe, must exist between successful CS and Engineering programs.

One of the first cultural shocks I encountered was the relatively inaccurate understanding of the discipline of Computer Science that permeated our institution then. One of the common beliefs equated Computer Science to Computer Programming. Others bounded Computer Science to the realm of "software". To some the idea that Computing may have existed long before the advent of the electronic computer was radical. "Computing without computers? Nonsense! Computing is about solving problems using the computer." Later on, I learned that these misconceptions were not a particular phenomenon in our institution, but were rather common outside the confines of Computer Science atmospheres.

What is Computing about? The discipline touches on all the areas mentioned above, so those inaccurate notions are somewhat justified. However, none of those notions by itself is comprehensive. I have seen several agreeable definitions of the discipline in the literature, including some rather long ones. The following definition offers a compact yet complete alternative:

Computing is the study of the phenomenon of Computation; the process of transforming information

Figure 1 explains how the computation process is used to solve real problems. First of all, an instance of the problem is encoded as input information. This input is transformed into some output information that, once decoded, yields a solution to the problem. A computing system or device can be said to *solve* a problem if and only if it is capable of producing the correct output for each possible input. For convenience I will use the term *computer* to refer to any device or system, not just electronic, capable of performing computation.

[†] COMPEL 2002 Plenary Talk

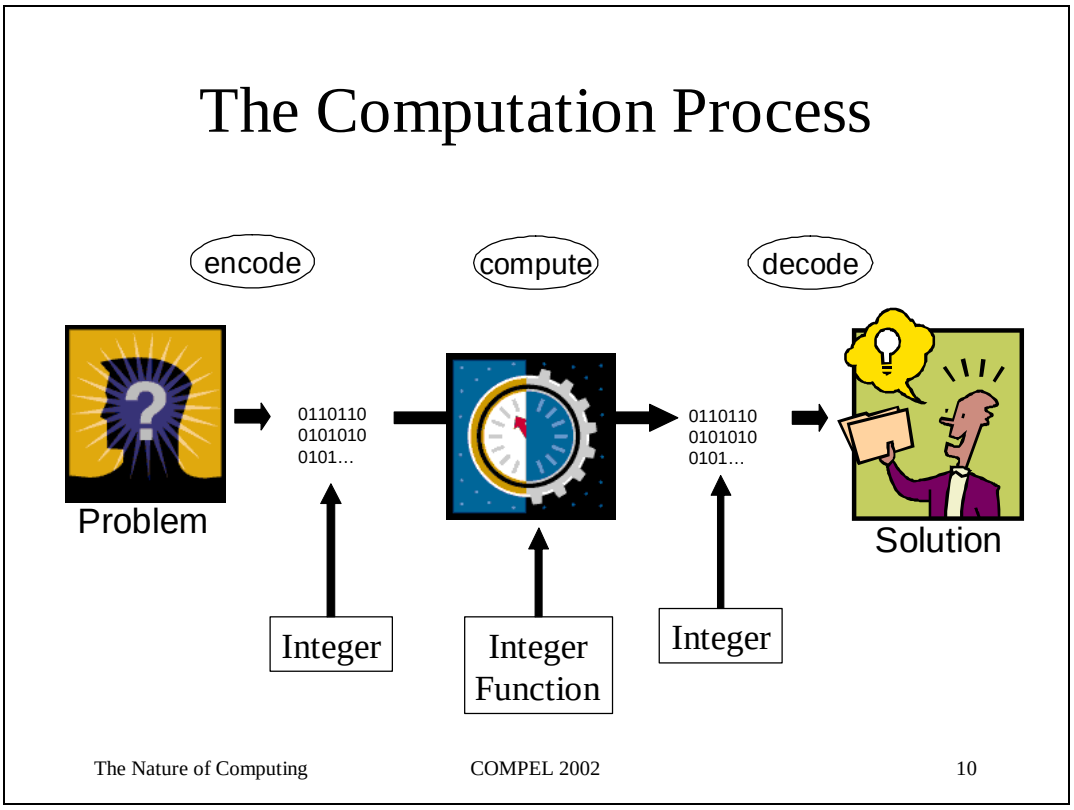

Figure 1. Steps involved in the process of computation

The discipline of Computing[1] addresses the following three fundamental questions:

- ? What can be computed? (computing models and computability)
- ? What can be computed efficiently? (complexity)
- ? How can we build practical computing devices and systems? (architectures and systems)

This definition clearly shows the lack of completeness of the common conceptions of the discipline. First, we can see that computer programming only encompasses the process of encoding algorithms into information. This is not to say that programming is a trivial process. Writing efficient, reliable programs remains a difficult, time-consuming and extremely expensive process. Second, we can see that computing is not exclusively concerned with software but rather with any device or system capable of performing computation.

The discipline of Computer Science has all the ingredients that comprise a traditional Engineering discipline. It is built on solid theoretical foundations which have been developed specifically to nurture the discipline and is concerned with the design and construction of real computing systems and devices. Consider the analogy that can be established with, say, Electrical Engineering (EE). Although built on solid theoretical

[1] In the remainder of this talk I shall use the terms Computer Science and Computing interchangeably.

foundations (e.g. Ohm's Law, Kirchoff's law, phasors) specifically developed for the discipline, EE is to a greater extent concerned with the design and construction of practical electrical systems and devices. The name Computer Science is mostly a historical accident. As Professor Gerald J. Sussman said during a lecture I once attended: "Computer Science is more about Engineering than about Science."

Due to space limitations I would like to organize the remainder of the talk as follows. First, I would like to explain in some depth the basic theoretical foundations of the discipline of Computing. Second, I will share with you what I think are some of the most ubiquitous concepts or ideas in Computing that I believe everyone willing to venture into the discipline should understand and readily apply.

Computing Models and Computability Theory

There are many ways of encoding problems into I/O information useful to a computer. In today's typical electronic computer, all information gets encoded as sequences of binary voltage levels, or bits (binary digits). But every binary pattern can be interpreted as a base 2 integer. Therefore, once a problem instance is encoded it can be viewed as an integer number. A computer is essentially an information transformation mapping, but since input information can be encoded as a number, every such mapping must have a corresponding integer function. A computation process implements an integer function mapping every possible input integer to the corresponding output integer. In light of these definitions, the first fundamental question of Computing can be rephrased as:

Which integer functions are computable?

In the late thirties the English mathematician Alan Turing proved that there were problems with no computational solution [2]. He proved that by introducing first a mathematical model of a computing machine that he called the logical computing machine (LCM). This model became later to be known as the Turing Machine. As shown in Figure **3**, the machine consisted of a controlling finite state machine and an infinite input/output tape. The machine always started from an initial state. On each transition the machine examined the contents of the tape under the tape head. Based on this symbol and the current state, the machine would decide the next state, an output symbol to write on the tape and the direction along which the tape head should be moved one position.

Turing provided rationale to justify that this simple model of a computer captured the whole nature of computation. He argued that the Turing Machine was powerful enough to compute anything that was computable by any realizable machine. In the same year of 1936, the logician Alonso Church [1] also conjectured that the lambda calculus, when viewed as a computational model, was also capable of computing all computable functions. Turing showed that his LCM was equivalent in computing power to Church's lambda calculus. In [2] Turing used his model to prove that there were problems that were *undecidable*, that is problems that no Turing Machine could solve. The most famous such problem became to be known as the Halting Problem (Figure 2):

Can we devise a Turing Machine to determine if a program ever terminates?

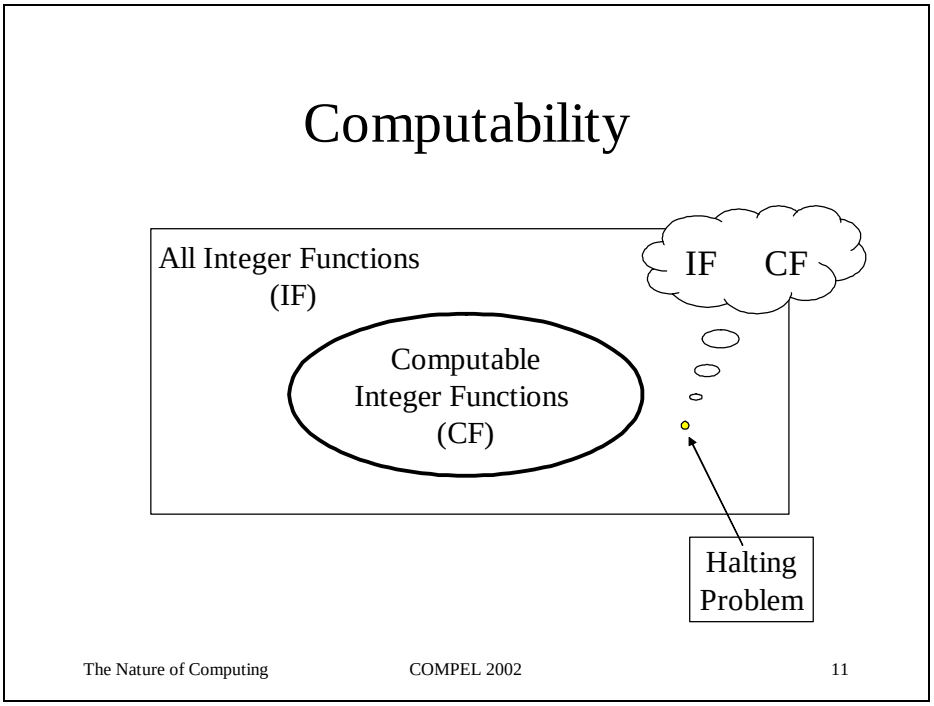

Figure 2. Relationship between integer and computable functions

In essence, Turing proved that if such a machine exists, then it would be possible to construct a "crazy" machine producing mathematically contradictory results. It is worth remarking at this point that the foundations of modern Computing were laid out even before the first electronic computer became available. Computing was born before the electronic computer.

Although not apparent at first glance, the undecidability of the Halting Problem has had and continues to have profound implications all throughout Computer Science. For instance, the undecidability of the Halting Problem implies that it is impossible to automatically determine if a program has any bugs. It also implies that it is not possible

to build a compiler that generates optimal code. In general, many interesting properties that we might want to determine about programs are known to be undecidable.

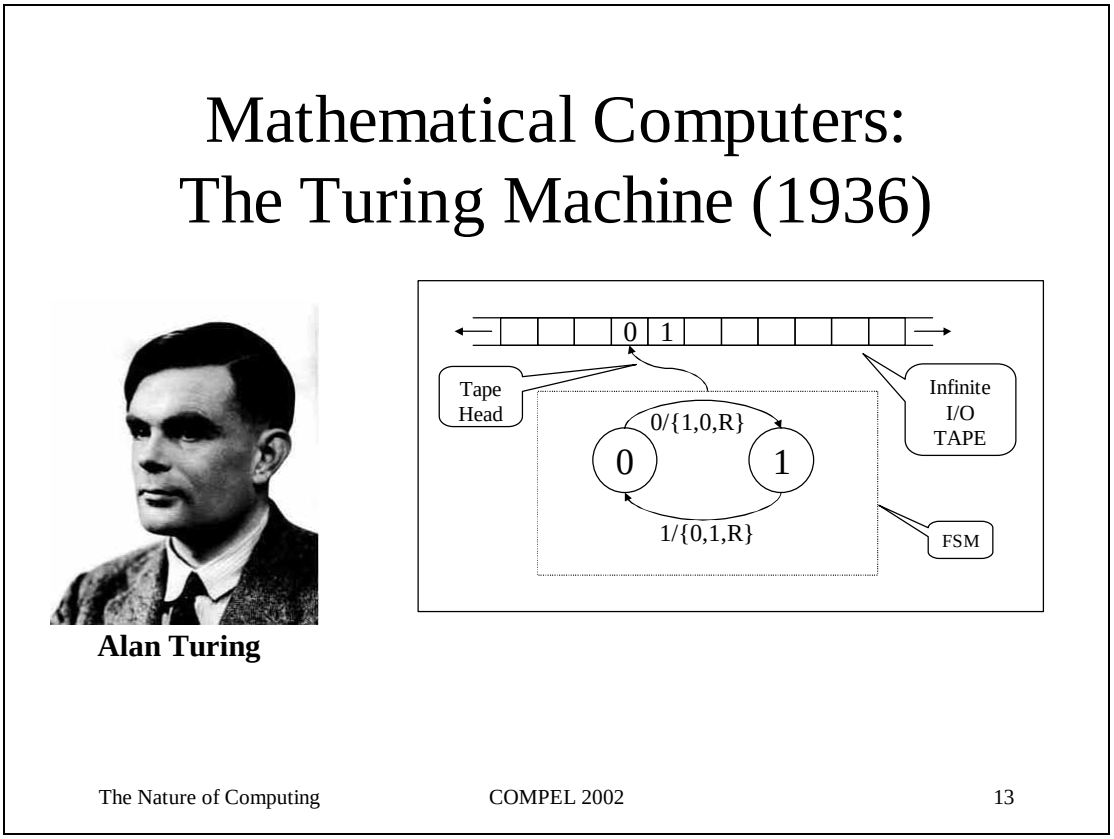

Figure 3. Turing Machines

Interpretation, Programmability and Universal Computation

Another remarkable contribution by Turing was the idea of a *universal Turing machine (UTM)*. A UTM is capable of simulating any other Turing machine. That is, given a description of an input machine M and its input tape T, the UTM can reproduce the behavior of M on T. By constructing a UTM Turing demonstrated that the problem of simulating any Turing machine was decidable. One of the most interesting implications of this result was a way of building a computer that can solve several problems. This laid out the foundation of the modern general purpose computer. A general purpose computer is thus an *interpreter* of *programs*, i.e. algorithms expressed in some *programming language*.

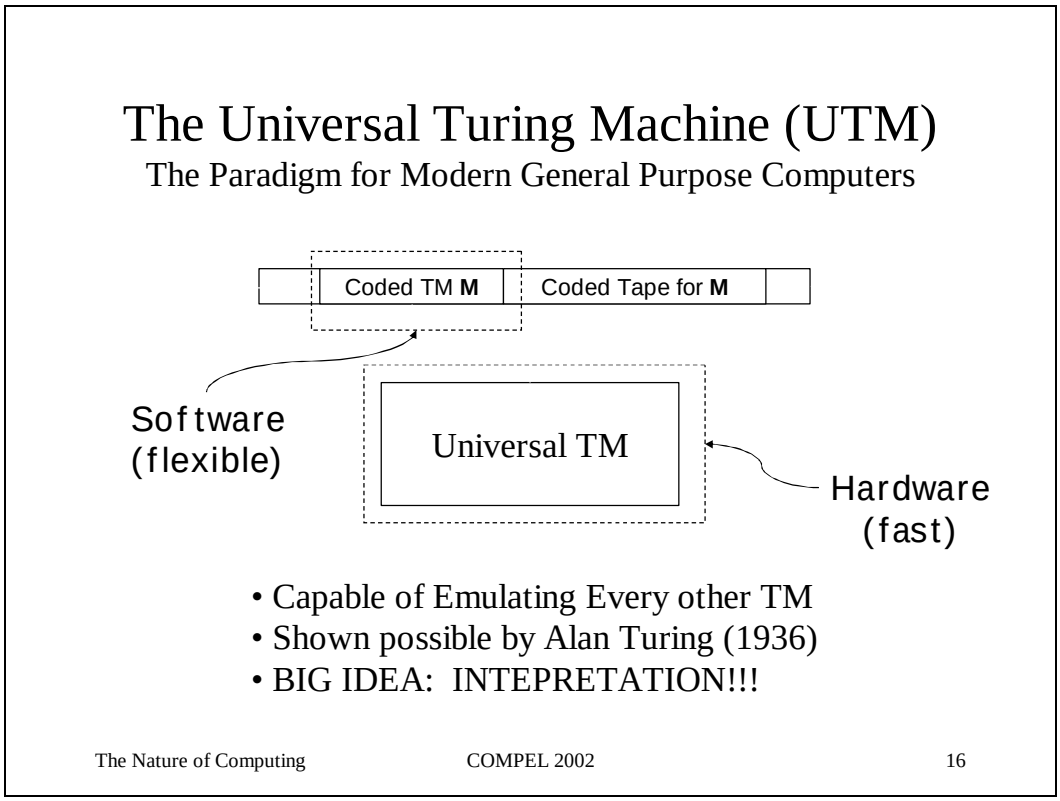

Figure 4. Universal Turing Machines

We can now clearly see that what we normally call programming is the process of encoding an algorithm (a Turing machine) for solving a problem on a general purpose computer. This idea of *programmability* is not only useful at the hardware level, but also at other levels of abstraction. For instance, many modern software applications provide a programmatic interface allowing users to customize or add functionality to the application making it unnecessary to change the software for this purpose. Examples of these types of applications include many editor (e.g. EMACS), mathematical processing (e.g. MATLAB) and circuit simulation packages (e.g. PSPICE).

Turing introduced the UTM to avoid designing a different Turing Machine for each problem. This became crucial to the design of practical computers. The need to reuse hardware for solving different problems became evident. The ideas of programmability and interpretation provided a solution. The same hardware would be designed to interpret a variety of software. Notice however, that the difference between software and hardware is one of level of abstraction. Both are simply different models of computing.

The architecture of the electronic computing instrument proposed by John von Neumann [1] and others (see Figure 5) turned remarkably similar to Turing's UTM. The now called von Neumann architecture organizes a processor into three main components: a central processing unit (CPU), a memory unit, and one or more input/output devices. The CPU is comprised of control unit (a finite state machine) and the data paths. A processor is said to be *universal* if and only if it can emulate a UTM. This doesn't necessarily mean that the processor is complex and has a rich instruction set. As many of you are probably

aware of, this architecture forms the basis of virtually all successful computer architectures to this day.

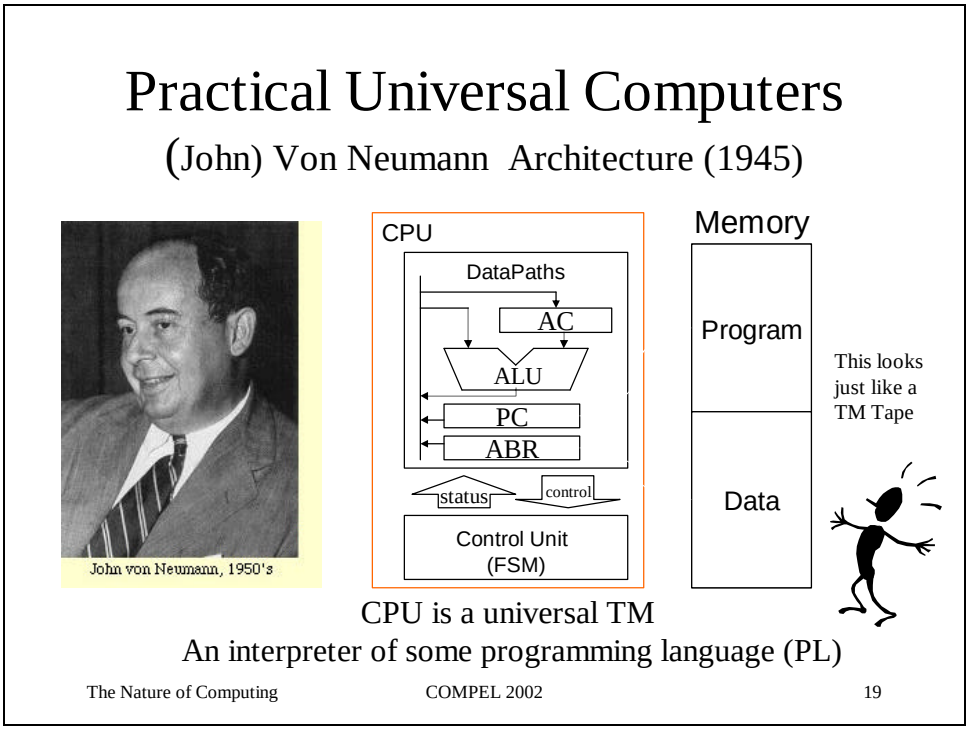

Figure 5. The von Neumann Architecture

Abstraction and Building Blocks

Besides the notions of programmability and interpretation another important idea commonly exploited in Computing is that of *abstraction*. It is my belief that the use of abstraction is common throughout engineering. According to Merriam-Webster's Dictionary abstraction is "the process of considering apart from application to or association with a particular instance".

Another word commonly used to name an abstraction is a *building block*. When designing a building block, one hides irrelevant details and keeps only what is essential to understand the role of the building block in higher and more complex abstractions. Thus, the process of abstraction is inherently recursive in nature. Perhaps what distinguishes Computing from other engineering disciplines that ubiquitously exploit abstraction is that the concept is as central to the discipline as the concept of interpretation itself. After all, the fundamental role of a programming language is to provide the abstractions that allow the most effective expression of general purpose algorithms.

The central nature of the abstraction in Computing has led to the study of a multitude of abstraction mechanisms. This, in turn has led to the development of general principles that guide the design of effective abstractions. These principles, I believe, apply and can

be extremely useful in other engineering branches. I have summarized these guidelines as follows:

- ? Provides a simple and easy to remember contract
- ? The contract hides details irrelevant to the effective application of the abstraction
- ? The contract is general and orthogonal
- ? The contract exhibits a closure property

Example of a successful abstractions often provided by programming languages are functions and types. In this context orthogonality implies that if functions allow parameters to have types, then any type should be permitted. Orthogonality is about avoiding special cases. The closure property is illustrated by languages that allow functions to call other functions. Closure facilitates the design of complex abstraction by combining many simple ones.

Contracts have different names in different disciplines. In electrical engineering they take the form of electrical components or devices. In industrial and chemical engineering they are embodied as industrial processes. In software development they show up as application development interfaces (API's) or abstract data types.

Summary

Computing is a young but rapidly maturing discipline with solid theoretical foundations. It is not surprising that much disagreement remains concerning its scope and extent. Some important Computer Scientists are beginning to view the discipline as evolving towards Engineering. Computing can contribute to Engineering in important ways, and vice versa. Although this symbiosis has already paid considerable dividends, the best is yet to come.

"Computer Science is no more about computers than Astronomy is about telescopes"

E. Dijkstra

References

1. Burks, A. W., Goldstine, H. H., and von Neumann, J. 1963. Preliminary discussion of the logical design of an electronic computing instrument. In Taub, A. H., editor, *John von Neumann Collected Works*, The Macmillan Co., New York, Volume V, 34-79.

2. Church, Alonso.'An Unsolvable Problem of Elementary Number Theory'. *American Journal of Mathematics*, 58, 345-363.

3. Turin, Alan M. 'On Computable Numbers, with an application to the Entscheidungsproblem'. In the *Proceedings of the London Mathematical Society*, ser. 2. vol. 42 (1936-7), pp.230-265; corrections, Ibid, vol 43 (1937) pp. 544-546.