Introduction to Computational Thinking

Author(s)
Silvio Peroni – silvio.peroni@unibo.it – https://orcid.org/0000-0003-0530-4305
Digital Humanities Advanced Research Centre (DHARC), Department of Classical Philology and Italian Studies, University of Bologna, Bologna, Italy

Keywords
Computational thinking; Language; Noam Chomsky; Programming

Abstract

These lecture notes introduce the main concepts related to computational thinking by providing a summary of relevant topics in the areas of Linguistics and Computing in the past 200 years. The historic hero introduced in these notes is Noam Chomsky, considered one of the fathers of modern linguistics. His works have been an enormous impact on the Linguistics domain as well as in the Theoretical Computer Science domain.

Historic hero: Noam Chomsky

Noam Chomsky (shown in Figure 1) is one of the most prominent scholars of the last one hundred years. His contributions and research works have been disruptive and have changed the way scholars have approached several domains in science and humanities. He is described as one of the fathers of modern linguistics with Ferdinand de Saussure, Lucien Tesnière, Luis Hjelmslev, Zellig Harris, Charles Fillmore. He is one of the very first contributors and founders of the cognitive science field[1].

His approach to linguistics has been revolutionary, even if linguists have also debated it. The central aspect of his approach to human language is that mathematics can be used to represent the syntactic structure of a human language. Also, such a structure is biologically determined in

[1] Cognitive science concerns the study of mind and its processes according to several interdisciplinary perspectives, including linguistics, psychology, and artificial intelligence.

all humans. It is already within us since our birth, and it is a unique characteristic of human beings only, and not of other animals. His view of human language is in high contrast with previous ideas about the evolution of languages, which intended a human being with no preconfigured linguistic structure. Thus, the language should have been a matter of learning a radically new endeavour from scratch.

Figure 1. A picture of Chomsky taken in 2011. Picture by Andrew Rusk, source: https://en.wikipedia.org/wiki/Noam_Chomsky#/media/File:Noam_Chomsky_Toronto_2011.jpg.

Among his extensive series of works in linguistics, the classification of formal grammars into a hierarchy of increasing expressiveness is undoubtedly one of his most important contributions, especially in the field of the Theoretical Computer Science and Programming Languages. A formal grammar is a mathematical tool for defining a language, such as English. This tool permits the creation of a finite set of production rules that enable the construction of any valid syntactic sentence.

Each formal grammar is composed of a set of production rules in the form `left-side ::= right-side` (according to the Backus–Naur form, or BNF), where each side can contain one or more symbols of one or more of the following types:

- *terminal* (specified between quotes in BNF), which identifies all the elementary symbols of the language in consideration (such as the nouns, verbs, etc., in English);

- *non-terminal* (specified between angular brackets in BNF), which identifies all the symbols in the formal grammar that can be replaced by a combination of terminal and non-terminal symbols.

Applying a production rule means that the sequence of symbols in the `right-side` part of the rule replaces those specified in the `left-side` part. The rewrite process done by the application of such production rules starts from an initial non-terminal symbol. In particular, one applies the production rules until he/she gets a sequence of terminal symbols only. For instance, the production rules `<sentence> ::= <pronoun> "write"`, `<pronoun> ::= "I"` and `<pronoun> ::= "you"` allows one to create all the two-word sentences having either the first or the second person singular pronoun accompanied by the verb write (e.g. “I write”). In addition, each formal grammar must specify a *start symbol*, that must be non-terminal.

The hierarchy proposed by Chomsky provides a way for describing formally the relations that may exist between different grammars in terms of the possible syntactic structures that such grammars are able to generate. In practice, they are characterised by which kinds of symbols one can use in the `left-side` and `right-side` parts of production rules. These grammars are listed as follows, from the less expressive to the most expressive – we use letters from the Greek alphabet for indicating any possible combination of terminal and non-terminal symbols, including the empty symbols (usually represented by ε):

- *regular grammars* – form of production rules: `<non-terminal> ::= "terminal"` and `<non-terminal> ::= "terminal" <non-terminal>`. Example:
```
<sentence> ::= "I" <verb>
<sentence> ::= "you" <verb>
<verb> ::= "write"
<verb> ::= "read"
```
- *context-free grammars* – form of production rules: `<non-terminal> ::=` γ. Example:
```
<sentence> ::= <nounphrase> <verbphrase>
<nounphrase> ::= <pronoun>
<nounphrase> ::= <noun>
<pronoun> ::= "I"
<pronoun> ::= "you"
<noun> ::= "book"
<noun> ::= "letter"
<verbphrase> ::= <verb>
<verbphrase> ::= <verb> "a" <noun>
<verb> ::= "write"
<verb> ::= "read"
```
- *context-sensitive grammars* – form of production rules: α `<non-terminal>` β `::=` α γ β. Example:
```
<sentence> ::= <noun> <verbphrase>
<sentence> ::= <subject pronoun> <verbphrase>
```

```
"I" <verb> <object pronoun> ::= "I" "love" <object pronoun>
"I" <verb> <noun> ::= "I" "read" "a" <noun>
<verbphrase> ::= <verb> <noun>
<verbphrase> ::= <verb> <object pronoun>
<subject pronoun> ::= "I"
<subject pronoun> ::= "you"
<object pronoun> ::= "me"
<object pronoun> ::= "you"
<noun> ::= "book"
<noun> ::= "letter"
```

- *recursively enumerable grammars* – form of production rules: α ::= β (no restriction applied). Example:

```
<sentence> ::= <subject pronoun> <verbphrase>
"I" <verb> <object pronoun> ::= "I" <verb> "you"
"I" <verb> <noun> ::= "I" "read" "a" "book"
<verbphrase> ::= <verb> <noun>
<verbphrase> ::= <verb> <object pronoun>
<subject pronoun> ::= "I"
<subject pronoun> ::= "you"
<object pronoun> ::= "me"
<object pronoun> ::= "you"
<verb> ::= "love"
<verb> ::= "hate"
```

What is a computer?

The English Oxford Living Dictionary defines the term *computer* as an "electronic device which is capable of receiving information (data) in a particular form and of performing a sequence of operations [...] to produce a result". However, the original definition of the same term, in use from the 17th century, is slightly different. It refers to someone "who computes" or to a "person performing mathematical calculations" – from Wikipedia. In these lecture notes, when we use the term "computer", we always consider the most generic definition: any *information-processing agent* (i.e. a machine or a person acting mechanically if appropriately instructed) [Nardelli, 2019] that can make calculations and to produce some output starting from input information.

Human computers, i.e. group of people who have to undertake long calculations for specific experiments or measurements, have been used several times in the past. For instance, in Astronomy, human computers have been used for calculating astronomical coordinates of non-terrestrial things – such as the calculation of passages of Halley's Comet by Alexis Claude Clairaut and colleagues. Napoleon Bonaparte used human computers, as well. He imposed the

creation of mathematical tables to convert from the old imperial system of measurements to the new metric system [Campbell-Kelly, 2009] [Roegel, 2010].

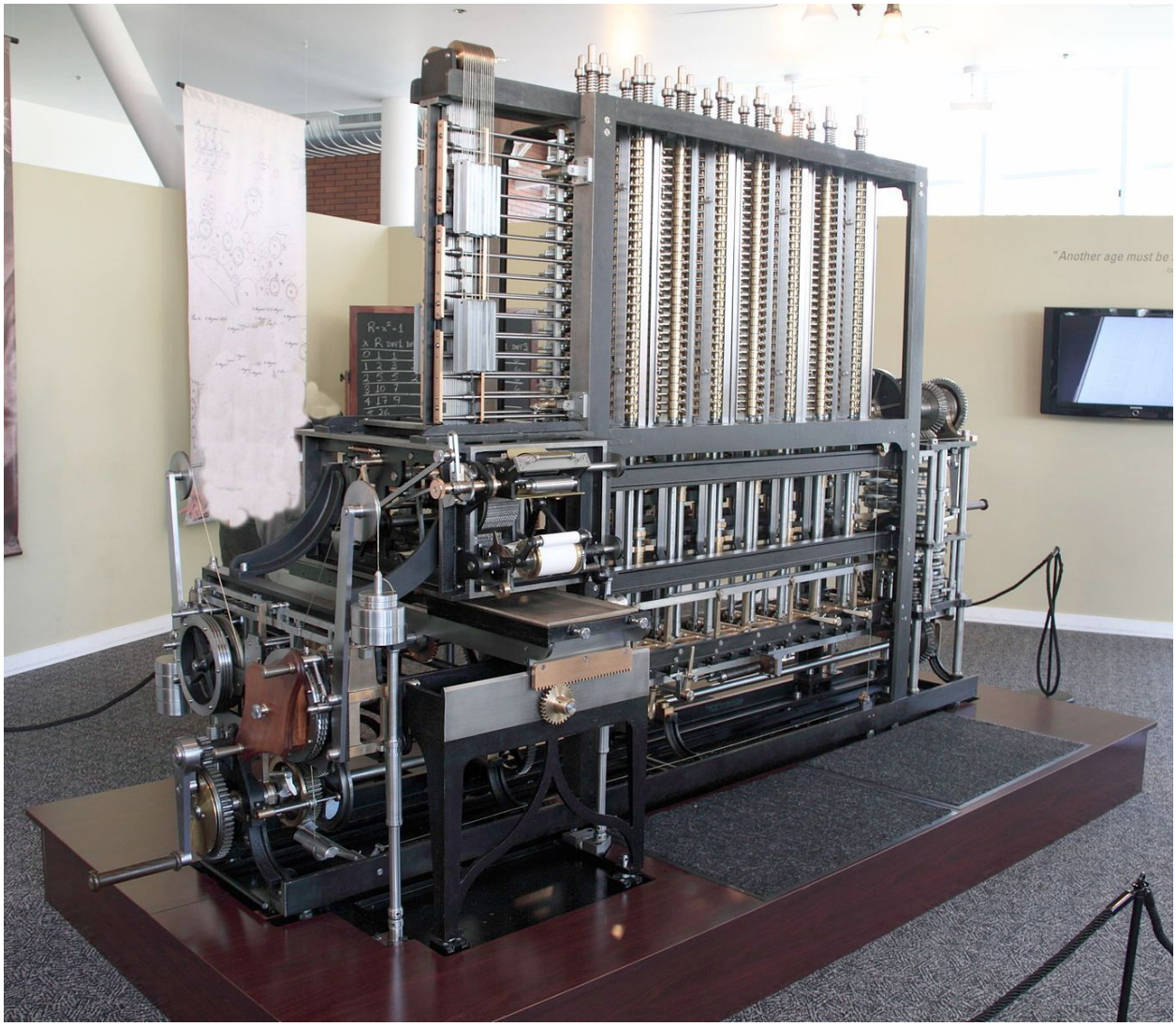

Figure 2. Babbage Difference Engine No. 2 built at the Science Museum (London) and displayed at the Computer History Museum in Mountain View (California). Picture by Allan J. Cronin, source: https://commons.wikimedia.org/wiki/File:Difference_engine.JPG.

In 1822, Charles Babbage, understanding the complexity of doing all these calculations by hand without introducing any error, started the development of an incredible machine. This machine was called the Difference Engine, a mechanical calculator, shown in Figure 2. It aimed at addressing similar tasks that were run by human computers, but in a way that was automatic, faster, and error-free. Babbage was able to build just a partial prototype of this machine, and, after the first enthusiasm, he was struggled by the limited flexibility that it offered. The Difference Engine was not a programmable machine. It was able to compute only a fixed set of operations on the inputs specified physically by changing specific configurations of the machine.

In order to address these limitations, in 1837, Babbage started to think a new machine, the Analytical Engine, summarised in Figure 3. No prototypes of this machine were built by Babbage. However, by using it, a user could create any possible procedural calculation, making it the very first mechanical general-purpose computer in history. In contrast to its predecessor, the Analytical Engine was able to receive the input instructions and data using punched cards. The use of such cards avoided the users to make any physical manipulation of the machine to get it working.

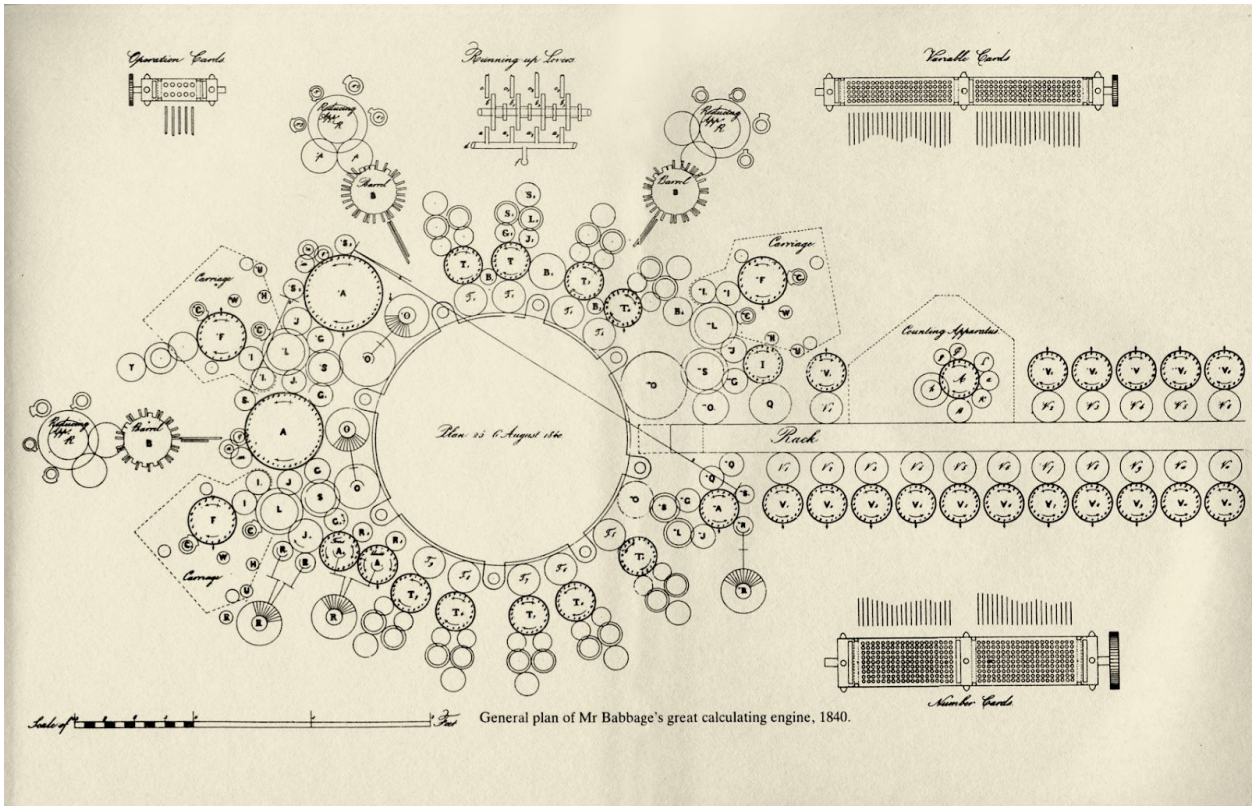

Figure 3. A sketch by Babbage that describes the architecture of the Analytical Engine. Source: The Analytical Engine: 28 Plans and Counting, Computer History Museum.

The ideas presented in the Analytical Engine were developed in a physical machine only one century later. The computing technology has had a drastic change as a consequence of World War II. Military research ordered the construction of several calculators. For instance, the Bombe (1940), designed by Alan Turing, was the main instrument used by a group of people living in the secret British military camp at Bletchley Park to decipher German's communications encrypted through the Enigma machine.

The Bombe was a handy and efficient machine. However, it was still partially based on mechanical components, and it allowed their users only a specific task. Although, if it was crucial from a purely historical point of view. The first fully-digital computer, as envisioned by

Babbage with his Analytical Engine, was developed in the United States only a few years later, in 1946. It was the Electronic Numerical Integrator and Computer (ENIAC), shown in Figure 4, that was programmable through patch cables and switches. This invention represents one of the most important milestones of the history of computers - the fixed point in time that generated all modern computers.

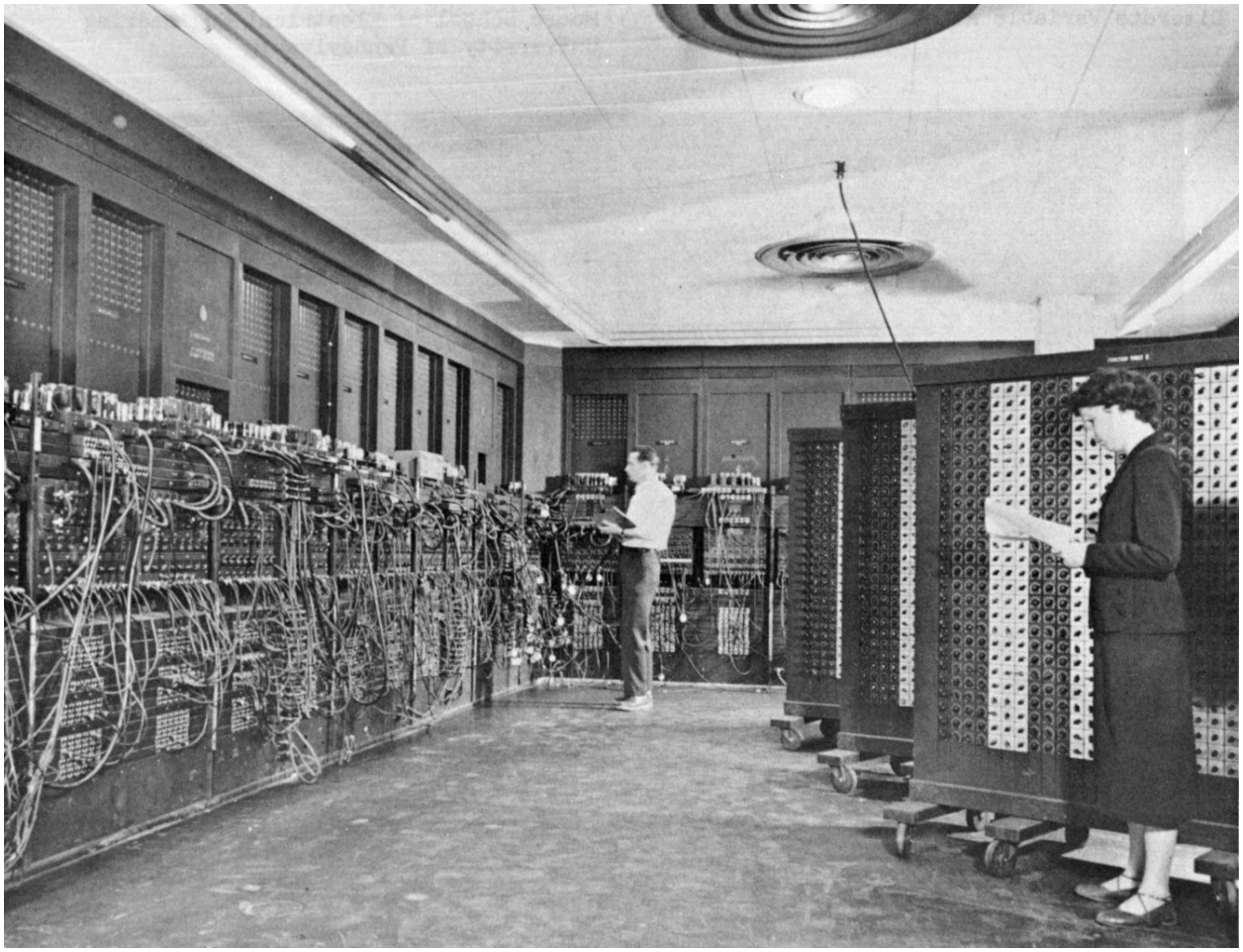

Figure 4. A picture of the ENIAC in the Ballistic Research Laboratory (Maryland). Source: https://en.wikipedia.org/wiki/ENIAC#/media/File:Eniac.jpg.

Natural languages vs programming languages

There is an aspect of *computers* (either humans or machines) that has not directly tackled yet: which mechanism can we use for asking them to address a particular task? relates to the particular communication channel we want to adopt. Considering human computers, we can use the natural language (e.g. English) to instruct them in addressing specific actions.

A natural language is just an ordinary language (e.g. English), either written or oral, that has evolved naturally in humans, usually without specific and premeditated planning. As we know

them, natural languages have the advantage (and, on the other hand, disadvantage) of being so expressive that particular instructions provided by using them can sound ambiguous. Consider, for instance, the sentence “shot an elephant in your pyjamas”. Does it mean one has to shot an elephant (with a rifle) while wearing pyjamas? Or that one should shot an elephant (with a water gun) drawn in pyjamas? We could come up with specific (e.g. social) conventions that allow us to restrict the possible meaning of a situation. In the previous example, the fact that one is in a bedroom and is not living in Gabon is enough for disambiguating the sentence. While natural languages are not formal by definition, several studies in Linguistics try to provide their formalisation using some mathematical tool, e.g. [Bernardi, 2002]. Even if one can provide a formal definition of a natural language, its intrinsic vagueness is still present in the language itself. For instance, one cannot use mathematics (or, better, logics) for removing (all) the ambiguities from a natural language.

Programming languages, on the contrary, are formal-born languages. They oblige to specific syntactic rules. Such rules avoid possible ambiguous statements, mainly by restricting the expressiveness of the language. Therefore, all the sentences in such language are conveying just one possible meaning. Usually, they are based on context-free grammars, according to the Chomsky's classification introduced in Section "Historic hero: Noam Chomsky". Also, they can have a large degree of abstraction. In particular, we can distinguish three macro-sets of programming languages:

- *machine language* is a set of instructions that can be executed directly by the central processing unit (CPU) of an electronic computer. For instance, the following code is the binary executable code (i.e. a sequence of 0 and 1) defining a function (i.e. a kind of tool that takes some inputs and produces some output) for calculating the n^{th} Fibonacci number:

  ```
  10001011010101000010010000001000100000111111010000000000111011
  10000011010111000000000000000000000000000000000000011000011100000
  11111110100000001001110111000001101011100000000001000000000000
  00000000001100001101010011101110110000000100000000000000000000
  00001011100100000001000000000000000000000000010001101000001000000
  11001100000111111101000000011011101100000011110001011110110011 0
  00100111000001010010101110101111110001010110111000011
  ```

- *low-level programming languages* are languages that provide one abstraction level on top of the machine language. Thus it allows one to write programs in a way that is more intelligible to humans. The most popular language of this type is Assembly. Even if it introduces humanly recognisable symbols, typically one line of assembly code represents one machine instruction in machine language. For instance, the same function for calculating the n^{th} Fibonacci number is defined in Assembly as follows:

  ```
  fib:
      mov edx, [esp+8]
      cmp edx, 0
      ja @f
      mov eax, 0
  ```

```
    ret

    @@:
    cmp edx, 2
    ja @f
    mov eax, 1
    ret

    @@:
    push ebx
    mov ebx, 1
    mov ecx, 1

    @@:
        lea eax, [ebx+ecx]
        cmp edx, 3
        jbe @f
        mov ebx, ecx
        mov ecx, eax
        dec edx
    jmp @b

    @@:
    pop ebx
    ret
```

- *high-level programming languages* are languages which are characterised by a strong abstraction from the specifiability of the machine language. In particular, it may use natural language words for specific constructs, to be easy to use for and to understand by humans. Generally speaking, the more the abstraction from the low-level programming languages is provided, the more understandable the language is. For instance, in the following example, we show how to use the C programming language for implementing the same function as before:

```
unsigned int fib(unsigned int n) {
    if (n <= 0)
        return 0;
    else if (n <= 2)
        return 1;
    else {
        unsigned int a,b,c;
        a = 1;
        b = 1;
        while (1) {
            c = a + b;
```

```
                if (n <= 3) return c;
                a = b;
                b = c;
                n--;
            }
        }
    }
```

We can also apply an additional level of abstraction to the previous example. For instance, we can provide instructions in a natural language for enabling a human-computer to execute the function mentioned above. Of course, none of the macro-sets mentioned above includes the natural language. However, the natural language would allow us to see how we can use even a more abstract language for instructing someone else to execute the same operation. In particular, a possible natural language description of the Fibonacci function could be:

`The function for calculating the n^th Fibonacci number takes as input an integer "n". If "n" is less than or equal to 0, then 0 is returned as a result. Otherwise, if "n" is less than or equal to 2, then 1 is returned. Otherwise, in all the other cases, associate the value "1" to two distinct variables "a" and "b". Then, repeat the following operations indefinitely. Set the variable "c" as the sum of "a" plus "b". If "n" is less than or equal to "3" then return "c", otherwise assign the value of "b" to "a" and the value of "c" to "b", and finally decrease the value of "n" by 1 before repeating.`

While the previous natural language definition maps perfectly the function defined in the machine binary code introduced above, other possible implementations of such Fibonacci function are possible. One of the most famous that uses the concept of *recursion* could be:

`The function for calculating the n^th Fibonacci number takes as input an integer "n". If "n" is less than or equal to 0, then 0 is returned as a result. Otherwise, if "n" is equal to 1, then 1 is returned. Otherwise, return the sum of the same function with "n-1" as input and still the same function with "n-2" as input.`

Abstraction is the key

We often say that we *program* a computer – where the word computer there refers to an electronic computer. However, according to the definition we have provided in this document, computers can be both humans and machines. Thus, the verb *to program* is not very well suited when we refer to human computers – we cannot program a person, can we? In this latter case, we usually say that we *talk with* a person to instruct her to execute specific actions, through a

(natural) language used as a communication channel. Thus, we think that, in this context, we should use the same verbs, i.e. *to talk* and *to instruct*, even when we refer to an electronic computer. Writing a program is precisely that: communicating to an electronic computer in a (formal) language that such an electronic computer and the human instructor can both understand [Papert, 1980].

First, we need to agree on the language to use for the communication between us and a computer (either human or machine). Then, we can start to think about possible instructions that, if followed systematically, can return the expected result to a particular problem. In order to reach this goal, we (even unconsciously) try to figure out possible solutions to such a problem by comparing it with possible other recurring situations that happened in the past. The idea is to find some patterns that depict a possible solution for a set of abstractly-homogeneous situations. Once found, the solution can be reused to reach our goal if it has been successful in the past. For instance, let us consider the actions that we perform at a post office. Some actions are similar to those we perform when we wait for our turn to play with a slide in the playground – as shown in Figure 5.

Figure 5. Two pictures that depict the same situation, i.e. queuing, in two different contexts: a playground (left) and a post office (right). Left picture by Prateek Rungta, source: https://www.flickr.com/photos/rungta/4409560365/. Right picture by Rain Rabbit, source: https://www.flickr.com/photos/37996583811@N01/6158491035/.

Considering the situations and contexts mentioned above, we call *computational thinking* a particular approach to "solving problems, designing systems and understanding human behaviour that draws on concepts fundamental to computing" [Wing, 2008]. Computational thinking is the thought processes that are involved when we formulate a problem and express the solution by using a language that a computer (either human or machine) can understand and execute.

Jeannette Wing provides an additional definition for clarifying what computational thinking is about [Wing, 2008]:

> Computational thinking is a kind of analytical thinking. It shares with mathematical thinking in the general ways in which we might approach solving a problem. It shares with engineering thinking in the general ways in which we might approach designing and evaluating a large, complex system that operates within the constraints of the real world. It shares with scientific thinking in the general ways in which we might approach understanding computability, intelligence, the mind and human behaviour.

It is important to stress that computational thinking is not a new subject at all. Instead, it focuses on specific aspects concerning computer science: the founding principles and methods instead of those merely related to particular tools and systems that people (often and erroneously) associate to any computer scientist, e.g. the electronic computer [Nardelli, 2019].

The primary notion related to computational thinking is *abstraction*: the "process of leaving out of consideration one or more properties of a complex object [...] by extracting common features from specific examples" [Kramer, 2007]. As highlighted in Figure 5, the skill of abstracting situations and notions into symbols is crucial for automating the execution of tasks using a computer that is responsible for interpreting such abstractions. However, usually, we use these abstractions unconsciously. One of the goals of computational thinking is to *reshape* the abstractions we have ingested as a consequence of our life experiences – that we are often unconsciously reusing. Thus, being again fully conscious of such abstractions, we can use an appropriate language for making them understandable to a computer, in order to automatise them.

The final goal of computational thinking is to make one think like a computer scientist. And it applies either to academic research (including in the Humanities, e.g. see the use of computational models and techniques in History research [Au Yeung, Jatowt, 2011] [Mullen, 2018] [Preiser-Kapeller, 2015]) or in real-life tasks. No one is saying that this way of thinking is the right one, of course. However, it surely offers a complementary tool to describe reality [Nardelli, 2019]. In the future, computational thinking "will be an integral part of childhood education" [Wing, 2008]. It will affect the way people think and learn and "the way other learning takes place" [Papert, 1980].

Exercises

1. What are all the possible sentences that one can produce by using the regular grammar introduced in Section "Historic hero: Noam Chomsky"?
2. What is the result of applying the latest natural language definition of the Fibonacci function in Section "Natural languages vs programming languages" using "7" as input?
3. Write down two objects or situations that are referring to the same pattern if analysed from an abstract point of view, as introduced in Section "Abstraction is the key". What features do they have in common?

Acknowledgements

The author wants to thank some of the students of the course, Severin Josef Burg and Yordanka Stoyanova, for having suggested corrections and improvements to the text of these lecture notes.

References

Au Yeung, C., Jatowt, A. (2011). Studying how the past is remembered: towards computational history through large scale text mining. In Proceedings of the 20th ACM international conference on Information and knowledge management (CIKM 2011): 1231-1240. DOI: https://doi.org/10.1145/2063576.2063755 - also available at http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.422.1205&rep=rep1&type=pdf (last visited 13 October 2019)

Bernardi, R. (2002). The Logical Approach in Linguistics. In Reasoning with Polarity in Categorial Type Logic. Ph. D. Thesis, Utrecht University. http://disi.unitn.it/~bernardi/Papers/thesis-chapter1.pdf (last visited 13 October 2019)

Campbell-Kelly, M. (2009). The Origin of Computing. Scientific American, 301 (September 2009): 62-69. DOI: https://doi.org/10.1038/scientificamerican0909-62 - also available at http://www.cs.virginia.edu/~robins/The_Origins_of_Computing.pdf (last visited 13 October 2019)

Kramer, J. (2007). Is abstraction the key to computing? Communications of the ACM, 50 (4): 36.42. DOI: https://doi.org/10.1145/1232743.1232745 - also available at https://www.ics.uci.edu/~andre/informatics223s2007/kramer.pdf (last visited 13 October 2019)

Mullen, L. A. (2018). Computational Historical Thinking: With Applications in R. https://dh-r.lincolnmullen.com (last visited 13 October 2019)

Nardelli, E. (2019). Do we really need computational thinking? Communications of the ACM, 62 (2): 32-35. DOI: https://doi.org/10.1145/3231587

Papert, S. (1980). Introduction: Computer for Children. In Mindstorms: children, computers, and powerful ideas: 3-18. New York, USA: Basic Books, Inc. ISBN: 0-465-04627-4. Full text available at http://worrydream.com/refs/Papert%20-%20Mindstorms%201st%20ed.pdf (last visited 13 October 2019)

Preiser-Kapeller, J. (2015). Calculating the Middle Ages? The Project "Complexities and Networks in the Medieval Mediterranean and Near East" (COMMED). Medieval Worlds, 2015.2: 100-127. DOI: https://doi.org/10.1553/medievalworlds_no2_2015s100

Roegel, D. (2010). The great logarithmic and trigonometric tables of the French Cadastre: a preliminary investigation. Research Report. INRIA. https://hal.inria.fr/inria-00543946

Wing, J. M. (2008). Computational thinking and thinking about computing. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences, 366 (1881): 3717. DOI: https://doi.org/10.1098/rsta.2008.0118 - also available at https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2696102/ (last visited 13 October 2019)