Introduction to Computational Thinking

Author(s)
Silvio Peroni – silvio.peroni@unibo.it
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 the father of modern linguistics. His works have been a huge impact in 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 has been described as the father of the modern linguistics, and he is one of the very first contributors and founders of the cognitive science field – that concerns the study of mind and its processes according to several interdisciplinary perspectives, including linguistics, psychology, and artificial intelligence.

His approach to linguistics has been really revolutionary. The main aspect of his approach to human language is that the principle underlying its structure is biologically determined in all humans – it is already within us since our birth – and, as such, it is a unique characteristic that has been evolved in time and that is shared by human beings only, and not by other animals. His view of human language is in great contrast with previous ideas about the evolution of languages, that want each human being as an empty-bucket mind, without any preconfigured linguistic structure, and thus the language should be 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 large 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 (e.g. a natural language, such as English) according to a finite set of production rules, that allows one to construct any syntactic valid sentence of such language.

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 words, 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.

In principle, the main idea of the application of a production rule is that the sequence of symbols in the `left-side` part can be replaced with those ones specified in the `right-side` part until the sequence includes only terminal symbols. 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 term *computer* is currently used to identify 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" – from the English Oxford Living Dictionary. However, the original definition of the same term, in use from the 17th century, is slightly different. In fact, 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 agent (i.e. anything that can act if appropriately instructed, such as a person or a machine) that is able to 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 certain 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 the Halley's Comet by Alexis Claude Clairaut and colleagues. Similarly, human computers have been used also for addressing Governmental issues, e.g. when Napoleone Bonaparte imposed the creation of mathematical tables for converting the values from the old imperial system of measurements to the new metric system [Campbell-Kelly, 2009] [Roegel, 2010].

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, called the Difference Engine (a mechanical calculator, shown in Figure 2), 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. In fact, the Difference Engine was not a programmable machine and, thus, it was able to compute only a fixed set of operations on the inputs specified physically by changing specific configurations of the machine.

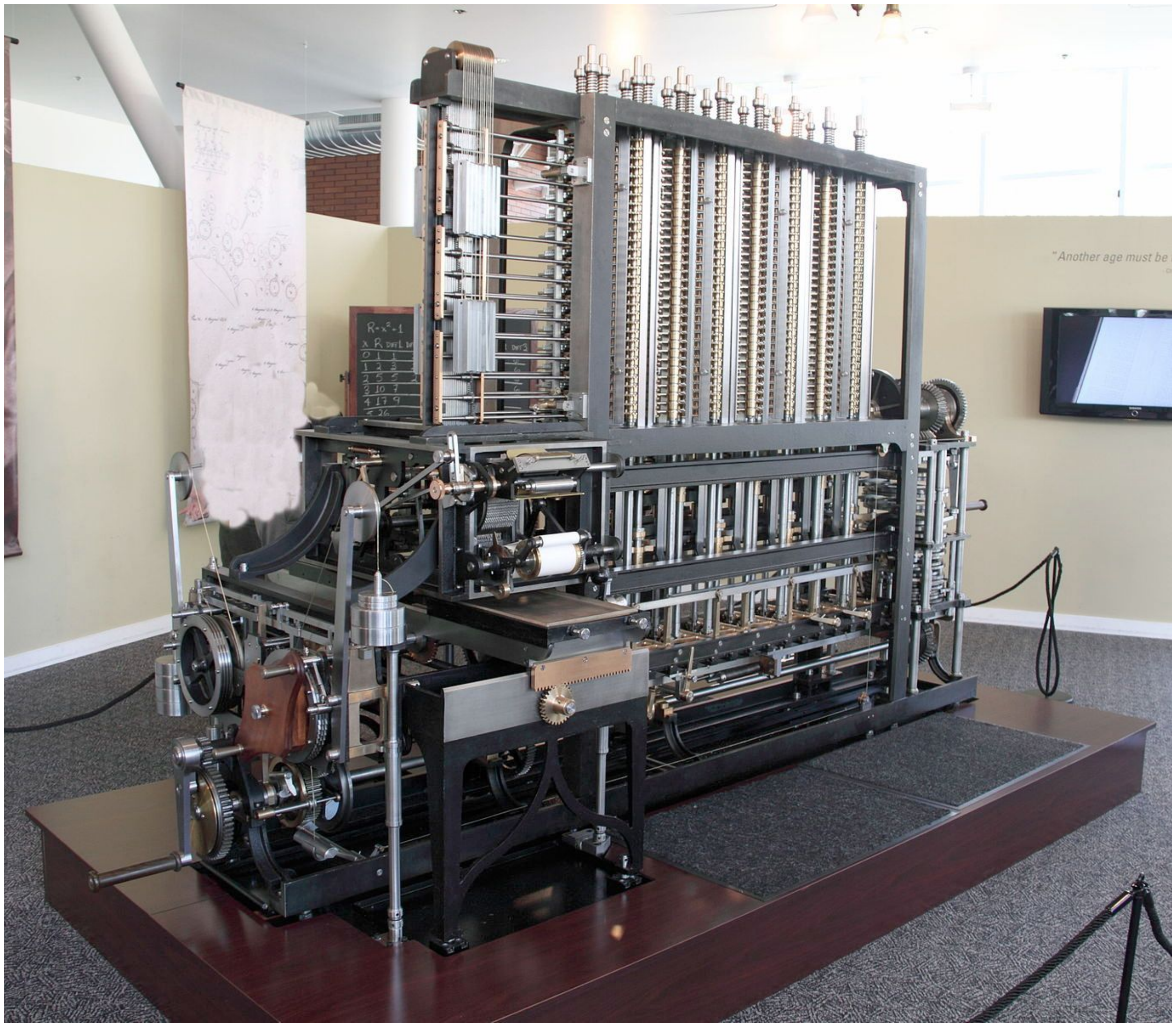

Figure 2. Babbage Difference Engine No. 2 built at the Science Museum (London) and displayed at the Computer History Museum in Mountain View (California). Picture by Allan J. Cronin, source: https://commons.wikimedia.org/wiki/File:Difference_engine.JPG.

In order to address these limitations, in 1837 Babbage started to think a new machine, the Analytical Engine, summarised in Figure 3. While no prototypes of this machine were built by Babbage, in principle it would have enabled a user to 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 by means of punched cards, without obliging its users to make any physical manipulation of the machine to get it working.

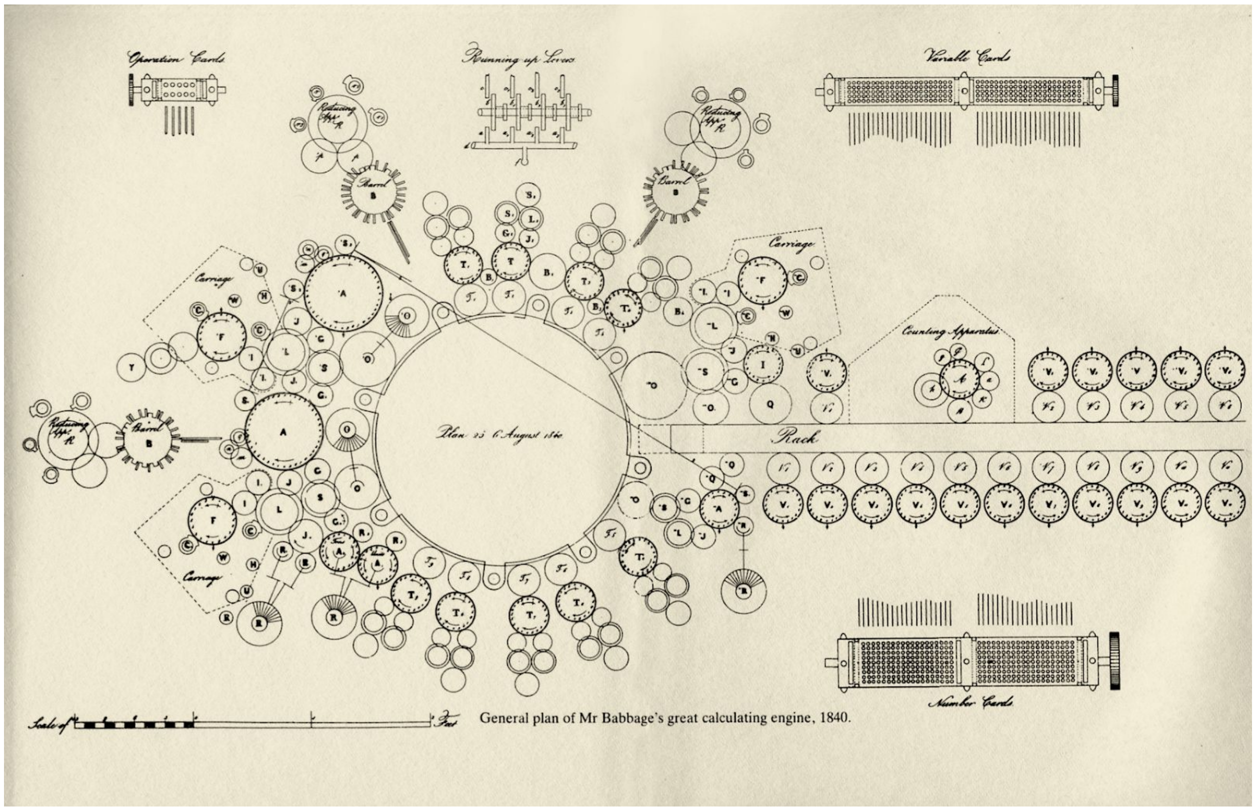

Figure 3. A sketch by Babbage that describes the main architecture of the Analytical Engine. Source: The Analytical Engine: 28 Plans and Counting, Computer History Museum.

More than one century was needed for seeing the ideas presented in the Analytical Engine developed in some physical machine. In fact, the computing technology had a drastic change as a consequence of the World War II. Several calculators were built for military reasons, such as the Bombe (1940), designed by Alan Turing, which was the main instrument that allowed a group of people, living in the secret British military camp at Bletchley Park, to decipher German's communications encrypted by means of the Enigma machine.

While the Bombe was a very effective and efficient machine, it was still partially based on mechanical components, and it allowed their users only a specific task, even 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 by means of patch cables and switches. This invention represents one of the most important milestones of the history of computers - the fixed point in time where all the modern computers have been then generated.

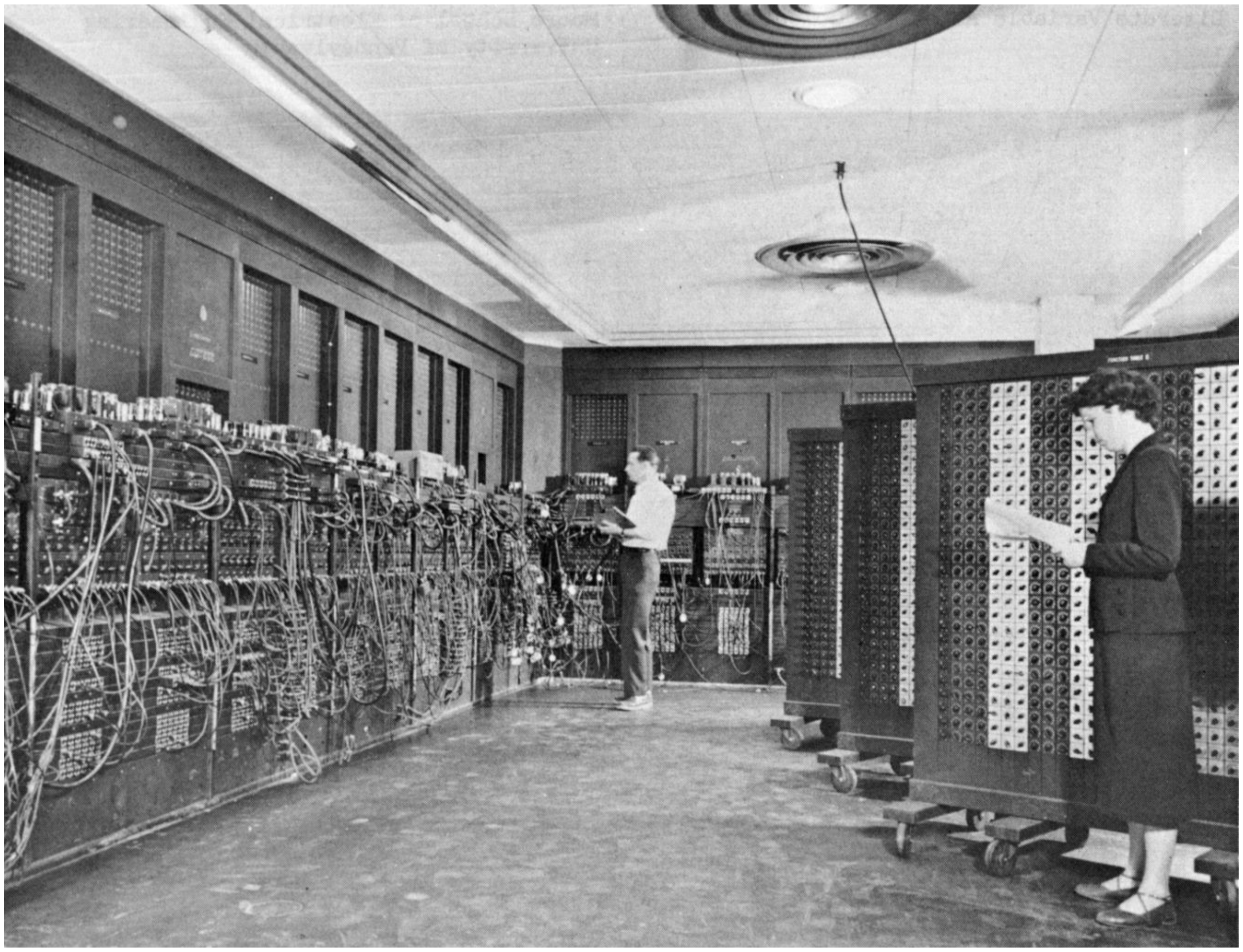

Figure 4. A picture of the ENIAC in the Ballistic Research Laboratory (Maryland). Source: https://en.wikipedia.org/wiki/ENIAC#/media/File:Eniac.jpg.

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? The way to approach this issue is extremely linked 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 a 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 that you have to shot an elephant (with a rifle) while you are wearing a pyjamas, or that you should shot an elephant (with a water gun) that is drawn in your pyjamas? However, often, we could come up with specific (e.g. social) conventions that would allow us to restrict the possible meaning of a piece of information – in the previous example, the fact you are in your bedroom and you are 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 by means of some mathematical tool, e.g. [Bernardi, 2002]. It is worth mentioning that, even if one can provide a formal definition of a natural language, its intrinsic vagueness are still present in the language itself – i.e. 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, and they are usually developed in a way that avoids possible ambiguous statements (mainly by restricting their expressiveness), so as that all the sentences in such language are clearly conveying just one possible meaning. They are usually based on context-free grammars, according to the Chomsky's classification introduced in Section "Historic hero: Noam Chomsky", and can have a large degree of abstraction. In particular, they are mainly grouped in three macro-sets:

- *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 nth Fibonacci number:
```
100010110101010000100100000010001000001111111010000000000111011
100000110101110000000000000000000000000000000000011000011100000
111111101000000010011101110000011010111000000000010000000000000
000000000001100001101010011101110110000000100000000000000000000
000010111001000000010000000000000000000000000100011010000010000 0
110011000001111111010000000110111011000000111100010111101100110
001001110000010100101011101011111100010101101111000011
```
- *low-level programming languages* are languages that provide one abstraction level on top of the machine language, and thus it allows one to write programs in a way that is more intelligible to humans. The most famous language of this type is Assembly. Even if it introduces humanly understandable symbols, typically one line of assembly code represents one machine instruction in machine language. For instance, the same function for calculating the nth 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 construct, so as 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, in order to provide natural language instructions for enabling a human computer, this time, to perform the operation depicted by the function for calculating the Fibonacci number. While natural language is not included in any of the aforementioned sets defining programming languages, it would allow us to see how we can use even a more abstract language for instructing someone else in performing the same operation. In particular, a possible natural language description of the Fibonacci function is shown as follows:

```
The function for calculating the nth Fibonacci number takes as input an integer “n”. If “n” is less than or equal to 0, then 0 is returned as 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 indefinitely the following operations: 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*, is introduced as follows:

```
The function for calculating the nth Fibonacci number takes as input an integer “n”. If “n” is less than or equal to 0, then 0 is returned as 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 really program a person, can we? In this latter case, in fact, we usually say that we *talk with* a person to instruct her to execute specific actions, by means of a particular (natural) language that is 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. Basically speaking, writing a program is exactly that: communicating to an electronic computer in a (formal) language that it and the human instructor can both understand [Papert, 1980].

Once agreed on which language to use for the communication between us and a computer (either human or machine), we should start to think about possible instructions that, if followed systematically, can return the expected result to a certain problem. In order to reach this goal, we (even unconsciously) try to figure out possible solutions to such given 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, so as to reuse the same strategy for reaching our goal, if that strategy has been successful in the past. For instance, it could be possible that some of the actions that we perform at a post office are quite similar to those ones we performed when we were a child waiting for our turn to play with a slide in the playground – as shown in Figure 5.

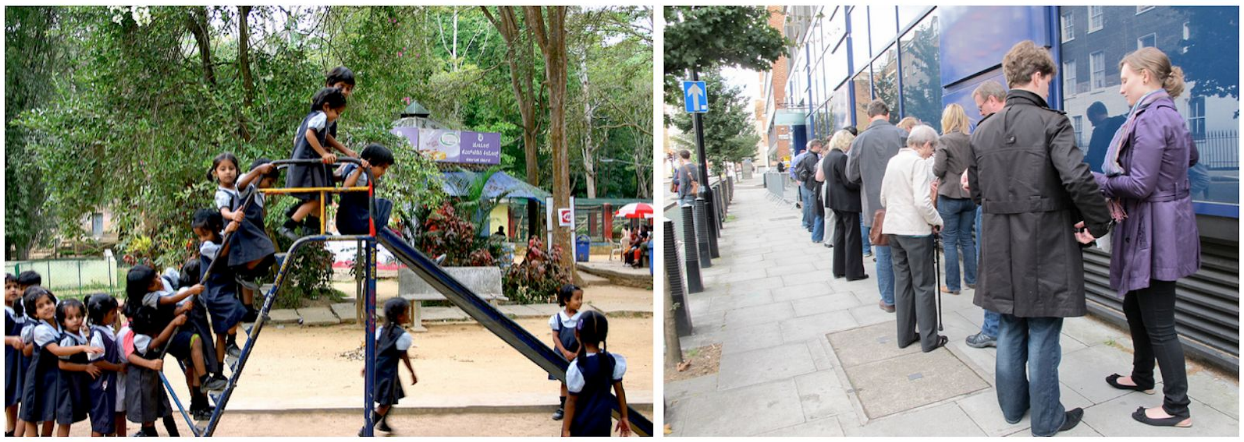

Figure 5. Two pictures that depict the same situation, i.e. queuing, in two different contexts: a playground (left) and a post office (right). Left picture by Prateek Rungta, source: https://www.flickr.com/photos/rungta/4409560365/. Right picture by Rain Rabbit, source: https://www.flickr.com/photos/37996583811@N01/6158491035/.

According to the aforementioned situations and context, 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] – where with the word *computing* we mean *calculating*. 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, thus, 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.

The main notion related to computational thinking is *abstraction*. As already highlighted in the aforementioned example in Figure 5, the skill of abstracting situations and notions into symbols is crucial in order to automatise the execution of certain tasks by means of a computer that is responsible to interpret 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 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.

Broadly speaking, the final goal of computational thinking is to make one think like a Computer Scientist, even when dealing with common tasks. In the future, computational thinking will become a course like Mathematics and Physics, it "will be an integral part of childhood education" [Wing, 2008], and 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 can be produced 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 situations that are actually referring to the same pattern if analysed from an abstract point of view, as introduced in Section "Abstraction is the key". What are their common features?

Acknowledgements

I would like to thank a of the student of the course, Yordanka Stoyanova, for having suggested corrections and improvements to the text of these lecture notes.

References

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 May 30, 2017)

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 20 August 2017)

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 20 August 2017)

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. https://doi.org/10.1098/rsta.2008.0118