Minds and Machines (2024) 34:S139–S170
https://doi.org/10.1007/s11023-023-09650-0

A Pragmatic Theory of Computational Artefacts

Alessandro G. Buda[1] · **Giuseppe Primiero**[1,2]

Received: 5 July 2022 / Accepted: 7 October 2023 / Published online: 22 November 2023

Abstract
Some computational phenomena rely essentially on pragmatic considerations, and seem to undermine the independence of the specification from the implementation. These include software development, deviant uses, esoteric languages and recent data-driven applications. To account for them, the interaction between pragmatics, epistemology and ontology in computational artefacts seems essential, indicating the need to recover the role of the language metaphor. We propose a User Levels (ULs) structure as a pragmatic complement to the Levels of Abstraction (LoAs)-based structure defining the ontology and epistemology of computational artefacts. ULs identify a flexible hierarchy in which users bear their own semantic and normative requirements, possibly competing with the logical specification. We formulate a notion of computational act intended in its pragmatic sense, alongside pragmatic versions of implementation and correctness.

Keywords Computational artefacts · Pragmatics · Correctness · Levels of Abstraction · User levels

1 Introduction

The current literature in the Philosophy of Computer Science presents two standard approaches to computational artefacts (Angius et al., 2021).

According to the first one, they have a dual nature, abstract and physical, characterized by structural and functional implemented properties (Kroes, 2009). Such duality

✉ Alessandro G. Buda
alessandrogiuseppe.buda@studenti.unimi.it

Giuseppe Primiero
giuseppe.primiero@unimi.it

1 PhilTech Research Center, Department of Philosophy, University of Milan, Via Festa del Perdono 7, 20122 Milan, MI, Italy

2 Logic, Uncertainty, Computation and Information Lab, Department of Philosophy, University of Milan, Via Festa del Perdono 7, 20122 Milan, MI, Italy

is expressed standardly by the distinction between hardware and software, but it is notoriously weak in light of the historical evolution, as once programs were executed by physical actions on levers, buttons, and wire. Moreover, the associated notion of implementation as the realization of a specification is not unique: it can be seen as semantic interpretation, forcing us to accept a continuous shift of syntax and semantics (Rapaport, 1999); or as the relationship between abstract and concrete (Turner, 2014).

According to the second, more recent approach (Primiero, 2016), computational artefacts are equipped with an ontology of several levels of abstraction (LoA), including layers otherwise ignored: the designer and its intention; the algorithm and its solution to the problem posed; the high-level vs. low-level language and their instruction-order distinction; finally, the user. Implementation is here generalised to a pair composed of an epistemological construct and an ontological domain for each level of abstraction (Primiero, 2019).

In both approaches, the notion of correctness is crucial and is always determined by the specification. In the LoAs approach, distinct notions of correctness can be identified: functional, procedural, and executional, all required for a correctly functioning physical computing system (Primiero, 2019). Both approaches provide logical-mathematical accounts of the relation between abstraction and implementation, but they leave out important aspects.

First, the role of linguistic metaphors in the design, implementation, and execution phases of such artefacts. The literature relying on the dual approach did not investigate the role of language almost at all, and works based on the LoAs hierarchy invests only high-level programming with a performative role. Nonetheless, linguistic aspects of computing are pervasive, starting from "Hello World!" programs, naturally presented as performative speech acts: they say something and, by saying it, they do something. At the very beginning of programming history, linguistic metaphors were widely used (Nofre et al., 2014) and the importance of pragmatic aspects of computation had already been taken into account (Zemanek, 1966). Recent studies in cognitive science consider the hypothesis that programming draws on some of the same resources as natural language processing (Fedorenko et al., 2019). Apart from rare (but periodic) attempts to stress the parallels between coding and literary expression (Cox & McLean, 2012), the linguistic nature of programming is taken for granted, and then almost forgotten. Philosophical approaches to computer science standardly dispute about its epistemological nature as a mathematical, scientific, or engineering discipline (Tedre, 2011, 2014; Primiero, 2019). We aim to complete this debate, considering the linguistic nature of computational artefacts.

The second forgotten aspect concerns the understanding and representation of all pragmatic, contextual, and expressive features in computational artefacts' ontology and epistemology. Such limitation is in line with the definition and description of computational artefacts limited to syntax and semantics. Contexts in which computations get different meanings, esoteric programming languages, change of intended use, and even the role that humans play in the results of modern data-driven applications fall under such pragmatic considerations, and seem to undermine the traditional independence of the specification from the implementation.

The present contribution aims to explain the interaction between pragmatics, epistemology and ontology in computational artefacts by recovering the role of the language

metaphor. This task has been partially considered for programming languages (Adamczyk, 2011), but never for computational artefacts at large. This allows extending the ontological and epistemological analysis of computational systems to include pragmatic aspects. To this aim, we introduce a User Levels (ULs) structure as a pragmatic complement to the LoAs-based structure already mentioned above. LoAs include agents operating in a hierarchical structure, in which syntax, semantics, and correctness are attributed to specific levels, with specification acting as a normative element. Using a distinction introduced in Tanaka-Ishii (2015), we might call LoAs a *constructive* system and ULs a *structural* one. Artificial sign systems are usually considered as constructive ones, where a sign depends hierarchically on other signs (Tanaka-Ishii, 2015, p. 997). Such a hierarchy of signs is not present in structural systems, typically natural sign systems, in which the signification is given *holistically* by the contribution and mutual influence of signs. Evolution is another aspect in which structural and constructive systems differ: constructive systems stop evolving once all the signs are clarified in *what they stand for*, converging to a *fixed point*; on the contrary, the mutual circular influence of signs drives structural systems towards a continuous evolution *in response to the environment*. Put differently, only structural systems are able to evolve. When dealing with linguistic and pragmatic aspects of computational artefacts, the LoAs model fails in describing them because it tries to apply a *constructive* system to a system that reveals *structural* characteristics. Therefore we aim to introduce structural aspects in the taxonomy of LoAs, in order to fill *"the gap between humanity and mechanical inhumanity"* (Tanaka-Ishii, 2015, p. 995).

ULs identify a communication structure with a flexible hierarchy. Users are bearers of their own semantic and normative requirements, possibly competing with those of the logical specification. While the physical level of the machine acts as ontological support for all other levels, correctness becomes heavily dependent on the UL under consideration. The hierarchy is flexible, as for each LoA it is possible to identify one or more users who, through a particular use of the computational artefact in a specific context, enable their level to be meaningful. The semantic and normative instances of users and of contexts in which they operate are taken to contribute to the design as well as to the extended semantics and correctness of the computational artefact. Such correctness is more expressive than, and in some cases possibly limiting, the artefact's standard semantic correctness as implementation of the given formal specification. We formulate a notion of computational act intended in its pragmatic sense, alongside pragmatic versions of implementation and correctness.

The paper is structured as follows. In Sect. 2 we overview the standard views on the notions of ontology and correctness as available in the current literature. In Sect. 3 we review the role that linguistics has had so far in the literature on the philosophy of computer science and introduce aspects of computing whose understanding seems to require a pragmatic approach, namely: development, deviant uses, esoteric programming languages, and data-driven computation. In Sect. 4 we answer to such a requirement for a pragmatic aspect in the ontology and correctness of computational artefacts with the formulation of a theory which accounts for users and their acts, allowing the formulation of pragmatic definitions of implementation and correctness,

and drawing the appropriate connections with the more standard Levels of Abstraction view. Finally, in Sect. 5 we summarize further lines of research for the pragmatics aspects of computational artefacts.

2 The standard Views on Ontology and Correctness

In order to introduce and justify pragmatic elements in a theory of computational artefacts, we first briefly review current approaches to issues of their ontology and correctness.

2.1 The Dualist View

Conventional academic wisdom, widespread in the early literature in the Philosophy of Computer Science, and possibly also common among the general public with a minimum understanding of computational systems, suggests that computational systems are characterized by a dual nature. This dualism can take the trivial form of the dichotomy hardware *vs.* software, but more refined interpretations are also possible: specification *vs.* implementation, physical machine *vs.* virtual machines, abstract *vs.* concrete. The nature of these systems appears double, if not even multiple, also, but not only, due to the fact that the concept of computational system is multiply realizable, hence, it can be analyzed from a functionalist point of view.

In this ontological framework, computational systems can be subsumed under the broader category of *technical artefacts*, i.e., intentionally produced entities characterized by a teleological nature. A technical artefact is defined not only by observables, such as appearance, chemical composition, or electric charge, i.e., its *structural properties*, but also, and above all, by its *functional properties*, i.e., the purpose for which that particular object was made. In the early literature, see (Angius et al., 2021, sec. 2.3), the study of the conditions under which a computational artefact possesses its functions, and how these functions are related with both its structural properties and the intentions of the agent, is limited to two main theories:

- the *causal theory of function* (Cummins, 1975), according to which the intentions of the agent play no role in defining the function, which are solely determined by the physical abilities and behaviours of the artefact;
- and the *intentional theory of function* (McLaughlin, 2000; Searle, 1995), according to which the intention of the designer plays a primary role in establishing the function of the artefact, and the structural properties are chosen only in relation to their ability to accomplish it.

The major drawback of the first theory lies in its inability to deal with *malfunctioning* (Kroes, 2009; Fresco & Primiero, 2013; Floridi et al., 2015), and *side-effects* (Turner, 2011): if the physical properties of a pocket calculator entail that, whenever it is used to perform the $2 + 2$ operation, the number 5 appears on the display as output, then, according to this theory, the device would function correctly despite the mathematical error, since the intention to produce a correct result plays no role in defining the function of this artefact.

On the other hand, the intentional theory, despite being able to deal with correctness, malfunctioning, and side effects, it is not able to identify where the function resides: if it is in the artefact, then we are pushed back to the question of how it possesses the intended function; if it is in the mind of the agent, then one can invoke the *Princess Elizabeth argument*, and ask for a better explanation for the connection between mental states of the agent and physical properties of artefacts (Angius et al., 2021, sec. 2.3). Therefore, a purely dualist approach, is not properly able to explain the relation between specification and function of a computational system, and how both are driven by the intention of the agent.[1]

Moreover, the distinction between hardware and software, is far from being satisfactory, and the history of its criticism has lasted for over four decades. In the late 70's, Moor (1978) pointed out that the hardware *vs.* software distinction, together with the digital *vs.* analogue, and the model *vs.* theory distinctions, constitute the three myths of computer science. In particular, the hardware *vs.* software distinction is ontologically irrelevant, and it has only, if any, a pragmatic significance. Furthermore, this distinction is imprecise not only from a historical point of view, since at the beginning of computing the act of programming was realized in particular configurations of pushed and pulled levers; but also because most modern software still mimics this type of interaction: we all *press buttons*, *move levers* selecting options and *scroll pages*. Thus, the strict abstract/concrete dichotomy seems insufficient to account for the required correctness of implementation.

In the scope of the dualist approach, the implementation, generically intended as the realization of the specification, can be interpreted in, at least, two different ways:

1. as the semantic interpretation of a syntactical model given by the specification, (Rapaport, 1999);
2. as the relation between the specification (abstract), and the artefact (concrete) (Turner, 2011, 2014, 2018).

Under the first reading, given its semantic role, the implementation should also provide the criteria for correctness. However this represents a serious issue for this account: the normative role of implementation does not give any tool to deal with *incorrect implementations* (Primiero, 2019). Consider again the pocket calculator that returns 5 when $2 + 2$ is performed: is this a correct result, given it is the semantic interpretation of the specification? Moreover, if we consider the case of the implementation of one language into another, this approach does not even account for correct implementations, since the implementing language should provide a semantic interpretation of the implemented language, and this is possible only if the former is associated with a semantics providing meaning and correctness criteria for the latter (Turner, 2018).

Under the second reading, the implementation does not bear any normative properties, which are entirely invested in the specification. However, this account is not able to deal with the complex and layered ontology of computational systems since

[1] Partly to answer to the mentioned limitations, the *evolutionary theory of function* was developed (Preston, 2022, sec. 2.3). The latter not only applies to organisms, but may also apply to the evolutionary development of how technical artefacts are used by agents over time. These three theories are not mutually exclusive but should rather be perceived as the end points of a threefold spectrum, allowing for a multitude of different theories within its limits. A case in point is the *ICE (Intention, Causal role, Evolution) theory of function* (Vermaas & Houkes, 2006), which will be taken into consideration in Sect. 3.2.

it identifies a unique implementation relation between a specification level and an artefact level (Primiero, 2019).

2.2 Layered Ontology

To answer the shortcomings of the dualist approach, a theory based on the methodology of the *Levels of Abstraction* (LoA) has been introduced. First successful especially in mathematical and philosophical research (Floridi, 2008), it has been recently applied to computational systems and to information (Primiero, 2016). In this theoretical framework, the ontology of computational systems extends the structural/functional one characteristic of technical artefacts (Kroes, 2009), since each level plays a functional role for lower ones, and a structural role for upper ones.

The *Intention*, defining the computational problem to be solved, expresses the functions that the system must achieve, and it is implemented in a set of requirements by the *Specification*. The latter plays a functional role, explaining the concrete functions that the software must implement, and it is realized by the procedure to solve the problem, namely an *Algorithm*. This represents the structural level with respect to the specification, and expresses the procedures that *High-level programming language instructions* must implement. The linguistic implementation defines and is realized by the functional properties for *Assembly/machine code operations*. Machine code, finally, identifies the functional properties implemented by the *Execution* level, expressing the physical structural properties of the running software.

Each LoA embeds a pair consisting of an epistemological construct and an ontological domain appropriately related by an instantiation relation of implementation across different LoAs (Primiero, 2019, p. 212). So the pair $\langle Problem, Intention \rangle$ is implemented in the pair $\langle Task, Algorithm \rangle$, and this in turn in the pair $\langle Instruction, Programming\ Language \rangle$. Consequently, a layered notion of correctness can be specified for each level: if the functionalities of the specification are displayed with the intended efficiency, then the computing artefact is *functionally correct*; if it displays correctly the functionalities intended by the underlying algorithm, then it is *procedurally correct*; if it runs correctly on the architecture on which it is installed, then it is *executionally correct*. A correct (or wellfunctioning) physical computing system is one which presents correct implementations at all the required levels. This view matches the understanding of miscomputations at different LoAs (Fresco & Primiero, 2013).

A theory of computational artefacts, articulated along different LoAs, requires an equally detailed analysis of the information and its propagation through these levels (Primiero, 2016). At the lowest level we are confronted with the physical machine and wired logic: here information is made up of a series of 0's and 1's in the form of bits, physically realized into configurations of matter, *e.g*,. electric charge flowing along wires, electronic configuration of silicon clusters. At this first level, we do not properly speak of information, but only of structured data. The introduction of syntactically well-defined low-level languages allows us to use textual instructions to perform mathematical operations on bits, which have a direct correspondence with the actions that can be performed on the physical device. The *operational information*

embedded in machine code is qualified as *well-formed performative data*, thus it is only required to be syntactically well formed: this type of purely quantitative information possesses neither semantics nor alethic value. The programming language level abstracts from the physical layer. At this level, *Instructional information* is described in terms of *well-formed and meaningful data*. Instructions *denote* the information content of appropriate executable strings and are satisfied by the operational information of the machine code. Like in the case of orders, there is still no alethic assessment. Alethic and epistemic considerations step in as soon as we analyze the level at which the instructions are evaluated in relation to the meaning *as intended* by the algorithm, and by the task it aims to correctly perform. The algorithm can be defined, in abstract terms, as a general recursive or Turing-computable function, implemented in a program, developed to perform a task represented by the expected behavior of the machine. Just as the algorithm is implemented in the programming language, it in turn also acts as an implementation for intentions aimed at solving a problem. The correctness of the algorithm, and of the program that implements it, will therefore be determined according to the intentions of the designer and its semantic value depends on the implementations realized at lower LoAs. The abstract informational content of an algorithm also defines *truthfulness* of its instances, and not just their correctness. An algorithm can then be defined as *abstract, correctness- and truth-determining information* for all previous levels, acting as a normative definition for the computational artefact. These considerations lead to a philosophical formulation of (digital) Computing as the systematic study of the ontologies and epistemology of information structures (Primiero, 2016, p. 122). However, the distinction between ontologies and epistemology, between object domains and linguistic means of control, is purely theoretical, and often blurred in practice. In the *Agile* methodology,[2] for example, each project is the result of a dialogue between parties, there are no clearly definable priorities between the phases, there is no hierarchical domain between the LoAs, the influence of the specification on the implementation is not any stronger than the reverse relation, in a context of continuous communication and collaboration between the several actors involved in the process. An example of such blurring of LoAs will be offered below in Sect. 3.1.

This has two important consequences. First, to sharpen the blurred distinction above, the introduction of pragmatic aspects can be useful, as we aim to do in Sect. 4. Second, if languages (from machine code to designers' intentional states) act as control means over objects domains (from bits to complex systems such as networks), limiting the performative role to programming languages seems reductive. In absence of assembly languages and high-level programming languages (as it was the case in the early days of computing when programming was done directly in machine language, and as it may become again the case with the new low code/no code paradigm, or in data-driven applications) the LoAs theory considers the information content of computational processes devoid of both operational and instructional information, limiting such content to structured data, abstract information and intentions. While possible, this might not be desirable, as it leads to two distinct conclusions: either the information content

[2] Agile methodology is a project management approach, opposed to the traditional *waterfall model* and based on the Agile Manifesto (Beck et al., 2001), that involves breaking the project into phases, emphasizing continuous collaboration and improvement.

is deprived of syntactic correctness and semantic value; or one is forced to move them to another LoA, falling back into the same issues that characterize the semantic interpretation of the dualist approach. Similar considerations can also be made in relation to the introduction of assembly languages, high-level programming languages, and automatic compilation. Furthermore, nowadays, the syntactic correctness of the written code is, in most cases, no longer checked during the compilation phase, but during the writing of the code itself, which again suggests a much more dynamic, less structured view on correctness. User-friendly interfaces and simplified dialects are increasingly pervasive in software development, underlining the importance of both the language metaphor and the users in the evolution of computational systems. We argue that these aspects have their own status and should integrate, rather than compete with, the standard layered ontology of computational artefacts.

3 Language and Pragmatics in Computer Science

Programs can be interpreted as performative speech acts which, in their announcement to the world, *say* something and, by saying it, they *do* something. Nonetheless, the linguistic metaphor has been given little consideration in recent philosophical and technical contributions in computer science, limited to programming languages, and preferring purely logical-mathematical approaches. By linguistic metaphor in computer science one refers to all syntactic, semantic and pragmatic aspects of the use of language that surrounds and is embedded in the process of design, development, and use of computational artefacts. While syntactic and semantic aspects have been more extensively considered, we focus here on the little investigated aspects of the pragmatics of language in this context.

Such reduced attention to pragmatics could be seen at least as an indication of disregard for both early computer science research and common sense, since the anthropomorphization of computers has always been a natural step, for both scientists and users, since Babbage, Turing and von Neumann (Nofre et al., 2014, p. 45). Together with the tradition of considering mathematics as "the language of nature", the anthropomorphic view of computers extended this linguistic analogy from pure mathematics to machine code. Starting from the mid-1950 s, with the appearance of different models of computers, the need to develop a machine-independent language was urgent. During this phase, *"the language metaphor lost its anthropomorphic connotation and acquired a more abstract meaning, closely related to the formal languages of logic and linguistics"* (Nofre et al., 2014, p. 41). Programming languages and algorithms emerged as independent of the hardware, a step which at the beginning of the next decade led programming languages to become connected with formal languages and with the notion of universality. In the 1960 s, with programming a discipline on its own, the language metaphor spread from computer science to involve other disciplines, from the analysis of natural language to molecular biology.

Meanwhile, the *Association for Computer Machinery* was clearly aware of the importance of the linguistic and pragmatic aspects of computing. In 1965, the title of the annual ACM Conference held in August in San Dimas, California, was "Programming Languages and Pragmatics". During the conference, (Zemanek, 1966) represents the

only attempt to fill the gap between semiotics and computer science for decades, stressing the importance not only of syntax and semantics, but also of pragmatics. The following terms and their definitions, first introduced in Morris (1938), were mentioned in (Zemanek, Zemanek (1966), p.139):

- pragmatics: deals with the origin, uses and effects of signs within the behavior in which they occur;
- semantics: deals with the signification of signs in all modes of signifying;
- syntactics: deals with combination of signs without regard for their specific significations or their relation to the behavior in which they occur.

From first attempt to formalize such a behavioral, human-inspired approach to semiotics due to positivist notion of *protocol*, (Zemanek, 1966) would bring semiotics to its natural end-point in programming.

According to his analysis, the relations between the three dimensions of semiotics is vague, and can be modified on the basis of the principles of the investigation (Zemanek, 1966, p. 141). If we consider the three semantic levels

- *object world, i.e.*, the objects of a real or invented world,
- *object language, i.e.*, their names, and
- *metalanguage, i.e.*, the names of the names,

as *primary* notions, syntax expresses the relation within any one level, semantics the relations between two levels, while pragmatics expresses the ability to construct relations from outside the scheme (Zemanek, 1966, p. 140). Moreover, both natural languages and programming languages are endowed with both prescriptive and descriptive components expressing declaration and command, dealing with states or with actions (Zemanek, 1966, p. 140). Such components cannot be treated independently of the semiotic triple, especially of pragmatics. Additionally, "formal" is not at all identical to "syntactical", nor to "artificial", as syntax can be treated informally and it is possible to formalize pragmatics, artificial languages can be informal and natural languages can be formalized. Finally, considering the performative aspect of languages, programming, like poetry, has a creative component, while, more than poetry, has a real effect on the physical world.

These pragmatics considerations should not be restricted to programming language, as they include all other aspects of systems such as the relation to the compiler, the machine language, the operating system, the hardware and the output. The relation between programming languages and humans offers another occasion for Zemanek to show his foresight. On the one hand, three years before the first public demonstration of a mouse controlling a computer system, he hypothesized the reduction of sophisticated languages to simple pointing, as shown by the billions of ostensive gesture made by all of us on our mobile phones, expanding pragmatics to involve not only *programmers*, but also *final users*. On the other hand, long before the introduction of contemporary voice assistants, he stressed how much conversational features would be essential when dealing with talking computational systems (Zemanek, 1966, p. 143).

Despite the relevance of pragmatics considerations in current computational trends already anticipated by Zemanek, after this initial approach there has been only scattered discussions about applying semiotics to programming languages. The language

metaphor, despite its importance, was taken for granted and set aside in the vast majority of investigations in computer science. The first attempt to study programming language systems, including compiler and interpreter, by means of semiotics is presented in Andersen et al. (1994). According to them, computers are symbolic machines constructed and controlled by means of signs not only in the obvious case of the interface, but also deep down *"in the intestines of the system"*, from program text to the compiler as meta-sign, from assembly to machine code, every part of the machine is text to be interpreted or prescription about states of the system and every such part can be treated semiotically (Andersen et al., 1994, p. 16). Such extended semiotic approach is complementary to the standard ontological and epistemological accounts of computational artefacts summarized in the previous section. However, while in agreement with the dualist theory, it is definitely at odds with the LoAs theory. The rigid demarcation between the different aspects of information doesn't allow a unique semiotic approach for any level, since semantics and syntax are strictly confined in specific level, and pragmatics is not even mentioned.

In cognitive science, the development of cognitive models of computer programming have been attempted since the 1970 s, and recently have established the connection between programming and reading abilities (Fedorenko et al., 2019, p. 4-7). Some further pragmatic approaches to programming languages have been proposed for implementation (Scott, 2009), actual use (Connolly & Cooke, 2004), understanding agent-oriented programming languages (Bordini et al., 2011) and software publishing (Lee, 2000), and to create programming languages based on speech acts (McCarthy, 2007).

The extensions of pragmatic and linguistic aspects from programming languages alone to computational artefacts at large is still missing though, despite the early intuitions by Zamanek. Even Turner (2018), which provides one of the most complete philosophical account of computational artefacts currently available, insists on the use of language in a rather standard way, referring to types of languages in use in the development of computational systems, programming languages paradigms, and the distinction between specification and implementation languages. Also the relation to Wittgensteinean themes concentrates on rule-following and understandability of proofs, leaving aside more pragmatic aspects. In the next sections, we consider several pragmatic aspects of computational systems which represent a serious challenge for the LoAs interpretation of specification and implementation, their relationship with correctness, and the origin of normativity. On the basis of the analysis of these cases, we will then propose a pragmatic refinement of the LoAs structure, in order to create a bridge between semiotics and standard theory of computational systems.

3.1 Development

From a pragmatic point of view, the standard interpretation of information technology increases the regulatory role attributed to functional specifications, while diminishing the normative role of economic, technological, legislative aspects, or those related to company policies.

Let us consider the development of a *Message Hub, i.e.,* an application that collects all the messages received and sent using other messaging systems. Its specification, thus expressed, does not seem to present issues. In technological terms, it is perfectly achievable, it does not violate any physical law, as shown by projects of this type already implemented in the past. Unfortunately, corporate policies and consequent software restrictions, led to the progressive elimination of *Application Programming Interfaces* (APIs) by many firms, making applications installed on our devices closed systems unable to communicate with each other, unless performing risky procedures, (*e.g., jailbreak*). Do company policies have a normative function? Arguably, their normative role is of a different order than what covered by the specification, namely they exercise a normative function on the specification itself. The same kind of considerations can be made for privacy laws, copyright laws, criminal law, and others: they all perform a normative function on the specification, determining the identity of the artefact (Angius & Primiero, 2018, 2020).

Not taking these aspects into consideration may mean investing resources, time and money in projects that will never be legally distributed. While this does not entail that company policies and institutional laws represent the regulatory element of computational artefacts, it shows that normativity does not only reside in the specification but must also be sought elsewhere. It could be objected that the definition of specification we are referring to is an informal definition, whereas normativity, and the consequent definitions of malfunctioning and correct functioning, classically are ascribed to the specification intended as a description of the logical function implemented by the artefact. We argue that any relevant normative element contributes in determining the specification against which eventually correctness is evaluated. Under this assumption, in the following pages, we will consider some issues that arise in the absence of proper consideration of relevant pragmatic elements, and the contexts in which they develop.

A first such element is the extreme dynamism and variability of the functional specification of the artefact, whether seen in formal terms or not (for a formal interpretation see Primiero et al. (2019a)). Only discontinued projects, i.e., obsolete and no longer maintained software, have static and immutable specifications: the greater the diffusion and success of a computational artefact, the greater the frequency of modification and revision of its specifications. The massive presence of Beta versions, or public and free test versions, distributed for the purpose of identifying bugs and any possible evolution through large scale use, seem to identify in pragmatic elements and in collective acts, rather than in the specification, the true normative element that guides the development and evolution of a computational artefact. Nowadays, the specification of any computational system germinates as an idea in the mind of some human being. In the very early stages, it typically takes on the appearance of an e-mail, requesting an economic quotation, sent to some software house, or to some digital media agency, in which a brief functional description is included. If the quote is accepted, and if the artefact is intended for human users, after a preliminary exchange of information between the customer and the development team, it is transformed into a visual mock-up, created by a designer, which simulates the *User Experience* (UX). The UX thus passes to the scrutiny of the programmer(s) to evaluate its technical feasibility. It is

only at this point that *Unified Modeling Language* (UML) diagrams, or other equivalent tools, come into play, with a use that, in most cases of software development, remains limited to managing data structures and *Database* (DB) design. At this point, even before starting the implementation, the specification may change again due to technological reasons, for example because the distribution of information within the UX does not allow for a performing DB, and changes may therefore be necessary that involve in the best case only the positioning of the information, in the most serious cases also the functionality of the artefact. At the same time, the UX is enriched with aesthetic elements that contribute to create the *User Interface* (UI). In this phase of development, the regulatory element is represented by the *Brand Identity*, namely the set of graphic and communication elements that participate in creating the perception and reputation of the computational artefact. In this context, a fundamental element is the name and its *bundle identifier* which, in carrying out the function of rigid designator of the computational artefact, succeed in a task precluded both to the specification and to the implementation. This process is recursively repeated, until the computational artefact is no longer maintained.

In this process, the identification of what a computational artefact is, seems to go beyond the abstract description of its functioning. Consider the general case of the development of an application for mobile phones. One must first decide for a *Native* development, typically by writing Java or Kotlin code using Android Studio for Android-based devices, and also by writing Objective-C or Swift code using X-Code for Apple devices; or for a *Cross-Platform* development, writing a hybrid application using a suitable, typically JavaScript-based, development framework. Despite having the same functional specification, the two native applications not only are written using different programming languages, and different SDKs, in order to be executed on different hardware systems; they also differ in both UI and UX. Consider a basic element for a mobile application, the navigation menu: to develop the Android version it will be necessary to use an element called *Navigation Drawer* which has completely different UI and UX from the *Tab Bar*, typical of iOS and iPadOS applications. It seems that we are facing two different computational artefacts, despite their specification being identical. Their identity is not difficult to reconstruct formally (Angius & Primiero, 2018), less so from the pragmatic point of view of the users involved.

In the case of applications for iOS or iPadOS devices, a particular hardware is also needed to develop them, i.e., a MacOS machine, or a so-called *Hackintosh* machine. Unfortunately, however, the relationship between dedicated software/hardware and hacking opens up new possibilities for indeterminacy, demonstrating how the specification alone might not be able – on its own – either to offer an adequate description of the correctness criteria or to identify the corresponding implementation. Consider the case of *jailbroken* iPhones: on most devices, it is possible to install unauthorized *firmware* and operating systems that enable features forbidden by the original ones, *e.g* to install applications not present on the App Store, allow shared data among several applications, and so on. It is important to notice that the act of jailbreaking a phone is not illegal *per se*, but it usually enables access to pirated or legally restricted content, and it goes against the producer policies, resulting in a forfeiture of the guarantee of the device. Under these circumstances, it becomes unclear whether the correctness of its functioning is established by the original firmware and its specification, or by

the unauthorized one. We could extend these questions to the most diverse types of computational artefacts such as *overclocked* processors, *modded* consoles, and so on. On the other hand, the name, the brand identity and, in more formal and contemporary terms, the aforementioned bundle identifier are able to provide an adequate criterion at least for the identification of a computational artefact, even if unauthorized or illegal, e.g., viruses, malware, *et cetera*, a family of artefacts which is acquiring more and more technical and conceptual relevance (see e.g. (Primiero et al., 2019b)).

Once the development has been completed, the product must be distributed. In the case of software for iOS devices, as in our example, it is necessary to publish the application developed through the App Store. Normativity is represented here by Apple's application development guidelines, but also by the discretion of the individual Apple employee assigned to carry out the verification. The online magazine Medium.com published a few years ago an article (Imaginovation, 2017) which lists various reasons why an application may not respect these guidelines which include, in addition to the most obvious crashes and bugs during the verification process, the lack of privacy policy or metadata, the use of offensive language or private APIs, graphics that are not in line with the Apple quality standards or long loading times. The 63,300,000 results produced by Google by carrying out the search "reasons why app could be rejected from app store" confirm that this is not a rare occurrence, and the variety of these results confirms the strong component of human discretion present in this process.

The above examples tell us that a wide range of components complement and modify the rigid formal structure of the functional specification of computational artefacts during their development and deployment. These include commercial and industrial policies, versioning processes determining continuously new improvements and extensions, the choice of compatible hardware for any given software, and finally the deployment strategies and requirements. These aspects, which we characterize as essentially pragmatic in nature, exceed the purely functional description while certainly they exert an essential role in determining well-functioning and correct use of the computational artefacts at hand. The human component, though, is also responsible for *idiosyncratic*, unconventional, deviant uses.

3.2 Deviant Uses

We now consider how the specification of a computational artefact can change under the pressure of more or less conventional uses.

The intentions of the anonymous developers of the famous Amiga 500 software called X-Copy were to create a backup software, certainly not to give birth to modern computer piracy, nor to contribute to the boom in sales of Commodore products between the 80 s and 90 s of the last century. The history of computing is full of episodes in which software behaviors that are initially considered as bugs, discovered by users or by the developers themselves through the test use of the artefact, reveal new and unexpected features, which are later included in the specification of later versions of the artefact itself. Consider the "unsend" functionality of GMail: lore has it that the first Google mail server architectures took more than 5 s to send an email. This would tend to be considered a malfunction, and as such it seems it was initially seen

by the creators of the famous e-mail service, until it was transformed into a feature by changing the text of the "send" button to "unsend", thus allowing users 5 s to change their mind and withdraw the sent message.

These considerations relate with a peculiar aspects of technical artefacts: the change of intended use. A person with a particularly refined taste could decide to use an old Olivetti Lettera 32 typewriter and an old Siemens S62 telephone as furnishing accessories; or, particularly appreciating their design, they could use a pair of old Apple Macintosh G3 as bedside tables. The worldwide presence of company museums, in which technical artefacts are exhibited as pieces of art, confirms that this function should not be neglected. This type of iconic information that some technical, and computational artefacts are able to *represent*, is closely related to the *meta-representation* generated in each individual user of such artefacts: a bottle of Coca Cola is not "just" a bottle, the Bataclan door painted by Banksy is not "just" a door, an iPhone is not "just" a phone, Deep Blue is not "just" an IBM computer; the message they represent is intelligible only on the basis of the user's meta-representation, which can obviously be different from user to user. It is therefore not just a simple change of intended use, on which however a few more words need to be spent: without changing the intended use, the computational artefacts would still be simple calculators, there would be no word processors, websites, web-televisions, video-games, social networks, *et cetera*. A pragmatic approach emphasizes the importance of the use of technical artefacts, not only the designed and preordained conventional use, but also the creative and the innovative use. In these cases, the definition of actual use does not need to match the definition of correctness for the artefact at hand. A lighter can act as a bottle opener and a shoe horn can be used to comfortably scratch your back, a coffee machine can be a work of art and a system bug can become a feature.

The ICE-theory of function mentioned in Sect. 2.1 seems a particularly suitable tool for modelling these kind of situations. The theory combines three general approaches to function: the I(ntetionalist), the C(ausal-role), and the E(volutionist) ones. The three approaches are embedded in a framework which theoretizes action with a partly normative and partly descriptive intent, see (Vermaas & Houkes, 2006, p. 9):

Definition 1 (ICE-function theory,) An agent a ascribes the capacity to ϕ as a function to an artefact x, relative to a use plan p for x and relative to an account A, iff:

I. the agent a has the capacity belief that x has the capacity to ϕ, when manipulated in the execution of p, and the agent a has the contribution belief that if this execution of p leads successfully to its goals, this success is due, in part, to x's capacity to ϕ;
II. the agent a can justify these two beliefs on the basis of A; and
III. the agents d who developed p have intentionally selected x for the capacity to ϕ and have intentionally communicated p to other agents u.

This theory models the evolution of technical artefacts starting from passive use, but it extends to a taxonomy of several types of designing. Consider everyday spreadsheet use: making additions and subtractions, changing the size of a cell to make it fit a huge amount of money, changing to red the background color of all the cells in the column labeled as "DEBTS", and so on. Everything goes as intended by the original design.

The first type in the array of (new) designs takes into account the fact that users can manipulate artefacts in a way that differs from the original use plan, for example when using our spreadsheet to create *pixel-art*. User u_1 may have the intuition that resizing all the cells in the same square shape, and changing color to their background in a certain way, allows her to create amazing pixel-artworks. To reach the second type in the design array, i.e., innovative using, communication with other users is necessary. User u_1 may share her discovery by means of several media: she can post pictures of her artwork on social media, she can talk/text to other users about her discovery, she can sell her artwork as *Non Fungible Token*, and so on. As a consequence, the expanded functionality is also made available to other final users.

Idiosyncratic and innovative using are usually not considered as cases of designing, because they are based on experience or luck, and they lack any scientific or theoretical knowledge. Expert redesigning fills this gap, adding the required *expertise* basis. Consider two spreadsheet users, d_1, d_2, who also are an independent graphic designer and an independent programmer respectively. On the basis of their scientific knowledge and of the new function spread throughout the final users community, d_1 can design and share picture templates, allowing unskilled final users to follow them and use the spreadsheet as a coloring book, while d_2 can write and share *macros* that allow final users to fill a cell or a group of cells with color gradients.

While the communities of final users made of independent designers and programmers (as well as other possible users communities) keep experimenting with new features (and possibly discovering new ones), also the original programmers and designers become aware of them. They may recognize the commercial potential of such features and decide to include them in the specification of the next release of the spreadsheet, moving to the final step of the array: product design. From the final user point of view, we are back to the passive using phase, and the process can start again from the beginning, with new features and behaviours.

3.3 Esolangs

Another pragmatic challenge, also related to creativity and agency, is represented by an operation that, *prima facie*, seems to involve only syntactical modifications, such as the translation from a programming language to another. We already mentioned that the problem of the translation from programming languages to machine code is a matter of pragmatics, however, one may argue that, when dealing with different programming languages and not with machine code, translation changes syntax while preserving semantic, so that function computation remains unaltered (Connolly & Cooke, 2004, p. 154). Unfortunately, there are several cases in which this is not true. For example, to translate a client-side program on a server-side data structure and architecture, not only syntactical changes are necessary, but also semantic ones, in many cases followed by essential human intervention.

In order to explain the concepts just expressed, a series examples consisting of various *Hello World* programs, developed using different *esoteric* programming languages (Esolang), will be proposed below. Esolangs are all those very complex programming languages, designed to be particularly cryptic and created for the purpose of testing

the limits of programming and programmers, for research, or just for fun. The already mentioned *Hello World Convention*, consists of the first program that is generally written when learning a new programming language; the result of this program is the "Hello World" output addressed to the human user through any output device.

Let's start by considering languages that are famous for the obfuscation of their code, i.e., the difficulties that human readers encounter when they try to understand them. Brainfuck, has an extremely minimal syntax, consisting exclusively of 8 admitted characters (> < + − . , [ ]), resulting in high lexical complexity of programs written in this language, and undermining the distinction between high-level and low-level programming languages. Befunge uses a two dimensional grid which allows the execution of the program to proceed in any direction of the grid, not only breaking with the convention that imposes a fixed direction in interpretation, but also annihilating the idea of a "line of code", by allowing the instruction pointer to change direction or flow off one side of the screen onto the other.

Example 1 "Hello World" in Brainfuck

```
++++++++++[>+++++++>++++++++++>+++<<<-]>++.>+.+++++++
 ..+++.>++.<<+++++++++++++++.>.+++.------.--------.>+.
```

Example 2 "Hello World" in Befunge

```
"dlroW olleH">:v
              ^,_@
```

Still in the scope of obfuscated code, another particularly famous esolang is Malbolge, designed to be as difficult as possible to program. Any operation performed by a Malbolge program involves a complex procedure, in which several aspects (e.g., the use of a primitive ternary CPU, the memory shared by code and data, the encryption/decryption of data, the use of a so-called *crazy operation*), combine to create a deliberately chaotic development environment.

Example 3 "Hello World" in Malbolge

```
(=<‘:9876Z4321UT.-Q+*)M’&%$H"!~}|Bzy?=|{z]KwZY44Eq0/{mlk**
hKs_dG5[m_BA{?-Y;;Vb’rR5431M}/.zHGwEDCBA@98\6543W10/.R,+O<
```

Given the extreme complexity of the language, no real programs have been written in Malbolge for a long time; the first working "Hello World" program came two years after the release of the language, produced by a Lisp program going through the logical space of all possible programs, see (Cooke, 2000). The first program "99 Bottles of Beer", closely linked to non-trivial conditions and cycles, was released only seven years after the birth of Malbolge by Hisashi Iizawa and is universally considered a masterpiece of the obfuscated code, also because it supports the hypothesis that Malbolge is Turing-complete. The temporal variable in this case seems to be a confirmation of the fact that the implementation between languages cannot be a process limited to purely syntactical translation.

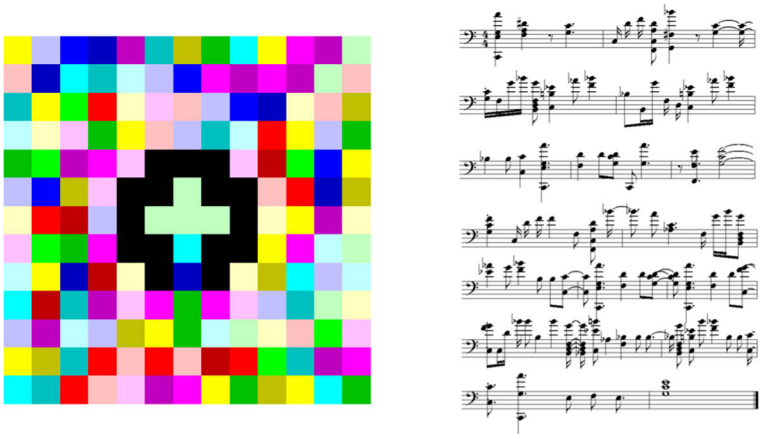

Fig. 1 "Hello World" in Piet (left) and Velato (right) - source: Esolangs

Elevating obfuscation to an art form, Piet and Velato fill the gap between programming language and artistic expression, see Fig. 1. Piet, named after Dutch painter Piet Mondrian, is a stack-based programming language, in which programs resemble geometric colorful paintings. Velato uses MIDI files as source code, defined by the intervals and the order of the notes encoded in them. By allowing some flexibility in composition, the resulting programs can be adapted not to sound like random strings of notes, in fact "[t]here is a tendency for Velato programs to have jazz-like harmonies" (Esolang, 2014). These examples highlight "the conflict between programming languages, designed with the assumption that human beings are capable of expressing logic, and the actual human beings who write code in them" (Temkin, 2017).

Entropy is a programming language which, despite its conventional syntax, has a peculiarity which makes it extremely interesting for our analysis: data remains uncorrupted until accessed and becomes more corrupt as it is used. An Entropy programmer needs to abandon the pursuit of precision, since "[a]ny output from an Entropy program will be approximate, and the more the data is accessed, the more random it will become" (Esolang, 2014). Let us consider again our *Hello World* program:

Example 4 "Hello World" in Entropy

```
Program MyNamespace MyProgram [
    print "Hello World";
]
```

The result of the execution of this program is a variation of the classic "Hello World" message such as "Hello + World" or "Eello Qorld" and so on. Obviously, the more complex a program is and the more it accesses data series, the more casual and divergent the result will be from initial expectations and specifications. It is evident that the definitions of correctness given above for the LoAs structure (Primiero, 2019) fail to include programs written in Entropy:

- the definition of *functional correctness* is inapplicable to Entropy programs as it is not possible to define with certainty the expected efficiency of their functions;
- Entropy programs can be *procedurally correct* only as long as data remains untouched, failing functionally the more procedural correctness is investigated;
- *executional correctness* is also unsuitable for the description of Entropy programs since, according to this definition, none of them will work correctly (functionally) on the installed architecture.

On the opposite side of the *spectrum* of correctness we locate FatFinger.JS. This JavaScript library allows the use of typing and spelling errors, which would normally be interpreted as syntax errors.

Example 5 "Hello World" in FatFinger.JS

```
<script type="text/javascript" src="fatfinger.js"></script>
<script type="text/javoscript">
    vart x = "herrrllo werld"
    dokkkkumint.rit3(xx)
</script>
```

Using FatFinger.JS these lines of code will be interpreted as:

```
<script type="text/javascript" src="fatfinger.js"></script>
<script type="text/javascript">
    var x = "herrrllo werld"
    document.write(x)
</script>
```

The considerations made in relation to Entropy can be transposed in this case, still confirming the limitation of the definitions of functional, procedural and executional correctness:

- it is difficult to establish with certainty the *functional correctness* of programs written using FatFinger.JS because the functional efficiency of such programs will generally (but not certainly) be higher than the functional efficiency of programs written without using it;
- it is not possible to clearly define the scope of definition of *procedural correctness* as both syntactically correct programs and programs containing various types of errors will show the functionalities intended by the underlying algorithm;
- similarly, regarding *execution correctness* we find ourselves in the curious situation in which both syntactically correct programs and programs containing syntax errors will work correctly on the considered architectures.

For the various cases of unintended uses, as well as for languages that deviate from a stable relation between specification and implementation, a pragmatic interpretation may become necessary. Users are traditionally ignored by computer theories, focusing on the production of computational artefacts and almost never on their use.

3.4 Classification by Data

A final case we want to consider briefly has less relevance for traditional digital computing systems, while it opens our pragmatic approach to describe novel computational technologies implementing learning based on data. Consider modern machine learning systems trained to classify data. Notoriously, a significant component in the result of such processes is represented by the data labeling of the training set. The input data on which the system learns, and the associated labeling, provides a significant component in the way the functional, procedural and executional correctness of such systems can be evaluated. Asking whether the result of a classification task with a given label set will be correct with respect to the (intuitive) function specification of the system ("label datapoint d_1 as label L_1 if its denotation can be correctly categorised as such") should therefore account for it, while the procedure it implements (e.g.: "label datapoint d_1 as label L_1 if a sufficient number of data in a specified neighborhood can be classified in a similar way") may diverge from it. In particular, deviant phenomena such biased labeling in the training set will contribute significantly to the evaluation of correctness and trustworthiness in their use. A training set in which (human) users have (even unconsciously) imported significant bias (e.g. by labeling images which account for social, cultural, religious or political judgements), will be reflected in the results of the classifier used on new inputs: shall we say that a classifier designed to classify humans as male/female is functionally incorrect when it systematically misjudges a young girl who happens to have (by chance, or by choice) non-feminine traits? Obviously so, but the training component that produces such a result is indeed part of the system's input. Only it appears to be originating from a user level (training data) which is far more hidden and less explicitly accounted for than other levels, and which is strictly related, closely talking to another user level (labeller) which influences and guides the computational result. Also establishing procedural correctness becomes difficult, as a label expressing a factual error by its (missing) association with a sufficient number of data will result in a correct functional result. The possibility that the trained algorithm might then run on different input sets, generating different results opens up the issue of executional correctness.

In the next section we introduce the concept of user levels (ULs) and investigate their relationship with the levels of abstraction (LoA) method applied to computational artefacts.

4 The UL Model

The considerations in the previous section lead to the conclusion that, at least for certain ways of intending computational artefacts, the semantics of information cannot be confined to any single LoA, but rather seems to propagate along all levels. In pragmatic terms: for each level of abstraction it is possible to identify one or more users who, through a particular use of the computational artefact, make this level significant. We organize such pragmatic component in a novel *User Levels Model*, which accounts for the contribution of different types of users in the definition of computational systems. We start off with some more simple examples.

For the programmers of ENIAC (Electronic Numerical Integrator and Computer) instructions and operations coincide in a single level of abstraction, thus making the wired logic meaningful. Or consider the following examples from Scott (2009):

Example 6 Euclid's algorithm for computing the greatest common divisor in machine language, expressed here as hexadecimal numbers (base 16), for the x86 (Pentium) instruction set:

```
55 89 e5 53 83 ec 04 83 e4 f0 e8 31 00 00 00 89 c3 e8 2a 00
00 00 39 c3 74 10 8d b6 00 00 00 00 39 c3 7e 13 29 c3 39 c3
75 f6 89 1c 24 e8 6e 00 00 00 8b 5d fc c9 c3 29 d8 eb eb 90
```

Example 7 Euclid's algorithm for computing the greatest common divisor in x86 assembly language:

```
pushl %ebp
movl %esp, %ebp
pushl %ebx
subl $4, %esp
andl $-16, %esp
call getint
movl %eax, %ebx
call getint
cmpl %eax, %ebx
je C
A: cmpl %eax, %ebx
jle D
subl %eax, %ebx
B: cmpl %eax, %ebx
jne A
C: movl %ebx, (%esp)
call putint
movl -4(%ebp), %ebx
leave
ret
D: subl %ebx, %eax
jmp B
```

For sufficiently competent users, it is possible to identify the semantics of the computational artefact, even at low levels of abstraction. Some users will be able to distinguish a program for calculating the greatest common divisor from another written, for example, to carry out sums, starting from the level of the machine language; others will need to move to the level of assembly languages, still others will have the need and the skills to use high-level languages, others will significantly operate at the algorithm level. Finally, end users will be able to distinguish and attribute the semantics of the artefact *via* UI. In all these case, the correctness criterion might still be characterised at the level of the algorithm, while pragmatically there might be no need to ascend through the entire hierarchy of LoAs from user interface to intention.

A first step to provide an integration of the LoA hierarchy which accounts for such pragmatic component consists therefore in identifying the concept of *user* who, as a particular signifier in context, is able to contribute to the meaning of the artefact at her own level. Such analysis of the notion of user extends the taxonomy presented in Fresco and Primiero (2013). A second step consists in adapting the common belief that the analogy between programming languages and speech acts can be extended to computational artefacts at large, especially with regards to performativity, dependence on context and the relationship with linguistic excess. Hence, by means of a series of practical examples, we will articulate the concept of *computational act* as an extension of the concepts of *code act* (Cox & McLean, 2012) and of *speech act* (Searle, 1969). Austin (1962) is usually considered to have offered the first complete formulation of Speech act theory. Sentences or utterances are not just a means to *say something*, but they are proper way to act, *to do something*. A sentence can be considered as a *performative act*, consisting of

- a *locutionary act*, i.e., the actual utterance and its meaning;
- an *illocutionary act*, providing the *force*, e.g., informing, ordering, asking, warning, etc;
- a *perlocutionary act*, representing the actual effect achieved by the utterance, e.g., scaring or persuade someone, making someone do or realize something, etc.

Under this context, speech acts and programs share a performative nature, with the latter endowed with the ability to both exceed and undermine the distinction between speech and writing, as they are able to address both humans ad machines, and make things happen, cf. (Cox & McLean, 2012, p. 35).

4.1 Users

The study of the use of technical artefacts in general, and of computational artefacts in particular, tends to be confined to the field of design, with few philosophical contributions, most of which focus on political and social considerations deriving from the linguistic-performative nature of code writing, and focusing exclusively on human users. The presence of human agents with the ability to act at certain LoAs also characterizes the taxonomy set out in Fresco and Primiero (2013). In summary, the taxonomy includes the following users:

1. the *Architect* operating at *Functional Specification Level* (FSL);
2. the *System designer* in charge of the *Design Specification Level* (DSL);
3. the *Algorithm designer* responsible for the *Algorithm Design Level* (ADL);
4. the *Engineer* in control of the *Algorithm Implementation Level* (AIL);
5. the *Algorithm Execution Level* (AEL), with no corresponding agent beyond the physical machine.

The attribution of syntax, semantics and normativity to specific levels makes the LoAs highly hierarchical: the physical machine will allow the program written by the Engineer to run on the basis of the algorithm designed by the Algorithm Designer. The latter, in turn, will work following the specifications imposed by the *System Design*

Description (SDD) produced by the System Designer on the basis of the *Software Requirements Specification* (SRS) established by the Architect.

On the contrary, the UL model is based primarily on the notion of (not necessarily human) user. By de-localizing the semantics involved in the use of the artefact, the UL model reveals a softer hierarchy: in this new structure all possible uses denote a semantic relation between signifier and signified. By signifier we intend the modes and requests of use of the artefact by any users; by signified we intend the collection of functional and non-functional requirements by all users converging in the specification. This type of layered organization is, by hypothesis, destined to be provisional, given its dependence on the technological and ergonomic advances of computational artefacts, as well as their deviant use. No matter how many ULs may be introduced to enrich the system at any given point, each of these will contribute to the definition of the peculiar meaning and correctness of the artefact through its use within a given context. Under this reading, the specification, and the underlying algorithm, loose their hegemony on correctness, to become the collector of the semantic contributions of the different signifiers conveyed by each UL, as well as of their regulatory requests.

The UL model qualifies as a structural system, in the sense intended by Tanaka-Ishii (2015) and illustrated above. As such, a list of its components does not have a fixed order, their roles may intertwine and overlap, each one communicates with the others and their nature may change. This is the case especially for the list of ULs of computational artefacts provided below, where for example some users that were typically humans in the past, have become today physical or virtual machines:

- the *developer (low-level, high-level, front-end, back-end etc.)* takes care of the implementation of the artefact. It communicates with other levels according to their tasks: e.g., the low-level developer communicates with assembler, compiler and system designer; the high-level developer communicates with system designer and algorithm designer; front-end developer and back-end developer communicate with each other; and the former communicates directly with the interface designer;
- the *designer (system, interface, algorithm etc.)* is responsible for designing the artefact, or part of it. In addition to the dialectics highlighted in the previous point, the system designer dialogues with the owner, whereas the interface designer and front-end developer dialogue with the user through their production of the UI;
- the *machine* corresponds to the level of the physical machine;
- the *operating system* is made up of many levels, and it constitutes the privileged way of man–machine communication;
- the *producer* deals with the production of the computational artefact (hardware and software);
- the *publisher* deals with the publication, marketing and dissemination of the artefact, also covering the role of policy maker with regard to its own distribution channels;
- the *compiler* role, once held by human users, is now the prerogative of machines, and it deals with the "translation" from high-level to low-level languages;
- the *assembler*, similar to the compiler and similarly once held by humans, refers to the lower levels of the artefact, dealing with the "translation" in assembly language;
- the *end user* is the final user of the computational artefact. Under appropriate contexts, any other UL, excluding the machine level, can be considered as an end user;

- the *policy maker* dictates the rules governing the production, dissemination and use of computational artefacts. These norms are created and modified as a result of the production, dissemination and use of computational artefacts, e.g. the changes in the music world induced by the advent of *peer-to-peer* software, the tightening of the rules on *hate speech* following the widespread diffusion of social networks, or the creation of the new legal category of computer crimes;
- the *owner* of the computational artefact dictates the general guidelines for the realization of the artefact and is responsible for the economic and legal aspects of its production, dissemination and use.

This is the most obvious list for the UL model of traditional digital computational systems. The introduction of novel technologies implies the addition of novel ULs: for example data-driven technologies will include novel users like the *data modeller*, the *data annotator*, the *data labeller* and so on. The semantic and normative instances of users, and of the contexts in which they operate, contribute through their interaction to the semantics and correctness of the computational artefact.

4.2 Computational Acts

Every computational act is a collective linguistic act that involves at least two ULs, with one of them always performed by the physical machine, the ontological support of all subsequent levels. This consideration seems to be applicable not only to digital artefacts, but to all computational artefacts, including the calculation with paper and pen or in mind, in which a single agent operates at different ULs (i.e., designer, developer, machine). We argue that meaning for a computer program, while formally expressed by the logic of the specification, it exceeds it in the many ways it is contextually considered by any given user.

For example, the regulatory role of contexts is not only limited to the developer UL, for whom such element allows the typing of variables and where the use of brackets determine variable declaration or function scope or interruption of logical flow. It rather extends to all levels, in a phenomenon similar to what we have seen with semantics. So the physical layer determines the operations that can actually be performed without running into *overflow*, (Connolly & Cooke, 2004); the firmware level is equipped with the so-called protection software, implemented in order to prevent *"untrusted programs"* or *"untrusted users"* from any access to the operating system kernel and input/output channels, (Kittler, 1996); and the level of the operating system and its graphical interface affect thoughts and actions of the users, (Cox & McLean, 2012).

Context is also important for ULs traditionally considered as *passive*, such as the compiler level. Consider the case of *polyglot* programs, where one or more algorithms are implemented using different programming languages, usually making creative use of comments, like in the following public domain "Hello World" written in ANSI C, PHP and bash:

Example 8 Polyglot "Hello World" in ANCI C, PHP e bash - source Wikipedia.

```
#define a /*
#<?php
echo "\010Hello, world!\n";// 2> /dev/null > /dev/null \ ;
// 2> /dev/null; x=a;
$x=5; // 2> /dev/null \ ;
if (($x))
// 2> /dev/null; then
return 0;
// 2> /dev/null; fi
#define e ?>
#define b */
#include <stdio.h>
#define main() int main(void)
#define printf printf(
#define true )
#define function
function main()
{
printf "Hello, world!\n"true/* 2> /dev/null | grep -v true*/;
return 0;
}
#define c /*
main
#*/
```

The influence of the compiler level becomes clear when we consider polyglots that implement different algorithms. Depending on the compiler that will translate the program, different results starting from the same program are possible. A (possibly extreme) example is the challenge launched on Stack Exchange to write a polyglot in n different languages. The program must return 1 when executed in the first language, 2 when executed in the second, 3 in the third, and so on. Currently, a correctly functioning program has been obtained in 377 different languages, many of them esolangs, making it practically impossible to reproduce the code on printed paper without modifying it, rendering the polyglot no longer functional. The challenge, which lasted for over five years with hundreds of contributors, further underlining the collective nature of the computational act, is still open, see https://codegolf.stackexchange.com/questions/102370/add-a-language-to-a-polyglot.

Another interesting case of contextual influence between ULs arises between the high-level developer and algorithm designer levels: it subverts the normative relationship between the two levels, making specification and high-level languages mutually dependent, undermining the independence of the former from the latter. First of all, high-level programming languages do not constitute a homogeneous and compact group. Even leaving aside esoteric languages, it is customary to divide them into different families based on the computation model: declarative (functional, dataflow, logic) languages and imperative (von Neumann, scripting, Object Oriented) languages. On the basis of this distinction, let's consider the example of Euclid's algorithm for

calculating the greatest common divisor (GDC) and its implementation in three programming languages: C representing the von Neumann family, Prolog for logic-based languages, and Scheme, chosen from functional languages.

Example 9 Greatest common divisor in C, Scheme, and Prolog - source (Scott, Scott (2009), p.13).

```
int gcd(int a, int b) {                                    // C
 while (a != b) {
   if (a > b) a = a - b;
   else b = b - a;
 }
 return a;
}

(define gcd                                                ; Scheme
 (lambda (a b)
   (cond ((= a b) a)
         ((> a b) (gcd (- a b) b))
         (else (gcd (- b a) a)))))


gcd(A,B,G) :- A = B, G = A.                                % Prolog
gcd(A,B,G) :- A > B, C is A-B, gcd(C,B,G).
gcd(A,B,G) :- B > A, C is B-A, gcd(C,A,G).
```

The von Neumann version of the algorithm shows clearly an imperative character. In the specification for Scheme the emphasis is on the mathematical relationship of the GCD function. In logic-based languages the programmer provides a set of rules for the system to find the right values. The resulting paradoxical situation is that if we want the specification to be independent from any linguistic implementation, then we are forced to accept a very general kind of specification, such as "write a 'Hello, World!' program" or "write a GCD program". The specification alone is not able to identify one program, or distinguish it from others written in different languages, or running on different hardware platforms. One rather needs the specification to be related to the actual implemented algorithm, which can be very different from language to language. In this case, implementation and specification undergo mutual influence both in informational and regulatory terms, strengthening the hypothesis of the contextual influence between levels and the relative collective nature of the computational act.

Like other collective speech acts, the computational act oscillates between process and expression. Like spoken languages, every computational act says something and does something. The similarity between programming and composing music or poetry formulated by Donald Knuth in *The Art Of Computer Programming* is enriched with meaning; the computational act, in its being poised between a readable state and an executable state, exceeds each and includes both. New meanings emerge from the normative and semantic instances of the different ULs, including those which, traditionally, are not considered active subjects in the definition of the artefact. Between

all these levels, the final UL seems to exert a particular type of thrust, especially in the unconventional, and *deviant* uses. At this point we offer a pragmatic definition of implementation which accompanies the various existing ones in the literature considered before, and a definition of pragmatic correctness which extends those based on functional, procedural and executional criteria:

Definition 2 (Implementation as relationship between signifiers) An implementation I pragmatically intended is the correspondence relationship between modes and requests of use of the artefact (signifiers) at different ULs of the same collection of functional and non-functional requirements (signified), converging in the specification.

Definition 3 (Pragmatic correctness) A physical artefact t is pragmatically correct if and only if there is convergence of all the modes and requests of use of the artefact expressed at the different ULs.

If the semantic and normative thrust of the deviant level exceeds that of the formal specification, i.e. if the uses or interpretation by some users do not conform with the intended one, but are not specifically excluded from the specification, they are not rejected, but rather extend the specification, thereby becoming correct uses. The general definition of correct physical computing system given for LoA model in Primiero (2019) remains valid also in the UL model.

Finally, we show how the notion of *intentional*, *computational* and *physical* information from the taxonomy defined in Primiero (2016) is complemented when analysed at distinct ULs (see Fig. 2):

- *intentional information*, which is *abstract* and truth-bearing from the point of view of the Levels of Abstraction model, satisfies a pragmatic aspect of meaning from the

Fig. 2 Information in UL model

semiotic point of view, and it defines the epistemological domain of the underlying levels. It represents the locutionary aspect of a computational act, i.e., its meaning in the broadest sense;
- *computational information*, which is *instructional*, meaning-bearing information from the point of view of the Levels of Abstraction model, is related to the semantic aspects of meaning, and it provides the language to connect the epistemological domain to its ontological counterpart. It represents the illocutionary aspect of a computational act, i.e., its force;
- *physical information*, which plays the role of *operational*, correctness-bearing information, is related to syntax, expressing the ontological support. It represents the perlocutionary aspect of a computational act, i.e., its actual effects.

The same exact structure is repeated recursively in the scope of computational information, in which the information of each UL, considered as *instructional information,* is the center of a triadic structure including the information of the levels immediately below, representing *operational information,* and above, playing the role of *abstract information*. To exemplify this analysis, let's consider a simple case in which we have only three ULs: Machine Language (ML), Programming Language (PL), and Algorithm. Starting from the higher level of abstraction we have (see Fig. 3):

- at Algorithm level: the information conveyed by the programming language will be operational, correctness-bearing with respect to the algorithm; the information conveyed by the algorithm is instructional, meaning-bearing; and the information content of the intention is abstract, truth-bearing;
- at Programming Language level: the information conveyed by the machine language will be operational, correctness-bearing with respect to the programming language; the information conveyed by the programming language is instructional, meaning-bearing; and the information content of the algorithm is abstract, truth-bearing;

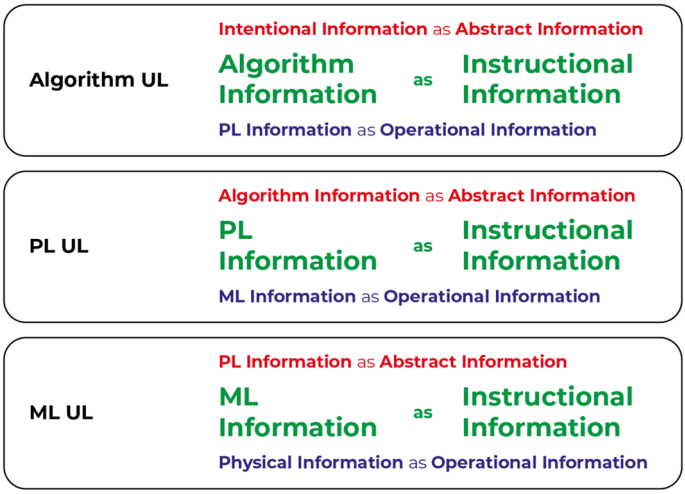

Fig. 3 Computational Information in UL model

- at Machine Language level: the physical information will be the operational, correctness-bearing with respect to the machine language information; the information conveyed by the machine language is instructional, meaning-bearing; and the information content of the programming language is abstract, truth-bearing.

Turning back to the considerations made at the end of Sect. 2.2, the UL structure succeeds where the LoA account fails. Consider, as an example, the recent introduction and growing diffusion of no-code systems among developers. LoA is not able to precisely define, in its linear hierarchy, this recent practice in the development of a computational artefact: the user is no longer dealing directly at the high-level language, nor formulating explicitly requirements for code to be correct. On the other hand, the recursive structure of the UL allows to model such new tools: the use of a platform that visually aids the end user to build her application by dragging-and-dropping preformulated components, can be seen as:

- the flow of the required processes at the abstract intentional information level (acting as algorithm);
- the interaction between user and machine, providing instructional information (acting as a programming interface);
- the lower level of operational information (when the components chosen and combined are seen at the programming language level).

Note that this analysis does not fix the user's actions at any given level, but it allows to shift them across levels and let them bear each time a different role.

The evolution of computational systems offers similar examples when a new feature, or combination of existing features, is discovered through use. In these cases, the conventional use (e.g. coloring the cells of a spreadsheet with the purpose of highlighting certain values) acts as instructional information for the intentional level regarding the new feature discovered (i.e. to display enjoyable graphic results), while the actual innovative result (the coloured spreadsheet on screen representing a Japanese landscape) reveals itself as perlocutionary act. The new feature, being excluded from the original specification, is not evaluated for correctness in the LoA model. On the contrary, according to the UL model, the correctness of the new feature – if not explicitly against the specification – is admitted by considering all modes of use expressed by all the different ULs involved (namely including the deviant use of colouring cells to make art). When a deviant use is explicitly against the specification, as in the cases of hacking, cracking, or uses that cause malfunctioning (e.g., SQL Injection or SSL Heartbleed), the specification can no longer act as *abstract, intentional information* with respect to such use, as the latter obviously contradicts some explicit requirement. Therefore, such forms of deviant use cannot be considered correct, unless the original specification is modified.

No matter how many ULs and sub-levels are added, due e.g. to technological and ergonomic evolution of artefacts, this taxonomical analysis still holds, as shown in Fig. 4.

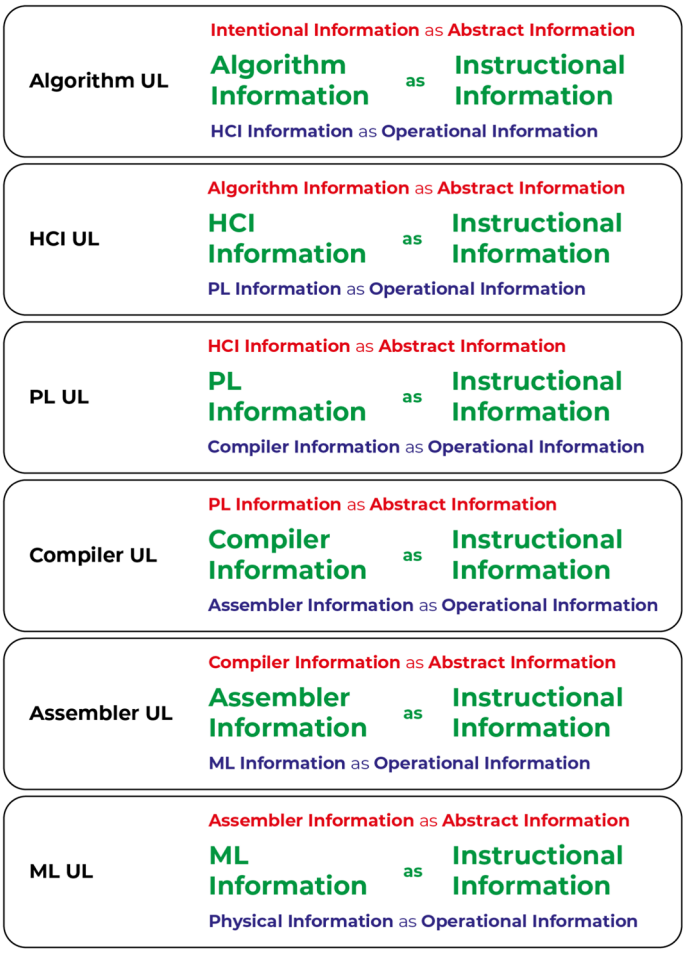

Fig. 4 The resilience of UL Structure to the evolution of computational systems

5 Conclusions and Further Work

A pragmatic approach to computational artefacts, based on their use intended as a performative and collective computational act, seems to offer a powerful tool to be placed side by side with the standard theory. In particular it is able to describe both peculiar phenomena such as esoteric programming languages, and still little explored aspects of philosophical research in computation, such as the change of intended use, the evolutionary drive that generates it, or the role of human components in novel data-driven technologies.

A required part of such theory is a formal elaboration, complete with syntax, and pure and applied semantics, as well as a complete taxonomy, as far as possible, of the different ULs. Another issue to be considered is the risk of admissibility of malfunctioning in the largely permissive nature of the pragmatic interpretation. Further

investigations may be devoted to applying the above mentioned theory of information to computational methods that already use independent agency, such as artificial neural networks, adversarial programs, genetic algorithms, and genetic programs.

Acknowledgments The authors thankfully acknowledge the support of the Italian Ministry of University and Research through the projects PRIN2020 n. 2020SSKZ7R (BRIO - Bias, Risk and Opacity in AI) and "Departments of Excellence 2023-2027" awarded to the Department of Philosophy "Piero Martinetti". The authors wish to thank two anonymous reviewers for their helpful and insightful observations.

Funding Open access funding provided by Università degli Studi di Milano within the CRUI-CARE Agreement. This research has been partially funded by the Project PRIN2020 BRIO "Bias, Risk and Opacity in AI" (2020SSKZ7R) awarded by the Italian Ministry of University and Research (MIUR).

References

Adamczyk, P. (2011). On the language metaphor. In Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, New York, pp. 121-128. Association for Computing Machinery.

Andersen, P. B., Holmqvist, B., & Jensen, J. F. (1994). *The computer as medium. Learning in doing: Social, cognitive and computational perspectives*. Cambridge University Press.

Angius, N., & Primiero, G. (2018). The logic of identity and copy for computational artefacts. *Journal of Logic and Computation, 28*(6), 1293–1322. https://doi.org/10.1093/logcom/exy012

Angius, N., & Primiero, G. (2020). Infringing software property rights: Ontological, methodological, and ethical questions. *Philosophy and Technology, 33*(2), 283–308. https://doi.org/10.1007/s13347-019-00358-7

Angius, N., Primiero, G., & Turner, R. (2021). The Philosophy of Computer Science, In The Stanford Encyclopedia of Philosophy (Spring 2021 ed.)., ed. Zalta, E.N. Stanford: Metaphysics Research Lab, Stanford University.

Austin, J. L. (1962). *How to do things with words*. Clarendon Press.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., & Thomas, D. (2001). Manifesto for agile software development. http://www.agilemanifesto.org/.

Bordini, R., Moreira, A., Vieira, R., & Wooldridge, M. (2011). On the formal semantics of speech-act based communication in an agent-oriented programming language. *Journal of Artificial Intelligence Research (JAIR), 29*, 221–267. https://doi.org/10.1613/jair.2221

Connolly, J. H., & Cooke, D. J. (2004). The pragmatics of programming languages. *Semiotica, 2004*(151), 149–161. https://doi.org/10.1515/semi.2004.065

Cooke, A. (2000). Andrew cooke: Malbolge hello world. https://web.archive.org/web/20191206191704/https://www.acooke.org/malbolge.html.

Cox, G., & McLean, A. (2012). *Speaking code: Coding as aesthetic and political expression*. The MIT Press.

Cummins, R. (1975). Functional analysis. *Journal of Philosophy, 72*, 741–64. https://doi.org/10.2307/2024640

Esolang. (2014). Esoteric programming languages wiki. https://esolangs.org/.

Fedorenko, E., Ivanova, A., Dhamala, R., & Bers, M. U. (2019). The language of programming: A cognitive perspective. *Trends in Cognitive Sciences, 23*, 525–528. https://doi.org/10.1016/j.tics.2019.04.010
Floridi, L. (2008). The method of levels of abstraction. *Minds and Machines, 18*, 303–329. https://doi.org/10.1007/s11023-008-9113-7
Floridi, L., Fresco, N., & Primiero, G. (2015). On malfunctioning software. *Synthese, 192*, 1199–1220. https://doi.org/10.1007/s11229-014-0610-3
Fresco, N., & Primiero, G. (2013). Miscomputation. *Philosophy and Technology, 26*, 253–272. https://doi.org/10.1007/s13347-013-0112-0
Imaginovation. (2017). 16 reasons the app store rejects mobile apps and how to avoid them. https://medium.com/@Imaginovation/16-reasons-the-app-store-rejects-mobile-apps-how-to-avoid-them-63f73fa33a3a.
Kittler, F. (1996). There is no software. In T. Druckrey & A. Stone (Eds.), *Electronic culture: Technology and visual representation, New York* (pp. 147–155). Aperture.
Kroes, P. (2009). Engineering and the dual nature of technical artefacts. *Cambridge Journal of Economics, 34*, 51–62. https://doi.org/10.1093/cje/bep019
Lee, T. (2000). Publishing software as a speech act. *Berkeley Technology Law Journal, 15*(2), 629–712.
McCarthy, J. (2007). Elephant 2000: A programming language based on speech acts. In Companion to the 22nd ACM SIGPLAN conference on object-oriented programming systems and applications companion, New York, pp. 723-724. Association for Computing Machinery.
McLaughlin, P. (2000). *What functions explain: Functional explanation and self-reproducing systems*. Cambridge University Press.
Moor, J. H. (1978). Three myths of computer science. *British Journal for the Philosophy of Science, 29*, 213–222. https://doi.org/10.1093/bjps/29.3.213
Morris, C. W. (1938). Foundations of the theory of Signs. University of Chicago Press Cambridge University Press.
Nofre, D., Priestley, M., & Alberts, G. (2014). When technology became language: The origins of the linguistic conception of computer programming, 1950-1960. Technology and culture 55: 40–75, 2 p preceding 1. https://doi.org/10.1353/tech.2014.0031 .
Preston, B. (2022). Artifact, in the stanford encyclopedia of philosophy (Winter 2022 ed.)., eds. Zalta, E. N. and Nodelman, U. Stanford: Metaphysics Research Lab, Stanford University.
Primiero, G. (2016), 06. Information in the philosophy of computer science, In The routledge handbook of philosophy of information, ed. Floridi, L., 108–125. London: Routledge.
Primiero, G. (2019). *On the foundations of computing*. Oxford University Press.
Primiero, G., Raimondi, F., & Chen, T. (2019). A theory of change for prioritised resilient and evolvable software systems. *Synthese, 198*, 5719–5744. https://doi.org/10.1007/s11229-019-02305-7
Primiero, G., Solheim, F. J., & Spring, J. M. (2019). On malfunction, mechanisms and malware classification. *Philosophy and Technology, 32*(2), 339–362. https://doi.org/10.1007/s13347-018-0334-2
Rapaport, W. J. (1999). Implementation is semantic interpretation. *The Monist, 82*, 109–130.
Scott, M. L. (2009). *Programming language pragmatics* (3rd ed.). Morgan Kaufmann Publishers Inc.
Searle, J. R. (1969). *Speech acts: An essay in the philosophy of language*. Cambridge University Press.
Searle, J. R. (1995). *The construction of social reality*. Simon and Schuster.
Tanaka-Ishii, K. (2015). Semiotics of computing: Filling the gap between humanity and mechanical inhumanity, In International Handbook of Semiotics, ed. Trifonas, P.P., 981–1002. Dordrecht: Springer Netherlands.
Tedre, M. (2011). Computing as a science: A survey of competing viewpoints. *Minds and Machines, 21*(3), 361–387. https://doi.org/10.1007/s11023-011-9240-4
Tedre, M. (2014). *The science of computing: Shaping a discipline*. CCRC Press, Taylor and Francis Group.
Temkin, D. (2017). New languages, Daniel Temkin Website. http://danieltemkin.com/Esolangs/.
Turner, R. (2011). Specification. *Minds and Machines, 21*, 135–152. https://doi.org/10.1007/s11023-011-9239-x
Turner, R. (2014). Programming languages as technical artifacts. *Philosophy and Technology, 27*, 377–397. https://doi.org/10.1007/s13347-012-0098-z
Turner, R. (2018). *Computational artifacts: Towards a philosophy of computer science*. Springer, Berlin Heidelberg.
Vermaas, P. E., & Houkes, W. (2006). Technical functions: A drawbridge between the intentional and structural natures of technical artefacts. *Studies in History and Philosophy of Science Part A, 37*, 5–18. https://doi.org/10.1016/j.shpsa.2005.12.002

Zemanek, H. (1966). Semiotics and programming languages. *Communications of the ACM, 9*, 139–143. https://doi.org/10.1145/365230.365249