10
INFORMATION IN THE PHILOSOPHY OF COMPUTER SCIENCE

Giuseppe Primiero

Introduction

During the last decade, the philosophy of computer science has carved an important space within the landscape of philosophical investigations. The range of questions and problems it addresses is wide and varied: the methodology of design, the ontology and semantics of computational artefacts, abstraction and implementation, to name a few. This chapter focuses strictly on the philosophical interpretation of the notion of information within Computer Science.

The centrality of information in Computer Science is indisputable: the discipline is hardly comprehensible when abstracted from the conceptualization and use of this notion. (Denning 1985) defined Computer Science as “the body of knowledge of information-transforming processes” and (Hartmanis, Lin 1992) as “the study of information” in itself. Although the debate on the nature of this discipline is far from being settled, these two early definitions refer to information as an essential concept. *A fortiori,* information represents an optimal conceptual tool to explore the philosophy of computer science.

Given its ubiquity, information risks to become a misleading concept. A philosophical approach to the role of information in Computer Science requires, in the first place, articulating the actual configuration of the discipline. A list of the main research areas within academic departments and research institutions can be roughly given as follows:[1]

1 Algorithms and Data Structures
2 Programming Languages
3 Architecture
4 Operating Systems and Networks
5 Software Engineering
6 Databases and Information Retrieval
7 Artificial Intelligence and Robotics
8 Graphics
9 Human-Computer Interaction
10 Data mining and Machine Learning
11 Bioinformatics

A quick overview of this list reveals the well-known methodology of the Level of Abstraction (LoA, see Chapter 7) at work, including all aspects from the very concrete to the formal, from the isolated act of computation to its complex environment: the formal structures underlying data and their algorithmic treatment (1); their implementation in language (2) and use for program design (5); the design and construction of (networks of) hardware to manipulate (3 and 4), visualize (8) and process data (6); data analytics and its use to lead automatic processes (10); the relation between machine and the user (9); the study of autonomous agents (7); the mechanical engineering of living systems (11).

This familiar way of representing the work of computer scientists tells us that the syntactic (see Chapter 4), the semantic (see Chapter 6) and the procedural (see Chapter 9) notions of information are all at work in different areas of Computer Science. Our task is to approach information focusing on the computational model, from the low-level processing of circuitry to the higher level of design, to sketch the philosophical issues that arise. For this reason, we will focus on the standard notion of digital computational system: we will show how the LoAs are structured, and will do so in terms of an *epistemology of control* and an *ontology of syntax and semantics* through the relation *abstraction-implementation,* i.e. the linking of a syntactic domain (abstraction, symbol manipulation) to a semantic one (domain of objects). This relation is considered at the core of the LoAs structure in Computer Science, see (Rapaport 1999). In the second section we start from the lowest possible level of abstraction, where information is electric inputs running on wires; in the third section we consider how that level is controlled through syntax; in the fourth section we move to the semantics of programming languages and their control of algorithmic structures; in the fifth section we investigate the intentional stance behind the programming and algorithm design practice, and in the sixth section we summarize our analysis of the information flow within the computational system, analyzing briefly how programs are interpreted and checked.

Information inside the computing machine: structured data

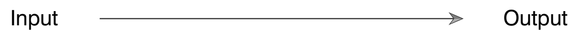
In the long-standing philosophical debate about what Computer Science is, the mechanical formulation of the computational process is central. (Newell *et al.* 1967) defined Computer Science as "the science of computers and related phenomena"; later (Newell, Simon 1976) rephrased it as "the empirical study of computer related phenomena". Under this interpretation, the core business of Computer Science is the material execution and mechanical realization of those information-transforming processes referred to by (Denning 1985). The physical core of a modern computer is the Central Processing Unit, roughly composed by:

- arithmetic and logic unit, for the data processing;
- registers, for their storage;
- program counter and instructions register, to store the machine state and current operation of the program;
- control unit: for the coordination of input/output devices.

While we are not strictly interested in the actual physical functioning of a CPU, we want to investigate which kind of information is at work at the physical level of the computing machine. The technical, well-known answer is that computing machinery at the physical level deals with *binary digits* (bits) expressing discrete, exclusive ON/OFF states of electrical-magnetic input. Philosophically, this description is still incomplete: the information flowing on wires is not simply 1s and 0s of bits randomly produced for the processing unit to operate on. These bits need to be structured and processed according to rules. Let us make an easy example. If we wire a switch to a LED (Light Emitting Diode) and connect it to a 5v supply on a breadboard, the effect is to turn the light on; if the switch is turned to OFF, so does the light. If we bypass first the wire through an *inverter* (a digital circuit which *inverts* the value passed to it, logically corresponding to a negation, see the diagram below), the effect is what we expect: when the switch is ON, the light will stay OFF; when the switch is OFF, the light will go ON (Figure 10.1).

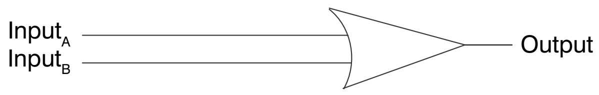
If we combine two wires, each connected to a switch, through an OR gate with one bit output to a LED, then the output will be ON if at least one of the two switches is ON (see Figure 10,.2).

With these simple cases in mind, we can already make some observations:



Input	Output
1	0
0	1

Figure 10.1 Inverter



A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

Figure 10.2 2-input OR gate

1 the input is given by a variable *x* (possibly composed by more than one bit) with value 0 or 1;
2 there is an output *y* whose value depends on *x*;
3 there is some rule establishing the dependency relation between *x* and *y*; for the *inverter* rule, such dependency is explained by saying that "if *x=0* then *y=1* and if *x=1* then *y=0*"; for the OR rule, the dependency is explained by saying that "if *x1 or x2=1* then *y=1* and if both *x1 and x2=0* then *y=0*".

The ontological domain of electrical inputs is an implementation; its structure is controlled in terms of value assignment, value dependency and rule execution.[2] A CPU and the board on which it is installed are just a more complex set of such structuring, including other essential Boolean circuits (implementing other logical operations, e.g. AND, XOR), further modified by the capacity to store values for future uses (memory), the ability to locate specific inputs (location assignment) and organize complex instructions (coordination). Hence, at this level, information corresponds to data as structured, physically evaluated variables, where structure control is meant to associate electrical charges to the realization of *actions* (Figure 10.1).

Look at Figure 10.3 below and let us unpack this definition. The pure syntactical, physical elements (electrical charge) create *distinction* or *difference* in the system, as the result of the evaluation of an empty element (value assignment function). The empty element in question, or variable, is an abstraction from the allocated memory space or physical wire taking a value (1,0). Its evaluation is the way difference is manifested. In (Floridi 2011, pp.85–86), difference is specified as *de re* (lack of uniformity in the real world) or *de signo* (lack of uniformity between at least two signals). In the context of electrical values manifested in bits, one is dealing with a difference of the second kind (*de signo*), coupled with a physical effect that occurs in the real world (*de re*), be it simply a charged wire or a lightened up LED. In the context of physical computing systems, a difference *de re* allows to trace a difference *de signo,* while it is always the latter that causes the former. While data processing is most commonly interpreted by digital bits, there are examples of analog computers that essentially change the understanding of the data worked on, by forms of mechanical (like the physical movement of objects) or hydraulic (water droplets) quantities. If we stick to the digital realm, the essential nature of data at this level is that of tokens *physically realized* in an electrical charge, possibly manifested in an actually visible state of the system (e.g. by pixels on a display or a LED). At higher levels, the analysis of information in the context of Computer Science allows, and in fact requires, abstracting away from the physical expression of our inputs.

The structuring of physical data at the level of digital processing is given by a *spatial and temporally determined execution,* essential to their correct manipulation in terms of dependency relations. Typically, it will make a difference if the value of *x* is accessed at a memory slot

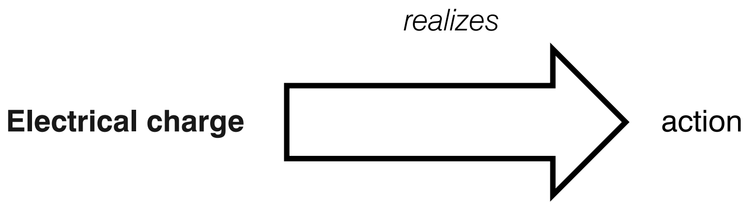


Figure 10.3 Structured physical data

before accessing a *y* from which it depends, or if the location is accessed after the relevant value is updated, or at which memory location one looks for a given value. The resulting structure can be taken either as an ontological or an epistemic property. In the former case (as argued in Fresco and Wolf 2013), data composed by logical operations and accessed by memory functions is structured in view of the capacity of the *abstract datum* to be so before any implementation. Under the epistemic interpretation (as argued by Floridi 2011), structure is determined by the knowledge process at stake at this specific LoA, i.e. when bits are taken as information. This means that structure is not inherited from abstract data as such, but only obtained in virtue of the operational view on data. While the epistemic reading makes the distinction between information and data essential, the ontological view reduces the possibility of structure to data themselves.

In both cases, the explanation of *how* structure is obtained requires higher levels of abstraction. The epistemic account of data structure leads to the notion of low-level instructions that allow action-control. The ontological account will lead us through higher aspects of data representation and control, corresponding to different philosophical characterizations of the notion of information.

Operational information: controlling structured data

The philosophical analysis of low-level information requires explaining the process of data structuring. The ontological view on structure refers to essential properties of data and it requires expressing how their properties are actualized. The epistemic view on structure sees it as a necessary result of our knowing process and it immediately leads to a procedural account of actions. Both require explanation of how communication of data at the processor level is obtained. Technically, this is explained by low-level languages. Conceptually, it requires defining *actions* in terms of *operations,* so as to justify our *knowledge-that* in terms of *knowledge-how*.

A low-level language is a program that takes textual instructions and turns them into appropriate arithmetical and logical operations on numbers and bits, then correctly executed by the processor. Such a program is called an *assembler.* An assembly program is thus a series of *operations* on values to be performed on the physical locations known to the processor. Its syntax includes instructions such as *jump*, *loop, load*; values such as *0xff* (255 in decimal) and actual register numbers, e.g. *16* or ports, e.g. *DDRB* (to control specific pins on an Arduino board). As an easy example, the programmer who expects the machine to sum together any two positive inputs from registers will work with few lines of code, e.g. in the format for 16bit addition:

```
DATA SEGMENT
NUM DW 1234H, 0F234H
SUM DW 2 DUP(0)
DATA ENDS
CODE SEGMENT
ASSUME CS: CODE, DS:DATA
START: MOV AX,DATA
MOV DS,AX
MOV AX,NUM ; First number loaded into AX
MOV BX,0H ; For carry BX register is cleared
ADD AX,NUM+2 ; Second number added with AX
```

```
JNC DOWN ; Check for carry
INC BX ; If carry generated increment the BX
DOWN: MOV SUM,AX ; Storing the sum value
MOV SUM+2,BX ; Storing the carry value
MOV AH,4CH
INT 21H
CODE ENDS
END START
```

An assembly program has its own syntax and construction rules, which determine the correctness of the execution. It allows an agent to structure physical data, i.e. it permits control over the physical layer by defining what *operation* (at machine-code level) is to be performed in order to execute the required *action* (at digital data level): for example, it allows assigning a value to an address or directing an output to a port.[3] We will say that machine code embeds a notion of **operational information** on structured, physical evaluated variables (Figure 10.2, see also Chapter 9).

Take a look at Figure 10.4, the essential element at this level is the use of a language to add structure to data. Such language is syntactically well-defined, hence structure is defined by correctness. Moreover, the language imports semantics as a way of denoting the physical entities that constitute the ontological domain of the relation language-objects. The semantics of data structuring, i.e. the range of available operations definable at machine-code level, is fixed by the physical layer information operates on: code 0xFF can only mean the decimal translation of 255; DDRB can only be used if the underlying hardware has a port that deals with a given set of pins; *jump* will correctly work if the specified address exists, and so on. Change the physical elements and the semantics changes accordingly. Here the semantic relation is not intended in terms of compositional rules and instructions that can be freely designed and modified by the agent, but is fixed by the ontological domain. This specific syntactic-semantic structure qualifies operational information as **well-formed performative data**.

The switch from information as structured data to operational information as its control device establishes therefore a semantic relation which maps ontology (of physical entities and physical actions) to a language (of operations). The corresponding satisfiability relation

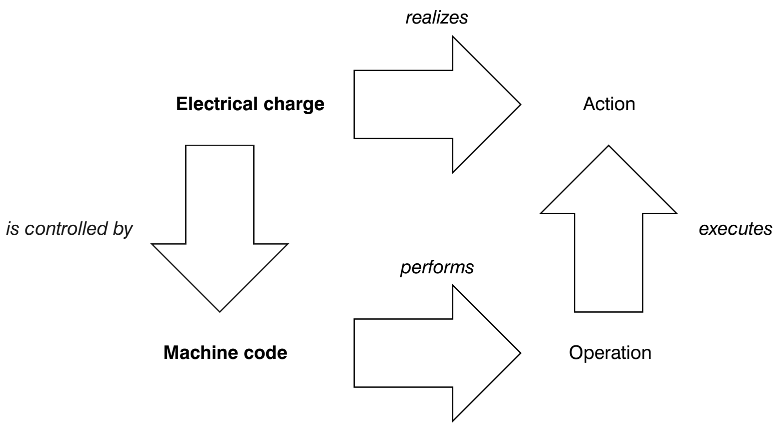


Figure 10.4 Operational information

of operations by actions can be interpreted in terms of execution. The ontology and the epistemology of structuring assign different priorities to action and operation. If the relation of structuring performed by the language of machine code is intended as inherent to data (i.e. understood from an ontological viewpoint), it will not be *provided by* operations, but only *described by* them. This implies that abstract data (like a value jump to an address) enjoys a structuring property that exists independently of the contingent action actually performed (with a given value and at a given address). The action, in turn, is explained in terms of implementation of the abstract case, corresponding to a valid change generated by the action in the system. Hence, for the ontological view, structure is *a priori* and inherent to data. If, instead, one looks at the structure as the result of the operation obtained by execution of strings of machine-code language, the result is *a posteriori* in the domain of reference of the language and it is *assigned* by operations.

Instructional information: programs and their semantics

Syntactic correctness and a denotational relation of satisfiability are the elements implemented at the level of actions-structuring by machine-code language operations. When data is abstracted from the physical layer, i.e. the specifics of the material execution and the ontology denoted by machine code are forgotten, a new control device is required to account for the meaning of computation. This time, the ontology of operations is understood as the reference domain of *instructions;* the latter, in turn, constitute a control device which corresponds, in the practice of Computer Science, to establishing the *interface* between the user and the machine language by means of a *programming language*.

A high-level program *denotes* the executable strings of low-level information of machine code. Each family of such languages interprets differently their linguistic constructs and the related semantics, here simplified in the following main distinction:

- *Declarative programming* refers to languages describing what computations (i.e. the low-level operations) should be performed. Functional languages (like Haskell, LISP, or JavaScript, although with relevant differences) are members of this family, with functions semantically defined by their input-output types (*signature*);
- *Imperative programming* refers to languages where the program is construed around states and actions telling *how* to change such states. Object oriented languages (like C or Java) are members of this family, with objects used to define every element and making use of various concepts and operations to re-use them.

A programming language offers a new control structure to be analyzed by properties of the program. While machine-code language provides a semantics for the execution (action), a programming language provides a semantics for the computation (operation). This is explained usually by referring to two main categories:

- *operational semantics:* it syntactically proves properties of the program in terms of rules from logical statements that formalize its states and procedures;
- *denotational semantics:* it compositionally expresses properties of the program in terms of formal statements that map to each syntactic object a mathematical one.

Each syntactic construct in a programming language expresses a functional executable procedure on the lower level of the machine language. This association of the high-level to the low-level language is nowadays mostly executed automatically through the processes of compiling, followed by the process of linking for the creation of an executable file. As a language, this novel control structure requires (again) correctness of its syntactic nature: the *compiler* is charged with the task of discovering syntactically incorrect code. But while the semantics of machine code was defined by the physical level, this new language instantiates a novel semantic relation with a domain of *abstract entities*. Let us extend the example from the previous section. To automatically produce and control the machine code to structure together the input values of two registers, the programmer will write some program, e.g. in C:

```
int main() {
int a, b, c;
printf("Enter two numbers to add\n");
scanf("%d%d",&a,&b);
c = a + b;
printf("Sum of entered numbers = %d\n",c);
return 0;
}
```

The C code above is syntactically well-formed, which guarantees control over operations at machine-code level (i.e. the code will compile correctly). Moreover, it *expresses* an instructional information of the form: "given positive integers *a,b,c* perform the operation $a+b=c$ and print the result *c*." The elements denoted are no longer the implementable names for physical locations and circuit-closing operations, but abstract objects and their properties, such as *integer number* and *sum operation*. We will refer to this new level of abstraction as **instructional information** (Figure 10.3).

The information content of the machine language (Figure 10.5) is now *denoted by* the language of the program in terms of instructions;[4] and the latter is *satisfied by* operational information in machine code. The new control device is represented by *syntactically correct* strings of programming code matching *appropriate* executable strings of machine code; the *meaning* of the former strings being given in terms of a domain of interpretation for the data as abstract objects and their properties, the new ontology of the language at hand. This syntactic-semantic structure qualifies instructional information as **well-formed, meaningful data**. Meaning is acquired at this stage by evaluating whether an operation (and in turn an action) is obtained at the implementation level. There is still no alethic assessment: like in the case of an order to a person, it makes no sense to ask of a piece of code in itself whether it is true or false.

But the programmer does not only want to know whether the produced code will execute some operation in a given domain of objects, making *something* happen at machine-code level. The latter evaluates instructional information in view of the implementation. Instructions need to be evaluated also with respect to their *intended* meaning, i.e. the abstract objects and the operations defined on them. These reflect the *algorithm* implemented by the current program: the programmer wants to know whether the code written will make happen what she is expecting. In the following section, we consider this further abstraction level, to evaluate computation and its informational content in view of purpose and design, and to explore how further epistemic and alethic conditions are involved.

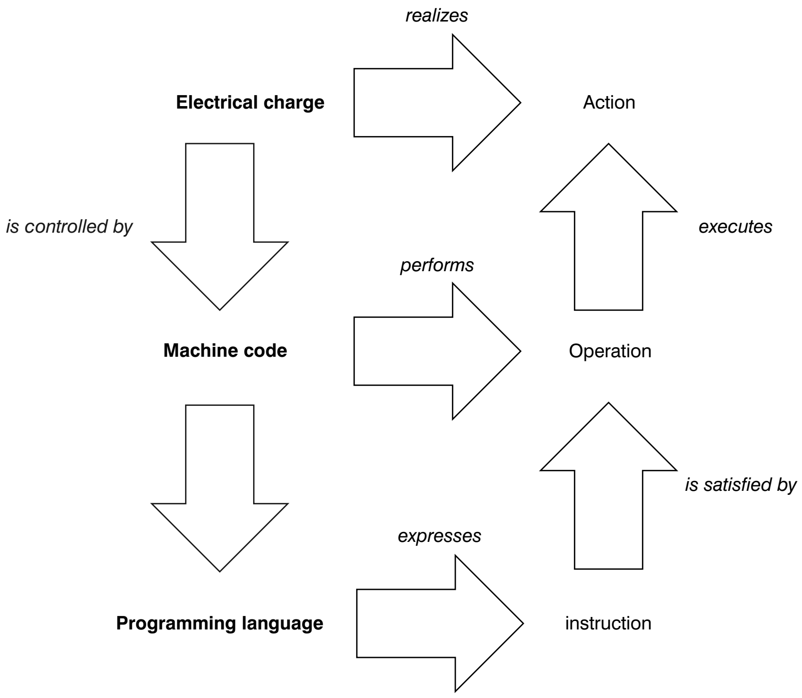


Figure 10.5 Instructional information

Abstract information: algorithm, design and purpose

The presence of a language at machine code and programming levels allows a mapping between symbols and meaning. When instructional information is not taken as an abstraction of the operational level, but as an implementation itself, the meaning of instructions is given by what *they are supposed* to make happen. This next level of interpretation for the notion of information within the computational paradigm is given by the purpose and design according to which a program implements (as efficiently and precisely as possible) an algorithm. This requires a further re-definition of information in view of its abstract content and epistemic value.

An algorithm is the abstract representation of a mathematical function required to fulfil a task. In our case, such task is represented by the *expected* machine behavior. Although it is debated how algorithms in Computer Science should be formally understood,[5] an abstract definition is in terms of a Turing computable or general recursive function implemented by a program, say for example one from natural numbers to natural numbers such as the sum of two integers.[6] A programmer who wants the machine to sum together any two positive inputs will write a program which implements a set of rules, for example as follows:

```
1  Read the Values of A and B
2  If A and B $\geq 0$, SUM = A+B. Display SUM. Stop.
3  Otherwise, Return ERROR: `No positive inputs'. Stop.
```

Besides the quantitative notion of algorithmic information,[7] the informational content of an implemented algorithm can be defined by abstraction and expressed in terms of the designer's intention to solve a formulated problem. For example, the designer might wish to know some physical quantity, for which she devises rules to perform a task, offering a corresponding solution, in our case the sum of two positive integers. The choice of which step to apply is crucial, because some solution will be *correct* to solve the problem at hand, others will not. It is in view of the designer's intention that the algorithm is considered a correct mathematical representation of the intended task and, in turn, the written program is deemed correct or not. Conversely, the program has to implement correctly the instructions expressed by the algorithm. Correspondence by implementation here is a relation of adequacy, not of mirroring. Accordingly, the content of an algorithm can be defined as **abstract, correctness-determining information** (see Figure 10.4).

As the informational content of the algorithm determines the correctness of all the lower level implementations, its semantic value cannot be dependent from those implementations. It has to be evaluated in terms of the ontology of the algorithm and the epistemology of its design. The former refers to the abstract representation of a function*,* akin to a mathematical model. The relation between mathematical and computational abstraction is not trivial. The algorithm is the mathematical description of the program functional specification and as such it establishes correctness for the material artefact running the program. In this sense, it represents a normative definition of the computational instrument, see (Turner, 2011). In software engineering, the design of the specification of a system is a process that precedes the design of the algorithm that has to satisfy it. Such a process, performed with semi-formal or formal methods such as the Unified Modeling Language (Fowler 2003), State Transition Diagrams or flowcharts, is meant to offer a representation of the system as intended by the designer. The result is a representation that expresses the intended meaning of the system ("what is that the system should do?", "which problem should it solve?"). The algorithm then expresses the required instructional setting ("how is the system supposed to work?", "how should it solve the given problem?") in an *abstract* way, independent from any language-specific syntax.

From the point of view of our analysis of information in Figure 10.6, the specification and the algorithm together offer a definition of the computational system and an image of the artefact that has to implement it. Besides, this pair has to match the intention of the designer. Here the epistemic reading completes the picture. Correctness becomes not only an analytic definitional property, in the sense of being the correct decomposition of concepts required to define a function (and eventually its execution). It has to be designed correctly to satisfy the required function for the problem at hand, a synthetic composition of problem and solution.

The correctly designed algorithm acts now as an information-hiding device with respect to both implementation (the instructional information level) and the working device (the operational information level). When defining the algorithm, details essential to the implementation and execution level are ignored, but their details can be reconstructed when required.[8] The designed algorithm immediately determines the correct implementations in any given algorithm and then in any language. From the point of view of the properties of the information, each construction made according to the rules defined by the algorithm represents a *true instance* of the abstract model defined by the specification and the algorithm. At this stage, the abstract nature of the informational content of an algorithm also defines *truthfulness* of its instances, and not just their correctness. The defining property of the informational content at the level of the algorithm is that every lower layer can be defined as a *true* and *correct instance* only in view of this one. Hence, the (full) informational content of an algorithm can be defined as

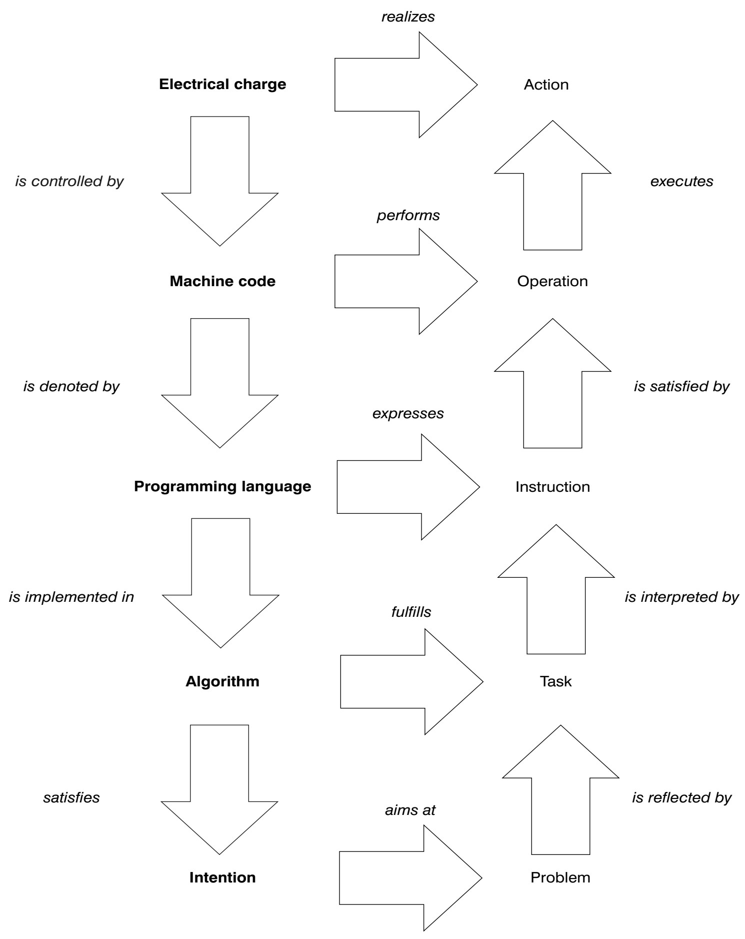


Figure 10.6 Abstract information

abstract, correctness- and truth-determining information. To further qualify this claim we shall reconsider the full computational process in the next section.

The information flow of the computational process

The various notions of information from the literature in mathematics, epistemology and philosophy arise within the computational process. Each notion emerges naturally, defining the ontology and the epistemology of computation. The former is given in view of the relation of *abstraction-implementation* present at each level of the computational process, realizing *syntax* and *semantics*. The latter is instantiated in terms of the *control structure* that each abstraction level performs on the lower one(s).

Let us recall the abstraction-implementation relations. Structured physical data is the informational content where only quantitative on/off relations of bits are at stake. Electrical charges are the domain of reference of machine code, which represents its control structure. At this higher level, the only requirement is that assembly language is guaranteed syntactically well-formed by automatic compilation. This quantitative information has in itself no semantic, nor alethic value. This abstraction level sees action-control performed by operations of machine code in terms of signaling and communication, a task notoriously fulfilled by the Mathematical Theory of Communication, or Shannon's Information Theory (Shannon 1948), see Chapter 4. Next, programs are abstractions from machine code. The syntax of any given correct string of instructions can be different, depending on the specifics of the language, but it will denote the same operation to be performed at machine-code level. For the corresponding control structure, strings of instructions in a programming language range over operations of machine code. But programs are also implementations of algorithms. Any instruction interprets a task and as such it is loaded with the meaning defined by the designer's task. This requires that the *correct* syntactic string in the language expresses the *appropriate* instruction from the algorithm.

The implementation side of this relation is instantiated as follows:

bits – machine code – programming language – algorithm.

Its informational content is characterized by syntactic correctness (at the level of actions, instructions and operations) and a meaning relation (by interpretation of a task by instructions and their implementation by actions): a correct operation is evaluated in terms of the instructional information expressed in some programming language; and a correct language implementation is evaluated in terms of the abstract information expressed by the algorithm. Information is composed here by: physical data in the circuitry of the machine, taken as a relational entity that establishes ontological and epistemic difference; correctness as a property of controlled operations, which relate to the physical layer and can be interpreted independently from the actual operation of coding; finally, the proper meaning is given in the relation to instructions, evaluated in view of the task to be performed, fulfilling also a performative role.

This qualification of the information flow corresponds to *well-structured meaningful data*, which is usually given as the *standard definition of information* (SDI), see (Israel, Perry 1990), (Devlin 1991), (Floridi 2005, 2014), see also Chapter 6. This notion of meaningful data satisfies all cardinal principles of SDI:

- *Typological Neutrality*: information cannot be dataless, and everything can be a datum;
- *Taxonomical Neutrality*: a datum is a relational entity, so is information;
- *Ontological Neutrality*: data implementing information are physical;
- *Genetical Neutrality*: data (and therefore information) can have a semantics independently of any informer;
- *Alethic Neutrality*: meaningful and well-formed data qualify as information, no matter whether they represent or convey a truth or a falsehood or have no alethic value at all.

It is thus clear that *meaning and correctness* are here the basic criteria of evaluation: is a given set of machine-code operations correct to perform some desired physical actions in a given physical environment? Does a given program express the correct instructions to obtain a certain output (functional correctness)? Finally, when the algorithm is taken operationally, is a given set of abstract rules the correct way to characterize the intention to obtain a result or resolve a problem?[9] No truth or falsity is conveyed, as it makes no sense to predicate truth of a list of instructions, be those expressed in common natural language or in the syntax of a given programming language.

The corresponding abstract relation is instantiated by the informational flow between

output – program – algorithm – intention.

This abstraction requires considering the output, the program and the algorithm as mathematical entities, realizing a definition of the specification. At this stage, an analysis in terms of correctness and meaning seems to remain unsatisfactory. As a mathematical model, the algorithm has instances that satisfy it, and some that do not, while each of those can be taken as correct on its own (when accounting for different tasks). This happens when the algorithm is interpreted as a recursive definition of a function in a specification (for example of the sum operation in terms of the successor function) defining the *model* of the task intended by the designer; such model will have (possibly many) abstract machine(s) as its instances and any implemented expression (for example in the C language, but also in natural language) of that algorithm will be a *true* realization of any such instance. This way of expressing the relation between abstract implementation, algorithm with its output and intention is akin to the definition of semantic satisfiability in a model, which does not define correctness, but *truth*. The informational content of such relation requires now for its definition reference to the semantic conception that encapsulates truth, see (Floridi 2011) and Chapter 6. The information flow of the full computational process is thus based on meaningfulness, correctness and truth at different stages. We recall all these stages again in Figure 10.5. This description emphasizes the duality inherent to computation as an abstract process that is instantiated in a mechanical artefact, an issue that has affected crucially the idea of program verification, see (Fetzer 1988).

Verifying information

The informational view of digital computation presented here can be matched against formal verification as the reconstruction of the correct mapping between the instructional information level of the executable program and the abstract information level of the algorithm and its intended design. The practice of verification is the process of testing, which includes also empirical verification. It consists in checking whether the execution of a given program is error-free, in the sense of returning the expected behavior as by definition of its design and by realization of its purpose. Software testing is the general process of checking that the requirements guided by the design are met and is divided into various tasks:

1 check that the program responds to all valid inputs;
2 check that the computation is performing with respect to time;
3 check that the system is acceptable for a standard user;
4 check that the system runs well on the intended environment (physical and virtual).

These tasks describe a modular enterprise: from the verification of the functionality of a single piece of code, e.g. implementing a function or an object (unit testing); through checking the interface of different units (integration testing); to checking validity of data passing among interfaces (component interface testing); up to the verification of the fully integrated system (system testing).

Compilation and linking processes are nowadays mostly automatically generated; hence, the verification of the operational information results in a combination of testing the functionality of the automatic compiler software that generates the machine code, and of the various techniques comprising hardware testing. For the former, one is considering another piece of software on its own, hence the following steps from unit testing onwards apply. For the latter, it mainly consists of so called stress testing, to establish the limit of system's stability and performance, focusing on the physical execution. Unit testing is the basic verification of the correctness of instructional information (i.e. at program's level). It is in the first place an evaluation of the purely syntactic structure of the instruction and, in turn,

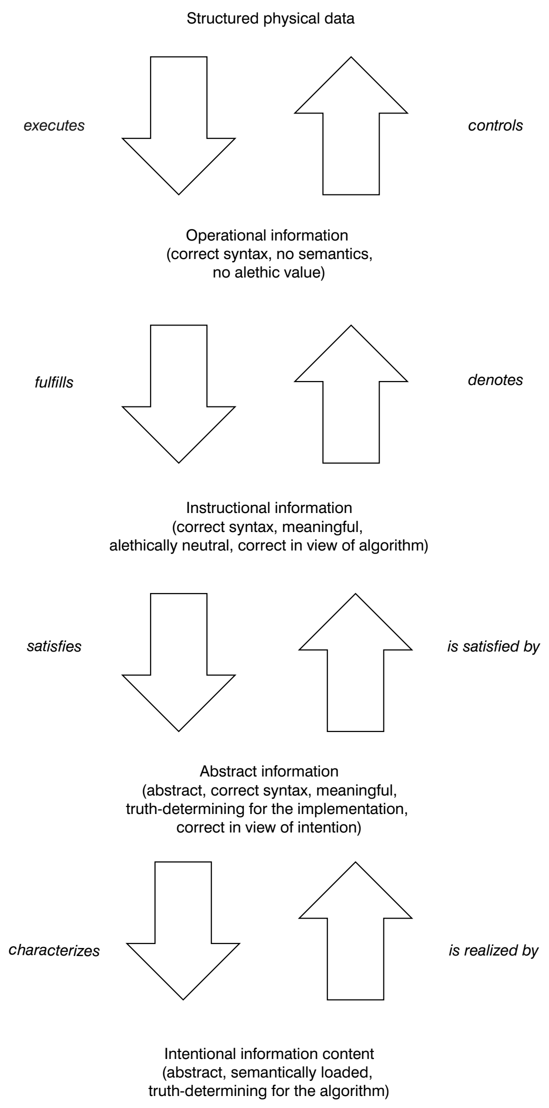

Structured physical data
executes
controls
Operational information (correct syntax, no semantics, no alethic value)
fulfills
denotes
Instructional information (correct syntax, meaningful, alethically neutral, correct in view of algorithm)
satisfies
is satisfied by
Abstract information (abstract, correct syntax, meaningful, truth-determining for the implementation, correct in view of intention)
characterizes
is realized by
Intentional information content (abstract, semantically loaded, truth-determining for the algorithm)

Figure 10.7 Syntax, semantics and control

of its execution as operation. The syntactic correctness check is required to establish whether the chosen procedures are implemented correctly according to basic syntactic rules of the language, e.g. by ensuring that C expressions terminate with a semi-colon, that opening brackets always have matching closing ones, and that operating on positive integers has the starting input defined using a failure on the (n<=0) expression. To check the meaningfully loaded instructional information, one requires not only to know whether a specific function (or object) is correctly coded; one also requires to know whether the function (or object) is what it is needed in order to express the instruction for the desired operation. Integration testing can be seen as the checking of meaningful composition of elements of an analytic definition, and as such it is meant to verify truthfulness. For example, given the intended operation is the one of sum on integers, it requires using the appropriate input/output signature int->int (i.e. that the function takes integers as inputs and returns an integer as output) and the use of the sum = a + b function. For this purpose, a composition that uses an input/output signature real->int (i.e. from real numbers as input to integers as output) using the function a * b (i.e. by multiplication) would not satisfy the analytic definition of sum of integers. While correct in view of another algorithm, this implementation would not be a true instance of the model defined by the intended one. Component interface testing is then a similar analyticity test in the composition of different functions for the purposes of defining a more complex algorithm. Finally, testing the integrated system has to reflect the physical execution and its effectiveness at the implementation level, the correctness of the instruction and of the rules that define them at the abstraction level.

From the informational viewpoint, verification corresponds therefore to checking meaningfulness, correctness and truth at both the implementation level (*bits – machine code – programming language – algorithm*) and the abstraction level (*output – program – algorithm – intention*). The implementation level requires checking that physical execution restores downwards the valid reference domain for the program, going from instructions to operations to actions. The abstraction level moves upwards from the physical (operational) and syntactical (instructional) formulation of the computation to reconstruct the semantic information content at the level of algorithm, intention and purpose. This means also verifying that the defined computation can be subsumed as a true instance of the (intended) definition of the system.

Computing as science of information

Our view on information in the computational process touches on Computer Science and its philosophy, including aspects related to Computer Engineering, Information Systems, Information Technology and Software Engineering.[10] For this larger understanding of the discipline, the term Computing seems preferable. In this context, the view that Computing can be understood as a Science of Information is not new. Since the late 1960s, practitioners and philosophers have presented views based on a notion of information.[11] It is striking, though, that these views focus over largely distinct aspects of computing, fitting different notions of information to their tasks. In some cases, information refers to data processing; in other cases, physical, methodological or working principles on data structures are intended.[12] Our analysis has highlighted the need for conceptual precision in identifying what *information* means at each level of abstraction, hence for each of the different sub-disciplines in Computing. We examined information structures in view of a syntax-semantic relation between a domain of objects and a language at various LoAs. Object domains, in terms of implementations, represent the *ontologies* of Computing (from bits to algorithmic constructs, but also including their physical implementations, such as networks). Languages constitute control means (from machine-code strings, through representations such as FSMs, to designers' intentional states), expressing the *know-how* over ontologies; in this sense, they can be subsumed under the general heading of epistemology. Such a conceptual distinction is often blurred in practice, where a design task can be updated many times in view of insights coming directly from prototyping. A philosophical formulation of (digital) Computing can be presented then in terms of information as follows:[13]

> Computing is the systematic study of the ontologies and epistemology of information structures.

It is essential that the qualification of *systematic study* appealed by in this definition be understood in a broad methodological sense: it refers to designs, formal models, blueprints, testing, up to include those fragments of the discipline that deal with evaluation, behavioral and experimental methods, including e.g. Human-Computer Interaction and Computational Simulations.

Concluding remarks

The standard digital computational process includes all the various aspects of the notion of information: the quantitative definition of bits, the syntactic construction of operations, the meaning of instructions, the abstract format of algorithm and the epistemically loaded designer's intention. We have analyzed both its ontology through the syntax-semantics divide, and its epistemology, in terms of control structures. The information flow that results from them is based on the relation abstraction-implementation. At each level, important philosophical issues arise, in particular related to correctness, meaning and truth. The relational notion of correctness is clearly a common trait to evaluate information throughout the whole computational process. Meaning is a distinctive step to move from mechanical operations to their instructional counterparts. Truthfulness arises at the highest level of abstraction, when algorithms are accounted as mathematical structures defining models. Each such description defines a different, essential format of the notion of information within the philosophy of computer science and in turn offers a better definition of Computing as a Science of Information.

Acknowledgements

I wish to thanks Liesbeth De Mol, Matti Tedre and the participants at lectures given at IHPST, Paris and DSV, Stockholm University where preliminary versions of this work have been presented.

Notes

1 This list is essentially incomplete. A more systematic presentation is available through the Association for Computing Machinery Classification System, see www.acm.org/about/class/ccs98-html.
2 Under the general heading of *rule* and *dependency*, we subsume the whole fetch-decode-execute-check cycle of the program instructions performed by the CPU.
3 Under the general heading of *operation*, we subsume the whole set of arithmetical operations performed e.g. on the stack memory structure or on their simulation by CPU machine registers.
4 Under the general heading of *instruction*, we subsume the basic list of structures sufficient to program any algorithm in any language: assignment, sequencing, branching and iteration.
5 The two most known and alternative views see algorithms respectively as *abstract machines* or as *recursive definitions*, see e.g. (Moschovakis 2001), (Blass, Gurevich 2003).
6 For an easy introduction to the notion of Turing Machine and the computable functions in relation to information, see (The Pi Network 2013), Chapter 12.
7 For Algorithmic Information Theory see Chapter 5.
8 For different views on this debate, see e.g. (Colburn, Shute 2007), (Turner 2011).
9 The algorithm *characterizes* the intention of the designer, in the same way as a characteristic function is a way of defining precisely a set by saying for any possible object whether it is a member of that set or not.
10 See e.g. https://www.acm.org/education/curricula-recommendations for the ACM Curricula Recommendations related to each sub-field.
11 A good selection of brief articles introducing different positions is available in (Denning 2010).
12 For an overview of interpretations on the nature of Computing as a discipline, see (Tedre 2014).

13 The present definition of Computing as a discipline dealing with information structures at ontological and epistemological levels is justified in the present chapter only in view of the digital computational model. It could be argued that paradigms such as bio-computing and analogue computing deal with a similar conceptual structuring of information in some format (e.g. referring to chemicals and water droplets as computational objects hiding more basic informational quantities). A further argument to restrict the present analysis to digital computing only is historical: certain older models of computing would struggle inside this definition, and so might be for future computational models. A more extensive analysis addressing these issues is outside the scope of the present contribution.

Further reading

For an overview of the topics and debates within the philosophy of Computer Science, see (Rapaport 2005) and (Turner, 2014). For an advanced introduction to assembly for some common types of processors, see e.g. (Dandamudi 2005). For a more extensive analysis of the semantics of programs, see Turner (2007) and White (2008). For an analysis of errors in information systems, see (Primiero 2014); for one specifically devoted to computational systems, see (Fresco, Primiero 2013). (The Pi Network, 2013) offers an overview of the philosophical issues related to information and Chapter 12 is specifically devoted to information and computation. For an overview of the information-hiding process by abstraction in terms of information, see (Primiero, 2009). (Angius 2013) analyzes software verification in relation to the philosophical categories of abstraction and idealization.

References

Angius, N, 2013. Abstraction and Idealization in the formal verification of software systems. *Minds & Machines*, 23(2):211–226.

Blass, A., Gurevich, Y. (2003). Algorithms: A Quest for Absolute Definitions, *Bulletin of the EATCS*, 8: 195–225.

Colburn, T., Shute, G. (2007). Abstraction in Computer Science, *Minds and Machines*, 17(2): 169–184.

Dandamudi, S.P. (2005). *An Introduction to Assembly Language Programming,* Texts in Computer Science, New York: Springer.

Denning, P.J. (1985). What is Computer Science? *American Scientist*, 73: 16–19.

Denning, P.J. (2010). Ubiquity Symposium `What is Computation? (ed.), *Ubiquity,* vol. 2010, no. October (2010).

Devlin K. (1991). *Logic and Information*, Cambridge: Cambridge University Press.

Fetzer, J.H. (1988). Program Verification: The Very Idea, *Communications of the ACM*, 31(9): 1048–1063.

Fowler, M. (2003). *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, Reading, MA: Addison-Wesley.

Fresco, N., Primiero, G. (2013). Miscomputation, *Philosophy & Technology*, 26: 253–272. DOI: 10.1007/s13347-013-0112-0.

Fresco, N., Wolff, M. (2013). Information Processing and the Structuring of Data, *The Israeli Society for History & Philosophy of Science Fourteenth Annual Conference.*

Floridi, L. (2005). Is Information Meaningful Data?, *Philosophy and Phenomenological Research*, 70(2): 351–370.

Floridi, L. (2011). *The Philosophy of Information*, Oxford: Oxford University Press.

Floridi, L. (2014). Semantic Conceptions of Information, *The Stanford Encyclopedia of Philosophy* (Spring 2014 Edition), Edward N. Zalta (ed.), http://plato.stanford.edu/archives/spr2014/entries/information-semantic/.

Hartmanis, J., Lin, H. (1992). What is Computer Science and Engineering? In Hartmanis, J. and Lin, H., (eds), *Computing the Future: A Broader Agenda for Computer Science and Engineering*, Washington, DC: National Academy Press, pp. 163–216.

Israel D., Perry J. (1990). What is Information? In Hanson, P. (ed.) *Information, Language and Cognition*, Vancouver, University of British Columbia Press, pp. 1–19.

Moschovakis, Y.N. (2001). What Is an Algorithm? In Engquist, B. and Schmid, W. (eds.) *Mathematics Unlimited – 2001 and Beyond*, New York: Springer, pp. 919–936.
Newell, A., Perlis, A. J., Simon, H. A. (1967). Computer Science. *Science*, 157(3795): 1373–1374.
Newell, A., Simon, H.A. (1976). Computer Science as Empirical Inquiry Symbols and Search. *Communications of the ACM*, 19(3): 113–126.
Primiero, G. (2009). Proceeding in Abstraction. From Concepts to Types and the Recent Perspective on Information, *History and Philosophy of Logic,* 30(3): 257–282.
Primiero, G. (2014). A Taxonomy of Errors for Information Systems, *Minds and Machine,* 24(3): 249–273.
Rapaport, W.J. 1999, Implementation Is Semantic Interpretation, *The Monist*, 82(1): 109–130.
Rapaport, W.J. (2005). Philosophy of Computer Science: An Introductory Course, *Teaching Philosophy* 28(4): 319–341.
Shannon, C.E. (1948). A Mathematical Theory of Communication, *Bell System Technical Journal*, 27: 379–423 and 623–656, July & October, 1948.
The Pi Network, (2013). *The Philosophy of Information – A Simple Introduction*, Society for the Philosophy of Information, available at http://www.socphilinfo.org/teaching/book-pi-intro.
Tedre, M. (2014). *The Science of Computing: Shaping a Discipline*. Abingdon: CRC Press/Taylor & Francis.
Turner, R. (2007). Understanding Programming Languages, *Minds & Machines*, 17(2): pp. 203–216.
Turner, R. (2011). Specification, *Minds & Machines*, 21(2): 135–152.
Turner, R. (2014). The Philosophy of Computer Science. In Zalta, Edward N. (ed.), *The Stanford Encyclopedia of Philosophy* (Summer 2014 edition), http://plato.stanford.edu/archives/sum2014/entries/computer-science.
White, G. (2008). The Philosophy of Computer Languages. In Floridi, L. *The Blackwell Guide to the Philosophy of Computing and Information,* Oxford: Blackwell Publishing, pp. 237–247.