Philos. Technol.
DOI 10.1007/s13347-012-0098-z

SPECIAL ISSUE

Programming Languages as Technical Artifacts

Raymond Turner

Received: 28 September 2012 / Accepted: 26 December 2012

Abstract Taken at face value, a programming language is defined by a formal grammar. But, clearly, there is more to it. By themselves, the naked strings of the language do not determine when a program is correct relative to some specification. For this, the constructs of the language must be given some semantic content. Moreover, to be employed to generate physical computations, a programming language must have a physical implementation. How are we to conceptualize this complex package? Ontologically, what kind of thing is it? In this paper, we shall argue that an appropriate conceptualization is furnished by the notion of a *technical artifact*.

Keywords Programming languages · Semantics · Technical artifacts · Philosophy of mathematics · Philosophy of technology

1 Programming Languages

What kind of thing is a programming language? Is it a language defined by a formal grammar, a mathematical object, something only brought into existence by a physical implementation, or somehow a combination of all three?

On the face of it, it is just a language determined by a formal grammar. However, there must be some account of what the constructs of the language are taken to mean. Without such, we would not be able to articulate the meaning of any program. At least we would not under the rather plausible assumption that the meaning of a complex program is compositionally dependent upon the meanings of its components. Presumably, programs are normally intended to meet some independently given specification. But without some semantic account of the program, we would

R. Turner (✉)
CSEE, University of Essex, Wivenhoe Park, CO43SQ Colchester, UK
e-mail: turnr@essex.ac.uk

Published online: 13 February 2013

not be able to articulate the required relationship between what a program means and what its specification insists it should achieve. So we would have no notion of semantic correctness for programs. This is not only a practical demand but also a conceptual one: without a notion of what it is for a program to be correct relative to a given specification, any program would be as good as any other. And this is nonsensical. Programming would be a trivial activity with no notion of correctness.

Moreover, unless one is only concerned with the construction of programs for purely mathematical purposes (e.g., to demonstrate the existence of an algorithm) even more is required. Programmers write programs to carry out physical computations, presumably ones that cannot be done by hand. For example, a program that processes visual information must be underpinned by a physical device that actually performs the task. For this, a physical implementation of the language is required.

So the purely grammatical thing that we might initially think of as the programming language is not the complete thing that is used by programmers. We suggest that this consists of the package that is constituted by the grammar/semantic definition on the one hand and a physical implementation on the other, i.e., these packages are made up of an abstract mathematical notion (grammar[1]/semantics) and a concrete physical one (implementation).

At the level of individual programs, this duality has been observed by many authors (Colburn 2000; Colburn and Shute 2007; Irmak 2012; Moor 1978). From the early days of the subject, we have the influential paper of Moor, who, in the course of demolishing the apparent clarity of the software/hardware distinction, writes:

> It is important to remember that computer programs can be understood on the physical level as well as the symbolic level. The programming of early digital computers was commonly done by plugging in wires and throwing switches. Some analogue computers are still programmed in this way. The resulting programs are clearly as physical and as much a part of the computer system as any other part. Today digital machines usually store a program internally to speed up the execution of the program. A program in such a form is certainly physical and part of the computer system. (Moor 1978)

The following is of more recent origin and more explicitly expresses the duality thesis.

> Many philosophers and computer scientists share the intuition that software has a dual nature (Moor 1978; Colburn 2000). It appears that software is both an algorithm, a set of instructions, and a concrete object or a physical causal process. (Irmak 2012)

Seemingly, programs are both abstract objects and physical causal processes. However, it is from the whole language that the duality emanates. Although many authors talk as if programs may be detached from host languages and seem to assume

[1] We shall use the terms *grammar/grammatical* and *syntax/syntactical* interchangeably.

that individual programs have stand alone meanings and implementations, this is not coherent. Programs have their meanings given by the meanings of their contained constructs, and, generally, the semantics must preserve the meanings of the constructs across programs. Confusion over this issue stems from a failure to cleanly distinguish the inherited semantics of the program from an external specification of what it is supposed to do (Sprevak 2010).

However, the same conceptual questions arise for both languages and programs. How are the two manifestations, the abstract and the concrete, related? We shall argue that the appropriate way to conceptualize matters is via the notion of a *technical artifact* (Franssen et al. 2009).

2 The Duality of Technical Artifacts

Technical artifacts include all the common objects of everyday life such as televisions, computer keyboards, paper clips, telephones, smart phones, dog collars, and cars. They are intentionally produced things. This is an essential part of being a technical artifact. For example, a physical object that accidently carries out arithmetic is not by itself a calculator. This teleological aspect distinguishes them from other physical objects.

Natural objects just happen. Descriptions of them as kinds, or in the case of biological objects in terms of their function, are post hoc and best seen as theories about them. Moreover, technical artifacts can only fulfill their intended function because of their actual physical structure. This is quite different to social constructions such as money where there is no necessary connection between function and physical structure.

This insistence on the need for a purpose or function has led philosophers to argue that technical artifacts have a dual nature fixed by two sets of properties (e.g., Kroes 2010, 2012; Meijers 2001; Thomasson 2003, 2007; and Vermaas and Houkes 2003):

- Functional Properties
- Structural Properties

Functional properties relate to what the artifact does. For example, a kettle is *for boiling water* and a cycle is *for transportation*. In engineering, Kroes (2010) functional properties are articulated as black box specifications in which the object of design is specified only in terms of its input and output behavior. For example, a kettle might be schematically specified in terms of its function as follows.

Input : *water* **Output** : *boiling water*

On the other hand, structural properties pertain to its physical makeup. They include its weight, color, size, shape, its chemical constitution, etc. In particular, we might insist that our kettle is to be made of copper and holds two pints.

In summary, technical artifacts are individuated by the two descriptions: the functional and structural. The physical thing by itself is not a technical artifact. And of course there can be many such implementations. And each functional description and implementation determines a different artifact.

How does this duality unpack in the case of a programming language package? To address this, we shall take the *engineering design* (e.g., Kroes 2010, 2012) view where the focus is on the specification, design, and construction of the artifact. All sorts of issues arise.

> Technical artefacts are indeed often characterized as intentionally made physical constructions that, on condition that they are functioning and used properly, support users in realizing their goals. Such a rough characterization raises questions about whether technical artefacts are mere physical constructions, about what it means for a technical artefact to function properly or to be used properly, about how technical artefacts are related to human intentions or human goals, or whether there is a clear demarcation line between technical artefacts and natural objects. (Kroes 2012)

These questions have analogs in connection with a programming language package.[2] In particular, we must spell out what constitutes the functional properties, what constitutes the structural ones, examine the various approaches to the relationship between the two, address the question of correctness, and discuss how human intention gets into the picture. These correspond to the following sections.

3 Function

In computer science a functional specification (Turner 2011) informs us how input and output are to be related. This relationship is expressed in a large variety of ways using a host of formalisms and languages. Specialized specification languages such as Z (Spivey 1988) and VDM (Jones 1990) have been constructed to support software specification, and HOL (Gordon 1986) to facilitate the specification of hardware. In a similar manner, formalisms and mathematical systems have been introduced and used for the functional definition of whole programming languages.

On the one hand, the definition of a programming language is the definition of a mathematical object. It functions as a specification when it is given governance over the construction of an implementation of the language. In this role, it provides the criterion of correctness for the implementation. So how is the appropriate concept of functional specification be provided? Before we address this question, we must clear the ground a little.

Cars and computers are complex artifacts. The latter may have a CPU, a motherboard, some memory, a hard drive, a GPU, etc. These various components collaborate to produce the function of the whole artifact. In a similar manner, a programming

[2] Of course programs are artifacts in their own right, but in an inherited way. The distinction might be best seen in terms of a simple analogy. Consider a sophisticated knitting machine with its own input interface. As an artifact, it must be distinguished from any cardigans it produces. Indeed, the situation is even more complex than this: the knitting machine is more like a universal Turing machine, the knitting-machine program is like a computer program (written in its own programming language), and the cardigan is its output.

language is made up of syntactic constructs such as *while loops, conditionals, recursive definitions*, etc. Each of them has an individual function that contributes to the function of grammatically correct programs in the language. So one must provide a functional specification for each such component. In the context of the computer science notion of functional specification, we must provide the input/output behavior for each construct.

A standard way of achieving this is in terms of the impact of each of the constructs on an underlying abstract machine and the functional description takes the form of an abstract semantics for the language (Fernandez 2004; Gordon 1979; Milne and Strachey 1977; Schmidt 1988; Stoy 1977; Tennent 1991; Turner 2007; White 2004; Winskel 1993).

To illustrate matters, we shall employ the following simple abstract machine whose underlying state stores numerical values in locations. We shall represent this as a finite partial function from a finite set of locations ($Location$) to a finite set of natural number values ($Number$).

$$State \triangleq Location \dashrightarrow Number$$

It is partial since it is not assumed that all locations have values in them.[3]

There are two main contemporary approaches to semantics: operational (Fernandez 2004; Landin 1964; Plotkin 1981) and denotational (Gordon 1979; Schmidt 1988; Stoy 1977). In order to illustrate some conceptual issues, we shall briefly discuss them both. We shall use some very simple programming constructs to illustrate what we shall argue yields the appropriate notion of function for programming languages.

3.1 Operational Semantics

We begin with a modern form of operational semantics (Fernandez 2004; Plotkin 1981) where the impact of the constructs on the state is given via rules of evaluation. The primary judgment is written as[4]

$$< P, s > \Downarrow s'$$

and pronounced as $< P, s >$ leads to s'. It is taken to express the judgment that evaluating the program P in state s terminates and returns the state s'. In the following, E are numerical expressions and B are Boolean expressions. We shall ignore their structure and concentrate on the actual $Programs$. Their semantics is given by operational rules. The idea behind them is simple enough: if the premises represent the state changes of the components of the program, then the conclusion represents the state changes of the whole program.

[3]This may be formalized in the abstract data type of finite sets or any functional language; one certainly does not need to assume the whole of set theory (ZF).

[4]This is related to the Hoare notation

$$s\{P\}s'$$

but in our case, the state s' has to be a terminating state. The Hoare notation is closer to the so called *small step* operational semantics.

The most basic program corresponds to the update operation of the machine: it is the simple assignment statement. It is governed by the following rules.

$$\frac{x : Location \qquad E : Exp}{x := E : Program} \qquad\qquad \frac{Val(E, s) = v}{< x := E, s >\Downarrow Updates[s, x, v]}$$

where $Val(E, s)$ is the value of the expression E in the state s and $Updates[s, x, v]$ is the partial function that is identical to s except perhaps that $s[x] = v$. The first rule is a grammatical rule and insists that it generates a program. The second, the semantic one, unpacks the meaning of assignment in terms of the update operation of the machine. According to it, if the execution of E in the state s returns the value v, then the execution of $x := E$ in a state s returns the state $Update(s, x, v)$.

Our first complex construct allows the sequencing $(P; Q)$ of programs.

$$\frac{P : Program \qquad Q : Program}{P; Q : Program} \qquad\qquad \frac{< P, s >\Downarrow s' \quad < Q, s' >\Downarrow s''}{< P; Q, s >\Downarrow s''}$$

The first rule informs us that programs can be sequenced. The second, the semantic rule, demands that sequencing is executed by first executing P in state s. If this yields the state s', then we execute Q in s'. If this returns the state s'', then the execution of $P; Q$ in s returns the state s''.

Finally, to put a bit more meat on things, we consider a slightly more complex construct. The initial rule informs us that we can form an iterative program from a Boolean expression and an existing program. The semantic rules split according to whether the Boolean is $false$ or $true$. If the execution of B in s returns $true$ and the execution of P in s returns s', and the execution of **while** B **do** P in s' yields s'', then the execution of **while** B **do** P in s returns s''. If the execution B in s returns $false$, then the execution of **while** B **do** P in s, returns s.

$$\frac{B : Boolean \qquad P : Program}{\textbf{While } B \textbf{ do } P : Program} \qquad\qquad \frac{< B, s >\Downarrow false}{< \textbf{While } B \; do \; P, s >\Downarrow s}$$

$$\frac{< B, s >\Downarrow true \qquad < P, s >\Downarrow s' \qquad < \textbf{While } B \; do \; P, s' >\Downarrow s''}{< \textbf{While } B \; do \; P, s >\Downarrow s''}$$

To unpack and emphasize the mathematical nature of the semantics, we shall explore matters a little. Observe that we may define

$$< P, s >\Downarrow \triangleq \exists s' \cdot < P, s >\Downarrow s'$$

This defines what it is for a program to terminate when it is evaluated in a given state. We may also define a notion of *equivalence* for programs/operations.

$$P \simeq Q \triangleq \forall s \cdot \forall s' \cdot < P, s >\Downarrow s' \leftrightarrow < Q, s >\Downarrow s'$$

i.e., they are taken to be equal if we cannot tell them apart in terms of their extensional behavior.

Of course, we could carry out a more adventurous mathematical investigation, e.g., prove that the system is consistent, etc. But we have done enough to make the obvious point: the language and its definition constitutes an axiomatic theory where the rules provide an axiomatic treatment of evaluation. From this perspective, a programming

language is an axiomatic *theory of operations* where the relation $\Downarrow$ is taken to be axiomatized by the rules.

Of course, one might claim that these axiomatizations are not always made in the precise axiomatic style given here. Often, the operational definitions are given in natural language. For example, we might just be given something of the following form:

- If the execution of B in s returns $true$ and the execution of P in s returns s', and the execution of **while** B **do** P in s' yields s'', then the execution of **while** B **do** P in s returns s''. If the execution B in s returns $false$, then the execution of **while** B **do** P in s, returns s.

But it is clear that our axiomatic description is nothing more than a reformulation where $< P, s > \Downarrow s'$ expresses the statement that evaluating the program P in state s terminates in the state s', and the conditional statements are recast as rules with the antecedent functioning as the premise, and the consequent as the conclusion. But, usually, both formal and informal accounts are given side by side (Börger and Schulte 2007).

3.2 Denotational Semantics

The alternative approach, the denotational one, provides a direct interpretation into another mathematical theory (e.g., set theory). For the constructs of our language, this might take the following form.

$$\|x := n\|\, s \simeq Updates[s, x, n]$$
$$\|P_1;\, P_2\|\, s \simeq \|P_2\|\, (\|P_1\|\, s)$$
$$\|\textbf{While } B \textbf{ do } P\|\, s \simeq \left\{ \begin{array}{l} s \; if \;\; \|B\|\, s = false \\ \|\textbf{While } B \textbf{ do } P\|\, (\|P\|\, s) \; if \;\; \|B\|\, s = true \end{array} \right\}$$

where

$$\|P\|\, s$$

is the state that results from the evaluation of the program P in state s and where $\simeq$ is partial equality that allows for the fact that programs may not terminate. This yields a version of denotational semantics (Gordon 1979; Gunter 1992; Stoy 1977; Milne and Strachey 1977; Tennent 1991; Winskel 1993) where each program is associated with a partial state to state (set theoretic) function. Notice how the latter requires some justification since there is a hidden *fixed-point* construction.

In some cases, the denotational notion of equality is as finely grained as the operational one of observational equivalence, where, informally, we say that two programs are observationally equivalent if there exists no closed program context C that can operationally distinguish between them. But, sometimes, it is not. In these circumstances, which account is taken to define the programming language as a mathematical entity?

In the way that the two are used in relationship to each other, the way they are treated in the literature (Abramsky et al. 1994; Abramsky and McCusker 1995), the operational account is in the driving seat: the denotational one must agree with the

operational one, i.e., the operational account is taken to define the programming language as a mathematical object.[5] The operational account defines the language as a theory of computation in much the same way that the Lambda calculus is defined by its rules of reduction. On this perspective, the main role of the denotational account is not definitional but to facilitate mathematical exploration of the language.[6]

3.3 Axiomatization

Actually, the operational account may be fundamental in a more traditional mathematical way: it is more fundamental because it seeks to axiomatize the notions of the language directly; it is ontologically more sensitive. Translations into other mathematical systems do not take the primitives of the language as fundamental notions axiomatized by the rules. For example, in the case of our simple constructs, the translations into set theory are not basic axiomatizations of the any primitive notion of *operation*; they are set theoretic interpretations of such a notion as partial functions in ZF. Their success is measured by how well they preserve basic computational intuitions: programs represented as infinite sets do not generally capture the computable nature of programs.[7]

The Lambda calculus is not defined by its set theoretic models but by its rules. Similarly, Turing machines are defined by their rules, not by any extraneous mathematical interpretation. This is consistent with Gödel's remark:

> The greatest improvement was made possible through the precise definition of the concept of finite procedure, which plays a decisive role in these results. There are several different ways of arriving at such a definition, which, however, all lead to the same concept. The most satisfactory way, in my opinion, is that of reducing the concept of finite procedure to that of a machine with a finite number of parts, as has been done by the British mathematician Turing (Gödel 1934).

Presumably, it is in the details of the primitive notion of a Turing machine that made it definitive, not any set theoretic model of Turing machines. In particular, Gödel did not accept that the Lambda calculus version of computability was equally good as a direct formalization of the notion of finite procedure. They may be extensionally equivalent, but they are intensionally different. Turing machines and the Lambda calculus represent different computational paradigms. Interpretations into the same mathematical theory may well obscure this. But to be clear,

[5] Indeed, much work has been done to construct denotational definitions that are in harmony with the operational ones. This often requires the use of a version of denotational semantics based upon games (Abramsky et al. 1994; Abramsky and McCusker 1995; Stoughton 1988; Mulmuley 1987).

[6] This is unlike the classical role of the Tarski notion of truth for predicate logic which is taken to fix the meaning of the logical constants.

[7] Category theoretic interpretations fair much better in giving a more intensional flavor to the interpretation (White 2004) and could be argued to provide an alternative framework for the axiomatization of computational notions.

the main conceptual point of this section does not depend upon the particular technique of formalization; however, they are articulated, the operational rules define the language.

Of course, modern programming languages involve much more complex notions than the toy ones that we have chosen to illustrate our main conceptual point. They involve sophisticated notions of *type*, *process*, *objects*, *classes*, *monads*, *abstract types*, etc. But underlying these more complex notions are fundamental mathematical concepts. For instance, the recent history of type theory (Barendregt 1992; Girard 1971; Mitchell 1996; Pierce 2002; Reynolds 1994) has provided a host of new notions of type (Girard 1971; Reynolds 1994). Cardelli and Abadi (1996) introduces a wide range of theories of objects and classes. The $\pi-$calculus (Milner 1993) belongs to the family of process calculi, mathematical formalisms for describing and analyzing properties of concurrent computation. In turn, these new theories, often modified and simplified, find their way into actual programming languages; they form their conceptual core. For example, the Lambda calculus inspired a whole generation of functional languages, such as Miranda (Thompson 1995), SML (Hanson and Rischel 1999), Haskell (Thompson 2011), etc.

In their definitional/functional guise, programming languages are complex and rich axiomatic theories of *operations*, processes, objects, or whatever fundamental notions are taken to be part of the language. These theories are definitional. This much is in agreement with Hilbert's view of axiomatic systems.

> In my opinion, a concept can be fixed logically only by its relations to other concepts. These relations, formulated in certain statements, I call axioms, thus arriving at the view that axioms (perhaps together with propositions assigning names to concepts) are the definitions of the concepts. I did not think up this view because I had nothing better to do, but I found myself forced into it by the requirements of strictness in logical inference and in the logical construction of a theory. I have become convinced that the more subtle parts of mathematics ... can be treated with certainty only in this way; otherwise one is going around in a circle. (Hilbert 1899).

The axiomatic theories fix the meaning of its internal concepts as implicit definitions. And this is precisely what operational accounts do for programming languages. Potentially, each one offers us an axiomatic theory of computation.

4 Structure

Pure mathematical objects may be given and employed independently of any physical realization. But while in their functional guise programming languages have the form of axiomatic theories, they are more than this: they are intended to be used. But, in order to be used, they have to be *implemented*. A *structural description* of the language must say how this is to be achieved. It must spell out how the constructs of the language are to be physically instantiated.

To illustrate the simplest case, consider the assignment instruction.

$$x := E$$

An implementation might take the following form.

- Physically compute the value of E.
- Place the (physical token for) the value of E in the physical location named x; any existing token of value to be replaced.

This is a description of how assignment is to be physically realized. It is a physical description of the process of evaluation. Of course, a complete description will spell out more, but presumably not what the actual machine is made of; one assumes that this would be part of the structural description of the underlying computer. The task of the structural description is only to describe the process of implementation on a class of physical machines where there is already some given assumptions about the latter's physical structure.

In the case of more complex expressions, one must stipulate how the constructs of the language are to be implemented. For example, to execute commands in sequence, we could add a physical stack that arranges them for processing in sequence. Of course, matters are seldom this straightforward. Constructs such as iteration and recursion require more sophisticated treatment.

4.1 Interpreters, Compilers and Virtual Machines

In general, implementations may be *direct* or *indirect*. In the former, an interpreter takes as input a program in the language and directly performs the operations of the language on a physical machine. In the indirect case, a compiler translates the programs of the source language into the programs of a target language. Indeed, actual compilation (Aho et al. 1992) may involve several passes (e.g., code generation, assembling, linking, and loading).

Furthermore, modern programming language implementations include layers of compilation. Part of this will almost certainly involve an interpretation on a virtual machine that runs as a program inside a host operating system. Such a machine is intended to be independent of any particular physical hardware and allows a program to execute in the same way on any platform. For example, the Java programming language is implemented using the Java virtual machine.

Perhaps the most conceptually significant virtual machine is the stack, environment, code, dump machine (SECD) intended as a target for functional programming language compilers. The machine was designed to evaluate Lambda calculus terms (Landin 1964). There are now many hardware implementations of the machine. This raises the following issue.

4.2 Abstract and Concrete Machines

What kind of thing is a virtual machine? At the top level, the SECD machine is an abstract machine, a mathematical object. However, its role during the implementation process may be twofold. Aside from its abstract role, it may have an existence as a piece of software running on an existing operating system. As a piece of software, it gets its functional definition from the semantic definition of the host programming language and its underlying abstract machine. Indeed, an implementation may

involve layers of virtual machines written in different programming languages with their associated semantics, so that each of the machines has an existence both as an abstract machine and as a software machine, and eventually as a physical device. However, in turn, via its abstract semantics, any software machine can serve as the functional specification of a more concrete device where, eventually, every machine in the sequence will be transformed into an actual physical device. Throughout the whole process, it is the abstract mathematical machines that determine function and the correctness of the various levels of implementation.

Generally, implementations include all the processes involved in getting new software or hardware working. The specific implementation of a programming language is supported by a background system that has to include a mass of subsidiary software, including the underlying operating system. We are not saying that all of this is fixed by the language implementor; much will already be fixed by the encompassing environment. The actual language implementation is constructed upon this base. But this complexity of the implementation process does not affect our central philosophical claims. While this may be quite hard to isolate, at some level, the structural description must provide a description of how the function of each component of the language is to be implemented on a physical device.

Finally, one might be tempted to claim that it is the physical implementation of the language that is the technical artifact. But this would be to deny the dual nature of technical artifacts: they must have a function. And in the case of the implementation of a language, this is given by the semantic definition of the language. Indeed, this confusion is the root of an even more pernicious perspective.

5 Idealization

Causal theories of function insist that actual physical capacities determine *function*. Cummins's theory of functional analysis is an influential example of such a *causal theory*. The underlying intuition is that, without the physical thing, and its actual properties, there can be no artifact: *no material object, no physical capacity, no technical function* (Cummins 1975).

In the case of programming languages, this amounts to the view that the physical implementation somehow defines the semantics. This is closely related to some comments found in the philosophy of computer science literature, i.e., to those views that seem to advocate a causal semantic theory. For example, Fetzer (1988, 1999) observes that programs have a different *semantic* significance to theorems. In particular, he asserts:

> ...programs are supposed to possess a semantic significance that theorems seem to lack. For the sequences of lines that compose a program are intended to stand for operations and procedures that can be performed by a machine, whereas the sequences of lines that constitute a proof do not. (Fetzer 1988)

This seems to suggest that the physical properties of the implementation somehow contribute to the meaning of programs written in the language. Colburn is more explicit when he writes that the simple assignment statement $A := 13 \times 74$ is

semantically ambiguous between something like the abstract account we have given and the physical one given as

> physical memory location A receives the value of physically computing 13 times 74. (Colburn 2000)

The phrase *physically computing* seems to imply that what the physical machine actually does is semantically significant, i.e., what it actually does determines or contributes to the meaning of assignment. But this has some queer consequences. In particular, it implies that, to fix what assignment means, we have to carry out a physical computation. This way of unpacking the physical significance of programs suggest that it is semantic in content. It also suggests that assignment is semantically ambiguous between its abstract and physical meanings. But, in what sense can the physical account contribute to the meaning of assignment?

If an actual physical machine is taken to fix or contribute to the meaning of the constructs of the language, then their meaning is dependent upon the contingencies of the physical device. But this entails that meaning of the simple assignment statement may well vary with the physical state of the device and, in particular, with contingencies that have little to with the semantics of the language, e.g. power cuts. Even the evaluation of arithmetic expressions is subject to physical interference: multiplication does not mean multiplication but rather what the physical machine actually does when *it multiplies*. Consequently, 13×74 might be 16. But this makes calculation an activity that is subject to causal interference. But this must be wrong. As Wittgenstein emphasizes, calculation is a mathematical activity, not an empirical one.

> There are no causal connections in a calculation, only the connections of the pattern. And it makes no difference to this that we work over the proof in order to accept it. That we are therefore tempted to say that it arose as the result of a psychological experiment. For the psychical course of events is not psychologically investigated when we calculate. (Wittgenstein 1939) 382.
>
> You aren't calculating if, when you get now this, now that result, and cannot find a mistake, you accept this and say: this simply shows that certain circumstances which are still unknown have an influence on the result. This might be expressed: if calculation reveals a causal connection to you, then you are not calculating.... What I am saying comes to this, that mathematics is normative. (Wittgenstein 1939) 424.

The problem of taking the empirical notion as the basis of a semantic account is made explicitly by Kripke in his analysis of Wittgenstein's rule following considerations.

> Actual machines can malfunction: through melting wires or slipping gears they may give the wrong answer. How is it determined when a malfunction occurs? By reference to the program of the machine, as intended by its designer, not simply by reference to the machine itself. Depending on the intent of the designer, any particular phenomenon may or may not count as a machine malfunction. A programmer with suitable intentions might even have intended to make use of the fact

> that wires melt or gears slip, so that a machine that is malfunctioning for me is behaving perfectly for him. Whether a machine ever malfunctions and, if so, when, is not a property of the machine itself as a physical object but is well defined only in terms of its program, stipulated by its designer. Given the program, once again, the physical object is superfluous for the purpose of determining what function is meant. (Kripke 1982)

The central issue is that such an account provides no notion of malfunction and correctness. Any implementation will be vacuously correct since it itself determines the function of the constructs of the language. There is no external demand; the physical device is all there is. The implementation determines its own function. But to provide an account of correctness, a functional specification must be normative (Turner 2011).

In the case of languages, we require an account of meaning that enables us to determine correctness. Indeed, this is a minimal requirement on any notion of meaning.

> The fact that the expression means something implies that there is a whole set of normative truths about my behavior with that expression; namely, that my use of it is correct in application to certain objects and not in application to others. .... The normativity of meaning turns out to be, in other words, simply a new name for the familiar fact that, regardless of whether one thinks of meaning in truth-theoretic or assertion-theoretic terms, meaningful expressions possess conditions of correct use. Kripke's insight was to realize that this observation may be converted into a condition of adequacy on theories of the determination of meaning: any proposed candidate for the property in virtue of which an expression has meaning, must be such as to ground the 'normativity' of meaning—it ought to be possible to read off from any alleged meaning-constituting property of a word, what is the correct use of that word. (Boghossian 1989)

It is very tempting to think that physical structure determines the meaning of the constructs of the language, and it is a common view among some working computer scientists. But the causal approach misplaces the semantic account of the language. Semantic accounts are not empirical claims about the behavior of a physical machine.

However, there is still a residual puzzle. Despite these considerations, when we talk about what the constructs of the language are supposed to do, we seemingly refer to their impact on a physical machine. This thought might be behind the quoted remarks from Fetzer (1988). So what sense, if any, can we make of this intuition? A way forward can be found in the writings of Wittgenstein.

> In kinematics we talk of a connecting rod—not meaning a rod made of brass or steel or what-not. We use the word 'connecting rod' in ordinary life, but in kinematics we use it in a quite different way, although we say roughly the same things about it as we say about the real rod; that it goes forward and back, rotates, etc. But then the real rod contracts and expands, we say. What are we to say of this rod? Does it contract and expand?—And so we say it can't. But the truth is that there is no question of it contracting or expanding. It is a picture of a connecting rod, a

> symbol used in this symbolism for a connecting rod. And in this symbolism there is nothing which corresponds to a contraction or expansion of the connecting rod. (Wittgenstein 1939)

In kinematics, one abstracts away from actual physical properties in much the same way that functional properties are separated from structural ones. The same theme appears in the writing of the physicist Duhem who argues for the need to abstract and produce an ideal version on which to carry out the reasoning processes necessary to conceptualize and employ any physical device.

> When a physicist does an experiment, two very distinct representations of the instrument on which he is working fill his mind: one is the image of the concrete instrument that he manipulates in reality; the other is a schematic model of the same instrument, constructed with the aid of symbols supplied by theories; and it is on this ideal and symbolic instrument that he does his reasoning, and it is to it that he applies the laws and formulas of physics. A manometer, for example, is on the one hand, a series of glass tubes, solidly connected to one another filled with a very heavy metallic liquid called mercury and on the other by the perfect fluid in mechanics, and having at each point a certain density and temperature defined by a certain equation of compressibility and expansion. (Duhem 1906) pp. 156–157.

The schematic model is an abstraction, a mathematical idealization of the physical object. Moreover, once this idealization is complete, there is no semantic role left for the physical device. As Kripke puts it: *Given the program, once again, the physical object is superfluous for the purpose of determining what function is meant.* Although we intend the language to be implemented on a physical machine, the semantic account refers to an abstraction of that machine.

So the picture seems clear: the physical implementation has no semantic import. Instead, the semantic account of the language has normative governance over the implementation. However, this raises the following question.

6 Correctness

When is an implementation of a language correct relative to its semantic account? Certainly, an implementation that turns every program into the identity state to state transformation will rarely be so. A correct implementation must meet the desiderata of the functional description that is constituted by the grammatical and semantic accounts of the language. But what exactly does this amount to? We shall address this question by reference to assignment and the abstract machine. The correctness of the implementation of the language is then built upon this: correctness is determined by the operational rules which provide the norms of correctness for the physical implementation of the constructs. But the central philosophical issue already arises with the simple assignment statement.

The most famous account of correctness is given by the simple mapping account (SMA) of Putnam (1967). A clear exposition is provided in Piccinini (2008). This demands that a physical system is correct relative to an abstract one if the former can

be mapped onto the abstract one in such a way that the state transitions are duplicated in the physical version. More exactly, this is expressed as follows.

> A physical system P **Implements** an abstract one A just in case there is a mapping I from the states of P to the states of A such that: for any abstract state transition $s_1 \curvearrowright s_2$ **if** P is in the physical state S_1 where $I(S_1) = s_1$, then it goes into the physical state S_2 where $I(S_2) = s_2$.

where the clause *"if the system is in the physical state S_1 where $I(S_1) = s_1$, then it.."* is to be interpreted as the material conditional.

For simplicity, assume that there are only two locations l and r which may contain two possible values 0 and 1. This generates four possible states (0, 0), (0, 1), (1, 1), and (1, 0). The following table then represents the complete input/output behavior of our abstract machine induced by updating.

Start	$r := 1$	$r := 0$	$l := 1$	$l := 0$
(0, 0)	(0, 1)	(0, 0)	(1, 0)	(0, 0)
(0, 1)	(0, 1)	(0, 0)	(1, 1)	(0, 1)
(1, 0)	(1, 1)	(1, 0)	(1, 0)	(0, 0)
(1, 1)	(1, 1)	(1, 0)	(1, 1)	(0, 1)

In principle, to ensure that we have an implementation of assignment that is in accord with the abstract device, we would need to check that there is an exact match between the states of the system and the physical and abstract update tables. For example, it is consistent with the constraints imposed by SMA, that a physical implementation could take the states of the machine to be stones that change color with temperature. I observe the stones and construct the following table to record the color changes with temperatures T_1, T_2, T_3, and T_4, taken at hourly intervals during a sunny day.

Stones	T_1	T_2	T_3	T_4
red	*blue*	*red*	*green*	*red*
blue	*blue*	*red*	*yellow*	*blue*
green	*yellow*	*green*	*green*	*red*
yellow	*yellow*	*green*	*yellow*	*blue*

At T_1 degrees, the stones change color as indicated by the column labeled T_1. Similar goes for the other columns. According to the SMA, the system of colored stones implements the abstract machine, and the sun shining on the stones implements assignment. Of course, it is not essential that the T_i are temperature changes; they could be any physical process. According to SMA, with its material notion of implication, they have only to be in accord with the table.

The upshot of this is that SMA is a very weak requirement; it only demands that the two tables be in accord/extensional agreement with each other. It is this notion of implementation that leads some authors to conclude that almost every physical system implements almost every specification (for more discussion, see Piccinini 2008 and Putnam 1988).

What has gone wrong? First, observe that in order to set up the correspondence between the two tables, we have to compute the whole state table for our abstract

machine.[8] Consequently, the relationship is created post hoc. This point is made by Copeland (1996) who observes that the mappings of SMA are *illegitimate* because they are constructed after the computation has been performed. Others offer solutions in terms of a lack of causal input in SMA (Chalmers 1996). Chalmers argues that this can be rectified by changing the material conditional for a more causally substantive one (e.g., a counterfactual conditional). The other solution insists that the account must be enriched by adding more semantic content over and above that given by the semantic definition of the language (Sprevak 2010). This semantic representation has something to do with the specification of the program rather than its semantic account. There is something to be said for each of these views, but we shall take a different tack that is dictated by the view of programming languages as technical artifacts.[9]

7 Intention

According to the dualism of technical artifacts, an object cannot be a technical artifact by virtue of just its physical properties. To be such, an object must have a function, and somehow the latter must involve human intentions. But whose intentions are relevant?

> Two specific contexts of intentional human action are of particular interest, namely the engineering design context and the user context. In the engineering design context the focus is on inventing/constructing a physical structure that will realise a given function (or that satisfies a list of functional requirements or of design specifications). (Kroes 2010)

We shall follow the *maker* or engineering perspective (Kroes 2010, 2012).[10]

If I observe a man arranging stones in a way that is consistent with the extensional arrangement determined by some abstract machine, I may not legitimately infer that he is building a device according to the abstract machine viewed as a functional specification. He might just be arranging them for esthetic reasons. In general, a physical device may be in accord with an abstract machine without being governed by it. How might we tell the difference? How can we tell if she is under the authority of the abstract machine or building a work of art? Presumably, we can ask: "what are your intentions?", "why did you do that?" For the former, the answer must refer to the abstract specification. However, the essence of this *maker* use of function is not

[8] One might note in passing that this seems to make computers redundant.

[9] What makes programming languages special as constructed language? Is it possible to see a language like Esperanto as a technical artifact? Does the latter have a physical realization?

[10] In the case of programming languages, the user is the programmer and the maker is the implementor. The normative use of the definition of the language in the user case is complex and the issue of correctness brings in the specification of individual programs. We shall not discuss this further here but see (Turner 2011).

in the details or complexity of the process of construction or implementation. Indeed, the whole process may be very rudimentary: in the limit, the process might amount to no more than taking an existing physical object to be an X. For example, suppose that I discover a large log in a field and decide that I can use it as a doorstop. I have circumvented the construction stage, but still the notion of doorstop determines what I am looking for and provides the criterion of correctness. If my log does not work as a doorstop, it is the notion of *doorstop* that tells me this and my experimentation with the log that demonstrates that it does not.

The essence of this intentional notion of function consists in taking it as a yardstick of correctness (Turner 2010, 2011). In the case of our abstract machine, it is to take it as a functional specification of a physical one. The function is a rule that must be followed, and the relation between it and the physical object is manifest in using the rule as a canon of correctness for the physical object. If I ask, *does it work?*, I must be able to justify my reasons relative to the abstract device.

We seem to have embraced some version of the intentional approach to function (McLaughlin 2001; Searle 1995). It is generally agreed that there must be structural agreement between the function and its physical structure. But if all that we require is that the agent take the object as an X, there is no necessary imposition of physical structure. One might attempt to enforce this by postulating complex mental states. In the present context, it would amount to the claim that the meaning of a programming language exists as something in the mental structure or history of the agent. But, in the case of programming languages, this is problematic.

> Given ... that everything in my mental history is compatible both with the conclusion that I meant plus and with the conclusion that I meant quus,[11] it is clear that the sceptical challenge is not really an epistemological one. It purports to show that nothing in my mental history of past behavior—not even what an omniscient God would know—could establish whether I meant plus or quus. But then it appears to follow that there was no fact about me that constituted my having meant plus rather than quus. (Kripke 1982).

Kripke argues Wittgenstein that the meaning of plus cannot be fixed by the past or present mental states of any agent. It is not a description of such mental states. Rather, the meaning of plus is fixed by the agreed rules of addition. The abstract structure is in the mathematical object itself.

Programming languages have a mathematical semantics and an implementation. It is in the mathematical object that structure is located. And the semantics can stand alone as a mathematical object, i.e., to be investigated as such. But when used as the specification, it is given governance over the implementation. The intentional aspect then comes into play: the agent gives the abstract semantics normative force over the physical implementation: they must be in extensional accord. Abstract structure and physical structure are linked by the intention to take the former as having normative

[11] Quus: a function that is only identical to plus up to a certain pair of numbers.

governance over the latter. It is at this point that the piece of abstract mathematics takes on its functional guise. The agent's intention relates the abstract and the concrete.

To unpack this perspective more explicitly, it will be helpful to compare the notion of functional specification to the scientific concept of *theory*. Wandering through a field, I discover a collection of colored stones. I notice that they change color according to my table of temperature changes. As an observer, I am intrigued by this thing: what is it? Subsequently, I attempt to construct an account of what it does. Essentially, I try and construct a *theory* of the device. After some reflection, I postulate that it is a simple store machine. But this is not a normative activity; it is a descriptive one that involves a theoretical characterization of the physical thing. To construct such a theory or *schematic model* (Duhem 1906), one must choose the objects, properties, and relations that make up the theory. These are the features that will be subject to testing and verification. The abstraction is a mathematical idealization where not only are some physical aspects ignored, but they are idealized.

However, despite these analogies and similarities between such theory construction and normative function, there is a fundamental difference. In the theory case, any lack of accord is laid firmly on the shoulders of the theory. When the theory diverges from the physical thing, it is the theory that is at fault. For instance, when an abstract machine is taken as a theory of a physical one, it is the physical device that is given to us. When there is a mismatch, it is the theory, the abstract machine, that must be sacrificed or modified. In contrast, in the case of specification, any lack of accord is blamed on the physical object. Matters are reversed. Here, the function takes charge: when there is disagreement, we blame the physical device. It is the physical device that is correct or not. It is the physical object that is to be changed.

Of course, I may change my intention towards a given physical object or device. I may, on finding my device and formulating a theory of it as a store machine, decide to use it as a computer store. At this point, it changes from being a physical thing with an attached theory to an artifact with a function. Provided it is in accord with any abstract notion, the intentional act of using the latter as a functional specification of an object creates a technical artifact. My intention has changed. What I previously dubbed a defeasible theory is now a normative function. These different intentions illustrate how impoverished the extensional/in accord account of correctness is. Such a simplistic approach, ignores any form of intentional attitude between the abstract and the physical one.

8 Computational Artifacts

The things that computer scientist build,[12] programs, data types, type inference systems, etc. seem to have an abstract guise that enables us to reflect and reason about them independently of any physical manifestation. For example, the data type of lists consist of the set or type of lists together with operations that enable the formation of

[12] What I would like to refer to as *computational artifacts.*

lists and their deconstruction. These are governed by several axioms that relate them. For example,

$$tail(append(a, l)) = l$$
$$head(append(a, l)) = a.$$

One can reason about these in a mathematical way that is independent of any physical representation. Much the same applies to programs, types, compilers, virtual machines, interpreters, etc. All of these notions seem to have an abstract guise that is independent of their physical realization or implementation.

On the other hand, these objects must have a physical implementation that enables them to be used as artifacts in the physical world. For instance, a program that has no physical realization is of little use as a practical device for performing humanly intractable computations. Computer science is not just an abstract discipline that is independent of the physical world: it produces technical artifacts.

These brief concluding remarks need to be supported by a more detailed analysis of the other artifacts of computer science. But if programming languages and programs are typical of the technical artifacts of computer science, then it seems to be a strange blend of pure mathematics and technology (Strachey 2000), and consequently, the philosophy of computer science, or at least the central part of it, should be a fusion of the philosophy of mathematics and the philosophy of technology.

References

Abramsky, S., Jagadeesan, R., Malacaria, P. (1994). In M. Hagiya, & J.C. Mitchell (Eds.), *Full abstraction for PCF in Theoretical aspects of computer software*. London: Springer.

Abramsky, S., & McCusker, G. (1995). Games and full abstraction for the lazy lambda-calculus. In D. Kozen (Ed.), *Proceedings of the 10th annual symposium on logic in computer science*. IEEE Computer Society Press.

Aho, A.V., Lam, S., Sethi, R., Ullman, J.D. (1992). *Compilers: principles, techniques, and tools, 2007*. Boston: Pearson.

Barendregt, H.P. (1992). Lambda calculi with types. In S. Abramsky, D.M. Gabbay, T.S.E. Maibaum (Eds.), *Handbook of logic in computer science* (Vol. III). Oxford: Oxford Univeristy Press.

Boghossian, P. (1989). The rule-following considerations. *Mind*, *98*(392), 507–549.

Börger, E., & Schulte, W. (2007). A programmer friendly modular definition of the semantics of java. *Lecture Notes in Computer Science*, *1523*, 353–404. ISBN: 3-540-66158-1.

Cardelli, L., & Abadi, M. (1996). A theory of objects. *Mongraphs in computer science*. New York: Springer. IBSN: 0387947752.

Colburn, T. (2000). *Philosophy and computer science*. New York: London.

Colburn, T., & Shute, G. (2007). Abstraction in computer science. *Minds and Machines*, *17*(2), 169–184.

Cummins, R. (1975). Functional analysis. *Journal of Philosophy*, *72*, 741–765.

Chalmers, D.J. (1996). Does a rock implement every finite-state automaton. *Synthese*, *108*, 309–333.

Copeland, B.J. (1996). What is computation?. *Synthese*, *108*(3), 335–359.

Duhem, P. (1906). *La Théorie physique*. Paris: Chevalier & Riviere, Editeurs. http://www.ac-nancy-metz.fr/enseign/philo/textesph/Duhem_theorie_physique.pdf.

Fernandez, M. (2004). *Programming languages and operational semantics: An introduction*. London: King's College Publications.

Fetzer, J.H. (1988). Program verification: the very idea. *Communinications in ACM*, *31*(9), 1048–1063.

Fetzer, J.H. (1999). The role of models in computer science. *Monist*, *82*(1), 20–36.

Franssen, M., Lokhorst, G., Poel, I. (2009). *Philosophy of technology*. Stanford Encyclopedia of Philosophy. http://plato.stanford.edu/entries/technology.

Girard, J.-Y. (1971). Une Extension de l'Interpretation de Gödel à l'Analyse, et son Application à l'Élimination des Coupures dans l'Analyse et la Théorie des Types. In *Proceedings of the 2nd scandinavian logic symposium* (pp. 63–92), Amsterdam.
Gödel, K. (1934). *Some basic theorems on the foundations of mathematics and their implications*, (pp. 304–323).
Gordon, M.J.C. (1979). *The denotational description of programming languages*. Berlin: Springer.
Gordon, M.J.C. (1986). *Hardware verification using higher-order logic*.
Gunter, C.A. (1992). *Semantics of programming languages: structures and techniques*. Cambridge, MIT.
Hanson, M., & Rischel, H. (1999). *Introduction to programming using SML*. Boston, Pearson.
Hilbert, D. (1899). *The foundations of geometry*, 2nd edn. Chicago: Open Court.
Houkes, W., & Vermaas, P.E. (2010). Technical functions. *On the use and design of artefacts. Philosophy of engineering and technology* (Vol. 1). Dordrecht: Springer.
Irmak, N. (2012). Software is an abstract artifact. http://miami.academia.edu/NurbayIrmak/Papers/1314637.
Jones, C.B. (1990). *Systematic software development using VDM*, 2nd edn. Upper Saddle River: Prentice Hall International.
Kroes, P. (2010). Engineering and the dual nature of technical artefacts. *Cambridge Journal of Economics*, *34*(1), 51–62. doi:10.1093/cje/bep019.
Kroes, P. (2012). *Technical artefacts: creations of mind and matter: a philosophy of engineering design*. Dordrecht, Springer.
Kripke, S. (1982). *Wittgenstein on rules and privaye language*. Cambridge: Harvard University Press.
Landin, P.J. (1964). The mechanical evaluation of expressions. *Computer Journal*, *6*(4), 308–320.
McLaughlin, P. (2001). What functions explain. *Functional explanation and self-reproducing systems*. Cambridge: Cambridge University Press.
Meijers, A.W.M. (2001). The relational ontology of technical artefacts. In P.A. Kroes, & M.A.W.M. Amsterdam (Eds.), *The empirical turn in the philosophy of technology research in philosophy and technology* (Vol. 20), (series editor Carl Mitcham). Amsterdam: JAI/Elsevier.
Milne, R., & Strachey, C. (1977). *A theory of programming language semantics*. New York: Halsted.
Milner, R. (1993). The polyadic $\pi - Calculus$: A tutorial. In F.L. Hamer, W. Brauer, H. Schwichtenberg (Eds.), *Logic and algebra of specification*. Berlin: Springer.
Mitchell, J.C. (1996). *Foundations for Programming languages*. Cambridge: MIT.
Moor, J.H. (1978). Three myths of computer Science. *British Journal for the Philosophy of Science*, *29*(3), 213–222.
Mulmuley, K. (1987). *Full abstraction and semantic equivalence*. Cambridge: MIT.
Piccinini, G. (2010). Computation in physical systems, Stanford encyclopedia of philosophy. http://plato.stanford.edu/entries/computation-physicalsystems.
Piccinini, G. (2008). Computation without representation. *Philosophical Studies*, *137*, 205–241.
Pierce, B.C. (2002). *Types and programming languages*. Cambridge: MIT. ISBN 0-262-16209-1.
Plotkin, G.D. (1981). *A structural approach to operational semantics.* Tech. Rep.19. Denmark, Aarhus: Computer Science Department, Aarhus University.
Putnam, H. (1988). *Representation and reality*. Cambridge: MIT.
Putnam, H. (1967). Psychological predicates In W.H. Capitan, & D.D. Merrill (Eds.), *Art, mind, and religion* (pp. 37–48). Pittsburgh: University of Pittsburgh Press. Reprinted in Putnam 1975a as "The Nature of Mental States," pp. 150–161.
Putnam, H. (1975). *Philosophical papers: Volume 2. Mind, language and reality*. Cambridge: Cambridge University Press.
Reynolds, J.C. (1994). *An introduction to polymorphic lambda calculus in logical foundations of functional programming* (pp. 77-86). England: Addison-Wesley.
Searle, J.R. (1995). *The construction of social reality*. London: Penguin.
Schmidt, D.A. (1988). *Denotational semantics: a methodology for language development*. Boston: Allyn & Bacon, 1986. Reprint, 1988.
Spivey, M. (1988). Understanding Z: a specification language and its formal semantics. In *Cambridge tracts in theoretical computer science* (Vol. 3). Cambridge: Cambridge University Press. ISBN 978-0-521-05414-0.
Stoughton, A. (1988). *Fully abstract models of programming languages*. London: Pitman/Wiley.
Sprevak, M. (2010). Computation, individuation, and the representation condition. *Studies in History and Philosophy of Science Part A*, *41*(3), 260–270. Computation and cognitive science.

Stoy, J. (1977). *Denotational semantics: the Scott-Strachey approach to programming language semantics*. Cambridge: MIT.

Strachey, C. (2000). Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, *13*, 11–49. Manufactured in The Netherlands: Kluwer Academic.

Thompson, S. (2011). *Haskell: the craft of functional programming*. England: Addison-Wesley. ISBN: 0-201-34275-8.

Thompson, S. (1995). *Miranda: the craft of functional programming*. England: Addison-Wesley. ISBN: 0201422794.

Thomasson, A.L. (2003). Realism and human kinds. *Philosophy and Phenomenological Research*, *67*(3), 580–609. CrossRefWeb of Science.

Thomasson, A.L. (2007). Artifacts and human concepts. In S. Laurence, & E. Margolis (Eds.), *Creations of the mind: essays on artifacts and their representations*. Oxford: Oxford University Press.

Tennent, R.D. (1991). *Semantics of programming languages*. Upper Saddle River: Prentice Hall.

Turner, R. (2007). Understanding programming language. *Minds and Machines*, *17*(2), 129–133.

Turner, R. (2010). Specification. *Minds and Machines*, *21*(2), 135–152.

Turner, R. (2011). *Computable models*. London: Springer.

Vermaas, P.E., & Houkes, W. (2003). Ascribing functions to technical artefacts: a challenge to etiological accounts of function. *British Journal of the Philosophy of Science*, *54*, 261–289.

Winskel, G. (1993). *The formal semantics of programming languages: an introduction*. Cambridge: MIT.

Wittgenstein, L. (1939). *Wittgenstein's lectures on the foundations of mathematics*. Cambridge: University of Chicago Press.

White, G. (2004). The philosophy of computer languages. In L. Floridi (Ed.), *The Blackwell guide to the philosophy of computing and information* (pp. 318–326). Malden: Blackwell.