philosophies

MDPI

Article

Conceptualising Programming Language Semantics

Troy Kaighin Astarte

School of Mathematics and Computer Science, Swansea University, Swansea SA2 8PP, UK; t.k.astarte@swansea.ac.uk

Abstract

The semantics of programming languages tend to be discussed with high levels of formality; much of the previous research—both philosophical and historical—has investigated them from this perspective. In this paper, I draw on the philosophical and cognitive theories of metaphor and use the early work of Adriaan van Wijngaarden as a historical case study to explore the *conceptual and discursive surroundings* of semantics. I investigate the relationships between the texts of semantics, the abstract entities they denote, and the metaphors, analogies, and illustrative language used to accompany or explain the same. This serves to further understanding of the historical developments of work in this area, the nature of programming languages and their semantics, and the importance of the communicative methods used in dissemination and education of computer science.

Keywords: programming languages; semantics; computer science; metaphor; van Wijngaarden; machines; machine language; programming; IFIP; ALGOL

Academic Editors: Raymond Turner and Henri Stephanou

Received: 1 May 2025
Revised: 24 July 2025
Accepted: 31 July 2025
Published: 9 August 2025

Citation: Astarte, T.K. Conceptualising Programming Language Semantics. *Philosophies* **2025**, *10*, 90. https://doi.org/10.3390/philosophies10040090

1. Introduction

> "If, however, the user does not trust his intuition or does not understand what the short description on the lid implies in a particular case, he can open the machine to inspect the precise working. To his surprise, he finds there are actually two machines inside, named *P*1 and *M*1. The working of the machines is explained in much more detail on the lids of the machines. The machine *P*1 is a so-called preprocessor, which chews the offered text and produces another text in a more basic language which is evaluated by the processor, i.e. machine *M*1."

What is the entity being described here? This passage ([1], p. 18) describes the core of the approach to programming language semantics presented by Adriaan van Wijngaarden in two papers in the first half of the 1960s [1,2]. This may not be immediately apparent, since the passage is redolent in non-literal language: an extended machine metaphor—itself somewhat anthropomorphised—chewing a key object within its jaws: text. The presentation of this image of programming languages provoked consternation when Van Wijngaarden presented it to an audience of programmers, language theorists, and mathematicians. Why did Van Wijngaarden choose this way of demonstrating his work? Why was the reaction so confused? And what can be learnt by studying the language used?

The present paper will explore this early effort to understand the semantics by examining the use of conceptuality: metaphor, analogy, and example—what I call "illustrative devices", as a reflection of "rhetorical devices", but with an emphasis on their explanatory nature. My previous work has explored the history of programming language semantics, and concurrency in programming, from a history of science perspective [3,4]. As proposed in the second cited paper, this promises to be a rich vein of study and one which I intend to

explore in future research projects. Drawing on considerations of metaphor and conceptuality in mathematics and science, I will take Van Wijngaarden's work as a case study and argue that by exploring the illustrative devices used by computer science, we can better understand the nature of the field, and its history. In this paper I will use "semantics" to mean that of programming languages except where disambiguation is required.

The discourse around computers is packed with non-literal language, at the level of hardware (BUS), software (SEMAPHORE), interface (WINDOW), application (FILE) and socio-technical systems (FOLLOW).[1] One important and relevant illustrative device is LANGUAGE itself, examined carefully by Nofre, Priestley, and Alberts [6].[2] The authors track and explain the rise of LANGUAGE to describe the notations used in formulating the instructions given to computers. The metaphor emerged towards the end of the 1950s as the expressive power of the computer was being realised, and as comparisons with "giant brains" [9] fell out of fashion. As LANGUAGE was evoked to conceptualise programming, notions were brought along not just from linguistics—like syntax and semantics—but from formal logic too [10]. This not only provided programmers and computer scientists with a new set of tools for discussing their objects of study[3] but shaped how these objects were formed and handled epistemically [3]. This carefully explored example shows the value of considering illustrative devices to aid in understanding the ontology of computational objects and critically also the history of computing, as the authors showed how the use of the LANGUAGE metaphor was contingent on the intellectual context in which the historical actors operated.

Most philosophy written about programming language semantics is about the ontology of programs and concerned almost exclusively with the technical properties of the entities.[4] For example, White wrote a brief overview of how semantics might be used and some challenging ontological aspects, but the material is technical and overly focused on one specific approach to semantics [13]. Rapaport [14,15] argued that semantics describes the way that given abstractions can be interpreted to form an implementation. In his textbook *On the Foundations of Computing*, Primiero mentioned semantics only in the context of its role in the program correctness effort [16]. Probably the most prolific writer on the philosophy of programming language semantics is Turner. In [17], he laid out a series of motivating questions, many of which still remain open, and of which many are considered in the present special issue:

> "Knowing a language involves more than knowing its syntax; knowing a language involves knowing what the constructs of the language do, and this is semantic knowledge. In general, without such knowledge it would not be possible to construct or grasp such programs. But how is this semantic content given to us; how is it expressed? What constraints or principles must any adequate semantic account satisfy? What are the theoretical and practical roles of semantic theory? Is natural language an adequate medium for the expression of semantic accounts? These are some of the questions that any conceptual investigation of programming language semantics must address. [...] What are the relationships between [the] various approaches? Do they complement each other or are they competitors? Is one taken to be the most fundamental?"

I contend that by investigating the illustrative devices used in semantics we will be able to provide a different set of answers to these questions to those already being posed.

The literature on philosophy of mathematics is likewise very concerned with ontology, as noted by Pérez-Escobar, who observed that discussion of mathematicians' practice is regularly neglected in favour of testing broader philosophical questions about ontology and metaphysics [18]. One notable exception is a study by Sha of the a-linguistic writing and performance practices of differential geometers [19], a deep and unconventional analysis of

the way in which these practices allow the exteriorisation of mathematical thought, which may be communicated with others or, once modified, reinteriorised. By considering the mathematical practice of performance—meaning a combination of acts of speech, writing, and bodily movements—a different ontology of mathematics is formulated. A similar analysis, with a more historio-sociological focus, is that of blackboard practice by Barany and MacKenzie [20]. They explored the specifics of chalk-on-board and its accompanying gestures, discursive shapes, and behaviours; interestingly, while the authors hinted at the metaphorical power of the chalk, they did not go on to analyse it in much depth, being concerned more with the graphic, spatial, and spoken aspects of the practice.

Even Maddy [21], commended by Pérez-Escobar for paying attention to mathematical practice, discussed it in the context of advocating for a realist ontology. Maddy highlighted the tension between the behaviour of mathematicians suggesting that their objects of study really exist and the formalist position of meaningless symbol manipulation as a theoretical fallback when paradoxes emerge. Shapiro [22] calls this "Benacerraf's dilemma" (after [23]), arguing that an antirealist philosophy provides a strong epistemology but makes it hard to keep a continuity between semantics of mathematics and everyday language, which seems necessary due to their combined usage in the sciences. Shapiro's own solution is to favour structuralism, in which mathematics becomes the study of patterns (on structures) and its objects are places within those structures, given properties only by their relationship to other things in the structures.

Computer science is even further within the grip of formalism than mathematics, as argued by MacKenzie [24]. Those interested in using mathematical/logical tools for verifying program correctness (typically called "formal methods")[5] tend to believe more in the value of symbol manipulation rules for reasoning about programs than the social processes of mathematical proof. Programming language semantics fits within this framework as a method for establishing the intended meanings of programs; historically, the semantics effort predates most formal methods work but is closely tied in terms of goals, methods, and involved people [3].

Rather than ontology, then, in this paper I will consider programming language semantics from the perspective of its illustrative devices. My discussion centres on the Van Wijngaarden case study to inform the investigation and to allow its history to be illuminated by the philosophical discussion. I begin by exploring some key frameworks for understanding non-literal language, sketch some of the historical background on programming languages before heading into the case study, and finish by looking at a number of reasons for further studying the illustrative devices of computing.

One final note: texts will appear regularly in this paper. We should be straight about the difference between (a) the text of the formal object which constitutes the semantics; (b) the accompanying discussion which serves to explain and illustrate (a); (c) the text in the metaphor A PROGRAM IS A TEXT used by Van Wijngaarden to illustrate his work. In this paper, I intend to focus my analysis on (b), although analyses of the others may occur from time to time, or in future work.

2. Metaphor

One major work studying the effects of metaphors is the *theory of embodied cognition*, the work of Lakoff and Johnson [27,28] (helpfully summarised by [29]). The theory states that human cognition and understanding is *embodied* and *metaphorical*. Metaphors shape the way we form conceptual links between ideas and create understanding, particularly for abstract concepts which use metaphors to transfer structures and properties from simpler source to more complex target domains (for example, LOVE IS BURNING).The embodiment comes from the ultimate base-level concepts, which are formed from bodily

experiences, such as HUMAN INTIMACY IS WARMTH (arguably from the bodily proximity of a newborn to a parent).Alongside the first joint book, further work by Lakoff, Johnson, and additional authors explored the use of metaphors in mathematics [30], philosophy [31], and science [32]. Gibbs [33] collects the perspectives of many authors applying the theory in various areas.

Their ontological position is called "embodied realism", as a way to balance objectivism and relativism, a pragmatic attitude where truth is judged to exist when a model works sufficiently in a given (social) situation, stands up to scrutiny from different perspectives, and its tenets are formed from the convergence of diverse methods. This informs their "empirically-responsible philosophy" ([34], p. 3), the principle that philosophy should (as much as possible) align with the results of empirical science.

In more abstract domains, however, like that of semantics, truth becomes more ambiguous and comes to rely more on internal consistency than anything empirical; here, the theory tells us less about how to use metaphors to aid in understanding. We must take care, since abstraction is a key principle in computing [11,35] and especially central to semantics ([36], Ch. 1). Since levels of abstraction are also a way to understand the ontology of computation itself [16], we must ensure we can understand how abstraction is dealt with as effectively as possible. There is an interesting parallel here to the mathematics of Ancient Greece as presented by Netz, where the necessity of a proof was built from the necessity of its opening statements, properties taken for granted by the writer and audience, and maintained only through the combination of these in appropriate ways [37]. This particular proof practice worked due to its social context: belief in the value of oral persuasion and recognised expertise.

Lakoff and Johnson's framework, however, centres embodied interpretation above all other potential influences on metaphor choice and usage, such as cultural contingency. Alongside other criticisms (some of which are summarised in [29]), Friedman [38], citing Lassègue [39], notes that the approach "flattens" notions of concept by reducing everything to embodied and sensorimotor roots, which ignores the socio-cultural origins of metaphors in mathematics.[6] Embodied cognition focuses heavily on the universality of embodied experiences, yet as more abstract metaphors tend to be built from other metaphors, there is far more opportunity for cultural contingency. Further issue is taken with the embodied basis by Sha, who notes that certain examples of their supposedly basic and universal concepts, such as "chair" and "blond", are in fact heavily culturally contingent [19]. Sha also challenges the idea that the human body should be the model for the understanding of every concept. The absence of any other medium for interpreting concepts, such as those present in mathematics, could lead to a narrow and overly anthropocentric ontology; while I promised to limit ontology, it is worth pausing a moment to note that the inclusion of computers as non-human agents suggests we might consider whether other frames of interpretation might be necessary.

In contrast to Lakoff and Johnson, Friedman outlines the position of Blumenberg. Here, the central notion is "concept", which may be literal but is usually non-literal and thereby closer to my "illustrative device". For Blumenberg, the development of concepts is characterised by an exploration of an unknown space, a *Vorfeld* (forefield) of nonconceptuality. This process is immediately partial and may take quite some time (and frequent repetition). The role of metaphor here is to provide the linguistic and cognitive tools to explore the *Vorfeld*.

Returning to philosophy, the theory of embodied cognition is an important challenge to the idea that logic is a priori, disembodied, and literal [34]. However, we must recognise that much of the work on programming language semantics (if not all) is in the formalist tradition, with its notions of set-models and meaningless symbols given meaning only

via relation with these models. An alternative, which may be of use when we think about illustrative devices, is to consider simulation semantics, originating in the work of Barsalou and colleagues in cognitive science (e.g., [40,41]). Their framework for conceptual understanding of words, supported by some empirical study, combines a linguistic system which recognises the form and linguistic relations of words, and a situated simulation system which in which the mind recreates images of experiences related to the concept. The situation of the simulation is relevant, with the same word producing different simulations depending on situation; the situation may be in the moment, in previous experiences, or in the linguistic context of the concept to be understood. Simulation semantics, as explained by Johnson, is the way in which meaning is given to entities by mental simulation of engagement with the affordances of those entities (and further empirical study suggests that limited neuronal activity in this situation mirrors that of taking the affordance) [32]. With formal objects like programming languages, one simulates these affordances by imagining oneself writing a statement; calculating an output value; or determining how many times a loop runs in a particular case. The choice of the words used to evoke these concepts, consequently, matters, as this influences the affordances present in the situated simulation.

This can be seen as a case of extended cognition [42], where aspects of the cognitive processes of thought, understanding, and meaning are externalised from the brain and placed in outside entities (or, more accurately, in the human–entity nexus). This could be in dialogue with another human, written words on paper, chalk on a board, or even virtual assistants.[7] Why could these external entities not include programming language semantics? Understanding cognitive practice must include understanding its metaphors, Friedman argues [38], taking the example of oceanic metaphors in mathematics as a way to frame encounters with the unknown.

Knowability is a key aspect of the philosophical study of metaphor presented by Ricoeur [44]: while a metaphor's interpretation is questionable, as in the *Vorveld* of Blumenberg, it is considered "living"; once its meaning is understood without ambiguity, the metaphor is dead. Writing about this work, Vincent noted "the foundation of hermeneutics is the discursive operation of all language, to which metaphor is the indispensable key." ([45], p. 413). Hermeneutics is the study of interpretation [46], and programming language semantics often attempts to resolve many questions of interpretation, albeit in a somewhat constrained manner. One kind of operational semantics, started by McCarthy [47], is known as "abstract interpreters" ([36], p. 38), and more generally, many writers (including Van Wijngaarden ([1], p. 17)) wrote of "interpreting" programs using either a written semantics or an intuitive understanding of the language.

Therefore, studying the metaphors used to accompany semantics could aid in our understanding thereof, as they shed light on the interpretative positions of the semantics writer. While Ricoeur wrote only about the use of metaphor in a purely natural linguistic context, interesting parallels can yet be drawn to programming language semantics. He argued that analysis cannot be purely based on the pair of denotation and connotation (my words); reference must be included. He also argued that a model can be judged on its authenticity—how effectively it redescribes reality. These questions will recur in the study of programming languages, both because programming languages are the tools used to create models of reality (i.e., programs) and because semantic descriptions of programming languages are themselves models of an implementation (and perhaps also vice versa).

Another direction which we might take from philosophy of language is to consider the use of rhetoric and its associated devices, such as the four "Master Tropes" of Burke [48]: alongside metaphor and metonymy, synecdoche and irony are also present. We have discussed metaphor; Burke's definition is one of *perspective*, a reframing of one concept in

terms of another in order to highlight some particular quality of the concept in question. Irony, or dialectic, presents an interesting direction: by putting the historian in the position of Burke's privileged observer, we are afforded the ability to grasp at the true nature of something—in this case, the semantics of programming languages—by examining dialectic formed by the interaction of the perspectives of each historical actor. That, however, would require the historian to have a comprehensive overview of those perspectives and a firm confidence in their interpretive analysis; at this stage in my research, I have neither. Synecdoche, per Burke's definition, comes closest to my interests here: the representation of a concept by some related aspect of it; specifically, metonymy is a *reduction*, a "convey[ing of] some incorporeal or intangible state in terms of the corporeal or tangible" ([48], p. 424). If we allow abstract concepts to be these incorporeal or intangible states, then metonymy could be at play when the semantics of programming languages is described with some other device. Metonymy is a key component of another rhetorical analysis: Netz's study of the development of deduction in Ancient Greek mathematics [37]. He argues that the lettered diagram is a metonym of the mathematical argument, serving to represent and illustrate without capturing the entirety of the proof. Netz shows the value of close reading to the "cognitive historian" (his term) and develops a convincing case to explain how the persuasiveness and generality of Greek proofs emerged from their social structures and oral traditions. This suggests that a reading of computer science texts as rhetorical works, replete with rhetorical devices, could be a worthwhile direction of study.

However, within the current paper, I intend to keep my focus on illustrative devices. I justify this by arguing that computer science texts are replete with complex illustrations containing analogies, scenarios, and examples. Consider the dining philosophers scenario [49], as analysed in [4]. Can this really fit within the framework of embodied metaphors? Yet, all the key aspects of representation (processes as philosophers), embodiment (resource utilisation as eating), and culture (spaghetti) are clearly present. Surely, this is just as deserving of study as the idea that a programming notation is a language. In the remainder of this paper, then, I shall draw from the ideas presented in the various analyses of metaphor mentioned in this section and recognise and acknowledge the presence of further literature or rhetorical devices where they are emergent. I intend not limit myself to (for example) the purely embodied basis of Lakoff and Johnson; and rather to take a broad view of illustrative devices in considering how we can understand the nature of semantics and its history—starting by situating it within the story of programming languages themselves.

3. Machines, Notations, and Languages

The history of programming languages really requires a full-length book study yet has to cope with four internalist retrospective volumes [50–53], and assorted papers on specific topics (e.g., [54–57] on ALGOL alone). A short introduction here must suffice.

Early computers were given "orders" encoded into machine-readable commands, with a structure closely mirroring that of the hardware [58]. As the complexity and power of machines grew, programs became longer, and there was a desire for more compact and human-intelligible programs [59]. Storing program texts on the computer enabled such richer programming systems [10]. One example was the "subroutine", a frequently used sequence of commands bundled together; these were originally saved in a (paper) notebook before later being loaded from machine storage [60]. Programs were made by compiling various such subroutines, providing the origin for "compiler". An important routine was the "interpreter routine" which could translate programs written in a pleasant notation into machine code on the fly [61].

Hopper later referred to one such system, the MIT Summer Sessions Computer, as a "mask" being worn by Whirlwind, the computer on which it was running in 1952 [62]. This

put a different face on the machine, one with a friendlier demeanour towards the students using it. It is worth observing that Hopper referred to the program providing this "mask" not as a system or notation or language, but actually as a different machine.

"Coding" entered the lexicon of the computer user from that of the telegraph operator, and denoted the (often repetitive, boring, later gendered, and even later "reclaimed") task of translating a desired behaviour into a machine-readable (but human-unreadable) notation [63].[8] Coding systems, programs which automated some of the task of coding, grew in popularity, flexibility, and number throughout the 1950s, with some even still enduring today. The programs created with these systems became increasingly important to their users, who wanted to be able to run them on newly acquired machines, or to share programs across organisations with different machines—ignoring differences in hardware. Thus grew the search for an ultimate, or universal, language in which such programs could be easily expressed and could be easily compiled or interpreted for any machine [6,11].

The goal of an ultimate universal language has a longer history, much of which is covered in Umberto Eco's *Search for the Perfect Language*, though he does not consider programming or other scientific languages [64]. Blumenberg wrote that mathematics had such an ideal since the time of Descartes, yet it proved repeatedly unreachable [38]. This, argued Friedman, was unavoidable: a universal language with no ambiguity, as Descartes desired, would be without metaphor, and it is in fact the power of both metaphor and ambiguity which enables nonconceptuality—encounters with the not-yet-known—providing the discovery mechanism for new knowledge. The LANGUAGE metaphor for programming, for example, enabled ideas of translation,and new definitions of equivalence as "mutually inter-translatable" [11].

Another important intellectual development, enabled by the separation of programming notations from specific machines, was the reification of programming languages as independent entities; together with the LANGUAGE metaphor, this opened up new ontologies of programming [3].[9] However, the introduction of intermediaries between program and machine brought dangers: the very programs which translated "high-level languages" into machine code (compilers) could contain errors like any other program. Without the operations of a machine to inspect, how could these errors be found—independent of errors in programs? How, indeed, could the correctness of a language itself be ascertained? Working on independent high-level languages led some to feel what Strachey characterised as "a rather vague feeling of unease, and though we think we know what we mean about [various language features] we are not altogether happy that we have really got to the bottom of the concepts involved" [66]. The realisation towards the end of the 1950s that trying to develop a semantics from an implementation was extremely challenging [13] led some, like McCarthy [67], to start searching for the basic first principles from which a properly mathematical theory of computation could be formed.

This short introduction to the history of programming languages and how it created a context in which work began on formal semantics is expanded in my earlier work [3,56,68] and is mentioned in other contexts by others [10,16,24,69]. In brief, the need for clearer ways to understand programming languages led to the creation of specification mechanisms characterised by concepts brought from mathematics, logic, and linguistics: formal, precise notations, and abstract entities whose behaviour or properties were related to that of the programming language. We shall see one such example in the next section.

4. Texts and Machines

I turn now to the early work of Van Wijngaarden for a historical case study. His later work on ALGOL 68 is perhaps better known (see [58,70] for some historical accounting).

That controversial programming language, widely disliked by the academic computing community who had embraced ALGOL 60 [71,72], was presented along with a novel and complicated formal definition; I argued that this unfortunate pairing led to the non-acceptance of both ([68], §7.1). Van Wijngaarden's earlier work on formal definition and semantics specifically, however, is much less known. The current section is divided into four parts. First, I sketch the background and context of Van Wijngaarden's work. Then, I consider the major components of his approach to programming language definition. Finally, I discuss and analyse the reaction to this approach, alongside its key illustrative devices.

4.1. Background

Adriaan van Wijngaarden (1916–1987) graduated from the Technical University of Delft with a degree in mechanical engineering in 1939 [73]; he worked there through the Second World War on fluid mechanics calculations, which he found tedious and rejected writing up because "it lacked beauty" [74]. Following the War, he was sent on a tour of the UK, including to Cambridge, to see what could be learnt about mechanical and naval engineering. Instead, he returned full of excitement about electronic calculating machines. When a new research institute, the *Mathematisch Centrum* (Mathematical Centre), was founded in 1946, Van Wijngaarden was invited to run its computing department, starting in 1947 [73].[10] There, he worked on building machines, and the first, the *Automatische Relais Rekenmachine Amsterdam* (Automatic Relay Calculator Amsterdam, ARRA), was electromechanical in nature. Van Wijngaarden launched the programming career of a young physics graduate, Edsger Dijkstra, hiring him in 1952 to help with the machine's programming even before its dedication that year [75]. It is less well known that Van Wijngaarden also hired twelve young women[11] to undertake calculation and programming work; these *rekenmeisjes van Van Wijngaarden* (computing/calculating girls of Van Wijngaarden) helped create the ARRA and its successor ARRA II [77].

While those around him worked on programming, Van Wijngaarden's work focused on mathematics and engineering until a sudden shift in 1958 towards programming languages following a visit to the UK in which his wife was tragically killed in a car accident [74]. In this period, the International Algebraic Language, later known as ALGOL 58, had just been published,[12] and Van Wijngaarden became deeply involved in developing its successor ALGOL 60. He was an author of the 1960 Report [80] and the 1963 Revised Report [81] which were the language's defining documents (from this point on, I shall use "ALGOL" to mean "ALGOL 60", as was the terminology of the time). Together with Dijkstra, he made an important contribution to ALGOL by insisting the language must have recursive procedures: "Asked why recursion was such a big issue to the Amsterdam team, van Wijngaarden responded in terms of ethos, "a matter of honour and intellectual decency" ([82], fn. 27).[13]

At Mathematisch Centrum, Dijkstra and Jaap Zonneveld implemented a (indeed, the first) compiler for ALGOL 60 to run on the newly developed Electrologica X1 machine. Contrasted with the second compiler, the ALCOR effort led by Bauer and Samelson across a number of institutions, which implemented a subset of ALGOL for efficiency's sake, the X1 compiler was slow but implemented the whole language. This was typical of Van Wijngaarden's lead: he favoured flexibility in a language over restriction in the name of efficiency. As a result, Samelson called him a "liberalist" or "trickologist" [79]. The term "generalist" is preferred by historians Alberts and Daylight [82]: Van Wijngaarden insisted on the equal treatment of all language constructs—some would say to a fault with ALGOL 68 later—and this was also remarked upon by Zemanek ([84], pp. 1–2):

> "From the very beginning I have sensed the dual character of his unique personality: the large mind which has always extended beyond my horizon, and the

> sharp brain that can suddenly focus on the smallest detail, but will illustrate by it some general aspect; the "generalizer" who generalized even a general purpose programming language, and the "specializer" whose production of sentences and questions has often reminded me of a pencil sharpener."

What is meant by "generalising a general purpose programming language"? This describes the two major early works of Van Wijngaarden which are the focus of my case study. The context is the search in the early to mid 1960s for a successor to ALGOL 60, from which Van Wijngaarden's proposal eventually was chosen ([68], §7.1). In the early works, however, he was taking his ideas about ALGOL and working on developing something broader and more universal for a new language.

Van Wijngaarden presented "Generalized ALGOL" [1] in Rome, 1962, at *Symbolic Languages in Data Processing*, the first symposium established by a new UNESCO-sponsored International Computation Centre [85]. The Centre was an effort to establish a collection of top-end computing machines which could be used and shared by research establishments across Europe; its symposia were intended to raise the profile of the Centre. "Recursive definition of syntax and semantics" [2] was presented two years later at the 1964 *Formal Language Description Languages* in Baden-bei-Wien. This conference, the first IFIP Working Conference, brought together programmers, language designers, machine creators, and formal language theorists; it had a catalysing effect on European computer science as I outlined in [3].[14] It is a particularly valuable historical record, since post-presentation discussions were recorded and transcribed for the proceedings; sadly, the discussions following Van Wijngaarden's earlier presentation were not recorded.

4.2. Processing and Preprocessing

Despite his input, Van Wijngaarden was unhappy with the generality of ALGOL, and wanted it unburdened from syntactical niceties. As part of the move towards developing independent mechanisms for specifying programming language semantics highlighted above, Van Wijngaarden laid forth his ideal ([1], p. 17):

> "the definition of the language should be the description of an automatism, a set of axioms, A MACHINE or whateverone likes to call it that reads and interprets a text or program, any text for that matter, i.e., produces during the reading another TEXT, called the value of the text so far read."

My emphasis is added here to highlight the central illustrative devices: A PROGRAMMING LANGUAGE IS A MACHINE and A PROGRAM IS A TEXT.

Let us start with the latter. It is clear that Van Wijngaarden is not talking merely about the syntax of ALGOL, which was defined using BNF.[15] This had provided a simple and flexible way to summarise which strings of symbols constituted valid ALGOL programs and also coincided with developments in formal language theory in pleasing ways [87]. However, Van Wijngaarden was uninterested in syntax, calling his approach "syntax-free" ([1], p. 17); instead, he viewed the text of the program as its true form, its sole medium of import, and its ultimate meaning.

What form did this vital text take? The quotation above tells us: the text of the written program itself, plus a second (which he referred to as **V**), a "sequence ... consisting of truths separated by commas" ([1], p. 20). The truths are propositions, mostly equalities and memberships, over "names", which entities Van Wijngaarden described carefully, though informally. A name is some reserved symbol (like $+$), or a character from which an identifier[16] might be built, but names can also be concatenated directly, sequentialised with commas, and grouped with brackets. The end result of this is that a full program text is in the end itself a name, and therefore there is no real distinction between a program text and a name in **V**—though Van Wijngaarden does not clearly highlight this identity

nor explore its implications.[17] The meaning of names in **V** is given by the presence of the reserved word **value** in equalities, which indicates that the value of the name following is that of the right-hand side of the equality. See an example truth from **V** in Figure 1.

At the beginning of interpreting a program, **V** is not empty. Some truths are predetermined by the language, such as the rules for performing addition. Many others are added through the course of interpreting the program. Van Wijngaarden is careful to note that the order of **V** matters: it is arranged by *complexity of names*, with the most general at the "bottom", considered last during interpretation.[18]

value {⟨sum 1⟩ + ⟨term 1⟩} = **value** {**value** ⟨sum 1⟩ + **value** ⟨term 1⟩}

Figure 1. A truth from **V** necessary for determining the value of sums ([1], p. 20).

To understand the use of **V** we must turn to the other chief metaphor, A PROGRAMMING LANGUAGE IS A MACHINE. But this is not a specific, physical machine since this would be insufficiently general for Van Wijngaarden who wrote ([1], p. 18):

> "We rather see the language as a machine *M*0 which is fed with the program at one end and produces the value at the other end. The rules of the language, i.e. a rough description of the working of *M*0 is printed on the lid of the machine".

The passage here is that which opens the current paper; I chose it as an epigraph since it is deeply redolent with illustrative devices. Now, in context, we can pick it apart.

The language description printed on the machine's lid is simple, and high-level, so the user may wish to know in more detail exactly how the language is working. For this, they "can open the machine to inspect the precise working". Inside, surprisingly, there are two more machines, named *P*1 and *M*1: a preprocessor and a processor. *P*1 "chews the offered text" to make a more basic input for *M*1. Each has a further textual description on its lid, more basic but less intuitive: "harder to understand" but "more uncertainties are settled" ([1], p. 18). Eventually there comes a machine with no further simplification possible and its lid cannot be opened.[19]

It is the operation of the processor machine which deals with **V**. To create the ultimate value of the program—itself a name in **V**—the machine dynamically reads the program, running through the truths in **V** until it finds one which is applicable, and performs a symbolic transformation on the program text. The search for applicable truths is the closest Van Wijngaarden comes to caring about syntax, since certain rules like "*a* **in** ⟨letter⟩" indicate that *a* is a suitable value for the meta-linguistic variable ⟨letter⟩ ([2], p. 16).

The presence of **value** in a truth means "look again in **V** to The name is repeatedly rewritten by the processor machine until no more rewriting is possible; What remains is the meaning of the name, its semantic interpretation; and since a program is itself a name, in this way the meaning of a whole program is determined.[20] If the resulting text is nonsensical, this indicates the program was meaningless, i.e., likely contained a syntactic error. Any new truths found along the way—like an assignment statement causing a new equality between an identifier and its value—these are added to **V**.

The final text of **V** contains not just equivalences showing the final value of all variables, but also a kind of history of computation, and all the rules of the program as they are created. Van Wijngaarden even notes that the rules in **V** can be altered before a program text is entered, providing different operating rules for the machines, or to say that another way, a different semantics for the language, but it is not clear how this is different from the way in which **V** is altered during a program interpretation. Perhaps the only difference is that of precedence: the truths introduced by the language designer having more power. It also implies that since **V** controls the operation of the machine, and the MACHINE IS THE

PROGRAMMING LANGUAGE, then a language is changed during the course of a program execution.[21]

A sample of the value list is shown in Figure 2.

value {⟨simple name 1⟩, ⟨sequence of symbols 1⟩} =
{**value** ⟨simple name 1⟩, **value** ⟨sequence of symbols 1⟩},
if {⟨name 1⟩ = ⟨name 2⟩} **in V then value** ⟨name 1⟩ =
value ⟨name 2⟩,
value {⟨variable 1⟩ := ⟨expression 1⟩} = {⟨variable 1⟩ =
value ⟨expression 1⟩},
if {⟨variable 1⟩ = ⟨variable 2⟩} **in V then**
value {⟨variable 1⟩ := ⟨expression 1⟩} = **value** {⟨variable 2⟩ :=
⟨expression 1⟩},

Figure 2. An example **V** containing truths about assignment for the processing machine to use ([1], p. 23).

The centrality of the program text here provides surprising power. As an example, the use of quotation marks in truths suffices to handle procedures, including parameters called by name (procedures without parameters are already handled by the second half of Figure 2—the procedure definition would be in the form ⟨ variable 1⟩ = ⟨variable 2⟩). A truth like *s* := '*t*' delays the evaluation, the first pass of the processor machine simply removing the quotation marks, such that "a note is left in **V**" ([1], p. 23) to be evaluated at the next pass.

Together, the processor machine, the value list **V**, and the operation of the machine when it hits **value** create something approaching the "state" as found in many other semantics methods (described in, e.g., ([36], §3.3)). Yet, everything here is textual; the only entity is a text being manipulated at the level of character recognition. While ideas such as equivalence between procedures and blocks with name substitution for parameters were not unique to Van Wijngaarden, he was working at a purely textual level. There are no meta-equivalences in this world.

In the later paper, Van Wijngaarden's discussion of the processor gave another indication of his view of the power of pure symbol manipulation, as he defined decimal addition and subtraction in this manner ([2], pp. 17–18). The metalanguage of **V** is also extended with the introduction of logical operators, and the ability of the writer to define new operations.

Meanwhile, what of *Px*? It is dealt with in more depth in "Recursive definition of syntax and semantics" [2]. Van Wijngaarden gives his motivation for the processor machine in the discussion ([2], pp. 18–19):

> "If I look at the [ALGOL] Report, I say to myself, must I define all this by the processor—all these rules? This is far too much for me! So I say, let's first take all the nonessential things out of ALGOL. Now, this is a task for the preprocessor—to look at this text and say, 'I'll translate this text into reduced ALGOL and then define only reduced ALGOL'."

Specifically, the preprocessor breaks down ALGOL concepts and replaces them textually. As an example, a switch statement is removed by transforming it into a procedure which accepts an integer parameter (the switch variable) and which contains a series of conditional statements leading to go to statements. In another example, a function is transformed into a procedure with an extra variable in the surrounding context to which the desired return value is ultimately assigned. In this way, transformation *becomes* definition:

"By such an intricate but still lexicographical process, one not only eliminates the function designator, but actually defines what it means" ([2], p. 14). Note here the emphasis remains on the text as the ultimate medium of meaning.

It is critical, therefore, that the preprocessor preserves meaning; the program cannot be changed by the preprocessing. When asked to confirm no loss of meaning, Van Wijngaarden replied "Sure. You see, I do not change any identifier" ([2], p. 19). Here, we can see a subtle but important distinction being made between *parts of the text*: those which preserve meaning and those which do not.

Notably, the illustrative device of a machine with lids, instructions, and jaws is de-emphasised in the second presentation; although Van Wijngaarden still talks of a machine in two parts, processor and preprocessor, the MACHINE is mentioned specifically only twice. The preprocessor does "stand by" ready to preprocess new pieces of text when a language is being considered that can generate new texts, but it does not chew them in the same way. The presentation rather becomes more anthropomorphic, with the narrator taking actions to change the text just as much as a machine does. There is one more visual flourish worth noting: the preprocessor has the ability to change the colour of the text as a visual aid: "One might visualize the unpreprocessed text as written in black ink, whereas the preprocessor turns out text in red ink" [2].

The most provocative and historically significant aspect of the preprocessor is the way Van Wijngaarden uses it to remove functions and go to statements, transforming them all into procedures. The idea, which he outlined in the same informal but precise language used throughout the papers, is to link the end of each block with the start of another, through careful lexical transformations. He explained ([2], p. 24):

> "if you do this trick I devised, then you will find that the actual execution of the program is equivalent to a set of statements; no procedure ever returns because it always calls for another one before it ends, and all of the ends of all the procedures will be at the end of the program: one million or two million ends. If one procedure gets to the end, that is the end of all; therefore, you can stop. That means you can make the procedure implementation so that it does not bother to enable the procedure to return. That is the whole difficulty with procedure implementation. That's why this is so simple; it's exactly the same as a goto, only called in other words."

4.3. Discussion

Van Wijngaarden's presentation at *Formal Language Description Languages* provoked a very lengthy discussion—approximately half the length of the published chapter is a transcription of the discussion—despite being the middle talk in a block of three [92]. It is worth noting that full papers were circulated to attendees in advance [93], and on the day each author delivered a presentation, to be followed by discussion. We do not have the presentation on hand, and this could of course have differed from the paper in the proceedings. In this way, we see a strange mixture of a finished piece of prose and the reaction to an oral/visual demonstration; we are left to infer, as did Netz with the mathematics of ancient Greece [37], exactly what that demonstration entailed.

The major reaction was puzzlement; Dijkstra began by stating "I am somewhat baffled, I might say, in many ways." ([2], p. 20) Many of the participants seemed to struggle to understand the core of Van Wijngaarden's idea, which caused him quite some annoyance, provoking exclamations of frustration, as in the exchange below about the go to transformation quoted above ([2], p. 21):

> DIJKSTRA
> I just don't understand. You have your text. You make another text that remains

> without procedure calls. Now, you say that somewhere or another you make the insertions; so that we do without procedure calls. Now we have only goto's.
>
> VAN WIJNGAARDEN
> What! What? What? You have only *statements*—sequences of *statements*. You have no goto's whatsoever! You have only sequences of statements, and these sequences of statements are *exactly* the same sequences of statements that would have been there in the other case. ([2], p. 21)

The transformation of go to statements into procedures (and then into the sequence of statements described above so vehemently) has an interesting subsequent history. While McIlroy challenged this as meaning "the entire history of the computation must be maintained" ([2], p. 24), in fact *that* was the approach put forth by McCarthy in his paper at the same conference [47].[22] What Van Wijngaarden did was repeatedly change the program text, not so much maintaining the history as flattening it out. Reynolds argued that this was the first published example of what was later called "continuation passing style" [94], although he is reinterpreting the work, sixty years later, in light of many subsequent results in programming language theory.[23] It is in some sense surprising that none of the influential computer scientists present at Van Wijngaarden's talk, including Landin and Strachey, both of whom later worked with continuations, made any connection back to Van Wijngaarden. But, as Reynolds observed, the prose was opaque and contained a minor technical error in the description of the transformation., I would add that as Van Wijngaarden was not really concerned with programming language theory in the same way—and saw the function as less basic than the procedure—he did not present his work with the illustrative devices which would have been amenable to the later functional programming community. The go to handling did make a more immediate historical impact, however: Dijkstra spent the next coffee break thinking about what Van Wijngaarden had said and scribbled some ideas on a napkin; this led to his famous letter to the editor of *Communications of the ACM* making a case against the go to statement (noted by McIlroy, reported in [94]).

Let us now start to unpick the illustrative A PROGRAMMING LANGUAGE IS A MACHINE device. An astute question was raised by Hoare, who asked why a distinction was made between the preprocessor and the processor; could the processor not do all the actions of both? Van Wijngaarden replied that the difference was being made only "for psychological reasons", with the preprocessor having the job of removing "wild ideas" which "people might introduce in languages" ([2], p. 21). Thus, the preprocessor machine acts as a normalising force, reducing a complex and unpleasant world into a more straightforward one. It is surprising that Van Wijngaarden did not return to his chief illustrative device of the earlier paper, with the lidded machines; he might have told Hoare that if he preferred, he could think of $P1$ as sitting within $M0$. Had Van Wijngaarden removed this nesting from within his own mental picture of the semantics? Without a reference to the illustrative device, it is not clear; indeed, the paper has more references to the machines' juxtaposition than their recursive structure—despite the paper's title! It is also noticeable that the anthropomorphism of the processor and preprocessor grows significantly within the discussion; when discussing his work orally, Van Wijngaarden repeatedly identifies himself with these, speaking of their functionality in the first person. Here is a kind of synecdoche, with the author standing in for the chief agent of his work.

There is a deeper point underlying Hoare's question. Which operations of the machine, or machines, constitute semantics? Many later approaches to programming language semantics were concerned with producing an ultimate value or denotation; but, from Van Wijngaarden's work, one feels it is rather the ongoing removal and replacement of text which is the act of finding meaning. It is therefore more like an operational semantics, with the meaning of the programming language invested in the MACHINE. Since there is,

however, a difference between the parts of the text that are fair game for being chewed by the machine and those that are not (only identifiers?), this distinction is important but not clear. Wirth, in a publication responding to Van Wijngaarden's papers [96], notes that it is not certain when a portion of text is regarded as fully dealt with by the preprocessor and when it needs further attention (should it go red as soon as the preprocessor has touched it, or only when it is finished?).

Van Wijngaarden's machine, at least in [1], takes on some additional embodied characteristics, in Lakoff and Johnson's language "personification" ([28], Ch. 7): the processor machine is "fed" a program and the preprocessor machine "chews" the text. We see not just the ascription of agency to the machine, but also a kind of animalistic hunger: there is also metonymy here as the machine's behaviour is reduced from full personhood to merely a masticator. This contrasts with the apparently helpful nature of these machines; recall the distinction between processor and preprocessor introduced in ([1], p. 18) to assist the user who wants to understand in more detail how the language is working: deeper nested machines are "harder to understand" but "more uncertainties are settled". At the same time as reducing uncertainty, the machine(s) remove intuitiveness: we see here an embodied mechanisation of the role of semantics in providing interpretation between levels of abstraction—as in the work of Rapaport [14]. Further, we can observe a juxtaposition between the way in which Van Wijngaarden's textual manipulation happens at a very basic level by the processor, providing a kind of "semantics of the word" in the Ricoeur sense [44,45]: names in the program and in **V** are given meaning via low-level transformations; one might view the preprocessor then as a "semantics of discourse", giving meaning at a higher level. But, a continuity exists between these machines, even as they appear and disappear throughout Van Wijngaarden's work. Indeed, this appearance and disappearance of the machine recurs throughout the early years of computer science, and the slow disappearance of the same is a characteristic of the maturing intellectual field [11]. As such, the machine becomes increasingly less attractive as a metonym to illustrate the computer–program–abstraction complex. Van Wijngaarden was clearly a machine-in person in this period (though perhaps less in his later work) and he was not alone in this view.

Peter Landin, who also spoke at the 1964 conference [97], developed a semantic approach involving translation of program constructs into "applicative expressions" which are interpreted using a combination of states and transition rules which Landin called a "machine" [98]. Though he was typically reluctant to take credit for ideas, he did allow himself the invention of the "abstract machine" as a way to interpret a program. This work was influential in particular on the IBM Laboratory in Vienna, whose subsequent work creating a "Universal Language Document", a formal specification of PL/I, referred to their semantic mechanism as a "machine (interpreter)" [99].[24] In the early work of Dijkstra, he saw a close correspondence with languages and machines, writing "A machine defines (by its very structure) a language, viz. its input language; conversely, the semantic definition of a language specifies a machine that understands it" ([100], p. 1).

Another conception of machine was put forward by Gorn in a summary of the 1963 *Working Conference on Mechanical Language Structures* [101]. Gorn wrote of a "background machine" which he felt must be in the mind of every programmer who looks at a program and imagines it being run; however, unlike the Landin and Vienna machines, which were abstract, formalised entities, Gorn's had clear hardware aspects: control counters, instruction registers, and address selectors. Indeed, he claimed "I am one of those extremists who feel that it is impossible to separate a language from its interpreting machine" ([101], p. 133). In other words, Gorn takes a position like that advanced by simulation semantics: meaning is inextricably linked to a mental imagining of the entity's affordances [40]. Van Wijngaarden, however, had already written that he did not wish to use "precise axioms" or a "precise

description of the machine" since the definition of a language should allow "distinction between fundamental concepts and useful but logically unnecessary conventions" ([1], pp. 17–18)—thus making an identity between a precise machine, and a precise syntax.

So, we can see a variety of different machines acting as illustrative devices even in this small corner of computing history; while the name is shared, the characteristics are very different, which suggests that to understand more deeply A PROGRAMMING LANGUAGE IS A MACHINE we need to interrogate the machine in question. Van Wijngaarden's machines were strongly focused on symbol manipulation, even to the extent that they disregarded numbers ([2], p. 23):

> "Now, you say that numbers are not strings in that sense. Now, I know exactly what a number is; it is a string of digits. It may be preceded by a plus or minus sign, and it may be preceded by a decimal point. There is no other thing in ALGOL that is a metaconcept called 'number' of which this is the number. To me the number 13 is just the sequence of symbols 1, 3. I have never seen a 'number'."

This intense focus on the text is reflected by the memories of contemporary Dana Scott, who wrote that Van Wijngaarden claimed "If you can't do it by symbol manipulation, then it's not worth doing it" adding that Van Wijngaarden was next to Haskell Curry in being the most symbolically minded people he knew [102]. Van Wijngaarden himself stated this, telling Landin "I can't do any kind of computation apart from acting on symbols" ([97], p. 294).

This heavily textual attitude to programs is not one shared by many—if any—other writers of semantics. Indeed, McCarthy directly opposed the use of strings of symbols as a fundamental basis for semantic interpretation, which he felt was too connected to implementation specifics, and contrasted Van Wijngaarden's attitude with his own desire to use an abstract syntax ([2], p. 23) (as in, for example, [47]). Dijkstra also opposed the focus on text, writing later [103]:[25]

> "It is practically impossible to give such a mechanistic definition without being over-specific. The first time that I can remember having voiced these doubts in public was at the W.G.2.1 meeting in 1965 in Princeton, where van Wijngaarden was at that time advocating to define the sum of two numbers as the result of manipulating the two strings of decimal (!) digits. (I remember asking him whether he also cared to define the result of adding INSULT to INJURY; that is why I remember the whole episode.)"

Yet, these were conflations; Van Wijngaarden had the same desire to avoid implementation specifics, wishing that "the language should not be burdened by syntactical rules which define meaningful texts" ([1], p. 17). He simply saw strings of symbols as the appropriately abstract core basis. Such a basis, however, was confused by the multiple roles being played by the text. As Wirth raised, it was not obvious which rules should be seen as syntactic and which semantic [96]. Van Wijngaarden would likely argue that he wanted to avoid a distinction here: all he works with is text, but all of it could be value producing. Only when the processor finds no applicable rule in **V** is the text found to be meaningless (and therefore syntactically invalid). So, syntax was managed in some way by Van Wijngaarden, despite his desire for a syntax-free language.

A PROGRAM IS A TEXT will take some more analysis here than A PROGRAMMING LANGUAGE IS A MACHINE. It could be considered in the Lakoff and Johnson mould as part of an objectivist theory of communication, in which a version of the LANGUAGE IS CONDUIT metaphor is used. Meanings are objects and so are linguistic expressions; communication happens when the latter carries the former from the speaker to the hearer ([28], Ch. 26). But, there is something else going on here, because the text is at the same time the static

program, the ongoing meaning of the program, and the ultimate semantic basis, all mixed together. In this sense, it brings something of the character of mathematics, as explained by Sha ([19], p. 39): "simultaneously concept and technology [...] both the generation of the concept and the means to articulate and actualize these concepts as material." While Sha presented this as a special character, the same remarks could apply to the study of natural languages, or even neuroscience. The strength of text as a basis was its flexibility; but, as an illustrative device, it caused problems: being too reductive, being an unhelpful metonym, and bringing conflation rather than distinction. Without an extra abstract framework (like McCarthy's abstract syntax [47]) naming the parts of the program, and with **V** containing program text as well as meta-identities, operators, and dynamic values, and a lack of clear distinction between the processor and preprocessor machines—there is a great deal of conflation and confusion. As Wirth wrote, "The reader is left with the uneasy feeling of having been told only half the story" [96].

Here is where a strong illustrative device could have helped clarify certain matters: but MACHINE had so many different meanings, even amongst the small group of attendees in Baden-bei-Wien in 1964, and TEXT carried many undesired connotations. The presentation format of Van Wijngaarden's work, with his unusual choice of illustrative device, obscured some of the intellectual merit in ideas which went on to have great penetration in computer science [94]. Considered from Lakoff and Johnson's perspective of embodied realism [34], the truth of Van Wijngaarden's models was lacking since his illustrative devices failed to convince in the social situation of academic presentation; taking his presentation as a rhetorical argument, in the style of Greek mathematical proofs as discussed by Netz [37], the "atoms of necessity" which begun his argumentation were not agreed as necessary by his audience, and they were not convinced his transformations preserved any necessity present. Van Wijngaarden's work needed to change to reach a larger audience—and ultimately convince the ALGOL committee to entrust its successor project with him. There is evidence of this beginning even in the 1964 presentation, as some of the machinic imagery diminishes in favour of a focus on language, and the trappings of the nascent formal semantics field appear. His later work on ALGOL 68 would continue this trend, making great use of powerful "two-level grammars" for transforming program texts, and only a little attention paid to an underlying interpreting machine [104].[26]

The two core metaphors in Van Wijngaarden's work are not terribly uncommon in the intellectual context of the time, and nor is their progeny too hard to grasp. Machines were a physical presence in the room, too large, noisy, and smelly to ignore; and programs as textual or metatextual objects is a consequence of autocoding, even the stored program concept itself. The more mathematically inclined people in the room may have read Church or Post and been familiar with the ideas of textual rewriting. Yet, perhaps due to the changing nature of computer science in this moment, away from the machine and towards something more vauntedly abstract than "mere" syntax (for which Van Wijngaarden's text was mistaken), the metaphors failed to communicate effectively with Van Wijngaarden's audience. Further historical work could uncover deeper heritages for illustrative devices in semantics and help take apart their composition and consequences—the role played by TEXT is clearly linked to the role of text in human–human communication, but philosophical analyses of the same miss of the distinctive aspects of program texts. More work is needed to uncover, classify, and unpack the vast array of illustrative devices in computing—and this work could have many important uses.

5. Future Work

It is noted by Nofre that metaphors play a "normative and epistemic" role in science, and their use in computer science (and its history) is largely unstudied.[27] Hans Blumen-

berg's characterisation of the space of mathematical knowledge as "controlled ambiguity" follows the late Wittgenstein: rather than fossilised in theorems, knowledge is open for re-exploration and redefinition in multiple forms, which forms may themselves unlock new knowledge [106]. This period in the history of semantics shows multiple cases of experimentation with knowledge forms—to varying levels of formality, and consequent varying levels of openness. These forms, such as the illustrative devices of MACHINE and especially TEXT from Van Wijngaarden were contested and argued over, provoking strongly felt responses from other participants and Van Wijngaarden himself. Johnson [32] insists that such emotions, feelings, and aesthetics are a crucial part of knowledge, with the embodied qualities of many metaphors having power precisely because of these evocations.

Another direction for more research, for which I am grateful to a suggestion from an anonymous reviewer, is an exploration more that is informed by the frameworks of literary analysis and rhetorical devices, which I have but hinted at in this paper.

Evidence from recent cognitive science suggests that language has a particularly strong role in shaping abstract conceptualisation and, possibly as a consequence, understanding of abstract concepts tends to be more culturally and linguistically contingent than concrete components [107]. Given the high preponderance of abstract ideas in computer science, analysis of the language is likely to be fruitful for further work in understanding computer science. One direction for study here could be the use of spatial and motion-based metaphors, which are prevalent in mathematics [30]. Recent studies in computer science education suggest that improved spatial skills correlate with STEM achievement in general [108], and effective grappling with computer science concepts in particular [109].

The importance of metaphors in education more broadly is highlighted by Low [110] who points to their role in replacing unfamiliar abstract concepts with more familiar ones, and the transfer of structural relationships from source domain to target (though noting this can also be problematic when unwanted properties are transferred). Ricoeur argued that metaphors have a particular importance in teaching when they are fresh and "living", since they offer the student a feeling of participation in the learning experience [44]. Beyond metaphors, Eriksson et al. [111] insist on the need for analogy competence in science educators; following my point above about the Dining Philosophers scenario, I contend that this should be broadened to the full illustrative device category. There is some empirical work undertaken already applying conceptual metaphor theory in computer science education: Sanford et al. [112] interviewed educators on an introductory module about their metaphors, and Harper et al. [113] undertook classroom observations of a similar module, specifically observing the use of metaphors to describe recursion.

With these works and that of Nofre showing the value of examining metaphors in computing and the present paper showing one case study, I propose there is space for a long-form examination of illustrative devices across the history of computing, and a consideration of their impact in the classroom and beyond. For example, a metaphor that has given rise to an entire theory of computation is that centered on actor/script/message, used by Hewitt [114] in his attempt to understand Alan Kay's ideas on object-oriented programming.[28] I am grateful to another anonymous reviewer for providing this suggestion. One angle which may prove valuable is considering those devices which show particular cultural contingency, such as Dijkstra's SECRETARIES and DIRECTORS for hierarchies of concurrency [49]; the MISSIONARIES AND CANNIBALS algorithm presented in, for example, [116,117]; and the widespread (though today diminishing) MASTER and SLAVE [118].

Funding: The author would like to thank the Isaac Newton Institute for Mathematical Sciences, Cambridge, for support and hospitality during the programme Modern History of Mathematics, where work on this paper was undertaken. This work was supported by EPSRC grant EP/Z000580/1.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: No associated research data.

Acknowledgments: Thanks to Cliff Jones for supporting the original research leading to this material, and to Michael Friedman for pointing me to his work which broadened my understanding (and literature!) significantly. Thanks also to two anonymous referees whose reports were very valuable in improving the analytical framing and the text itself.

Conflicts of Interest: The author declares no conflicts of interest.

Notes

1 Lakoff and Johnson [5] highlighted metaphors with small caps, and I use the same typography for illustrative devices.
2 Other examinations of the terminology of computing include an ethnography of Ruby coders [7] and a queering of the stack rhetoric [8].
3 The encounter provided new epistemic tools and practices for the study of human languages as well [11].
4 A useful overview of key concerns in the philosophy of computer science is given in [12].
5 See [25] for a history, or [26] for a more irreverent framing as a history of errors.
6 Netz applies this criticism to *all* cognitive science, arguing that attempting to make universal statements about cognition is very hard (perhaps impossible), but cognitive *history* (study in a specific context) may be performed fruitfully [37].
7 Here, in addition to the previously mentioned work on inscriptive and performative mathematical practice [19,20], we of course are reminded of Latour [43].
8 The cited chapter presents a historico-philosophical study of other names—actually, illustrative devices in themselves—for programming entities (ALGORITHM , CODE, and PROGRAM).
9 This distancing from the computer was not unique to programming; as argued by Dick [65] and Nofre [11], much of the nascent science of computing was legitimised exactly by its removal from any physical actuality in favour of abstraction, generalisability, and the enviable Queenship of mathematics.
10 Van Wijngaarden went on to lead the entire organisation in 1961 and remained there until his retirement twenty years later, just before its name change to *Centrum Wiskunde & Informatica* (Centre for Mathematics and Computer Science, CWI).
11 Eddy Alleda, Dineke Botterweg, Ria Debets, Marijke de Jong, Bertha Haanappel, Emmy Hagenaar, Truus Hurts, Loes Kaarsenmaker, Corrie Langereis, Reina Mulder, Diny Postema, and Trees Scheffer. Save for Eddy, who became a mathematics teacher, the rest left professional work for the family home. This is a sadly common pattern in the time period; for more on the ways women were forced out of computing work, see [76].
12 For more history of the ALGOL effort, see [55,78,79].
13 Further discussion is provided by Van den Hove, who observes that since the syntax of ALGOL is defined recursively, it would in fact require serious effort to make an ALGOL implementation which did *not* support recursive procedures [83].
14 A fuller description of the conference and its effect on the field of programming language semantics is available in ([68], Ch. 4).
15 Backus–Naur Form, or Backus Normal Form—see [86].
16 The ALGOL term given to named entities in a program, like variables and procedures.
17 This could be due to Van Wijngaarden's familiarity by this point with the stored program concept: a computer program may be stored in the same memory as any other data and operated on by another program likewise. The idea is traceable to Turing, but it was Von Neumann who operationalised it in a computer [88].
18 One might observe here some interesting similarity with Scott domains for semantics, which are partially ordered by information content [89]; of course at the time, Van Wijngaarden would not have known this.
19 Van Wijngaarden does not clarify whether Px and Mx can each be opened to find two more machines; the implication from the rest of the paper is that it is only Mx.
20 This is reminiscent of the Turing Machine, or perhaps Post's machine [90], although Van Wijngaarden did not cite either author in his work.
21 Jones and I make a similar observation that "programming languages provide a repertoire of basic operators and, crucially, put in the hands of programmers ways to express functions that extend this repertoire." ([91], p. 179).
22 The full program was a parameter to his semantic function, remaining such throughout interpretation; and this style went on to affect the subsequent "Vienna Definition Language"—see [56].
23 Influential historian of mathematics and computing Mike Mahoney would not approve; he criticised this kind of reinterpretation of past work in terms of modern concepts, writing "Historically, a rose by another name may have a quite different smell." [95].

[24] For more on this approach, historically interesting but lacking in much later influence, see ([56], §3).
[25] By the time he wrote this in 1974, Dijkstra had moved from being interested in programming language definition to program correctness and was typically scornful of earlier work.
[26] For more on this, see ([68], §7.1).
[27] Nofre's own work provides some exception; as well as co-authoring [6] on language, a recent work explores the power of the Tower of Babel metaphor in 1960s programming language communities [105].
[28] SmallTalk has a long history, and rather than citing any one publication on the language I offer Kay's own historical account [115].

References

1. van Wijngaarden, A. Generalized ALGOL. In Proceedings of the Symposium on Symbolic Languages in Data Processing, sponsored and edited by the International Computation Centre, Rome, Italy, 26–31 March 1962; pp. 409–419.
2. van Wijngaarden, A. Recursive definition of syntax and semantics. In *Formal Language Description Languages for Computer Programming*; North-Holland Publishing Company: Amsterdam, The Netherlands, 1966; pp. 13–24.
3. Astarte, T.K. "Difficult things are difficult to describe": The role of formal semantics in European computer science, 1960–1980. In *Abstractions and Embodiments: New Histories of Computing and Society*; Johns Hopkins University Press: Baltimore, MD, USA, 2022.
4. Astarte, T.K. From Monitors to Monitors: an Early History of Concurrency Primitives. *Minds Mach.* **2023**, *34*, 51–71. [CrossRef]
5. Lakoff, G.; Johnson, M. Conceptual Metaphor in Everyday Language. *J. Philos.* **1980**, *77*, 453–486. [CrossRef]
6. Nofre, D.; Priestley, M.; Alberts, G. When Technology Became Language: The Origins of the Linguistic Conception of Computer Programming, 1950–1960. *Technol. Cult.* **2014**, *55*, 40–75. [CrossRef] [PubMed]
7. Heurich, G.O. *Coderspeak: The Language of Computer Programmers*; UCL Press: London, UK, 2024.
8. Nelson, S.L. Computers Can't Get Wet: Queer Slippage and Play in the Rhetoric of Computational Structure. Ph.D. Thesis, University of Pittsburgh, Pittsburgh, PA, USA, 2020.
9. Berkeley, E.C. *Giant Brains: Or Machines That Think*; Wiley: New York, NY, USA, 1952.
10. Priestley, M. *A Science of Operations: Machines, Logic and the Invention of Programming*; History of Computing; Springer: London, UK, 2011.
11. Nofre, D. "Content Is Meaningless, and Structure Is All-Important": Defining the Nature of Computer Science in the Age of High Modernism, c. 1950–c. 1965. *IEEE Ann. Hist. Comput.* **2023**, *45*, 29–42. [CrossRef]
12. Angius, N.; Primiero, G.; Turner, R. The Philosophy of Computer Science. In *The Stanford Encyclopedia of Philosophy*, Summer 2024 ed.; Zalta, E.N., Nodelman, U., Eds.; Metaphysics Research Lab, Stanford University: Stanford, CA, USA, 2024.
13. White, G. The philosophy of computer languages. In *The Blackwell Guide to the Philosophy of Computing and Information*; Floridi, L., Ed.; Blackwell Publishing: Hoboken, NJ, USA, 2004; pp. 237–247.
14. Rapaport, W.J. Implementation is semantic interpretation. *Monist* **1999**, *82*, 109–130. [CrossRef]
15. Rapaport, W.J. Implementation is semantic interpretation: Further thoughts. *J. Exp. Theor. Artif. Intell.* **2005**, *17*, 385–417. [CrossRef]
16. Primiero, G. *On the Foundations of Computing*; Oxford University Press: Oxford, UK, 2019.
17. Turner, R. *Computational Artifacts: Towards a Philosophy of Computer Science*; Springer: Berlin/Heidelberg, Germany, 2018.
18. Pérez-Escobar, J.A. Showing Mathematical Flies the Way Out of Foundational Bottles: The Later Wittgenstein as a Forerunner of Lakatos and the Philosophy of Mathematical Practice. *KRITERION—J. Philos.* **2022**, *36*, 157–178. [CrossRef]
19. Sha, X.W. Differential Geometric Performance and the Technologies of Writing. Ph.D. Thesis, Stanford University, Stanford, CA, USA, 2001.
20. Barany, M.J.; MacKenzie, D. Chalk: Materials and Concepts in Mathematics Research. In *Representation in Scientific Practice Revisited*; The MIT Press: Cambridge, MA, USA, 2014. [CrossRef]
21. Maddy, P. *Realism in Mathematics*, Paperback ed.; Clarendon Press: Oxford, UK, 1992.
22. Shapiro, S. *Philosophy of Mathematics: Structure and Ontology*; Oxford University Press: Oxford, UK, 1997.
23. Benacerraf, P. Mathematical truth. *J. Philos.* **1973**, *70*, 661–679. [CrossRef]
24. MacKenzie, D. *Mechanizing Proof: Computing, Risk, and Trust*; MIT Press: Cambridge, MA, USA, 2001.
25. Jones, C.B. The Early Search for Tractable Ways of Reasoning about Programs. *IEEE Ann. Hist. Comput.* **2003**, *25*, 26–49. [CrossRef]
26. Petricek, T. Miscomputation in software: Learning to live with errors. *Art, Sci. Eng. Program.* **2017**, *1*, 14-1–14-24. [CrossRef]
27. Lakoff, G.; Johnson, M. *Metaphors We Live By*; University of Chicago Press: Chicago, IL, USA, 1980.
28. Lakoff, G.; Johnson, M. *Metaphors We Live By*, 2nd ed.; University of Chicago Press: Chicago, IL, USA, 2003.
29. Clark, K.M. Embodied Imagination: Lakoff and Johnson's Experientialist View of Conceptual Understanding. *Rev. Gen. Psychol.* **2024**, *28*, 166–183. [CrossRef]
30. Lakoff, G.; Núñez, R. *Where Mathematics Comes From*; Basic Books: New York, NY, USA, 2000.
31. Johnson, M. Philosophy's debt to metaphor. In *The Cambridge Handbook of Metaphor and Thought*; Cambridge University Press: Cambridge, UK, 2008; Chapter 2.

32. Johnson, M. *The Aesthetics of Meaning and Thought: The Bodily Roots of Philosophy, Science, Morality, and Art*; University of Chicago Press: Chicago, IL, USA, 2018.
33. Gibbs, R.W., Jr. (Ed.) *The Cambridge Handbook of Metaphor and Thought*; Cambridge University Press: Cambridge, UK, 2008.
34. Lakoff, G.; Johnson, M. *Philosophy in the Flesh: The Embodied Mind and Its Challenge to Western Thought*; Basic Books: New York, NY, USA, 1999.
35. Tedre, M. *The Science of Computing: Shaping a Discipline*; Chapman and Hall/CRC: Boca Raton, FL, USA, 2014.
36. Jones, C.B. *Understanding Programming Languages*; Springer Nature: Cham, Switzerland, 2020.
37. Netz, R. *The Shaping of Deduction in Greek Mathematics: A Study in Cognitive History*, 1st ed.; Number 51 in Ideas in Contex; Cambridge University Press: Cambridge, UK, 1999.
38. Friedman, M. On metaphors of mathematics: Between Blumenberg's nonconceptuality and Grothendieck's waves. *Synthese* **2024**, *203*, 149. [CrossRef]
39. Lassègue, J. La genèse des concepts mathématiques: Entre sciences de la cognition et sciences de la culture. *Rev. Synthèse* **2003**, *124*, 223–236. [CrossRef]
40. Barsalou, L.W. Perceptual symbol systems. *Behav. Brain Sci.* **1999**, *22*, 577–660. [CrossRef] [PubMed]
41. Barsalou, L.W.; Santos, A.; Simmons, W.K.; Wilson, C.D. Language and simulation in conceptual processing. In *Symbols and Embodiment: Debates on Meaning and Cognition*; Oxford University Press: Oxford, UK, 2008. [CrossRef]
42. Clark, A. Embodied, embedded, and extended cognition. In *The Cambridge Handbook of Cognitive Science*; Cambridge University Press: Cambridge, UK, 2012; p. 275.
43. Latour, B.; Woolgar, S. *Laboratory Life: The Social Construction of Scientific Facts*; Sage: Thousand Oaks, CA, USA, 1979.
44. Ricoeur, P. *La Métaphore Vive*; Seuil: Paris, France, 1975.
45. Vincent, G. Paul Ricoeur's "Living Metaphor". *Philos. Today* **1977**, *21*, 412. [CrossRef]
46. George, T. Hermeneutics. In *The Stanford Encyclopedia of Philosophy*, Winter 2021 ed.; Zalta, E.N., Ed.; Metaphysics Research Lab, Stanford University: Stanford, CA, USA, 2021.
47. McCarthy, J. A formal description of a subset of ALGOL. In *Formal Language Description Languages for Computer Programming*; North-Holland Publishing Company: Amsterdam, The Netherlands, 1966; pp. 1–12.
48. Burke, K. Four master tropes. *Kenyon Rev.* **1941**, *3*, 421–438.
49. Dijkstra, E.W. Hierarchical Ordering of Sequential Processes. *Acta Inform.* **1971**, *1*, 115–138. [CrossRef]
50. Wexelblat, R.L. (Ed.) *History of Programming Languages*; Academic Press: Cambridge, MA, USA, 1981.
51. Bergin, T.J.; Gibson, R.G. (Eds.) *History of Programming Languages—II*; ACM Press: New York, NY, USA, 1996.
52. Ryder, B.G.; Hailpern, B., Eds. *HOPL III: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*; Association for Computing Machinery: New York, NY, USA, 2007.
53. Steele, G. History of Programming Languages IV [Special Issue]. *Proc. ACM Program. Lang.* **2020**, *4*. Available online: https://dl.acm.org/do/10.1145/event-12215/abs/ (accessed on 30 July 2025).
54. Nofre, D. Unraveling Algol: US, Europe, and the Creation of a Programming Language. *IEEE Ann. Hist. Comput.* **2010**, *32*, 58–68. [CrossRef]
55. Nofre, D. The Politics of Early Programming Languages: IBM and the Algol Project. *Hist. Stud. Nat. Sci.* **2021**, *51*, 379–413. [CrossRef]
56. Astarte, T.K.; Jones, C.B. Formal Semantics of ALGOL 60: Four Descriptions in their Historical Context. In *Reflections on Programming Systems—Historical and Philosophical Aspects; Philosophical Studies Series*; De Mol, L., Primiero, G., Eds.; Springer: Cham, Switzerland, 2018, pp. 83–152. [CrossRef]
57. Alberts, G. Algol Culture and Programming Styles [Guest editor's introduction]. *IEEE Ann. Hist. Comput.* **2014**, *36*, 2–5. [CrossRef]
58. Peláez Valdez, M.E. A Gift from Pandora's Box: The Software Crisis. Ph.D. Thesis, University of Edinburgh: Edinburgh, UK, 1988.
59. Campbell-Kelly, M. Programming the EDSAC: Early programming activity at the University of Cambridge. *Ann. Hist. Comput.* **1980**, *2*, 7–36. [CrossRef]
60. Hopper, G.M. Keynote address at ACM SIGPLAN History of Programming Languages conference, June C1–3 1978. In *History of Programming Languages*; Academic Press: Cambridge, MA, USA, 1978.
61. Knuth, D.E.; Trabb Pardo, L. *The Early Development of Programming Languages*; Technical Report STAN-CS-76-562; Stanford University: Stanford, CA, USA, 1976.
62. Priestley, M. Synthetic machines and practical languages: masking the computer in the 1950s. In *Computing Cultures: Knowledges and Practices (1940–1990)*; Borrelli, A., Durnova, H., Eds.; Meson Press: Lüneburg, Germany, 2025.
63. De Mol, L.; Bullynck, M. What's in a name? Origins, transpositions and transformations of the triptych Algorithm—Code—Program. In *Abstractions and Embodiments: New Histories of Computing and Society*; Johns Hopkins University Press: Baltimore, MD, USA, 2022.
64. Eco, U. *The Search for the Perfect Language (Trans. James Fentress)*; John Wiley & Sons: Hoboken, NJ, USA, 1997.

65. Dick, S. Computer Science. In *A Companion to the History of American Science*; Montgomery, G.M., Largent, M.A., Eds.; John Wiley & Sons: Hoboken, NJ, USA, 2015; pp. 79–93.
66. Strachey, C. Towards a Formal Semantics. In *Formal Language Description Languages for Computer Programming*; North-Holland Publishing Company: Amsterdam, The Netherlands, 1966; pp. 198–200.
67. McCarthy, J. Towards a Mathematical Science of Computation. In *Program Verification*; Springer: Dordrecht, The Netherlands, 1962; Volume 62, pp. 21–28.
68. Astarte, T.K. Formalising Meaning: A History of Programming Language Semantics. Ph.D. Thesis, Newcastle University: Newcastle upon Tyne, UK, 2019.
69. Giacobazzi, R.; Ranzato, F. History of Abstract Interpretation. *IEEE Ann. Hist. Comput.* **2022**, *44*, 33–43. [CrossRef]
70. Lindsey, C.H. A History of ALGOL 68. In Proceedings of the Second ACM SIGPLAN Conference on History of Programming Languages, HOPL-II, Cambridge, MA, USA, 20–23 April 1993; pp. 97–132. [CrossRef]
71. Dijkstra, E.W.; Duncan, F.G.; Garwick, J.V.; Hoare, C.A.R.; Randell, B.; Seegmüller, G.; Turski, W.M.; Woodger, M. Minority Report. *ALGOL Bull.* **1970**, *31*, 7.
72. Landin, P.J. WG 2.1 Minority Report of Subcomittee on Needs of ALGOL 60 Users. Circulated to IFIP TC-2, 1969. Available online: https://ershov.iis.nsk.su/en/node/805980 (accessed on 30 July 2025).
73. Pérez, J.A. Report on CWI Lectures in Honor of Adriaan van Wijngaarden. *ACM SIGLOG News* **2017**, *4*, 40–41. [CrossRef]
74. Alberts, G. International Informatics—On the Occasion of Aad van Wijngaarden's 100th Birthday. *ERCIM News* **2016**, *106*, 6–7.
75. Dijkstra, E.W. The Humble Programmer. *Commun. ACM* **1972**, *15*, 859–866. [CrossRef]
76. Hicks, M. *Programmed Inequality: How Britain Discarded Women Technologists and Lost Its Edge in Computing*; MIT Press: Cambridge, MA, USA, 2017.
77. Belgraver Thissen, W.P.C.; Haffmans, W.J.; van den Heuvel, M.M.H.P.; Roeloffzen, M.J.M. Computing Girls. 2007. Available online: https://web.archive.org/web/20220120174546/http://www-set.win.tue.nl/UnsungHeroes/heroes/computing-girls.html (accessed on 20 January 2022).
78. Perlis, A.J. The American Side of the Development of ALGOL. In *History of Programming Languages*; Wexelblat, R.L., Ed.; Academic Press: Cambridge, MA, USA, 1981; Chapter 3, pp. 75–91.
79. Naur, P. The European side of the last phase of the development of ALGOL 60. In *History of Programming Languages*; Wexelblat, R.L., Ed.; Academic Press: Cambridge, MA, USA, 1981; Chapter 3, pp. 92–137.
80. Backus, J.W.; Bauer, F.L.; Green, J.; Katz, C.; McCarthy, J.; Naur, P.; Perlis, A.J.; Rutishauser, H.; Samelson, K.; Vauquois, B.; et al. Report on the algorithmic language ALGOL 60. *Numer. Math.* **1960**, *2*, 106–136. [CrossRef]
81. Backus, J.W.; Bauer, F.L.; Green, J.; Katz, C.; McCarthy, J.; Perlis, A.J.; Rutishauser, H.; Samelson, K.; Vauquois, B.; Wegstein, J.H.; et al. Revised Report on the Algorithm Language ALGOL 60. *Commun. ACM* **1963**, *6*, 1–17. [CrossRef]
82. Alberts, G.; Daylight, E.G. Universality versus Locality: The Amsterdam Style of ALGOL Implementation. *IEEE Ann. Hist. Comput.* **2014**, *36*, 52–63. [CrossRef]
83. van den Hove, G. On the Origin of Recursive Procedures. *Comput. J.* **2014**, *58*, 2892–2899. [CrossRef]
84. Zemanek, H. The Role of Professor A. van Wijngaarden in the History of IFIP. In *Algorithmic Languages: Proceedings of the International Symposium on Algorithmic Languages*; de Bakker, J.W., van Vliet, J.C., Eds.; North-Holland Publishing Company: Amsterdam, The Netherlands, 1981.
85. Pohle, J. From a Global Informatics Order to Informatics for Development: The rise and fall of the Intergovernmental Bureau for Informatics. In Proceedings of the IAMCR Conference 2013, Dublin, Ireland, 25–29 June 2013.
86. Knuth, D.E. Backus Normal Form vs. Backus Naur Form. *Commun. ACM* **1964**, *7*, 735–736. [CrossRef]
87. Lucas, P. On the formalization of programming languages: Early history and main approaches. In *The Vienna Development Method: The Meta-Language*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 1978; Volume 61, pp. 1–23.
88. Copeland, B.J. The Modern History of Computing. In *The Stanford Encyclopedia of Philosophy*, Winter 2020 ed.; Zalta, E.N., Ed.; Metaphysics Research Lab, Stanford University: Stanford, CA, USA, 2020.
89. Scott, D. *Outline of a Mathematical Theory of Computation*; Technical Report PRG-2; Oxford University Computing Laboratory, Programming Research Group: Oxford, UK, 1970.
90. De Mol, L. Towards a diversified understanding of computability or why we should care more about our histories. In Proceedings of the Computability in Europe '22, Swansea, UK, 11–15 July 2022.
91. Jones, C.B.; Astarte, T.K. Challenges for semantic description: comparing responses from the main approaches. In *Engineering Trustworthy Software Systems*; Lecture Notes in Computer Science; Bowen, J.P., Zhang, Z., Liu, Z., Eds.; Springer: Cham, Switzerland, 2018; Volume 11174, pp. 176–217. [CrossRef]
92. IFIP. Working Conference Vienna 1964 Formal Language Description Languages. Program. Oxford, Bodleian Libraries. MS. Eng. misc. b. 287/E.39, 1964.
93. Zemanek, H. Report on the Vienna conference, to IFIP TC2 Prague Meeting, 11 to 15 May 1964. Held in the Van Wijngaarden Collection, Universiteit van Amsterdam.

94. Reynolds, J.C. The discoveries of continuations. *Lisp Symb. Comput.* **1993**, *6*, 233–247. [CrossRef]
95. Mahoney, M.S. What Makes History? In *History of Programming Languages—II*; Bergin, T.J., Jr., Gibson, R.G., Jr., Eds.; ACM: New York, NY, USA, 1996; pp. 831–832. [CrossRef]
96. Wirth, N. Comments on "Recursive definition of syntax and semantics" by A. van Wijngaarden. *ALGOL Bull.* **1965**, 11–12.
97. Landin, P.J. A formal description of ALGOL 60. In *Formal Language Description Languages for Computer Programming*; North-Holland Publishing Company: Amsterdam, The Netherlands, 1966; pp. 266–294.
98. Landin, P.J. The mechanical evaluation of expressions. *Comput. J.* **1964**, *6*, 308–320. [CrossRef]
99. Lucas, P.; Alber, K.; Bandat, K.; Bekič, H.; Oliva, P.; Walk, K.; Zeisel, G. *Informal Introduction to the Abstract Syntax and Interpretation of PL/I*; Technical Report 25.083; ULD Version II; IBM Laboratory Vienna: Vienna, Austria, 1968.
100. Dijkstra, E.W. *Some Meditations on Advanced Programming*; Technical Report DR 30/62/R; Mathematisch Centrum: Amsterdam, The Netherlands, 1962.
101. Gorn, S. Summary Remarks (to a Working Conference on Mechanical Language Structures). *Commun. ACM* **1964**, *7*, 133–136. [CrossRef]
102. Scott, D.S. Looking Backward; Looking Forward. In Proceedings of the Domains13 Workshop at Federated Logic Conference, Oxford, UK, 6–19 July 2018.
103. Dijkstra, E.W. Letter to Hans Bekič. EWD 454, 1974. Available online: https://www.cs.utexas.edu/~EWD/ewd04xx/EWD454.PDF (accessed on 30 July 2025).
104. van Wijngaarden, A.; Mailloux, B.J.; Peck, J.E.L.; Koster, C.H.A. *Report on the Algorithmic Language ALGOL 68*; Second printing, MR 101; Mathematisch Centrum: Amsterdam, The Netherlands, 1969.
105. Nofre, D. A Compelling Image: The Tower of Babel and the Proliferation of Programming Languages During the 1960s. *IEEE Ann. Hist. Comput.* **2025**, *47*, 22–35. [CrossRef]
106. Friedman, M. On Blumenberg's Mathematical Caves, or: How Did Blumenberg Read Wittgenstein's Remarks on the Philosophy of Mathematics? In *Hans Blumenberg's History and Philosophy of Science*; Fragio, A., Ros Velasco, J., Philippi, M., Borck, C., Eds.; Springer Nature: Cham, Switzerland, 2024; pp. 29–51. [CrossRef]
107. Borghi, A.M. A Future of Words: Language and the Challenge of Abstract Concepts. *J. Cogn.* **2020**, *3*, 42. [CrossRef]
108. Margulieux, L.E. Spatial Encoding Strategy Theory: The Relationship between Spatial Skill and STEM Achievement. In Proceedings of the 2019 ACM Conference on International Computing Education Research, Toronto, ON, Canada, 12–14 August 2019; pp. 81–90.
109. Parkinson, J.; Margulieux, L. Improving CS Performance by Developing Spatial Skills. *Commun. ACM* **2025**, *68*, 68–75. [CrossRef]
110. Low, G. Metaphor and education. In *The Cambridge Handbook of Metaphor and Thought*; Cambridge University Press: Cambridge, UK, 2008; Chapter 12.
111. Eriksson, S.; Gericke, N.; Thörne, K. Analogy competence for science teachers. *Stud. Sci. Educ.* **2024**, 1–29. [CrossRef]
112. Sanford, J.P.; Tietz, A.; Farooq, S.; Guyer, S.; Shapiro, R.B. Metaphors we teach by. In Proceedings of the 45th ACM Technical Symposium on Computer Science Education, SIGCSE '14, Atlanta, GA, USA, 5–8 March 2014; pp. 585–590. [CrossRef]
113. Harper, C.; Mohammed, K.; Cooper, S. A Conceptual Metaphor Analysis of Recursion in a CS1 Course. In Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1, SIGCSETS 2025, Pittsburgh, PA, USA, 26 February–1 March 2025; pp. 457–463. [CrossRef]
114. Hewitt, C. Viewing control structures as patterns of passing messages. *Artif. Intell.* **1977**, *8*, 323–364. [CrossRef]
115. Kay, A.C. The Early History of Smalltalk. *SIGPLAN Not.* **1993**, *28*, 69–95. [CrossRef]
116. Simon, H.A.; Newell, A. Computer simulation of human thinking and problem solving. *Monogr. Soc. Res. Child Dev.* **1962**, *27*, 137–150. [CrossRef]
117. Amarel, S. On representations of problems of reasoning about actions. In *Machine Intelligence 3*; Michie, D., Ed.; Edinburgh University Press: Edinburgh, UK, 1968; Volume 3.
118. Eglash, R. Broken metaphor: The master-slave analogy in technical literature. *Technol. Cult.* **2007**, *48*, 360–369. [CrossRef]