Informatics in Eduaction, 2024, Vol. 23, No. 4, 837–868


DOI: https://doi.org/10.15388/infedu.2024.30

# The Two Powers

## How Pascal and Python Shaped Programming Education

Tobias KOHN[1], Jacqueline STAUB[2]

[1] *Karlsruhe Institute of Technology (KIT), Germany*
[2] *Trier University, Germany*
*e-mail: tobias.kohn@kit.edu, staub@uni-trier.de*

**Abstract.** The choice of programming language for education is an intensely debated topic. On the one hand, the programming language is supposed to be *relevant* in that its organisation, structures, and paradigms adhere to current standards and best practices in industry and academia. On the other hand, the programming language is expected to be *simple*, that is, easy to use and, of course, easy to learn. This increasingly means that we need an introduction to concepts shaped by a long history without all the historic baggage.

The flip side of these requirements of an educational programming language is its power to shape thought patterns, mental models, and best practices of the students, and thus of their future. Moreover, the aspect of simplicity may have the power to decide whether programming will become democraticed or remain the realm of a few highly paid specialists.

From the many programming languages that have been used in programming education, few stand out for being as popular and widely used as Pascal and Python. Decades apart, both have been designed with simplicity and relevance in mind, and yet it seems they could not differ more in so many design choices, most notably typing.

In this article we therefore consider the similarities and differences between Pascal and Python with emphasis on an educational perspective. We identify core features that might explain their suitability for education, compare their strengths and weaknesses, and discuss to what extent the premises behind their design still apply to the future of programming education.

**Key words:** Pascal, Python, programming.

### 1. Introduction

The ‘Big Bang’ of computing was followed by a rapid expansion: computers quickly became increasingly powerful and affordable. At the same time, programming became more language-oriented and started to adopt (linearised) mathematical notation. This development raised two important questions: what makes a ‘good’ programming language and how do we make programming accessible to an ever-growing group of computer users?

Even after the first programming languages were invented and widely used, the power of the computing machines meant that programs grew ever larger and more unwieldy, plaguing the newly forming computing industry. Initially a small group of computer scientiest including Edsger Dijkstra and Tony Hoare started to advocate what became

known as ‘structured programming’: a new way of devising programming languages and writing programs based on abstraction and mathematical rigour. The tinkering and craftsmanship of early programming was to turn into an academic discipline with scientific study and insights. In particular, programs themselves would become the subject of study.

A key aspiration of the community was not necessarily that the correctness of programs would become mathematically provable, but that abstraction and strict structural principles would bring order into chaos and make program code amenable to being fully understood and reasoned about. In other words, abstraction was seen as a tool to achieve urgently needed simplicity in spite of the fast growing complexity of these systems.

One of the very first programming languages that implemented those ideas of elegant simplicity and fully structured programming was *Pascal* by Niklaus Wirth. Pascal was clearly designed as a ‘teaching’ language, which meant much more than just being simple and approachable. Wirth repeatedly called out the role of universities to shape the future by daring to innovate and propagate the necessary change. He believed strongly that by building the notions of structured programming into a teaching language and thereby educating tomorrow’s programmers, the new school of thought would start to pervade the industry and herald the change needed to tackle the software crisis. Indeed, Pascal became a huge success in terms of its popularity and it might have contributed a great deal to the wide acceptance of structured programming as the way forward.

Pascal was, however, by far not the last word to be spoken on this matter. As much support as it enjoyed, it was also often criticised. (Even Wirth himself hoped that Pascal would make way some day to the next steps in evolution and not become a hindrance to progress itself (Wirth, 1975, 2002a).) Shortly after the publication of Pascal, a Dutch team set out to devise a ‘better’ programming language through an iterative design process that would last more than ten years. Essentially starting from Pascal and seeking to further simplify the language, the team eventually came up with *ABC*, a language that looked radically different and yet followed the same principles of structured programming as Pascal. Alas, ABC entered the scene when there was little interest or felt need for yet another teaching language; the overall tone was about to change and favour languages that were ‘relevant’ and actually used in industry.

One programmer who had worked on the implementation of ABC went on and incorporated many of ABC’s design ideas into a new scripting language for a different project. Guido van Rossum thus fueled the spirit of ABC with new life when he created his own language *Python*. Although it took a while for it to fully catch on, Python has by now not only become one of the most widely used programming languages in general, but also ranks among the most widely used educational languages, arguably being a worthy successor to Pascal.

Pascal and Python have had without any doubt a huge impact on programming education and thus the field of programming at large. They have, however, by no means been the only educational programming languages. BASIC was developed even earlier than Pascal, has enjoyed huge success as an ‘end-user’ programming language, but was often deemed inappropriate for serious teaching because of its lack of structure. Another example is Logo, a language that evolved out of LISP and was designed as a ‘learning’ language

rather than as a language for teaching. Like BASIC it lacks the 'structured' approach of Pascal and Python (particularly in terms of data types),[1] but with its radical simplicity and constructivistic approach, Logo may well have had one of the largest impacts on the field of computing education of all the languages and deserves some attention beyond a mere passing mention in this tale. However, Logo never made it out of the purely educational context and was thus often criticised as not 'relevant' enough with respect to teaching industry practices.[2]

With this article we aim to demonstrate how Niklaus Wirth has shaped programming education, honouring his great achievements and ideas. Moreover, by drawing a direct line from Pascal to Python via the programming language ABC, it should become clear Wirth's work also lives on in Python and is not limited to just 'his' programming languages. The exposition is deeply rooted in historical aspects, although we believe that many of the concepts and ideas are still relevant today and might provide an outstanding ground for a better understanding of programming and programming education in all its facets.

The discussion will start in earnest in Section 3 by highlighting the crucial role of 'abstraction' for programming. The then following two sections shall provide a more in-depth discussion of the design elements of both Pascal and Python (through ABC), before we contrast their designs with Logo as a natural complement in educational programming.

We are fully aware of the many omissions necessary by the restricted scope of such an article. A rigorous discussion of the programming language scene in the 1960s is not complete without also mentioning *Simula*, for instance. Among the educational programming languages we left out contributions as important as Scratch, Processing, Greenfoot, and many debates. However, we felt that our selection should provide the necessary background to understand Wirth's contribution and legacy to programming education—although we should definitely point to Wirth's later work in form of *Modula* and *Oberon* as successors to Pascal that have, alas, never really caught on.

## 2. Historic Background

With the rise of ever more powerful computing machines during the 1960s and early 1970s we saw a proliferation of new programming languages, some with a high and lasting impact on the field. The dominance of professional-oriented programming languages such as FORTRAN, COBOL, and ALGOL sparked the creation of new languages better suited for teaching and learning, among them Pascal, BASIC, and ABC.[3] In comparison to these languages, Python seems to be a fairly recent addition to the set of programming languages and is therefore rather an outlier in this section. Indeed, Python was developed quite exactly

[1] While LISP or Logo never had the problem of `Go To`s that Dijkstra referred to, there are other aspects such as macros and dynamic scoping that may be seen as 'problematic' with regards to structured programming.

[2] The power and popularity of the LISP-family of languages proves this perception wrong, of course.

[3] It was usual at the time to spell the names of programming language in capital letters (e.g., 'FORTRAN' instead of 'Fortran'). We keep that historic spelling for those languages that we provide for historical context here, but use the more modern forms for Pascal, Logo, etc., which are the focus of our discussion.

twenty years after Pascal. Its origins, however, reach back to the 1970s as well in form of the language ABC.

Given that most readers might not be familiar with concepts, ideas, and programming languages of that era, we provide a brief historic background of some of the programming languages that play a role in our story. By necessity, this account is incomplete and we refer the interested reader to literature such as, e.g., Guzdial and du Boulay (2019); Hoare (1980); Lorenzo (2017); Meertens (2022); Wirth (1996, 2008).

### 2.1. *FORTRAN, LISP, ALGOL, and COBOL*

In the 1950s and early 1960s there were four programming languages that stood out. *FORTRAN* was one of the first programming languages that abstracted from machine instructions. In order to convince programmers of the time of its usefulness, it was important to generate very efficient machine code—a feature that FORTRAN has kept until this day.

The 'European' programming language *ALGOL* was developed shortly after FORTRAN and was a much more ambitious project in terms of language design and features. ALGOL has greatly influenced subsequent programming languages, but implementing ALGOL-compilers was a non-trivial task and issues with the language led both to committees delevoping new versions as well as individuals devising their own 'simplified' languages—among them Pascal and CPL, the latter of which eventually leading to the development of C.

*LISP* differed radically from FORTRAN and ALGOL in that it did not try to provide infix notation, but relied instead of a unified concept of dynamic lists and functions operating on these lists. With its concentration on functions and 'high-level' program manipulation, LISP was a precursor to many functional languages, although it is itself not necessarily a functional language in the strict modern sense.

The three languages FORTRAN, ALGOL, and LISP all concentrated heavily on computers as machines for processing numerical calculations and were thus rather 'scientifically' oriented. In order to use computers efficiently for data processing such as business records, a different approach was needed, which led to the programming language *COBOL* with its focus on text and file processing.

### 2.2. *BASIC*

The *BASIC* programming language was developed in the early 1960s as an easy-to-use programming language for beginners (Lorenzo, 2017). The authors originally considered using a subset of either FORTRAN or ALGOL, but found it infeasible to cut out a meaningful subset or avoid some of the more obscure quirks of the languages. Design decisions included the use of a single numeric data type (floating point numbers), having one statement per line (thus eliminating the need for semicolons or other separation marks), beginning each statement with an easy-to-remember English keyword, and making it 'interactive,' i.e., eliminating the usual write-compile-link-run cycles and have immediate reaction from the machine instead (Biancuzzi and Warden, 2009; Lorenzo, 2017).

BASIC offers little in terms of data structures with its limitation to numbers, arrays, and text strings. In its original form it also lacked essential elements for structured programming such as a *while*-loop or proper subroutines with local variables. These were added only about twenty years later after much criticism in the discourse of structured programming and the success of Pascal (Lorenzo, 2017).

### 2.3. *Logo*

*Logo* is an educational programming language that originated in the late 1960s and was first implemented on a LISP system. From the very beginning, the language's design was specifically aimed at beginners. Both in terms of syntax and through its various 'mini-worlds,' Logo provides an engaging and condensed learning experience.

One of Logo's most famous 'mini-worlds' has been Turtle graphics, which was developed over several decades (Solomon *et al.*, 2020) and is still in use nowadays (Caspersen and Christensen, 2000; Staub, 2021). Logo offers several primitive data types and the linguistic elements for creating structured programs using sequence, selection, iteration, and procedures. However, `go to` is still presented as an alterative to `if else` (Feurzeig *et al.*, 1969).

### 2.4. *Pascal*

*Pascal* evolved from a proposed successor to ALGOL 60 and in large part as a necessity to have a programming language suitable for teaching novices at university around 1970 (Wirth, 1996). Two of the main pillars of its design were the implementation of *structured programming* as proposed by Dijkstra (1968, 1972) and of dyamic and structured data types as proposed by Hoare (1972).[4] Wirth has also attributed Pascal's popularity to the availability of text books and programming environments with rapid write-compile-run/test cycles.

### 2.5. *ABC*

Work on the *ABC* programming language began in 1975 at the mathematical centre Amsterdam in the Netherlands (Meertens, 2022). Early publications (Geurts and Meertens, 1976) already referred to both BASIC and Pascal as existing languages with often heavy criticism of BASIC for its lack of 'structured programming.'

During its long development and evolution, the language was initially known as $B_0$, $B_1$, etc., highlighting its deliberate iterative design process, until it finally adopted the name ABC for the stable version.[5] The originally intended target audience of ABC would

[4]Wirth mentioned his close collaboration with C. A. R. Hoare and E. Dijkstra in interviews (see, e.g., Lüthi (2021)), has cowritten his proposal for an extension to ALGOL with Hoare (Wirth and Hoare, 1966), and also refers to both Dijkstra and Hoare as sources for elements of Pascal (Wirth, 1996), even though the book by Hoare and Dijkstra on structured programming appeared two years after the design of Pascal had been finished.

[5]For reasons of simplicity and readability, we will commonly refer to the entire series of languages as 'ABC' and speak of the 'development of ABC' rather than $B_k$.

in today's terms be described as 'end-user programmers'; in particular *not* professional software-developers, but rather occasional programmers for which the simplicity and memorability of keywords plays an important role (Geurts and Meertens, 1976). Later publications primarily highlighted the goal of designing an educational language, though (Meertens, 2022; Pemberton, 1987).

### 2.6. *Python*

After having previously worked on the ABC project, Guido van Rossum developed *Python* as 'scripting language' for an experimental operating system (Biancuzzi and Warden, 2009; Meertens, 2022). While various design elements from ABC made its way into Python, there was a crucial difference: Python was meant to interface with various components, whereas ABC was intended as a closed system (Meertens, 2022).

Moreover, together with its simplicity, Python's ability to automate tasks over various software packages is arguably a main driver for its decisive success as one of the foremost and most popular programming languages today: in recent years, in particular, Python has become a de facto standard for interfacing with sophisticated machine learning libraries, completely abstracting away from the technical details of tasks such as running high-performance machine learning on GPUs.

In addition, Python has also found its way into computing education at large, although not necessarily for computer science majors due to its lack of data structuring.

## 3. Programming is Abstraction

What is programming and how does it relate to computers? In light of the well-known rally cry that "computer science is as much about computers as astronomy is about telescopes,"[6] programming as the activity of instructing machines (also known as 'coding') seems oddly out of place and time for modern computer science. Moreover, it begs the question of what role programming might have in general school education and hence also what an 'educational' programming language should provide.

In this section, we argue that programming goes far beyond 'coding' and is a combination of abstraction and reification: i.e., programming is the activity of abstracting a task manipulating concrete instances in terms of algorithms[7] and then encoding this abstracted task in terms of machine instructions. The 'machine' here, however, does not necessarily mean a concrete physical machine, but is a reference to the provided opportunities in terms of a set of available capabilities and instructions.

Moreover, taking the historic context as a frame of reference helps us understand some of the different approaches to programming. For instance, Tedre (2018) draws attention to

[6] This is often attributed to E. Dijkstra, but we could not find any reliable source for attribution.

[7] We are aware that 'algorithm' is a surprisingly problematic term here. The term 'computation' might be a better fit when dealing with programs (Knuth, 1997). However, we believe that 'algorithm' is a better fit for expressing the abstract and conceptual aspect and will therefore stick to it for reasons of readability.

the struggles of early computer scientists in establishing computer science as an academic discipline beyond tinkering with computing machines—which provides the necessary frame of reference for understanding the rally cry above. Yet, with the ubiquitous computing technology permeating society today, there is now little doubt that computer science is a highly relevant field of research and that computing machines make a worthwhile field of study within computer science. Hence, it is not that programming is out of place and time, but in fact the rally cry itself.

### 3.1. *The Nature of Programming*

Programming is a fundamental skill in computer science, up to the point where computer science tends to be reduced to programming in the public perception (although this has recently been challenged by the sudden rise of generative artificial intelligence in the public discourse, leading to remarks about programming being outdated and superfluous now). The *Computational Thinking* (CT) movement (Aho, 2012; Tedre and Denning, 2016; Wing, 2006) as well as the *CS Unplugged* approaches (Bell *et al.*, 1998, 2009; Gallenbacher, 2008) aim to address and rectify this perception by placing more emphasis on the underlying algorithmic ideas and their wide range of application in general education.

As the landscape of computing education further evolves, lines between approaches focusing on the algorithmic aspects versus those focusing on programming become increasingly blurred. Clearly, we should not consider programming as opposed to computational thinking or algorithmics, but rather understand these as complementary aspects of the field at large.

While computational thinking has started to place more emphasis on the algorithmic aspects, we have simultaneously witnessed a terminological shift from 'programming' to 'coding.' Historically, 'coding' referred to entering the instructions into the machine—originally a cumbersome and tedious process that has been supplanted by technology such as compilers—and still carries a strong machine-related connotation. The task of 'programming,' on the other hand, was originally much closer to drawing flowcharts and devising algorithms. As the technology and user interfaces evolved, 'coding' in the original sense became obsolete while programming took on more of a textual-linguistic flavour suitable for being entered into machines directly (see, e.g., Arawjo (2020) for a great account of these developments, but also Blackwell (2002b)). Remarkably, even the earliest programming languages like FORTRAN or LISP already strived for a high degree of machine-independence, thus emphasising the algorithmic aspect of programming over the coding of machine-specific instructions. Nonetheless, it seems programming has ever since been caught somewhere in the middle between the desire to express algorithms and the requirement to encode these algorithms in a machine-accessible format.

With the evolution of application software, programming skills in the original sense have become less important and computer proficiency no longer requires any knowledge of either the system or how to 'program' it. This shift was promptly reflected in education by moving away from teaching programming at schools to emphasising application-specific computing skills such as using 'office' software packages. The computational thinking

movement's emphasis that computing is more than 'just programming' is arguably best understood not as a direct critique of programming in education itself, but rather as a response to the dismissal of programming as a valuable skill while also addressing some of the valid criticisms.

Indeed, programming has not simply been supplanted by software applications. Rather, proficient and efficient use of computers often involves automating tasks and 'end-user programming' (Blackwell, 2002a,b). Hence, programming has not disappeared, but permeates modern professional computing. Programming is no longer the realm of professional software developers and system engineers, but an integral part of all sort of professionals who use it as an important tool.

This changing and diverse landscape of what is understood by 'programming' leaves us with the question of what it ultimately means and how it relates to the concepts of 'coding' and 'algorithms.' Blackwell (2002b) proposes to look at it from the user perspective and understand programming as a task of abstraction and automation with a notation suitable to express that abstraction. In particular, Blackwell stipulates that programming at its core "is the manipulation not of a concrete instance of the desired result, but an abstract notation defining behaviour in different circumstances" (Blackwell, 2002b). Note that this definition neither refers to algorithms nor machines, but builds on the involved cognitive activity required for the task. The defining characteristics of programming are thus that it provides a *notation* for manipulating the *abstraction* of a task.

According to Blackwell (2002a), programming offers itself as an alternative when a mundane task has to be done repeatedly so that developing an abstraction of the task pays off and offsets the initial cognitive investment of programming in the first place. We might derive from this that a good programming tool or environment is one that provides (a) a notation for dealing with abstraction of the tasks and the ensuing complexity and (b) a seamless transition from manipulation of concrete instances to abstractions of these instances.

Despite the strong emphasis on abstraction, however, programming is also rooted in a concrete machine and its set of predefined instructions available to the programmer for expressing the program. In other words, programming is not only an activity of abstracting a given task manipulating concrete instances, but also involves a concrete realisation of that abstraction on the basis of a machine, i.e., what we refer to as 'coding.' As alluded to before, the coding aspect might be transparent to a high degree and captured entirely by the notation and software tools such as compilers. Moreover, it would seem with the prevalence of 'high-level' programming languages that the underlying machine is already abstracted away. However, even such 'high-level' programming languages are based on an underlying conceptual machine that captures and expresses its capabilities and limitations: the *notional machine* (du Boulay *et al.*, 1981; du Boulay, 1986; Fincher *et al.*, 2020).

### 3.2. *Addressing Complexity*

During the 1960s computers had become powerful enough and programs accordingly large enough to cause a 'software development crisis': it was no longer possible to easily

understand the programs and many naïve approaches to software construction started to fail spectacularly (Hoare, 1980; Wirth, 2008). To address this issue, Dijkstra, Hoare, Wirth, and others sought to put programming onto firm mathematical foundations, prove the correctness of programs and introduce sophisticated means of structuring program code (Wirth, 1976a). In fact, Wirth (2008) suggested that mastering the complexity of computing machines can be done only through a single means: abstraction.

Abstraction is thus not only a way for automating tasks, but also a crucial tool for handling the complexity of machines. On the one hand, the notion of making the programming language independent of the underlying machine not only makes the programs 'portable,' but also allows the programmer to reason in terms of larger, abstract structures instead of having to deal with the details of implementation. In fact, this idea of layers of abstractions has become a corner stone of modern computer science where an application developer no longer needs to be concerned with caching strategies of CPUs, say, or how the operating system schedules threads. On the other hand, programming languages that build on abstract mathematical foundations are amenable to mathematical analysis themselves, such as proving the correctness of a program.

Blackwell (2002a) points out that abstraction requires that any action (on the machine) has a limited effect that can be understood by the programmer (called the 'frame problem'). A major benefit of abstracting from the actual underlying machine is that we can reason about programs in terms of structures and instructions with clearly defined actions and boundaries. While it may seem that the much simpler machine instructions are easily understood, too, the debate about `Go To` has shown that their combination may easily escape any actual understanding and render a program incomprehensible.

With the notion of writing programs in a 'machine-independent' and abstract manner, programs move closer to algorithms. Wirth (1976a) defines the relationship in that a program is an algorithm together with a concrete choice how to structure data. In his view, however, it is also still important that the data types and structures directly map to the underlying machine and are easily implementable; otherwise the programming language would not truthfully represent the underlying machine through its abstraction anymore. This has changed in ABC's and Python's approaches where the data structures have become entirely 'mathematical' and abstract.

Finally, it is noteworthy that Wirth (2008) drew attention to the fact that a programming "language represents an abstract computer whose objects and constructs lie closer to, and reflect more directly, the problem to be represented than the concrete machine." This links back to our previous discussion where abstraction is also closely related to solving problems and performing concrete tasks. Moreover, what Wirth refers to here has later become known as the 'notional machine' as mentioned above.

### 3.3. *Teaching Abstraction*

From an educational point of view, the emphasis on abstraction in programming is a problem.[8] We know from educational studies that learning needs to be rooted in direct

[8] In fact, we would even argue that this emphasis on abstraction is actually shared by computer science as a discipline and not only a feature of programming as such.

experience and concrete examples before it can move on to abstract concepts. For instance, conrete examples were found to be one of the most helpful learning aids to novices when learning programming (Lahtinen *et al.*, 2005); an effect well known in education and one that wears off as the student progresses in their curriculum and gain both experience and expertise.

Indeed, the learning process and path to a point where a student can proficiently deal with abstractions is much longer than was initially expected: a number of studies on novice programmers' capabilities revealed a surprisingly weak performance (Lister *et al.*, 2004; McCracken *et al.*, 2001). This led to the proposition of a 'Neo-Piagetian' framework for better describing the students' learning process in terms of 'stages' (Lister, 2011; Teague *et al.*, 2013; Teague and Lister, 2014). According to this theory and corresponding empirical findings, programming novices initially struggle to understand anything but isolated concrete instructions and must first learn to carefully trace a program with concrete values, before they will eventually start to develop more abstract notions and the ability to actually reason about program code.

This suggests that the transition from concrete instances to abstractions mentioned above may be even more important in an *educational* programming setting. There is, however, another aspect of abstraction that requires equal consideration: the (notional) machine. As quasi universal Turing machines, computers offer an overwhelming and seemingly limitless spectrum of possibilities. As pointed out before, 'high-level' programming languages move even further away from the mechanics of the underlying computer, resulting in an entirely abstract conceptual machine. Hence, in the spirit of addressing abstraction within an educational setting, we need to heed the issue of abstraction both in terms of the tasks to be performed as well as the machine on which the tasks are to be executed.

In light of the long time needed to properly learn programming with the abstractions, meaningful computing education can arguably not succeed within a single term or even year, but requires instruction over a number of years and grades—not unlike mathematics, say. Together with the necessity to slowly move from concrete examples to ever higher abstractions based on these examples, this strongly suggests that the right format for computing education is a spiral curriculum.

### 3.4. *Beyond Abstraction*

Without doubt, abstraction is a key concept in the field of programming. Programming involves abstracting from concrete tasks to achieve automation as well as abstracting from the machine to achieve program comprehension and (mathematical) reasoning about programs. Moreover, abstraction is also a powerful and important tool to deal with complexity.

On the flip side, programming equally involves reification and concretisations of abstract concepts. Programs are, according to Wirth, algorithms with a choice of concrete data types and structures. Moreover, programs are eventually meant to be run on a concrete machine and a good programming language is always designed with the implementation of the features in mind (Wirth, 1976a). Finally, instead of formally proving the correctness

of programs, modern software development is governed by concrete test cases.[9] It would therefore be wrong to reduce programming to abstraction only.

Particularly in an educational setting, yet another aspect of abstraction gains paramount importance: *simplicity*. Wirth (1974) highlights that abstraction is not only a key concept, but a necessary one by "admission of the limitation of our minds," i.e., abstraction is necessary to render programming simple enough so that we can actually deal with it. By choosing an abstraction of the complex computing machine where actions are strongly limited in their effects (both in terms of space as well as time), we may begin to reason about the overall effects of entire programs. At the same time, abstracting from the underlying machine necessarily hides part of the underlying mechanics, which might make it harder to understand the machine in the first place. The very programming language designed or chosen to simplify the task of programming might therefore become part of the 'black box'-problem (du Boulay *et al.*, 1981).

Simplicity is therefore not a single feature or dimension of measurement, but may be achieved through different, conflicting means. In fact, for starters, we may differentiate between *definitional simplicity* refering to systems with only a few (well-chosen) concepts provided and *conceptual simplicity* where a number of tasks are easily handled (Pemberton, 1987). However, having a small set of features is not sufficient for (definitional) simplicity, we must also consider how these features may interact and be combined to build up larger concepts and structures. Ideally, the basic features and concepts are 'orthogonal,' i.e., independent and freely combinable—a design philosophy that was picked up by Pascal, ABC, Python, and many other languages.

Another aspect with respect to abstraction is the *relevancy* of tasks and problems that can be (easily) solved with a programming language, but also the relevancy of a programming language in what might be called 'good practice.' For instance, Geurts and Meertens (1976), Pemberton (1987), and Wirth (2002b) all warn of 'bad habits' that the novice programmer might pick up without the right programming language or paradigm. Moreover, with the heavy emphasis of computing education geared towards professional usefulness (Guzdial, 2015), programming languages for education have been judged by whether they are used in industry or live up to current standards.

## 4. Finding a Notation

In order to be meaningful and amenable to manipulation, abstraction requires suitable representations or notations (Blackwell, 2002b; Hoare, 1972). The power of computing and mathematics lies in their abilities to manipulate abstract symbols to then infer properties about the 'real world' or to find a solution to the problem at hand. However, finding a notation that fully supports utilising this power is a non-trivial task (as seen by the centuries it took in mathematics). We therefore follow aspects of the design processes that

[9] Wirth (2008) seems a bit ambivalent in this regard as he argues for the need to formally prove the correctness of programs, but also points out that this seems unlikely to effectively succeed for any real system beyond some academic toy applications.

led to Pascal and Python, respectively, their similarities and differences. At its core, this is primarily a tale about structured programming, its origins, its implementations, and its limitations.

When comparing Pascal and Python with earlier design sketches and their predecessors ALGOL and ABC, there are indeed some striking similarities between these languages. Wirth and Hoare (1966), for instance, propose in their report about extending ALGOL the notation `S[i:j]` to indicate the subsequence of $S$ from $i$ to $j$, both for extraction and insertion (i.e., as an expression as well as an assignment target, respectively). Furthermore, Wirth and Hoare proposed to include complex numbers as a built-in datatype. Although neither of these two proposed features became part of Pascal, they have become part of Python, demonstrating that many ideas found in the much younger Python actually date back to the beginnings of Pascal.

### 4.1. *Structured Programming*

*Control flow* refers to the ability to not only execute the instructions of a program in a linear order, but allow for instructions to be skipped or repeated, thereby achieving conditional branching and loops, respectively. In assembly or machine code, control flow is expressed by 'jumping' to a specific instruction or address in the code. In terms of source code, this is expressed either by jumping to a predefined label or a specific line number. Early programming languages used the instruction `Go To` for this purpose.

Unfortunately, jumping via `Go To` is not only used for a variety of structural reasons (such as branching, looping, or error handling), which may already be hard to distinguish, but also invited a number of 'clever' tricks. A well-known modern example of such a trick is 'Duff's device' where a switch and a loop in C are cleverly interleaved to achieve partial loop unrolling. However, early high-level programming languages like FORTRAN did not yet provide while-loops, making the use of `Go To` therefore a necessity. In addition, subroutine calling and functions were not fully developed until the late 1960s. Many computers lacked proper support for stacks and programming languages often provided no support for recursion.[10] As a consequence, program code often ended up jumping all over the place in convoluted ways, which was dubbed 'Spaghetti Code.'

In a seminal article with the title 'Go To Considered Harmful,' Dijkstra (1968) highly criticised this culture of Spaghetti Code and called for what became known as *structured programming*.[11] Dijkstra suggested that a 'high-level' programming language should do without `Go To` entirely and consequently use constructs such as `if/else` or `while/repeat` instead. In his call for structured programming, Dijkstra did not advocate to minimise the number of structures and even considered that breaking loops might have a place, as long as understanding control flow was clear and straightforward. In the structural programming movement, this quickly evolved to demanding that each programming structure has unique

[10] The lack of recursion was actually a consequence of subroutines storing the return address in a variable and therefore not supporting reentry.

[11] Dijkstra mentions in his article that the issue with `Go To` had been brought up by Hoare before, but it is through Dijkstra's 'letter to the editor' that this issue seems to have gained wider traction.

entry and exit points (Dijkstra, 1972; Wirth, 1974), i.e., frowning on breaking loops (we will come back to this last point in Section 4.5).

Wirth (2002b) noted that "Pascal was designed explicitly to support this discipline [of Structured Programming]," although he retained the "famed go to statement" out of fear to antagonize influential ALGOL programmers. Moreover, Wirth (1996) also writes that "it was high time to not only preach the virtues of structured programming, but to make them applicable in actual practice by providing a language and compilers offering appropriate constructs." Together with Dijkstra and Hoare, Wirth was thus one of the pioneers of structured programming, and with Pascal he provided an actual implementation complementing the theoretical underpinnings. Following the rule of single entry and exit points, Pascal did not have a `return`-statement nor supported it leaving loops 'early' using `break`.

In addition to providing a working implementation of a language for structured programming, Wirth (1996) explicitly aimed to train future programmers in this methodology and thereby cause the necessary evolution in the software industry (in fact, he repeatedly returned to this notion that the universities have a responsibility to teach new approaches and methodologies to induce them into the industry (Wirth, 2002a, 2008)). Overall, the movement of structured programming must be regarded as a great success; there is hardly any modern 'high-level' language that still provides a `Go To` statement.

### 4.2. *Designing a Structured Programming Language*

The programming language Pascal was based on one of the very first designs following a structured programming approach, despite the elements that Wirth kept from its predecessors such as ALGOL. In a later assessment, Wirth (1975) further clarified that the aim of structured programming is being able to prove the correctness of a given program. Moreover, the static program code should reveal as much information about the dynamic process resulting from running the program as possible. However, Wirth was equally critical of introducing new features or deviating from established norms with regards to notation. For instance, even though he pointed out that parametrised types might solve some of the problems that plagued Pascal (see Section 5.4), he feared the far-reaching consequences that the introduction of such a feature might have.

A much less conservative route was taken by the team designing ABC (Geurts and Meertens, 1976). Their iterative design process was much more rooted in explorative studies and the expectation that features would change until eventually a stable version would be reached. The programming language was therefore called $B_0$ during the first iteration, $B_1$ during the second, and so on, until it resulted in ABC. Nonetheless, the entire design strictly followed structured programming as well, although in stark contrast to Pascal, even well-known and established programming constructs and features were to be examined and rejected or replaced if they were not found effective enough. Nonetheless, the first iteration $B_0$ included a number of features taken from other programming languages that were only dropped later in the design process. Also, like Wirth, Meertens (1981) cautioned against the ramifications that "seemingly innocent minor decisions" could have, advocating for a more careful design process.

As a measure of evaluation, Geurts and Meertens (1976) checked a proposed feature first with respect to whether it would aid in proving a program's correctness and second in terms of what kind of algorithmic concepts it would cater for. If the same algorithmic concepts could easily be implemented with existing features, the new feature would be dropped. If the algorithmic concept could be implemented more efficiently using an alternative or modified version of the feature, that alternative or modification, respectively, would be added to the language. Part of this design philosophy is still often cited in connection with Python these days as "there should be one—and preferably only one—obvious way to do it" (Peters, 2004).

One prominent example of the results of the this design process is the strong differentiation between a while- and a for-loop in ABC and Python. From the earliest design proposals on, for-loops in ABC would iterate over any given sequence of items, rather than increment and test a counter variable according to some arithmetic rule (which can easily be done with a while-loop) (Geurts and Meertens, 1976).

Even though the design process of ABC started merely a few years after Pascal's, it took over ten years before its first release in 1987. Python then evolved from ABC by loosening many of its strict design rules. For instance, ABC required each statement to start with a unique keyword (while no expression would start with a keyword) and therefore used `PUT 123 IN x` for assignment,[12] whereas Python used the more established syntax `x = 123`. Another example is Python's support for breaking out of loops (see Section 4.5 below).

Finally, it is noteworthy that both Wirth and the ABC team highlighted the requirement for simplicity in the language and differentiate between two kinds of simplicity. Wirth (1975) points to the conflict between the writer and the reader of a program in terms of simplicity, where the writer appreciates a rich set of features at their disposal, whereas the reader prefers a language with only few but clearly understandable features. Pemberton (1987) makes a very similar distinction between 'definitional' and 'conceptual' simplicity, where definitional simplicity refers to a language offering a small number of concepts while conceptual simplicity offers concepts more amenable to your needs. Geurts and Meertens (1976) also stress the importance of simplicity and elaborate that this includes a small number of programming constructs that are easy to learn, remember, and understand.

### 4.3. *Choosing Keywords*

Mainstream programming languages have converged on a number of 'keywords' that have become standard parlance in the field. Obvious examples for this are `if`, `else`, `while` or `for`. In other cases, such as definition of a function, procedure, or subroutine, say, we find a bewildering range of different keywords including `function`, `procedure`, `fun`, `def`, `to`, `how'to`, and others. The broad range of different keywords for similar concepts hints at the delicate consideration between factors such as readability, brevity, consistency

[12] From a pedagogical point of view it is interesting that ABC is so strongly based on the Box-metaphor for variables, which has been found to be somewhat problematic (Putnam *et al.*, 1986).

with existing conventions, and pedagogical considerations; at times these factors have conflicting priorities and result in trade-offs.

Geurts and Meertens (1976) point out that reserved (key)words come with the issue that they make a number of names unavailable for variables or procedures. When designing a beginner's language, this may be a particular issue because students might accidentally choose a name that is a reserved word, but corresponds to an advanced language feature that has not been discussed, yet. As a result ABC is designed so that each statement starts with a unique keyword and then uses further keywords in clearly identifiable positions, essentially alternating between keywords and expressions/parameters.[13] This allows the parser to distinguish between keywords and variable names, avoiding any possible interference and thus allowing the programmer to choose any name without restriction.

As a consequence, ABC followed BASIC in having a keyword for assignment, rather than using either '=' or '`:=`'. For "didactical reasons" the keyword pattern was chosen to be `PUT` ⟨`value`⟩ `IN` ⟨`var`⟩ with the idea of highlighting the 'algorithmic' nature of the variables and assignment in contrast to variables and equations in mathematics (Geurts and Meertens, 1976). While the confusion between 'mathematical' and 'algorithmic' variables is an issue, indeed (Kohn, 2017), so is the 'box-metaphor' that underlies the assignment statement here (Hermans *et al.*, 2018; Putnam *et al.*, 1986). It is therefore not entirely clear whether this choice is as pedagogically helpful as it was intended. However, one of the most common mistakes found in Python programs is based on the symmetry of the assignment operator (Sirkiä and Sorva, 2012), which is also something often brought up informally in discussions by teachers, sometimes with the desire to bring back Pascal's non-symmetric assignment operator. In this sense, ABC's design might be didactically helpful not necessarily because of the keyword, but because of its clear asymmetry.

Another interesting example with regards to keywords is the for-loop. While the exact syntax and semantics of a for-loop vary wildly between languages, we find the keyword itself to be quite uncontested and stable for decades. The name *for* seems to originate from Switzerland in the early 1950s (Rutishauser, 1952), where its meaning lay somewhere between that of a modern for-loop and that of an if- or case-conditional: *for* $k = 1$ *do this; for* $k = 2 \ldots n$ *do that*.[14] Despite the popularity of `for` as a keyword, Stefik and Siebert (2013); Stefik and Gellenbeck (2011) found that `for` ranked about worst of all the suggested keywords for 'intuitively' conveying the notion of a loop. We should therefore think that both Wirth and the ABC team missed an opportunity there. By now, of course, `for` is so much established in general programming language terminology that changing it is virtually impossible.

### 4.4. *Program Structure*

ABC initially followed ALGOL and Pascal in its use of `BEGIN` and `END` to delineate a block of code. However, it was quickly felt that the `BEGIN` was superfluous, anyway (cf.

[13] Expressions in ABC would never start with a keyword, though. This has survived in modern Python with the distinction between the *if-statement* and the *if-expression*.

[14] During our own teaching we found students sometimes confusing the meaning of `for` and using it as an if-statement.

BASIC, which only has `END`-statements), and that `END` was also "pure noise" once the block structure was outlined by indentation (Meertens, 1981). Moreover, `BEGIN` and `END` have more of a 'statement'-like character than a delimiting one (Geurts and Meertens, 1976). At this point, it might be worthwhile to point out that character sets and keyboards were not yet as uniform and standardised as today; curly braces, as adopted by BCPL (and later C), were simply not available on all systems at the time.

Wirth (1996) was similarly critical of the `BEGIN`–`END` blocks due to the 'dangling else'-problem. In the following code sequence it is not clear whether the `else` should bind to the first or second `if` and thus under what circumstances $D$ will be executed:

```
if A then if B then C else D;
```

As a remedy he suggested in hindsight that each control structure should have required a clear end marker by default (and thereby replacing the `BEGIN`–`END` block structure inherited from ALGOL).

Basing the program structure in ABC and Python on indentation rather than explicit markers has drawn some criticism, particularly from educators who feel that students are struggling with getting the indentation right.[15] The designers of ABC already pointed out that the indentation-based approach was possible because ABC had its own syntax-aware editor (Pemberton, 1987) (the idea of a syntax-aware editor has been picked up recently again for Java and later Python by Brown *et al.* (2016), who also argued that indentation-based structuring had no place in modern programming (Kölling *et al.*, 2017)). Unfortunately, there seems to be no empirical evidence that the indentation-style structuring is any better or worse than using markers. However, research in the 1980s found that indentation helps in program comprehension (Miara *et al.*, 1983), although this could not be confirmed by a much more recent study (Bauer *et al.*, 2019).

### 4.5. *Breaking Out of Loops*

In line with structured programming, neither Pascal nor ABC offered any means to leave a loop from the 'middle.' Wirth (1974) actually cautioned against such a construct and pointed out that it was merely a `Go To` in disguise (which, of course, is a very weak argument in itself). Geurts and Meertens (1976) also point out that 'escaping' from a while-loop violates the expectation that the while-loop's condition is no longer met after the loop.

Nonetheless, this strict adherence to structured programming with unique entry and exit points was challenged in experimental studies (Sheppard *et al.*, 1979; Soloway *et al.*, 1983). Sheppard *et al.* (1979) found that structured programming generally improved programmer's performance, but slight deviations from strict structured programming had no significant effect. However, Soloway *et al.* (1983) found that permitting to leave loops is actually highly beneficial for writing programs. Their 'averaging problem' has entered the

[15] In our teaching experience, students had at least as much problems getting the structure right using curly braces in Java.

**Program 1.** The 'rainfall problem' in two (Pascal) versions. On the left hand side in a strict structural programming style, on the right hand style using `break` to leave the loop.

```
value := input();
while value <> sentinel do
    begin
        data := data + value;
        count := count + 1;
        value := input();
    end;
```

```
while True do
    begin
        value := input();
        if value <> sentinel
            then break;
        data := data + value;
        count := count + 1;
    end;
```

computing education literature as the 'rainfall problem' and has been studied and applied numerous times over the years (Fisler, 2014; Seppälä *et al.*, 2015; Finnie-Ansley *et al.*, 2022). The rainfall problem asks that a program accepts numeric inputs one by one until a sentinel value is entered to end the data sequence. The program should then output the average of the values entered.

From a strict structural programming perspective, this requires that a first input is accepted before entering a while-loop. A less strictly structured approach, however, would permit an 'infinite' loop that is left in the 'middle' once the sentinel value has been detected (Program 1). Soloway *et al.* (1983) found that programmers wrote a correct solution significantly more often when allowed to leave ('break') from the loop. Moreover, they could refute the claims that strict structural programming improves readability and correctness of programs, thereby going further than Sheppard *et al.* (1979). Rather, the design of the programming language should also consider the cognitive load placed on programmers (also see Blackwell (2002a) on the importance of cognitive load in the context of programming).

An important rationale for structured programming is based on the belief that "a program that is easily proved correct is also easily understood" (Geurts and Meertens, 1976). However, the above findings suggest that this only holds up to a certain point and that, in the end, being able to express algorithmic ideas in a 'natural' and 'intuitive' way might be more important. In other words: simplicity is not merely a matter of 'sound design principles,' but perhaps better thought of as a balance and negotiation between the user and the problems to be solved by the tool.

### 4.6. *Designing for Education*

Pascal and ABC are both languages that were designed with education in mind. However, as subsequent empirical research has shown, both languages suffer from pedagogically problematic features. The example of escaping from loops shows that too rigorous and strict interpretation of a general principle may be problematic. The example of the `for`-keyword, on the other hand, also shows that programming languages contain historic 'baggage' that was kept without further examination in both Pascal and ABC—despite best efforts of the designers to eliminate problematic features from prior languages.

Neither the design process of ABC nor that of Pascal involved empirical studies about the learning process of novice programmers. Studies about the difficulties of learning to program emerged during the 1970s (Guzdial and du Boulay, 2019), but we could not find any indication that these findings had a significant impact on the design of ABC. Summarizing those difficulties, du Boulay (1986) points out that both common loop constructs (i.e., for- and while-loops) caused issues for learners: "Loops cause beginners all kinds of trouble." For-loops tend to update the loop variable "behind the scenes,"[16] thereby performing automatic hidden changes to the program's state. While-loops, on the other hand, suggest that the loop is left at the very instance the condition is no longer fulfilled.

In the end, little empirical research on learning has effectively found its way into programming language design. Exceptions include McIver and Conway (1996), who assembled a set of empirically motivated design principles and then created the programming language *GRAIL*, as well as Stefik and Ladner (2017), who designed the programming language *Quorum* based on empirical research and ensured that it is, among other things, particularly accessible to blind programmers. It seems, however, that none of these efforts has so far been met with high success, potentially owing to the perception of necessary 'industry relevance' of any language used in teaching.

## 5. Organising Data

Pascal was one of the very first programming languages that provided a unified, orthogonal, and rich type system. It not only included records for structuring data, but provided typed pointers (i.e., references) and even supported recursive data structures (a feature otherwise only known from linked lists in LISP).

ABC and Python followed suit with their emphasis on a powerful and versatile type system. However, Pascal as well as Python have often drawn criticism in particular for their choices with regards to typing, indicating that this is an important and contested topic. This section therefore provides an overview of how these type systems evolved.

### 5.1. *Designing Data Structures*

While the notion of structured programming began with control flow, it was quickly matched with contemplations about structured data organisation (Hoare, 1972). Early computers were considered pure computing devices and thus relied on numeric data types and arrays, vectors, or matrices. During the 1960s, text strings were established (early FORTRAN versions required to encode text as quasi-numeric 'Hollerith' constants), and towards the end of the decade, records/structs appeared in ALGOL and Pascal. Some few programming languages such as COBOL had a more 'business-oriented' approach and focused on text processing and data structuring rather than purely numeric computations, but were mostly ignored by the computer science community.

[16] A notable exception are for-loops in C, which are themselves problematic due to the non-linear control flow.

Whereas data types had primarily referred to the type of number (such as floating point, integer, or complex), data structuring sought to combine types to form new ones in a way inspired by mathematics; an idea that became known as *Algebraic Data Types* (ADT). Interestingly, while we would naïvely expect arrays to be interpreted as vectors and thus as a Cartesian product, Hoare (1972) suggested that arrays correspond to mathematical mappings. The Cartesian product, in turn, was realised through records.[17] Other proposed data types included sequences (such as text and bit strings) and sets. Finally, this new approach to data structuring also explicitly included *recursive* data structures, which indeed play a crucial role in defining trees, linked lists, and similar data structures.

It is hard to tell exactly where the idea for records came from. The extension proposed to ALGOL by Wirth and Hoare (1966) named Douglas T. Ross with the programming language *AED-I* as pioneer of this feature (who, in turn, refers to Hoare (Duncan, 1967)). Wirth (2002b) later also attributed it to COBOL. However, early drafts for PL/I not only described records (as structs) even slightly earlier (Radin and Rogoway, 1965), but also used the dot-notation for field access (which was also used for Pascal in contrast to the ALGOL proposal and ALGOL 68, which both used two different notations). It is clear, however, that Pascal was one of the absolute earliest programming languages that fully implemented structured data within a unified type system. Finally, given that records were an innovative feature in the late 1960s, it is hardly surprising that we find them neither in BASIC nor LISP or Logo, say, which clearly predate any of these possible sources.

### 5.2. *What is a Type*

Modern programming languages feature a sophisticated type system almost as a matter of course. We should be aware, though, that the nature and role of types vary considerably, depending on the abstraction level and point of view. Types…

1. provide a mapping from bit patterns to values (e.g., `1001010` might represent the number 74 as integer, the number 5.0 as 'minifloat,' the letter 'J,' etc.);
2. specify which operators can be applied to the value and how a specific operator is going to manipulate the value (the operator +, for instance, has completely different semantics depending on whether it is applied to numbers or strings, say);
3. may be seen as mathematical sets such as the integers $\mathbb{Z}$ or real numbers $\mathbb{R}$, although in programming these sets are all finite (or potentially infinite at most), e.g., signed 7-bit integers $\{-128, \ldots, 127\}$ or ASCII characters $\{A, B, C, \ldots, Z, a, b, \ldots, z, \ldots\}$.

Accordingly, a *variable* is then thought of as a 'cell' in memory with enough bits to hold and represent any value that a specific type could assume (i.e., a variable has a clear memory address and a size measured in bits), or it might be seen as a mapping from 'time' to values from a specific 'type' set: $var\colon \mathcal{T} \to \mathbb{U}$ where $\mathcal{T}$ refers to 'time' and $\mathbb{U}$ is a type (set), such as integers, floating point numbers, or strings.

This second understanding of variables as mappings from time to values lies at the core of the structured programming movement. In his letter against the `Go To` statement,

[17] These 'records' became 'structs' in ALGOL 68 and the C-family of languages.

Dijkstra (1968) argued that understanding a program strongly depends on being able to assign actual values to variables at different times (which is also backed up by pedagogical research (Lister *et al.*, 2004)). Dijkstra then concluded that a necessary prerequisite is the ability to infer the 'time' (a dynamic property of the running program) from a position in the source code (a static property). In other words: reading and understanding any given program crucially depends on whether we can figure out *when* a specific line or statement is being executed. Jumping around in code via `Go To` hampers or destroys exactly that ability, rendering it impossible to infer the values variables might have at specific points in the program code and thus reason about the program code's correctness.

Hoare (1972) also based his theory of structured data on this mathematical understanding of types and variables, looking for axiomatic ways of describing types; i.e., start with a few 'obvious' and simple basic types and build the rest by mathematical operations such as Cartesian products from there. This obviously captures arrays as Cartesian products $U \times U \times \cdots \times U$ and thus neatly integrates them into a unified type system, although Hoare and Wirth rather described them as mappings.[18] It is quite natural to then also include products of different types $U_1 \times U_2 \times \cdots \times U_k$, which leads to records (structs).

Pascal's type system was not universally appreciated, though, but received some criticism. In particular, Habermann (1973) felt that data structures and data types are two different concepts that should not be mixed. Although his criticism of Pascal received a strong rebuttal pointing out various straw men arguments and mistakes (Lecarme and Desjardins, 1974), one of Habermann's arguments stands out from a modern perspective: he reasoned that types are atomic entities with an associated set of operations, whose internal structure should neither be visible nor matter for the program. This notion has become a corner stone of *Object-Oriented Programming*—which may be understood as a dialectical combination of data structuring according to Hoare and Wirth with atomic types and information hiding.

Another line of criticism came from Kernighan (1981), who argued that Pascal lacked the necessary capabilities for building large software system or interacting with other parts of the computing environment. Three of the main issues raised by Kernighan are the fixed size of arrays (cf. Section 5.4), the lack of a means to leave a loop (cf. Section 4.5), and the strict type system (i.e., the lack of type casting). Unrelated to Kernighan's article, Wirth (2008) in turn criticised C for not adhering to a type system with strict type checking. According to Wirth, the advantages of abstract (structured) programs break down if these abstractions are mixed with 'lower level' concepts. From Wirth's perspective, C's mix of 'high-level' abstractions and 'low-level' system access did not follow the "spirit of structured programming."

*Polymorphism.* What has plagued type systems much more than the Cartesian product is the notion of type unions $U \cup V$, which can be translated to something like 'either an integer or a floating point number.' The problem arises from the ambiguity of how to

[18] In mathematics, the set of all functions $\mathbb{N} \to \mathbb{R}$, say, can also be thought of as a Cartesian product $\mathbb{R} \times \mathbb{R} \times \cdots$, written as $\mathbb{R}^{\mathbb{N}}$. Interpreting arrays as mappings $U \to V$ is therefore in line with mathematical practice and a natural extension of the Cartesian product.

correctly interpret the bit pattern representing values. In Pascal, type union has found its expression in 'variant records,' which can be tagged to indicate which of the possible types is currently/actually the correct one. Unfortunately, the compiler can no longer check or ensure the consistency of types, even with tags. Wirth (1996) has repeatedly stated his frustration about this fact and saw variant records as a major regret in Pascal's design.

It took some time for the programming concepts to catch up with the notion of type unions, which eventually led to polymorphism, both in functional as well as object-oriented languages. In contrast to Pascal's variant records, where different type interpretations of the data are all concurrently accessible through fields, the trick is to make data access conditional-based on the tag. Functional programming languages have tended to pair polymorphism with pattern matching whereas object-oriented programming binds operations directly to the objects to ensure type-safety.

Interestingly, this is exactly one of the main aspects where Python differs from Pascal. Python fully embraces the object-oriented design with a strict pairing of objects and operators.[19]

*Types in ABC and Python.* Like Pascal, ABC has based its type system on a strongly mathematical understanding, and yet came to entirely different conclusions, based on ideas taken from the programming language *SETL* (Meertens, 1981, 2022; Schwartz, 1975). While the first version of ABC's type system was clearly influenced by Pascal (Geurts and Meertens, 1976) and required variable declarations, later versions relied on type inference and structured data using 'tuples,' 'multi-sets,' and 'tables' (Meertens, 1981). Tuples were seen as a replacement for records as Cartesian products (without named fields, though), and tables generalised arrays as maps along the ideas of Hoare (1972). In today's understanding, tables correspond most closely to 'hash tables' or 'dictionaries.' Multi-sets are sets that allow an item to be included multiple times and were represented as automatically sorted lists. The notion of lists that are automatically sorted was later heavily criticised from an educational perspective as 'confusing' (McIver, 2000).[20]

Early iterations of ABC distinguished between integers and floating point numbers, although the final version of ABC only provided a unified data type 'number' with unlimited size and exact semantics whenever possible. Similarly, a single text string type was used with no specific character type. In the original report, the Boolean type was supposed to be implemented by the user as a 'range' (called an 'enumeration' in Pascal), but seems to have been dropped completely later on.

### 5.3. *From Arrays to Lists*

One of the contentious questions regarding sequential data structures in programming is whether to start at zero or one for indexing the first element. While zero-based indexing

[19] In this aspect, Python adheres even more to the idea that data is represented by objects than Java, which still has 'primitive' types although Java appeared a few years after Python.

[20] We agree with McIver that this feature of multi-sets or 'automatically sorted lists' is highly unusual and very badly named, making it pedagogically rather problematic.

seems to be widely spread in modern usage, languages like, e.g., BASIC use one-based indexing instead. ALGOL went a different route by declaring arrays with both a start- and an end-index, such as `a[1:10]`, say. In their enhancement proposal, Wirth and Hoare (1966) did not only retain this declaration of arrays including the specific range over indices, but also suggested that the same syntax might be used to denote a sub-sequence of an array, although rather as a possible further language extension in the future. The subsequent development and implementation of *ALGOL W* at Stanford (Bauer *et al.*, 1968) based on Wirth's and Hoare's proposal slightly changed the syntax to `a(1::10)`, i.e., using a double colon and round parentheses.

Pascal seems to have turned the colon by 90° so that array indices were now declared using two dots: `a[1..10]` (Wirth, 1971b).[21] Moreover, the expression `1..10` by itself denoted a scalar data type and could be used outside of array declarations. Likewise, instead of explicitly stating a range of possible indices, it was possible to use any scalar type, such as, e.g., `a[char]` to denote an array with character indices. While Wirth and Hoare (1966) still describe an array as an "ordered set of variables," arrays were "regarded as a mapping of the index type onto the component type" in Pascal a few years later (Wirth, 1971b). Hoare (1972) similarly then wrote that "an array may be regarded as a mapping between a domain of one type (the subscript range) and a range of some possibly different type (the type of the array, or more accurately, the type of its elements)."

The authors of ABC (Pemberton, 1987) took this notion of the array as a mapping even further and replaced arrays by 'tables,' which could map from any set of values to any other set of values—the values in either set do not even have to necessarily be of one type (it would therefore be wrong to say that tables can map from any type to any other). What was retained from Pascal was the syntax for ranges `1..10`, although now interpreted as values rather than types.

Python deviates from the minimalistic stance of ABC and uses two different data types: lists as zero-indexed sequences of values and dictionaries as a direct surrogate for the tables—although the two data types still share a lot of commonality in Python. Additionally, Python has also abolished the range-syntax in favour of returning to the colon for specifying subranges (e.g., `a[1:10]`), in line with the original proposal by Wirth and Hoare (1966). There is, however, another more subtle change in Python: the end-index is no longer included in the range: `a[1:10]` denotes all elements at indices starting from 1 and up to 9, but excluding the element at index 10.

### 5.4. *Issues with Arrays*

Interpreting arrays as mappings in the way Pascal implemented it causes some issues in application. With the ranges `0..3` and `0..5` constituting two different types, we find that the arrays `array[0..3] of T` and `array[0..5] of T` constitute two different (incompatible) types, too. Unfortunately, there is no type hierarchy in Pascal such that these two arrays would be subtypes of a more general `array of T`. As a consequence,

[21] We do not know, however, whether Pascal syntax really came from turning the colon or whether this is just a superficial coincidence.

any procedure or function could only accept array arguments of a very specific type and thus length. It is therefore quite impossible to write a general 'sort'- or 'search'-function, say:[22] the type system constrained each parameter to be of a fixed length.

Wirth (1975, 1976b, 1996) acknowledged the lack of 'dynamic arrays' for parameter passing as a major drawback and issue in Pascal's design. Even though parametrised types were already known and understood at the time of implementation, Wirth had decided against it, afraid that using parametrised types would lead to unforeseen ramifications and 'blow up' the language in direct contradiction to its design objective of simplicity. Later Pascal implementations, however, followed a different path and solved the problem through subtyping rather than type parametrisation.

It is interesting to contrast this overly restrictive array parameter declaration with 'procedure' parameters (i.e., passing a procedure or function as an argument to another procedure). Inheriting directly from ALGOL 60, procedure parameters in Pascal did not specify either the number or types of the arguments to the procedure parameter. In other words, all procedures were just of type `procedure` instead of, e.g., `procedure(Int, Int)`. Wirth (1996) also acknowledged this issue and together with `Go To` saw these as mistakes owed to a desire to stay compatible with previous programming language conventions. In hindsight, we might say with respect to parameters that the generalisation happened in the wrong place.

The notion of regarding arrays as mappings $\mathbb{V}^{\mathbb{U}}$ rather than simple Cartesian products $\mathbb{V}^n$ has one more ramification. We actually lose the intrinsic order of elements. In a Cartesian product (essentially a tuple or vector), it makes perfect sense to talk about the first and the second element. However, in a mapping this no longer holds. At best the domain of the mapping (that is the index type) induces an ordering (Pascal therefore required that the index type for arrays be an ordered type such as integers or characters). ABC went one step further than Pascal and replaced the classic arrays entirely by tables as mappings from any type to any other. It is quite clear that elements in such tables are not ordered and do not have an intrinsic sequence (with the now common implementations as hash tables, we find that items may end up in 'scrambled' order even if all the keys are consecutive integers).

So, conceptually, it is not really clear what it is meant by 'sorting a function.' Hence, by giving up the original sequentiality of arrays with their intrinsic order, Pascal and to a larger degree ABC have moved away from the basics of data processing to a degree where canonical tasks start to become meaningless.

Guido van Rossum's decision to reintroduce sequential data in the form of lists (essentially corresponding to dynamic arrays) should therefore be seen very favourably from an educational point of view, as they are a better representations of sequential data than tables.

[22] It was still possible to write more general functions by using pointers, of course, but this meant effectively to circumvent arrays.

## 6. Radical Simplicity in a Concrete World

The development of programming languages has brought with it various strategies for facilitating abstraction, such as programming paradigms, data structures, and type systems. In view of this development, the question arises to what extent these developments harmonise with the goal of *simplicity*. To complement the discussion we had so far, we take a closer look at Logo, a programming language that stands for exemplary simplicity, dates back to the era of Pascal, has left its trace in Python, and is still in active use.

Perhaps most strikingly and in stark contrast to Pascal and ABC, Logo was not developed as a 'teaching' language, but rather as a 'learning' language. So what is the difference? Pascal was specifically designed to teach programming at university, in particular the ideas of structured programming. Wirth aimed to educate the next generation of programmers. Logo, on the other hand, never had any ambition to teach principles of software design, but followed the constructionist ideas of Papert with the emphasis on the children's learning. Logo is therefore much more amenable to individual exploration. Moreover, the objective of Logo was to teach primarily mathematics through the means of programming a computer, rather than focusing on programming itself.[23]

### 6.1. *Logo's Roots in LISP*

Logo emerged in the late 1960s as a programming language that was specifically designed for children. The first implementation of Logo ran on a LISP system, a circumstance that can be attributed to its creators working in the field of artificial intelligence (researchers working in AI commonly used LISP at that time). Later implementations on different systems followed and kept a similar notation to the original 'Lisp Logo' (Solomon *et al.*, 2020).

LISP programs consist of expressions enclosed in parentheses, where each expression consists of a function or operator followed by a number of arguments. Using a prefix notation and parentheses, LISP has a notation that allows a direct mapping between a given program code and its corresponding parse tree (e.g., $((1-2)+3)*4$ is written as `(* (+ (- 1 2) 3) 4))` in LISP).[24] The language moreover facilitates the representation of both code and data within the same data structure, namely a list. With this syntax, LISP is easily extensible and highly expressive.

While expressiveness is a desired feature to have, there are some syntactic attributes that make LISP vulnerable to mistakes, especially when used by novices. Prefix notation, for instance, is uncommon for most children and thus likely to result in mistakes. Similarly, LISP relies on having all parentheses balanced and at the 'correct' positions (getting

[23] All this is *not* to say that Logo would be unsuited for teaching programming, but rather that its original design concentrated on other principles and goals than Pascal and Python, say, and therefore approached the idea of an educational programming language from an entirely different perspective.

[24] The prefix notation goes back to Polish logician Jan Łukasiewicz and is therefore also known as the 'polish notation.' The original strength of this 'polish' notation was that it did not require parentheses because of the unambiguous pairing of operators and operands (functions and arguments). Ironically, LISP, which has taken up this notation, however, is infamous for its excessive use of parentheses, which it owes in part to its variadic functions (i.e., the number of arguments can vary for almost any function).

**Program 2.** The same code once in Logo and once in LISP. The program defines and invokes a procedure that takes one argument `x` and returns the square of `x`. Upon invocation with value 5, both programs print the result 25 to the console. It is remarkable how close the Logo version is to ALGOL-based languages, despite its LISP-origins.

**Logo**

```
1 to square :x
2   output :x * :x
3 end
4
5 print square 5
```

**LISP**

```
1 (defun square (x)
2   (* x x))
3
4 (print (square 5))
```

the parentheses wrong either results in a syntax error or in a different calculation than intended). While designing Logo, Feuerzeig and his team thus moved from pre- to infix notation for common mathematical operations and allowed programs to not be fully parenthesised (Feurzeig *et al.*, 1969). Simultaneously, they decided to keep some features, such as LISP's read-eval-print-loop, its case-insensitive syntax, automatic memory management, as well as lists as a data structure to contain both code and data (Solomon *et al.*, 2020).

From an outside perspective, the syntactic changes from LISP to Logo seem rather large (see Program 2), and yet, if we are able to see past these superficial syntactic differences, there are fundamental similarities in how these two languages understand computation and also in their role as a programming languages. And yet, Logo is syntactically significantly closer to the ALGOL-family of languages such as Pascal and Python.

### 6.2. *The Language 'Logo'*

Logo's notation is dedicated to the principle of simplicity—its syntax is minimised (e.g., whitespace as a delimiter for both statements and arguments), thereby bypassing numerous difficulties that learners may encounter in other programming languages. For instance, du Boulay (1986) noted that the semicolon in Pascal caused a significant problem for learners. Later studies on syntax errors in Java also found that missing or misplaced semicolons and parentheses ranked among the most common errors (Becker, 2016; Brown *et al.*, 2014; Jackson *et al.*, 2005).

In comparison with Pascal and Python, two design choices stand out and warrant a brief description. For a more thorough discussion of Logo's design, see the excellent exposition by Solomon *et al.* (2020).

*The choice of assignment operator.* Pascal, Python, and Logo each handle variable assignment differently. Python uses the classical syntax of a single '=' sign, which (as discussed in Section 4.3) is linked to several misconceptions in novice programmers. Both the deceptive symmetry (`x=y` vs. `y=x`) and the proximity to the mathematical equality operator (`x=x+1` as an apparently unsolvable mathematical equation) are shown to be connected with prevalent misconceptions among novice programmers (Ma *et al.*, 2007;

**Program 3.** The underlying implementation of Logo's looping construct. No special syntax is required to express the concept of a loop.

```
1 to repeat :num :commands
2   if (:num=0) [stop] [eval :commands repeat :num-1 :commands]
3 end
```

Putnam *et al.*, 1986; Sirkiä and Sorva, 2012). Logo, on the other hand, introduces its own dedicated syntax: a special command `make :x 100` to assign a value, 100, to a variable, `:x` (this notation resembles the variable assignments in BASIC and ABC, although the order of argument is swapped). Pascal elegantly solves the problem of undesired symmetry using its assignment operator '`:=`', known for definitional assignment from mathematics.

*The choice of data structures.* Python and Pascal both offer a wide range of data types and structures. In contrast, Logo is limited to a single composed data structure: the list. This may seem as a major drawback for Logo, but in fact just emphasises Logo's focus on simplicity—Logo's lists serve a powerful bidirectional purpose: once for storing data (e.g., in `print [1 2 3 4 5]` the list is used as an array of numbers), and once as a container for code (e.g., in `repeat 4 [fd 100 rt 90]` the list is simply passed to the `repeat` command as a second argument. Program 3 shows the underlying implementation of the `repeat` command and thus highlights how iteration is a concept that can easily be reimplemented by the programmer using the concepts of selection, recursion, and code evaluation. Note how the `eval` command is used to parse and interpret the code provided within the list).

That is, while many other programming languages need to introduce additional syntax for their looping construct (and thereby create more syntactic blackboxes for the programmers), Logo's `repeat` is both simple, customizable, and expressive.

### 6.3. *The Turtle and Notional Machines*

Logo was designed with the idea of offering several 'mini-worlds,' that is, concrete and tangible application domains for the students to work in. Logo's language-based 'mini-world' with manipulation of words and sentences actually gave the language its name.

However, one of Logo's most famous mini-worlds is without doubt *Turtle graphics*, featuring a programmable agent (the 'Turtle') that reacts to movement commands issued by the programmer and draws lines by tracing its movement. The concept offers didactic benefits (Craver Jr. *et al.*, 1985) and was adopted by various other programming languages and systems. For instance, Python includes Turtle graphics as one of its standard modules whereas in other cases such as, e.g., Java, there exist a number of different implementations.[25] Turtle graphics has thus enjoyed a wide range of applications and huge

[25] Perhaps one of the most remarkable implementations is the 'cheloniidae'-package by Tipping (2009), which offers 3D Turtle graphics and describes itself as "absurdly overengineered Turtle graphics for Java," demonstrating the potential of the Turtle metaphor.

success in the field of programming education (Abelson and DiSessa, 1986; Caspersen and Christensen, 2000). Overall, the Turtle not only offers an approachable "object to think with" (Papert, 1980), but its main strength arguably lies in its capability of providing a notional machine.

*Notional machines* (du Boulay *et al.*, 1981; Fincher *et al.*, 2020) refer to the abstract conceptual machine underlying a specific programming language. The notional machine captures the capabilities and structures offered by a programming language; it is a conceptual machine that uses exactly the same abstractions and metaphors as provided by the programming language itself. Understanding the notional machine and its workings is therefore an important step in effectively using the programming language.

It is obviously quite impossible for a novice to understand the complete notional machine as implied by a 'whole' programming language, so that in practice, notional machines usually cover specific aspects or parts of a language. Still, the issue remains of how to make these notional machines observable and tangible, i.e., how to expose the mechanics of the running program to the novice programmer. This is were the Turtle as a notional machine comes in. The Turtle replaces the much more abstract, versatile, and powerful computer by a machine that is directly observable and whose capabilities and mechanics are immediately comprehensible (Kohn and Komm, 2018). The same idea of using a concrete 'actor' as carrier of the computational model and thus a notional machine for introductory programming has become quite pervasive: for instance, in Greenfoot, the role of the observable actor is played by a wombat and in Scratch it is a cat.

With its Turtle graphics, Logo has done something revolutionary for the field of computing education. Rather than focusing all the attention on the programming language, its syntax, semantics, and structure, Logo replaced the underlying machine, removing all unnecessary abstraction, and brought the machine on the same level with the tasks to be solved. In stark contrast to the notion of 'relevancy' in terms of industry practices, Logo's Turtle makes programming *relevant to the student* by empowering the student to fully comprehend the machine and devise new algorithms to solve problems.

### 6.4. *Programming Philosophies*

Despite Pascal, ABC, and Logo all aiming at novice programmers, there are some interesting differences in their fundamental understanding of their role as programming languages.

*Top-down vs. bottom-up.* Modern information technology (comprising enormous architectures such as the world wide web or large operating systems) is said to be the biggest construct humanity has built so far (Northwood, 2018). In order to continue from this point, programmers need to be able to decompose their programs into smaller building blocks as well as make use of the building blocks others have provided for them. Modular program decomposition can be taught from the start (Hromkovič *et al.*, 2016) and two common ways of developing a program for a given problem are bottom-up (i.e., constructing and testing smaller programs which then are composed to form the next layer of abstraction)

and top-down (i.e., starting from the high-level goal which is subsequently decomposed into smaller and smaller sub-modules).

While both Logo and Pascal allow modular program decomposition, however, they promote different approaches of doing so. Logo often takes a bottom-up approach (Clements and Meredith, 1993; Hromkovic *et al.*, 2017) whereas Wirth promotes top-down programming for Pascal (Wirth, 1971a). The debate whether it is better to teach novices via a top-down or bottom-up approach is complex and context-dependent. It is a question that may not have a definitive answer (Saito and Yamaura, 2014; Solomon *et al.*, 2020).

*Learning vs. teaching.* Logo was designed as a programming language to *learn* with rather than to *teach* with (this becomes evident once we compare the 651 references of the word "learn" to the 163 references of the word "teach" in *Mindstorms* (Papert, 1980)). In contrast, both ABC and Pascal were created to simplify teaching (Wirth, 1971b; Pemberton, 1987).

Ideally, teaching and learning should go hand in hand. However, if learning is promoted as a self-standing activity, it may imply a mistrust in how schooling works and show the author's desire for change. Sometimes, as in Logo, this desire results in educational ideas that stand the test of time.

## 7. Conclusion

In the history of computing, the importance of programming has changed its nature substantially. Whilst the initial desire to ease work through automation has remained unchanged, the way in which this is achieved has adapted over time. In the early stages of programming, tinkering and clever tricks resulted in workable solutions that more often than not lacked structure and mathematical rigour. In the middle of the last century, a shift to more structured methods has taken place, which also resulted in programs becoming more comprehensible with uniform principles underneath. With this, the start towards an independent academic discipline for the studies of programs and their notations was given.

With both industry and academia now involved, the question arises as to what role the two parties should play in the development of new programming languages. Should universities be primarily concerned with preparing the next generation of developers for the demands of industry? Or should they rather play a role in the development of novel notations and techniques to be used in industry? With the development of Pascal, Wirth has clearly taken a stand for the second path. And yet, he did so not primarily out of a conviction to create a language for industry, but rather to have a didactic tool for his teaching.

Python, on the other hand, has taken the opposite path: developed to fulfil a purely practical need, Python now has a firm place in both industry and academia (even programming education in general). However, Python certainly has had its influences from the educational sector, most directly through the educational programming language ABC, but also indirectly through Pascal and even other languages such as Logo.

Finally, we would like to address a debate that is more heated in computer science than in many other areas of science: to what extent does Pascal still add value to modern

programming education despite its age? To answer this question, we first acknowledge that the fast pace we are experiencing naturally includes the tools we use to express our ideas. The tree of programming languages and dialects is growing rapidly as we discover new applications, improve the machines we run our programs on, or as we find more elegant or error-resistant notations. But the fewest of all languages begin from scratch; new languages build on top of previous developments that have proven useful. The seeds Wirth has planted decades ago therefore still last on and serve both us and future generations of programmers with useful concepts and notations. The answer to the above question is as easy as this: 'Good ideas never age.'

## References


Abelson, H., DiSessa, A. (1986). *Turtle Geometry: The Computer as a Medium for Exploring Mathematics*. MIT Press.

Aho, A.V. (2012). Computation and Computational Thinking. *The Computer Journal*, 55(7), 832–835.

Arawjo, I. (2020). To Write Code: The Cultural Fabrication of Programming Notation and Practice. In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (CHI 2020)*, pp. 1–15.

Bauer, H.R., Becker, S.I., Graham, S.L., Forsythe, G.E., Satterthwaite, E.H. (1968). Algol W (revised). Stanford University, Department of Computer Science.

Bauer, J., Siegmund, J., Peitek, N., Hofmeister, J.C., Apel, S. (2019). Indentation: Simply a Matter of Style or Support for Program Comprehension? In: *Proceedings of the 27th IEEE/ACM International Conference on Program Comprehension (ICPC 2019)*. IEEE Computer Society, pp. 154–164.

Becker, B.A. (2016). An Effective Approach to Enhancing Compiler Error Messages. In: *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE 2016)*. Association for Computing Machinery, New York, NY, USA, pp. 126–131.

Bell, T.C., Witten, I.H., Fellows, M. (1998). Computer Science Unplugged: Off-line Activities and Games for All Ages. Computer Science Unplugged.

Bell, T., Alexander, J., Freeman, I., Grimley, M. (2009). Computer Science Unplugged: School Students Doing Real Computing Without Computers. *New Zealand Journal of Applied Computing and Information Technology*, 13(1), 20–29.

Biancuzzi, F., Warden, S. (2009). *Masterminds of Programming: Conversations with the Creators of Major Programming Languages*. O'Reilly Media, Inc..

Blackwell, A.F. (2002a). First Steps in Programming: A Rationale for Attention Investment Models. In: *Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments*. IEEE Computer Society, pp. 2–10.

Blackwell, A.F. (2002b). What is Programming? In: *PPIG* (Vol. 14), pp. 204–218.

Brown, N.C., Altadmri, A., Kölling, M. (2016). Frame-Based Editing: Combining the Best of Blocks and Text Programming. In: *Proceedings of the 2016 International Conference on Learning and Teaching in Computing and Engineering (LaTICE)*. IEEE Computer Society, pp. 47–53.

Brown, N.C.C., Kölling, M., McCall, D., Utting, I. (2014). Blackbox: A Large Scale Repository of Novice Programmers' Activity. In: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE 2014)*. Association for Computing Machinery, New York, NY, USA, pp. 223–228.

Caspersen, M.E., Christensen, H.B. (2000). Here, There and Everywhere – On the Recurring Use of Turtle Graphics in CS 1. In: *ACM International Conference Proceeding Series* (Vol. 8). Association for Computing Machinery, New York, NY, USA, pp. 34–40.

Clements, D.H., Meredith, J.S. (1993). Research on Logo: Effects and Efficacy. *Journal of Computing in Childhood Education*, 4(4), 263–290.

Craver Jr., W.L., Schroder, D.C., Tarquin, A.J., Hu, P.-W. (1985). LOGO as an Introduction to Fortran Programming. *Journal of Professional Issues in Engineering*, 111(3), 119–123.

Dijkstra, E.W. (1968). Letters to the Editor: Go To Statement Considered Harmful. *Communications of the ACM*, 11(3), 147–148.

Dijkstra, E.W. (1972). *Notes on Structured Programming*. Academic Press Ltd., GBR, pp. 1–82.
du Boulay, B. (1986). Some Difficulties of Learning to Program. *Journal of Educational Computing Research*, 2(1), 57–73.
du Boulay, B., O'Shea, T., Monk, J. (1981). The Black Box Inside the Glass Box: Presenting Computing Concepts to Novices. *International Journal of Man-Machine Studies*, 14(3), 237–249. https://doi.org/10.1016/S0020-7373(81)80056-9.
Duncan, F.G. (1967). ALGOL Bulletin no. 26. *SIGPLAN Notices*, 2(11), 1–49. https://doi.org/10.1145/1139498.1139500.
Feurzeig, W., Papert, S., Bloom, M., Grant, R., Solomon, C. (1969). Programming-Languages as a Conceptual Framework for Teaching Mathematics. Final Report on the First Fifteen Months of the LOGO Project.
Fincher, S., Jeuring, J., Miller, C.S., Donaldson, P., du Boulay, B., Hauswirth, M., Hellas, A., Hermans, F., Lewis, C., Mühling, A., Pearce, J.L., Petersen, A. (2020). Notional Machines in Computing Education: The Education of Attention. In: *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education (ITiCSE-WGR 2020)*. Association for Computing Machinery, New York, NY, USA, pp. 21–50. https://doi.org/10.1145/3437800.3439202.
Finnie-Ansley, J., Denny, P., Becker, B.A., Luxton-Reilly, A., Prather, J. (2022). The Robots are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming. In: *Proceedings of the 24th Australasian Computing Education Conference (ACE 2022)*, pp. 10–19.
Fisler, K. (2014). The Recurring Rainfall Problem. In: *Proceedings of the 10th Annual Conference on International Computing Education Research (ICER 2014)*, pp. 35–42.
Gallenbacher, J. (2008). *Abenteuer Informatik*. Spektrum.
Geurts, L.J.M., Meertens, L.G.L.T. (1976). Designing a Beginners' Programming Language. *IRIA. Rocquencourt*.
Guzdial, M. (2015). *Learner-Centered Design of Computing Education: Research on Computing for Everyone*. Morgan & Claypool Publishers.
Guzdial, M., du Boulay, B. (2019). The History of Computing. *The Cambridge Handbook of Computing Education Research*, *11*.
Habermann, A.N. (1973). Critical Comments on the Programming Language Pascal. *Acta Informatica*, 3, 47–57.
Hermans, F., Swidan, A., Aivaloglou, E., Smit, M. (2018). Thinking Out of the Box: Comparing Metaphors for Variables in Programming Education. In: *Proceedings of the 13th Workshop in Primary and Secondary Computing Education (WiPSCE 2018)*, pp. 1–8.
Hoare, C.A.R. (1972). Notes on Data Structuring. In: *Structured Programming*, pp. 83–174.
Hoare, C.A.R. (1980). The Emperor's Old Clothes. In: *ACM Turing Award Lectures*. Association for Computing Machinery, New York, NY, USA, pp. 75–83.
Hromkovič, J., Kohn, T., Komm, D., Serafini, G. (2016). Examples of Algorithmic Thinking in Programming Education. *Olympiads in Informatics*, 10(1-2), 111–124.
Hromkovic, J., Kohn, T., Komm, D., Serafini, G., et al. (2017). Algorithmic Thinking from the Start. *Bulletin of EATCS*, *1*(121).
Jackson, J., Cobb, M., Carver, C. (2005). Identifying Top Java Errors for Novice Programmers. In: *Proceedings of the 35th Annual Conference of Frontiers in Education*. IEEE Computer Society, pp. 4–4.
Kernighan, B.W. (1981). Why Pascal is Not My Favourite Programming Language.
Knuth, D.E. (1997). *The Art of Computer Programming*. Addison-Wesley Professional.
Kohn, T. (2017). Variable Evaluation: An Exploration of Novice Programmers' Understanding and Common Misconceptions. In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2017)*. Association for Computing Machinery, New York, NY, USA, pp. 345–350.
Kohn, T., Komm, D. (2018). Teaching Programming and Algorithmic Complexity with Tangible Machines. In: *Proceedings of the 11th International Conference on Informatics in Schools (ISSEP 2018)*, pp. 68–83. Springer.
Kölling, M., Brown, N.C.C., Altadmri, A. (2017). Frame-Based Editing. *Journal of Visual Language and Sentient Systems*, 3(1), 1.
Lahtinen, E., Ala-Mutka, K., Järvinen, H.-M. (2005). A Study of the Difficulties of Novice Programmers. *ACM SIGCSE Bulletin*, 37(3), 14–18.
Lecarme, O., Desjardins, P. (1974). Reply to a Paper by AN Habermann on the Programming Language Pascal. *ACM SIGPLAN Notices*, 9(10), 21–27.
Lister, R. (2011). Concrete and Other Neo-Piagetian Forms of Reasoning in the Novice Programmer. In: *Conferences in Research and Practice in Information Technology Series*.

Lister, R., Adams, E.S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J.E., Sanders, K., Seppälä, O., Simon, B., Thomas, L. (2004). A Multi-National Study of Reading and Tracing Skills in Novice Programmers. In: *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE-WGR 2004)*. Association for Computing Machinery, New York, NY, USA, pp. 119–150. https://doi.org/10.1145/1044550.1041673.

Lorenzo, M.J. (2017). *Endless Loop: The History of the BASIC Programming Language (Beginner's All-Purpose Symbolic Instruction Code)*. CreateSpace Independent Publishing Platform.

Lüthi, P. (2021). "I Always Saw Myself as an Engineer". ETH Zurich. https://inf.ethz.ch/news-and-events/spotlights/infk-news-channel/2021/11/niklaus-wirth-video-interview.html.

Ma, L., Ferguson, J., Roper, M., Wood, M. (2007). Investigating the Viability of Mental Models Held by Novice Programmers. In: *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2007)*, pp. 499–503.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y.B.-D., Laxer, C., Thomas, L., Utting, I., Wilusz, T. (2001). A Multi-National, Multi-Institutional Study of Assessment of Programming Skills of First-Year CS Students. *SIGCSE Bull.*, 33(4), 125–180. https://doi.org/10.1145/572139.572181.

McIver, L. (2000). The Effect of Programming Language on Error Rates of Novice Programmers. In: *PPIG*, p. 15. Citeseer.

McIver, L., Conway, D. (1996). Seven Deadly Sins of Introductory Programming Language Design. In: *Proceedings of the 1996 International Conference on Software Engineering: Education and Practice*. IEEE Computer Society, pp. 309–316.

Meertens, L. (2022). The Origins of Python. https://inference-review.com/article/the-origins-of-python.

Meertens, L.G.L.T. (1981). Issues in the Design of a Beginners' Programming Language.

Miara, R.J., Musselman, J.A., Navarro, J.A., Shneiderman, B. (1983). Program Indentation and Comprehensibility. *Communications of the ACM*, 26(11), 861–867.

Northwood, C. (2018). The Modern Web. *The Full Stack Developer: Your Essential Guide to the Everyday Skills Expected of a Modern Full Stack Web Developer*, 1–9.

Papert, S. (1980). *Mindstorms – Children, Computers, and Powerful Ideas*. Basic Books.

Pemberton, S. (1987). An Alternative Simple Language and Environment for PCs. *IEEE Software*, 4(1), 56.

Peters, T. (2004). PEP 20: Zen of Python. https://peps.python.org/pep-0020/.

Putnam, R.T., Sleeman, D., Baxter, J.A., Kuspa, L.K. (1986). A Summary of Misconceptions of High School Basic Programmers. *Journal of Educational Computing Research*, 2(4), 459–472.

Radin, G., Rogoway, H.P. (1965). NPL: Highlights of a New Programming Language. *Communications of the ACM*, 8(1), 9–17. https://doi.org/10.1145/363707.363708.

Rutishauser, H. (1952). Automatische Rechenplanfertigung bei programmgesteuerten Rechenmaschinen. *Journal of Applied Mathematics and Physics (ZAMP)*, 3, 312–313. https://doi.org/10.1007/BF02009622.

Saito, D., Yamaura, T. (2014). Applying the Top-Down Approach to Beginners in Programming Language Education. In: *Proceedings of the 2014 International Conference on Interactive Collaborative Learning (ICL)*. IEEE Computer Society, pp. 311–318.

Schwartz, J.T. (1975). Automatic Data Structure Choice in a Language of Very High Level. *Communications of the ACM*, 18(12), 722–728.

Seppälä, O., Ihantola, P., Isohanni, E., Sorva, J., Vihavainen, A. (2015). Do We Know How Difficult the Rainfall Problem is? In: *Proceedings of the 15th Koli Calling Conference on Computing Education Research (Koli Calling 2015)*, pp. 87–96.

Sheppard, Curtis, Milliman, Love (1979). Modern Coding Practices and Programmer Performance. *Computer*, 12(12), 41–49. https://doi.org/10.1109/MC.1979.1658575.

Sirkiä, T., Sorva, J. (2012). Exploring Programming Misconceptions: An Analysis of Student Mistakes in Visual Program Simulation Exercises. In: *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*, pp. 19–28.

Solomon, C., Harvey, B., Kahn, K., Lieberman, H., Miller, M.L., Minsky, M., Papert, A., Silverman, B. (2020). History of LOGO. *Proceedings of the ACM on Programming Languages*, 4(HOPL), 1–66.

Soloway, E., Bonar, J., Ehrlich, K. (1983). Cognitive Strategies and Looping Constructs: An Empirical Study. *Communications of the ACM*, 26(11), 853–860.

Staub, J. (2021). Logo Environments in the Focus of Time. *Bulletin of EATCS*, *1*(133).

Stefik, A., Gellenbeck, E. (2011). Empirical Studies on Programming Language Stimuli. *Software Quality Journal*, 19, 65–99.

Stefik, A., Ladner, R. (2017). The Quorum Programming Language. In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2017)*. Association for Computing Machinery, New York, NY, USA, pp. 641–641.

Stefik, A., Siebert, S. (2013). An Empirical Investigation Into Programming Language Syntax. *ACM Transactions on Computing Education (TOCE)*, 13(4), 1–40.

Teague, D., Lister, R. (2014). Manifestations of Preoperational Reasoning on Similar Programming Tasks. In: *Proceedings of the 16th Australasian Computing Education Conference (ACE 2014)*, pp. 65–74. Australian Computer Society.

Teague, D., Corney, M., Ahadi, A., Lister, R. (2013). A Qualitative Think Aloud Study of the Early Neo-Piagetian Stages of Reasoning in Novice Programmers. In: *Proceedings of the 15th Australasian Computing Education Conference*, pp. 87–95. Australian Computer Society.

Tedre, M. (2018). The Nature of Computing as a Discipline. *Computer Science Education: Perspectives on Teaching and Learning in School*, *2*.

Tedre, M., Denning, P.J. (2016). The Long Quest for Computational Thinking. In: *Proceedings of the 16th Koli Calling International Conference on Computing Education Research (Koli Calling 2016)*. Association for Computing Machinery, New York, NY, USA, pp. 120–129. https://doi.org/10.1145/2999541.2999542.

Tipping, S. (2009). Cheloniidae. https://spencertipping.com/cheloniidae/.

Wing, J.M. (2006). Computational Thinking. *Communications of the ACM*, 49(3), 33–35. https://doi.org/10.1145/1118178.1118215.

Wirth, N. (1971a). Program Development by Stepwise Refinement. *Communications of the ACM*, 14(4), 221–227.

Wirth, N. (1971b). The Programming Language Pascal. *Acta Informatica*, 1, 35–63.

Wirth, N. (1974). On the Composition of Well-Structured Programs. *ACM Computing Surveys (CSUR)*, 6(4), 247–259.

Wirth, N. (1975). An Assessment of the Programming Language Pascal. In: *Proceedings of the International Conference on Reliable Software*, pp. 23–30.

Wirth, N. (1976a). *Algorithms + Data Structures = Programs* (Vol. 158). Prentice-Hall Englewood Cliffs, NJ.

Wirth, N. (1976b). Comment on a Note on Dynamic Arrays in Pascal. *ACM SIGPLAN Notices*, 11(1), 37–38.

Wirth, N. (1996). Recollections about the Development of Pascal. In: *History of Programming Languages—II*, pp. 97–120.

Wirth, N. (2002a). Computing Science Education: The Road Not Taken. *ACM SIGCSE Bulletin*, 34(3), 1–3.

Wirth, N. (2002b). *Pascal and its Successors*. Springer-Verlag, Berlin, Heidelberg, pp. 108–119.

Wirth, N. (2008). A Brief History of Software Engineering. *IEEE Annals of the History of Computing*, 30(3), 32–39.

Wirth, N., Hoare, C.A.R. (1966). A Contribution to the Development of ALGOL. *Communications of the ACM*, 9(6), 413–432.