2 Mathematisation of Programming

Teacher: Although I want to focus on mathematical thinking about programming, it may be useful to start from the broader theme of professionalisation of programming. When did programming start turning into a respectable discipline?

Pythagoras: I have serious doubts whether programming is a respectable discipline even today, given that we are still unable to produce correct programs. But the turning point from which we can even think about making programming respectable is the publication of the preliminary report on the Algol programming language in 1958.

Diogenes: The proliferation of programming languages at the end of the 1950s was a gradual progression. We were slowly learning how to program better. What was so wrong with programming in the 1950s that suddenly changed with Algol?

Pythagoras: In the 1950s, programming lacked the sound body of knowledge that could support it as an intellectually respectable discipline.[1] Treating programs as mathematical entities, which was the intellectual achievement of Algol, made it possible to start building such knowledge in the form of fundamental algorithms...[2]

Archimedes: I'm not sure if algorithms are the right kind of fundamental knowledge, but I would agree that the problem with programming in the 1950s was that it was unsystematic. There were no accepted development methodologies, standard tools or processes and so programming was highly unreliable.

Xenophon: I agree about the unreliability of programming back then, except that it was not due to a lack of algorithms or development methodologies, but because of a personnel issue. The industry was growing too rapidly and there was a shortage of qualified programmers.[3]

Socrates: It is ironic that the attempts to 'professionalise programming' at the time of a labour crisis can also be linked to the exclusion of women from the workforce. Many of the professionalisation efforts tried to turn the perception of programming from low-skilled clerical work, associated with women, to high-skilled masculine work. What programmers actually did has not changed much. The perception did.

Teacher: I do not want to brush aside an issue that continues affecting programming to this day,[4] but let's get back to programming languages for a minute. Would you agree that the birth of programming languages was a notable milestone in the professionalisation of programming?

Xenophon: The interesting shift is that programming languages became entities separate from specific machines. This solved the portability problem that many businesses were facing in the 1950s. At least in principle, you could take your programs in a given programming language, compile them and run them on a new, more powerful, machine without having to start coding from scratch.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

Diogenes: You all keep talking as if programming languages were some sudden new invention at the end of the 1950s, but there were many systems that preceded the better-known programming languages. There was the A-0 system by Grace Hopper and Autocode by Alick Glennie, both created in 1952. Already at the end of the 1940s, the Electronic Delay Storage Automatic Calculator (EDSAC) programmers used an 'external form' of instructions and interpretive routines, which are early tricks resembling later programming language ideas.[5]

Pythagoras: You can trace the history of programming languages even further back. In the 1930s, lambda calculus, Turing machines and Post production systems captured the essence of later functional, imperative and concurrent programming languages. The definition of Algol also went hand-in-hand with mathematical work on formal language theory by Chomsky and others.[6]

Socrates: This is a nice story computer scientists like to tell themselves. The actual historical influence of logic is doubtful,[7] but you are right about formal languages.

Archimedes: I think programming languages illustrate a point that we made in Chapter 1. It is a single concept that we all share that brings together ideas coming from different backgrounds. It combines the theory of formal languages that *Pythagoras* talked about, the implementation techniques from earlier systems mentioned by *Diogenes* and the concrete business motivation that *Xenophon* found important.

Teacher: That is a nice observation. But do we also have a shared understanding of what programming languages are?

Socrates: I like to link programming languages to human thinking. In the 1950s, an early perspective was to see programming languages and their compilers as translators from a language that a human understands to a language that a machine understands.[8]

Xenophon: This is not a very practical perspective. The first programming systems in the 1950s were also called 'automatic coding' systems and they tried to make programming easier so that it does not have to be done by skilled programmers. It makes much more sense to see programming languages as one of the approaches to tackling the labour crisis of the 1950s.[9]

Pythagoras: Sure, programming languages involve translation and, sure, they make programming practically easier. But what makes them interesting is that they enable a new mathematical thinking about programming. They make it possible to treat programs as mathematical entities and formally analyse them using the devices of formal logic.

Teacher: Let's focus on this idea of treating programs as mathematical entities for a moment. What exactly does this mean?

Pythagoras: If you treat computer programs as syntactic structures in a formal language defined by the rules of logic, then you can infer properties of programs from their texts by purely deductive methods. In other words, you can write a program and then prove that it is correct.

Diogenes: But how do you make sure that your compiler does what your mathematical model says it does? You make it sound very easy to specify what a program means, but real programming languages and compilers are enormously complicated. After all, one of the criticisms of Algol was that it was very hard to implement properly.

Pythagoras: This is something that people soon started working on. Since the 1960s, there have been several efforts to define Algol formally, so that you could prove that a compiler correctly implements the specification and, consequently, that programs have no errors. It was a visionary idea at the time, but the CompCert project,[10] created in the early 2000s, does exactly this for the C programming language and has been used commercially in mission-critical systems.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

Diogenes: So, it took just 60 years to do this and only for a single programming language?

Pythagoras: The programming practice had to evolve in order to make such projects possible. We had to figure out how to write programs in a more compositional way, so that a correct program can be composed from individual correct components.

Teacher: Can we relate the ideas on better ways of organising programs to our more general discussion of professionalisation of programming?

Archimedes: Certainly! The motivation for structured programming in the 1960s was to help programmers write high-quality code by using logical structures with meaning, such as loops for control-flow and records and collections for representing data.

Diogenes: Structured programming is useful, but it is going a bit too far if you remove `GOTO` and other low-level features from a programming language, making it hard to express complex logic efficiently.

Pythagoras: I do not understand why you would want to have `GOTO` in a programming language. The Böhm–Jacopini theorem from 1966[11] clearly proved that any program containing `GOTO` can be transformed to an equivalent structured program...

Xenophon: Structured programming was valuable for a different reason. It made it possible to manage large development efforts in a top-down way. It inspired a new managerial approach to programming where a skilled programmer draws up the overall design and divides it into modules that are then assigned to junior programmers. This led to a successful development methodology adopted by IBM and others in the 1970s.[12]

Archimedes: The whole point of structured programming is to help individual programmers be more confident in the correctness of their programs. Treating structured programming as a managerial practice that turns programmers into unskilled workers goes directly against this idea.

Teacher: If you excuse me, I would like to get back to the issue of mathematical proofs of computer programs. It seems that the basic ideas necessary for this way of thinking have been around at least since the 1960s, yet proving programs correct is something that programmers rarely do today. Do you have any insights into why this is the case?

Pythagoras: This is just a matter of poor economic decisions. Proving programs correct increases the initial costs, so people do not do it. They foolishly ignore the fact that eliminating program errors would save them money in the long run...

Archimedes: I'm not sure if proofs about programs are really the same kind of thing as ordinary mathematical proofs. Unlike in mathematics, formal specifications of programs are often very long and proofs rarely reveal any deeper insights. Maybe people do not use them because they are just tedious...

Socrates: This is the objection raised by De Millo, Lipton and Perlis in 1977.[13] They pointed out that correctness of mathematical proofs is ensured by the social processes that surround them, such as mathematicians excitedly discussing new elegant proofs at the blackboard during a seminar. Proofs about computer programs are too long and boring to support similar social processes and so they cannot be trusted.

Archimedes: This is an interesting problem, but I would not throw proofs of programs out the window yet. Perhaps there is some other way of building trust in program proofs.

Xenophon: You have to realise that you are not trying to produce correct programs for some intellectual enjoyment. If a company decides it is economically worth it, it can ask programmers to write and carefully review the programs. You can set a small target error rate and adjust your development methodology to guarantee that.[14]

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

Pythagoras: I do not understand why you would accept 'small error rate' when the purpose of mathematical proofs is to give you an absolute certainty.

Archimedes: I thought we agreed that program proofs are not like mathematical proofs and cannot be trusted to the same degree.

Pythagoras: If you have a proof where each step is derived from the assumptions and the previous steps using an inference rule, then it is a proof. It does not matter who is talking about it and how! In fact, you can even automate proof checking and write a computer program that will check that each step of a formal proof is correct. That way, there can be absolutely no doubt about its correctness.

Socrates: Interestingly, De Millo et al. talk about automatic verifiers and they fear what they call the Titanic effect. A failure is still possible, but because of our uncritical trust in the system, it would have catastrophic consequences.

Pythagoras: Even if you have reservations towards full program verification, you should not underestimate the usefulness of small, elegant mathematical proofs about programs. If you study simple, formal models of programming languages, you may not be able to prove that the actual implementation is correct, but you will often find where the tricky corner cases are, so that you know where you need to pay special attention during the implementation.

Teacher: What I find most curious about our discussion is that mathematical knowledge appears as something that does not admit multiple interpretations, yet there are so many different ways of relating to it in the context of programming. It seems that this plurality of perspectives has sometimes been beneficial but it has also led to clashes once we started questioning the foundations of programming.

2.1 Giant Electronic Brains

On February 15, 1946, the recently built electronic general-purpose computer, ENIAC, completed a number of calculations during a public demonstration where it was unveiled in front of some 100 prominent guests. One of the example programs was a calculation of the trajectory of a shell, which it completed in 20 seconds. The same task would take female computers about 40 hours to complete using a mechanical desktop calculator, which is how the military made those calculations before.

As the ENIAC computed the trajectory, the attendees could literally see the numbers being processed on a front panel of the machine. The panel showed the contents of ENIAC accumulators using a grid of faint neon lights. For the demonstration, the inventors of ENIAC, John Mauchly and J. Presper Eckert placed halves of Ping-Pong balls over the lights and painted them with numbers each light represented. When one of the engineers turned off the lights in the room during the actual computation, the result was a captivating show of flashing lights produced by a machine that was soon nicknamed a 'giant brain' by the press.

Although the audience for the demonstration was almost exclusively male, the programmers who set up the machine to undertake the calculation were mostly female. One of them, Betty Jean Jennings (later Jean Bartik), recalled how she and a colleague, Betty Snyder, were busy making the last corrections on the day before the demonstration.[15] Last-minute debugging was apparently as common in the 1940s as it is 80 years later. According to Jennings, the machine was calculating the trajectory cor-

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

rectly, except that it did not stop when the shell hit the ground. When the programmers left around midnight to catch the last train home, the calculation was still buggy. By the morning, Betty Snyder had figured out what was wrong. The program was doing one additional iteration after testing for collision with the ground. She flipped one of the 3,000 ENIAC switches to correct the error and the calculation was ready for the presentation.

The ENIAC was initially designed and built to calculate artillery firing tables for the United States Army. But even before it was publicly unveiled in 1946, it was programmed to perform scientific simulations of neutrons passing through various materials to assist with the design of the hydrogen bomb. This classified work was going on, in parallel with firing trajectory calculations, at the time of the public demonstration of the machine.

The programming of the ENIAC was a complex process that involved novel mathematical analysis, devising computation plans, translation of such plans to ENIAC setups and, of course, all kinds of debugging. Much of the translation work was done by women (Figure 2.1), initially referred to as *operators*.[16] Many of them were recruited from a group of human computers hired earlier to calculate ballistics tables by hand, using mechanical desk calculators and a differential analyser. Women were, at least



Figure 2.1 Four women programmers holding parts of the first four US Army computers, from left to right, Patsy Simmers, Mrs. Gail Taylor, Mrs. Milly Beck, Mrs. Norma Stec. (Source: US Army/ARL Archives, Courtesy of Mike Muuss[17])

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

initially, hired for the work because of the uncertain status of the job and because of the labour shortage resulting from the ongoing war effort. To the surprise of some of the engineers and mathematicians, it soon turned out that the work of computer operators required ingenuity and posed at least as many problems as the engineering and mathematical work.

The ENIAC was, as Betty Jean Jennings eloquently put it, a 'son of a bitch to program'. In its early days, it was programmed by setting switches and plugging wires to connect individual components of the machine. Those included *accumulators*, which were used for addition, subtraction and number storage. Specialised components included *divider and square rooter*, *multiplier* for multiplication and *master programmer* that could be used to control looping and branching. The components operated in parallel and had to be synchronised by connecting cables from pulse outputs (indicating the completion of an operation) to pulse inputs (to start the next operation). This was further complicated by the fact that different components took different lengths of time to complete their work. For some, such as the divider, the time depended on the values with which they computed. Programming the ENIAC was more like constructing new elaborate scientific equipment than programming as we know it today.

The first computers also suffered from frequent mechanical or electronic failures. During the first runs of the ENIAC, the engineers were continually fixing the machine, replacing blown vacuum tubes, resolving short circuits and correcting other failures. But as the story told by Betty Jean Jennings shows, programmers of the first electronic computers also had to cope with some of the same difficulties as programmers do today. To make matters worse, early computers like ENIAC were idiosyncratic and had very limited computing power. This meant that programmers had to invent numerous clever tricks to make computers do even the most basic calculations. This included both technical inventions, to overcome machine limitations, and novel numerical analyses, to cope with the way the electronic computers calculated. At the same time, each machine was different and so required very different kinds of programming tricks. The programming tricks were rarely documented and became a part of each programmer's personal repertoire. This unwritten knowledge later contributed to their reputation for being 'geniuses and mavericks'.[18] In a later recollection, John Backus, creator of FORTRAN, said 'Programming in the 1950s was a black art, a private arcane matter.'[19]

Over time, the mechanical and electronic failures became less frequent. In the case of ENIAC, the engineers realised that vacuum tubes often blew because of power surges created when the machine was turned on. By keeping it always on, they managed to reduce the failure rate to one tube every couple of days. Many believed that programming errors would, over time, become as infrequent as hardware errors.[20] The industry needed almost a decade to fully realise that this was not going to be the case. One of the first computer pioneers who understood that programming was going to be a major concern was Maurice Wilkes, who designed and helped build the EDSAC computer at the University of Cambridge in 1949. Wilkes recalled that, while working on a programming challenge 'the realisation came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

my own programs'.[21] The importance of programming became even more apparent as computers entered the commercial sector, which started happening almost as soon as the ENIAC was built.

Following a dispute with the Moore School of Electrical Engineering about patent rights, the two ENIAC creators J. Presper Eckert and John Mauchly resigned from their university positions and formed the Eckert–Mauchly Computer Corporation. They hired three of the original ENIAC programmers and started building commercial computers designed for business applications. Their first client was the US Census Bureau, but later installations included a number of manufacturing and insurance companies, as well as other government departments.

As computers left the university, companies needed to recruit personnel to program the machines. The early programmers came from a variety of backgrounds including engineering, physics and mathematics, but the growth of the computing industry expanded the range of people who were involved with computers to include statisticians, secretaries, managers, salespeople and many others. All companies that were entering the computing industry faced the challenge of how to recognise who might have a talent for programming. One recommendation was to look for creative people who like intellectual challenge like chess players or mathematical puzzle solvers,[22] but the reality was that nobody knew what a programmer looked like.

Not only was it hard to find programmers, it was also difficult to train them. Much of the knowledge that early programmers needed to have was intuitive and local. It consisted of tricks that worked on specific machines, anecdotes and rules of thumb. Programming at the time had many of the typical characteristics of the hacker culture of programming. Hackers tend to rely on hunches that are derived from experience with a large number of past situations, but are hard to record in an explicit form. More often than not, the knowledge possessed by hackers is tacit knowledge,[23] which is acquired through practice or by working with those who already have the skills. The way hackers learn their craft is not through any formal education, but through practice and apprenticeships.

2.2 Sound Body of Knowledge

Programming as a discipline that relies on tacit knowledge of the hacker culture was good enough in the early days of digital computers when there were only a few distinctly unique machines. As the computer programming sector started growing and diversifying in the 1950s, programming increasingly became the key limiting factor. The growth of the electronic data processing industry in the early 1950s led to an imminent shortage of programmers. Throughout the 1950s, the shortage developed into an industry crisis referred to as the 'programming gap'.[24] The gap was widely reported and discussed in industry journals. It was certainly due to the novelty of the problems and the rapid growth of the industry, but the nature of the knowledge needed by the early programmers may have also played its part. Tacit knowledge that a programmer

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

in the 1940s and 1950s had to master is slow to acquire and it cannot be easily distilled into textbooks or training courses.

The ongoing industry crisis made many people involved with computers ponder how to make programming into something more than a mere practical skill learned through practice. The variety of ideas about what programming should look like served as an early example illustrating the differences between several cultures of programming.

Programmers with academic aspirations found the current state of the programming discipline lacking. One of the assessments of programming in the early 1950s came from Edsger Dijkstra, who was a theoretical physicist by training but started programming in 1952. He later recalled his dissatisfaction when he realised, some three years after he began working as a programmer, that programming lacked 'the sound body of knowledge that could support it as an intellectually respectable discipline'.[25] For those with academic interests, this was not only intellectually dissatisfying, but it also threatened to keep programming a low-status profession in academia.

In academic circles, many of those interested in computers joined the Association for Computing Machinery (ACM) after it was established in 1947. By the end of the 1950s, the members of the ACM became convinced that a more rigorous approach to programming should be modelled on applied mathematics. Although the initial motivation was practical, this focus later led to a disconnect and focus on computer science as an academic discipline as opposed to computer programming done by practitioners.

The focus point of the emerging academic discipline became the notion of an *algorithm*,[26] which was defined as a 'finite set of rules that gives a sequence of operations for solving a specific type of problem'.[27] The concept of an algorithm gave computer scientists a practical agenda with many interesting unsolved puzzles. A prime example of a work done within this newly created mathematical paradigm of computer science became the book series *The Art of Computer Programming* by Donald Knuth,[28] who set out to document the most fundamental algorithms. As of today, the first three and a half volumes of this ever-growing monograph have been published, consisting of over 3,000 pages. Similarly, *The Communications of the ACM*, a journal that was established in 1958, dedicated a large part of its content to the discussion of algorithms.

The practitioners also started calling for the professionalisation of programming, but their interpretation of what this means differed. For managers, the key concern was making programming less reliable on the personal skills of individual programmers. Programmers themselves were concerned with career paths and being recognised as skilled programmers. The topics discussed in the practically oriented computer magazine *Datamation*, launched in 1957, demonstrate all those interests. They include novel equipment for data management, articles on programming education and training, machine compatibility and management techniques. There were some early attempts to define a more practically oriented kind of programming knowledge that would form the middle ground between the managerial and mathematical conception. Professional organisations bringing together practitioners in the field of data processing started developing professional certifications such as the Certificate in Data

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

Processing, which originated in 1960,[29] while the development of the SAGE system in the 1950s was supported by early programming training efforts.

The ideas discussed so far are characteristic of three cultures of programming. The programmers belonging to the *hacker culture* keep mastering their craft and inventing new tricks to make programming more manageable. Those with commercial focus that I associate with the *managerial culture* want to make programming more predictable, so that software can be built according to a specification and within a fixed budget. Those with academic interests that I refer to as the *mathematical culture* focus on mathematical analysis of algorithms, which has enabled them to establish computer science as a reputable discipline in the modern university environment that values theory higher than practice.[30]

The developments also show how the different cultures began to diverge and clash.[31] In particular, the focus on abstract algorithms distanced academic computer science from the concerns of commercial programmers, while the managerial attempts to make programming more structured conflicted with the interests of the hackers. The certification and training efforts could be seen as early work of the *engineering culture*, but it would still be some time before programmers would start embracing this perspective.

Given their divergent interests, it seems unlikely that the different cultures could all contribute to a single programming invention. Yet, this is exactly what happened and it revolves around what is quite possibly the greatest invention in programming of the 1950s.

2.3 How Technology Became Language

Programming computers by plugging wires between components, as done in the early days of the ENIAC, did not stay around for long. Most later machines were programmed through numerical codes that were entered by flipping switches and using punched cards. However, each machine had its own encoding with different numerical codes, as well as its own idiosyncratic limitations. As a result, all programs were written for just a single machine.

In the 1950s, 'programming' referred to the process of devising an algorithm, while 'coding' was the more tedious task of encoding the steps of the algorithm as machine instructions. Programmers of the early computers soon realised that they can use the computer itself to make coding easier.[32] The Short Code system, built in 1950 for the UNIVAC I computer, enabled programmers to specify programs in a more human-readable, although still numerical, code that resembled infix mathematical expressions. The computer run a separate program, Short Code interpreter, which read such instructions and performed the specified computations. A more efficient approach implemented by other systems within a few years was to create a program that would read such human-readable code and translate it to the code used by the machine, which would then be executed directly. A computer scientist will recognise these two methods as predecessors of modern interpreters and compilers. The approach

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

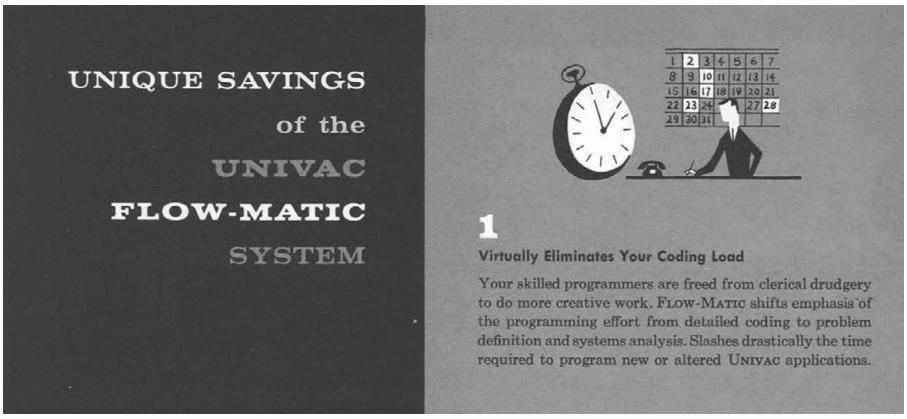


Figure 2.2 A page from a brochure 'Introducing a New Language for Automatic Programming Univac FLOW-MATIC' published by the Sperry Rand Corporation in 1957.[33]
(Source: Computer History Museum,[34] reproduced with permission of Unisys Corporation)

of writing programs using more expressive pseudo-instructions and translating those to the machine language became known as *automatic coding*. As illustrated by the FLOW-MATIC promotional brochure (Figure 2.2) it promised to 'virtually eliminate your coding load'.

A popular description of programming at the time was that the programmer had to come up with a program and then translate it into a language the machine understood. This used the anthropomorphic metaphor of computers as giant electronic brains, which was introduced by the media and was further popularised by ongoing developments in cybernetics. The metaphor made it natural to describe programming as a translation from human language into a machine language. It was not used just by popular media. A computer scientist Grace Hopper illustrated her 1954 programming course at MIT with a drawing (Figure 2.3) that describes the A-2 programming system, a predecessor to FLOW-MATIC, as a robot translating between two languages.

During the second half of the 1950s, the notations used by automatic coding systems became known as *programming languages*. This happened as the language metaphor used in the context of computers 'lost its anthropomorphic connotation and acquired a more abstract meaning, closely related to the formal languages of logic and linguistics'.[35] The FORTRAN programming language illustrates this gradual shift. The 'Specifications for the IBM Mathematical FORmula TRANslating System FORTRAN', published in 1954, did not use the term 'programming language' at that time, but it already consistently referred to the notation as the 'FORTRAN language'.[36]

The most revolutionary aspect of the development was that programming languages became standalone objects, independent of hardware that supported them, and began to be studied on their own. The FORTRAN language is, again, a witness of this development. It was first built for the IBM 704 machine in 1957, but by 1958, IBM was planning to make FORTRAN available for other machines. The early versions of FORTRAN for other machines were not fully compatible with each other, but it

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

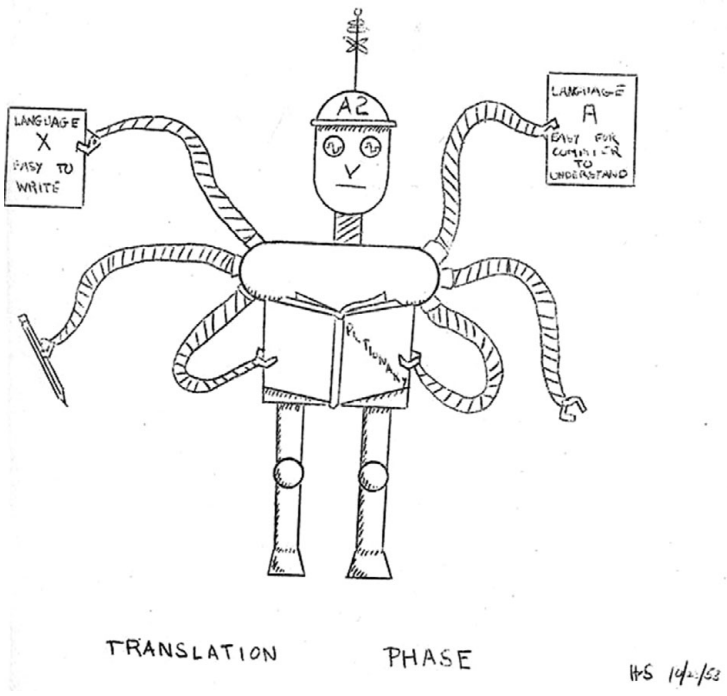


Figure 2.3 An illustration by Grace Hopper that describes the functioning of the A2 compiler for the UNIVAC computer.[37] (Source: MIT, Reproduced with permission)

became possible to think that they should be, and the managers of academic computing centres who wanted to share and reuse programs soon started requiring such machine independence.

At least three different cultures of programming played their role in the birth of the idea of a programming language. The managerial culture provided motivation for detaching programs from individual machines. A programming language makes this, at least in principle, possible and 'the early users of the term *programming language* were principally computer-user groups and computer-installation managers attempting to bring about the cross-machine compatibility of programs' who understood that 'common or universal notations would facilitate the exchange of programs among and within organizations, and would also provide a suitable vehicle for teaching programming in universities'.[38] Technically, programming languages were the next step of the development that started with automatic coding systems such as Short Code, A-2 and FLOW-MATIC. The 'black art' and arcane tricks of the hacker culture of programming were behind the implementations of the interpreters or compilers for the emerging programming languages.

The work on machine-independent programming languages was proceeding on two fronts. The largest computer-user group, SHARE, established a committee to study universal programming languages. The committee failed to make a specific language

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

proposal and some members felt this problem was a 'bucket of worms'.[39] The work was complicated by practical challenges, such as machine differences, but it may also have been hindered by the lack of suitable tools for defining a programming language.

The academically minded members of the Association for Computing Machinery (ACM) similarly established a committee in order to develop a universal language to be used in academic publications and for sharing scientific calculations. The ACM committee eventually developed the International Algorithmic Language (IAL), which was soon renamed to just Algorithmic Language, or Algol. The members of the ACM committee had various academic backgrounds, but unequivocally represented the mathematical culture of programming. This background had a strong influence on the definition of the Algol language. The language was defined using mathematical tools, such as formal grammar, that made it possible to treat programs and programming languages as mathematical objects.[40]

The managerial, hacker and mathematical cultures of programming all contributed to the idea of a programming language. They all recognise the importance of the idea, but they view it from a somewhat different perspective. To mathematicians, programming languages are abstract entities that can be formally studied; to hackers, they are tools to help them with coding and to managers, they are a way to make programs run on multiple machines. Those views are not incommensurable and many individuals span multiple cultures and combine their perspectives. The notion of a programming language plays the role of boundary object[41] in that it provides a common object of interest for multiple cultures. Although the birth of programming languages brought the different cultures closer, it did not permanently bind them together. The idea of programming languages is flexible enough to let the different cultures keep their own distinct perspectives. Three influential programming languages that were designed in 1958 and 1959 illustrate this point.

The Algol language originated with the mathematical culture. It contains a reference to the formal notion of an algorithm in its very name and emphasised the use of mathematical methods for programming. It was recognised as an 'object of stunning beauty'[42] and a remarkable achievement of computer science. Although it was never widely used for building commercial software, it became the de facto language in programming textbooks and academic publications. It was also at the core of many developments in mathematisation of programming, which I will follow in the rest of this chapter.

The language that eventually originated from the commercial efforts to create a portable language for business data processing was COBOL ('Common business-oriented language').[43] The language was born from the managerial culture of programming. Its designers came from commerce and government and did not include anyone with university affiliation. The language aimed to be easy to use, even if that made it less powerful. It aimed to broaden the range of who can use it, for example, by using English-like syntax. Once developed, COBOL was widely adopted by the data processing industry and remains in use in many legacy systems today. At the same

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

time, COBOL was criticised by academics for its poor structure and verbosity and has never been adopted and studied at universities. COBOL's emphasis on data processing meant that it had innovative language features for working with data, a topic that I will return to when discussing types in Chapter 5.

The third language appearing at the end of the 1950s was LISP, which was designed by John McCarthy at MIT.[44] The cultural origins of LISP are harder to untangle than those of Algol and COBOL. The popular story of LISP today is that it was a more or less direct implementation of the lambda calculus, a formal system developed in mathematical logic based on functions. This would place LISP firmly into the realm of the mathematical culture of programming. As pointed out by the programming historian Mark Priestley,[45] the reality is more complex. LISP was motivated by work on symbolic artificial intelligence (AI) that required sophisticated list processing. It was inspired by a range of existing practical AI projects, the FORTRAN language, as well as mathematical theories including recursive function theory and the lambda calculus. LISP certainly has mathematical origins, but it was always a pragmatic language that borrowed ideas from all possible sources in order to solve practical problems. However, LISP soon became the language of choice of the MIT hackers, possibly because it was suitable for the interactive and experimental way of programming favoured by the hacker culture, a history that I revisit in Chapter 3.

2.4 Mathematical Science of Computing

The Algol language may not have been adopted for commercial programming, but it had an enormous influence on academic computer science. The Algol report[46] is written in a combination of formal and informal language. For describing the language syntax, it uses a format that is very similar to that used in formal language theory. The semantics of the language, that is, what the individual constructs mean during execution, is specified using a very structured and precise English language description.

Although the report is formal, it also illustrates that the use of mathematical formalisms in computer science differs from their use in mathematics in that the report presents a definition rather than a proof of a theorem. Such structure would make for a very odd publication in mathematics! Yet, the semi-formal definition was enough to get computer scientists to think about programs in a mathematical way. It was also formal enough to get computer scientists to point out issues with the Algol language definition itself, leading to a revised version published three years later.[47]

The Algol report showed how to think about programs as formal entities and made it conceivable to use mathematical methods for their study. Academic computer scientists could now join the effort to formally study concrete programs, develop mathematical tools for proving their properties and mathematically analyse programming languages themselves. As pointed out by Mark Priestley, the Algol report played a crucial part in

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

transforming the vague feeling that programming should be seen as a mathematical activity into a concrete scientific research paradigm.[48] The paradigm provided the mathematical culture of programming with basic assumptions about what programs are. It identified problems that are worth studying and valid scientific methods for tackling those problems.

One of those who started exploring the possibilities of mathematical reasoning about programs was John McCarthy. In a manifesto of the Algol research program that is relevant to this day,[49] he identifies computer science as a mathematical science in which 'it is possible to deduce from the basic assumptions, the important properties of the entities treated by the science'. He sees programs as such entities and outlines a number of program properties that one can study: Are two procedures equivalent? Does one procedure take fewer steps than another? Does a translation algorithm correctly translate procedures between two programming languages? Many of the questions outlined by McCarthy are still studied by academic computer scientists 60 years later. McCarthy also hopes that the new mathematical science of computing will almost eliminate debugging and claims that, 'instead of debugging a program, one should prove that it meets its specifications'.

To give a sense of the leap from plugging cables on the ENIAC and writing idiosyncratic numerical codes to formal reasoning about Algol programs, let's go through one example discussed by McCarthy. He illustrates the various formal tools he proposes by proving that a program implements the mathematical factorial function. A factorial, written $n!$ is defined as 1 when n is 0 and $(n-1)! \times n$ when n is greater than 0. McCarthy defines a function $g(n, s)$, which computes $n! \times s$ as:

$$g(n, s) = \texttt{if}\ n = 0\ \texttt{then}\ s\ \texttt{else}\ g(n - 1, n \times s).$$

The function calculates the factorial by going down from the highest number to 0. To obtain a factorial of n, you call $g(n, 1)$. In each step, the function multiplies s by the current n and then it continues for $n - 1$ until it counts down to 0 and then it returns the current s. When calling $g(3, 1)$, the value of s at the end will be $1 \times (2 \times (3 \times 1)) = 6$.

The definition is recursive, meaning that g is defined in terms of g and it uses the conditional expression `if` introduced in the Algol report. The conditional expression evaluates to the result of one of the branches, depending on whether the condition $n = 0$ is true or false. This may not seem revolutionary, but it is quite a step from the instructions that machines actually execute. Only some 10 years earlier, EDSAC programmers had to program using memory addresses and two special accumulators instead of variables. To encode conditions, they had to use the `EnS` instruction, which checked if the value in the accumulator is positive and if so, jumped from the current location to the memory location `n` and then continued executing more instructions.

McCarthy proves that the $g(n, s)$ function actually implements mathematical factorial. As we will discuss proofs about computer programs repeatedly in the rest of this chapter, it is useful to see how this is actually done. I will show this in more detail than McCarthy, so that each step involves just one idea. We start with the obvious truth that

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

$n! \times s = n! \times s$. We refine the formula in a number of steps, making sure that we always replace it with one that means the same thing, until we get to our definition of g:

$$
\begin{aligned}
n! \times s &= n! \times s, && (1)\\
n! \times s &= \texttt{if } n = 0 \texttt{ then } n! \times s \texttt{ else } n! \times s, && (2)\\
n! \times s &= \texttt{if } n = 0 \texttt{ then } 1 \times s \texttt{ else } ((n-1)! \times n) \times s, && (3)\\
n! \times s &= \texttt{if } n = 0 \texttt{ then } s \texttt{ else } (n-1)! \times (n \times s), && (4)\\
g(n,s) &= \texttt{if } n = 0 \texttt{ then } s \texttt{ else } g(n-1, n \times s). && (5)
\end{aligned}
$$

The fact that proving the correctness is quite tedious, even for such a simple example, is partly the point of this example, so please bear with me. We start with an obvious truth (1) and then introduce a conditional expression (2). The conditional expression evaluates to either the `then` or the `else` branch, but those are both the same original expression. Next, in (3) we rewrite the expressions using facts about $n!$ In the `then` branch, we know that $n = 0$ and so $n! = 1$. In the `else` branch, we know that $n \neq 0$ and so we can replace $n!$ with $(n-1)! \times n$. We then simplify the expressions (4) and now we have our program, if we just replace $n! \times s$ with $g(n,s)$ on the left-hand side of the formula and $(n-1)! \times (n \times s)$ with $g(n-1, n \times s)$ on the right-hand side of the formula.

The example shows that one can mathematically prove the correctness of a simple program, but if we look at the articles published in the *Communications of the ACM*, a primary journal of the mathematical culture of programming throughout the 1960s, we can see that writing tedious formal proofs had not become a commonplace activity. The journal, however, adopted Algol as the language for publishing algorithms and those were written in a formally precise style. The *Communications* included a column containing a number of new algorithms each month. Between 1961 and 1973 the algorithms were numbered and reached number 472 when the column stopped being regular. This confirms the point, made by Nathan Ensmenger[50] that the notion of algorithm has became central for the mathematical culture of programming. The magazine, however, simply published useful algorithms with a comment on what they do and did not study them formally. Publications with complete formal proofs of program correctness were relatively rare. Even papers that contain proofs, such as the 1969 paper 'Prevention of System Deadlocks',[51] which presents a resource allocation algorithm that never gets into a stuck state known as deadlock, focus more on the general conditions of the mathematical model, rather than on a line-by-line analysis of the code of the algorithm.

Being aware that proving actual programs correct was very tedious, a number of computer scientists focused on finding easier and more practical methods for doing so. Most of those were centred around some way of attaching formal mathematical propositions to specific locations in a program. The idea first independently appeared as *snapshots* in the work of Danish computer scientist Peter Naur[52] and *interpretations* in the work of American Robert W. Floyd.[53] As was typical at the time, both Naur and Floyd got involved with computers through a programming job after their

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

studies of a different scientific discipline, astronomy and physics, respectively. However, the method that gained notoriety in the mathematical culture of programming was described by C. A. R. Hoare in 1969 and became known as 'Hoare triples'. Hoare studied classics and philosophy, followed by statistics, but became a programmer for a small British computer manufacturer in 1960. A year later, he started implementing a compiler for Algol and eventually became chief engineer at the company. In 1968, he moved to academia and began working on an axiomatic description of programs. Building on the work of Naur and Floyd, he proposed a method for formally reasoning about programs in 'An Axiomatic Basis for Computer Programming'.[54] Here, Hoare introduced a notation written as $\{P\}c\{Q\}$. In the formalism, c is a program instruction also called a command and P, Q are formulas of mathematical logic called pre-condition and post-condition, respectively. A Hoare triple $\{P\}c\{Q\}$ asserts that, if a pre-condition P holds before the execution of a command c, then the post-condition Q will hold after the program runs.[55]

There are two interesting aspects of the axiomatic approach introduced by Hoare. The first aspect is that, the pre-conditions and post-conditions P, Q in $\{P\}c\{Q\}$ do not need to talk about everything that is happening in the program. They can only involve aspects that are necessary for the proof and ignore anything that is irrelevant. The second aspect is that the approach is compositional. Let's say that we have two commands c and d with properties $\{P\}c\{Q\}$ and $\{Q\}d\{R\}$ where Q is the post-condition of the first command and, at the same time, the pre-condition of the second command. Now, if we write a program $c; d$ that runs the two commands in a sequence, we know that $\{P\}c; d\{R\}$ because the post-condition of the first command is the same as the pre-condition of the second command. In the first paper, Hoare used his approach to prove the correctness of a simple program that performs division with a remainder, but he later used a similar approach to prove the correctness of a somewhat more complicated program FIND that partially sorts an array of data, which was published in the Communications of the ACM.[56]

To offer a sense of proofs about programs based on Hoare triples, let's look at a different way of calculating the factorial of a number n. This time, the program is written as a loop that iterates from $i = 1$ to $i = n$ and, in each step, multiplies the result s by i:

$s = 1; i = 1;$
`while` $i < n$ `do`
  $i = i + 1;$
  $s = s \times i;$
`end`

The value s is first set to 1. It is then multiplied by $2, 3, \ldots n$ and so the final value will be $s = 1 \times 2 \times 3 \times \cdots \times n$. But how can we formally prove that this actually works? Using Hoare triples, we can focus on the key part of the program, which is the body of the loop. To make the logic easier to follow, I will write i' and s' for the new state of the variables after the body of the loop executes. It turns out that, for the two assignments in the body it holds that $\{s = i!\}i' = i + 1; s' = s \times i\{s' = i'!\}$. This

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

means that, if s is a factorial of i before the body runs then, after the two assignments are executed, s' will again be the factorial of i'. The trick is that i is incremented by 1 and s' has been multiplied accordingly. We also know that, as the program starts, it holds that $\{\}s = 1; i = 1; \{s = i!\}$. This simply tells us that, when we set s and i to 1, it will be true that $s = i!$ because $1 = 1!$ The compositionality of the method means that we can now put all these arguments together. We start with $s = i!$ and this is also the pre-condition and post-condition of the loop body, because the values s', i' will become the new values of s, i of the next iteration. This means that $s = i!$ will be true after the loop finishes looping. But then, $n = i$ and so at the very end $s = n!$ which is exactly what we wanted to show.

It may not seem like that from my examples, but the axiomatic approach did make reasoning about more complex programs easier. This is due to the two aspects that I mentioned earlier. We need to choose our axioms so that they talk only about what matters and the final proof can be composed from proofs about individual parts of programs. Together, these two aspects also enabled later developments of semi-automatic program verification tools, a topic that I will return to at the end of this chapter.

In retrospect, it is difficult to say what the expectations of the authors contributing to the mathematical culture of programming were. Hoare treats programming almost as a branch of mathematics. His hope likely was that programmers would use deductive reasoning themselves to ensure their programs are correct. McCarthy was aware that this is a lot of work and suggests that programs should be involved in checking of the proofs, 'because we can require the computer to do much more work in checking each step than a human is willing to do'.[57] This idea is closer to the way mathematical methods are used in software verification tools today. However, building such tools required both further theoretical innovations and the involvement of the engineering culture of programming in order to build tools that can be used to check non-trivial practical software systems.

2.5 Many Definitions of Algol

Early examples of rigorously applied formal reasoning to computer programs involve very simple programs or algorithms. Even the aforementioned proof of the FIND program by Hoare, which shows 'the construction of the proof of a useful, efficient, and nontrivial program'[58] studies an algorithm that has some 20 lines of code. In practice, even the proponents of the mathematical culture relied on empirical testing. The *Communications of the ACM* magazine confirms this fact. In addition to publishing the algorithms themselves, the magazine also published 'certifications' of previous algorithms, such as the one in Figure 2.4. Those were certainly not proofs. They were personal reports confirming that the reader implemented the algorithm as described previously, tested it, possibly corrected it and obtained the expected result.[59] The practice of publishing certification of algorithms bears a resemblance to the scientific practice of reproducing an experiment and publishing a reproduction study. The

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

CERTIFICATION OF ALGORITHM 58
MATRIX INVERSION (Donald Cohen, *Comm. ACM 4*, May 1961)

RICHARD A. CONGER
Yalem Computer Center, St. Louis University, St. Louis, Mo.

Invert was hand-coded in FORTRAN for the IBM 1620. The following corrections were found necessary:

The statement $a_{k,j} := a_{k,i} - b_j \times c_k$ *should be*

$$a_{k,j} := a_{k,j} - b_j \times c_k$$

The statement **go to** back *should be changed to*

$$i := z_k ; \quad z_k := z_j ; \quad z_j := i; \quad \textbf{go to} \text{ back}$$

After these corrections were made, the program was checked by inverting a 6×6 matrix and then inverting the result. The second result was equal to the original matrix within round-off.

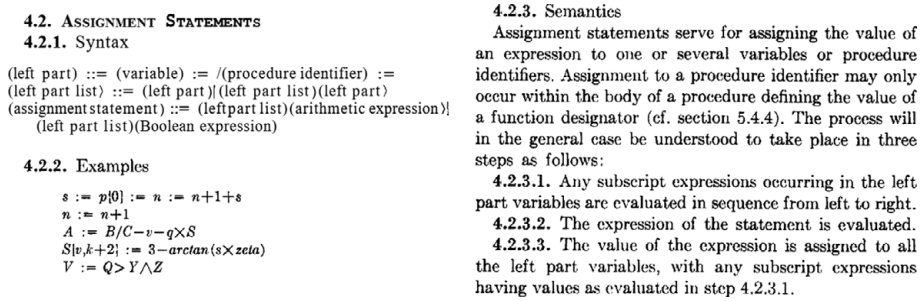
Figure 2.4 An algorithm certification published in the *Communications of the ACM*. (Source: Used with permission of ACM,[61] permission conveyed through copyright Clearance Center, Inc.)

work thus combines aspects typical for the mathematical culture with aspects that will later become typical for the engineering culture.[60]

A prime example that illustrates the struggles with applying rigorous mathematical methods to large software systems is the quest for a formal description of the Algol programming language. Algol emphasised its mathematical structure. As an object of stunning beauty for the theoreticians, it is an obvious target for formal mathematical treatment. Yet, formally defining Algol turned out to be more difficult than expected. The full story, recounted meticulously by Troy Astarte,[62] sheds light on problems that computer scientists still face 60 years later.

Algol was developed by a committee of American and European computer scientists in a meeting in 1958 in Zurich and revised in a subsequent meeting in 1960 in Paris. Figure 2.5 shows an excerpt from a canonical description of Algol, 'Revised report on the algorithmic language Algol 60',[63] which 'gives a complete defining description of the international algorithmic language Algol 60'. To a present-day computer scientist, a language specification would have to define two aspects of the language. The syntax would define the syntactic structures of the language and how they can be composed when writing a program, whereas the semantics would define how programs written using the permitted syntacitc structures evaluate, that is, what happens when a program runs. The Algol report defines the syntax formally using the Backus–Naur form, a notation technique developed for the report and still in use in modern programming language specifications. It provides numerous examples and then proceeds to describe the language semantics. This is done in English which is precise enough for a human reader, but does not define the semantics of the programming language in formal mathematical terms.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003



4.2. Assignment Statements

4.2.1. Syntax

(left part) ::= (variable) := /(procedure identifier) :=
(left part list) ::= (left part)|(left part list)(left part)
(assignment statement) ::= (left part list)(arithmetic expression)|
(left part list)(Boolean expression)

4.2.2. Examples

$s := p[0] := n := n+1+s$
$n := n+1$
$A := B/C-v-q\times S$
$S[v,k+2] := 3-arctan(s\times zeta)$
$V := Q > Y \wedge Z$

4.2.3. Semantics

Assignment statements serve for assigning the value of an expression to one or several variables or procedure identifiers. Assignment to a procedure identifier may only occur within the body of a procedure defining the value of a function designator (cf. section 5.4.4). The process will in the general case be understood to take place in three steps as follows:

4.2.3.1. Any subscript expressions occurring in the left part variables are evaluated in sequence from left to right.

4.2.3.2. The expression of the statement is evaluated.

4.2.3.3. The value of the expression is assigned to all the left part variables, with any subscript expressions having values as evaluated in step 4.2.3.1.

Figure 2.5 An excerpt from the Revised report Algol 60 showing the syntax and semantics of assignment statments. (Source: Used with permission of ACM,[64] permission conveyed through copyright Clearance Center, Inc.)

Contrary to the expectations about mathematical publications, the report does not specify any theorems about Algol. It is merely a 17-page-long definition. This illustrates an important discrepancy between mathematics and programming. Mathematical theorems are shaped by their proofs, counter-examples and the mathematical striving for simplicity and elegance.[65] By contrast, the structure of theorems about programs is mainly shaped by the programs they talk about. The programs are, in turn, shaped by the aim of building software that can be run to achieve some task. Algol may be a carefully designed simple programming language, but as a mathematical object, it remains very complex.

The questions posed by McCarthy, which I discussed earlier, make it clear that the proponents of the mathematical culture were interested in proving properties about the Algol language and programs. To do this formally, they first needed to produce a formal definition not just of the language syntax, but also of the language semantics. An informal English language description of the semantics was not enough. McCarthy made the first step in 1964 by defining a formal semantics of a minimal subset of Algol that he called Microalgol.[66] Microalgol includes the tricky `goto` construct, but omits many other features of Algol. The semantics is defined as a recursive function $\texttt{micro}(\pi, \xi)$, which defines a single step of the program execution. Given a program π and a current state of execution ξ, the result of $\texttt{micro}(\pi, \xi)$ is a new state ξ' that describes what has changed after running a single statement of the program. The states ξ, ξ' consist of vectors of values assigned to variables together with a number representing the next statement to be executed.

McCarthy acknowledges that Algol is considerably more complicated and lists a number of issues, but believes that 'those difficulties can be resolved and that a clear description of the state of an Algol computation will clarify the problem of compiler design'. The case of Microalgol illustrates two important aspects of programming language definition. The first is the meta-language used. This is the language used to talk about the programming language. McCarthy uses normal mathematical notation with concepts like vectors and functions. The second aspect is, how is the meaning of programs defined? In the case of Microalgol, the mathematical definition works as a kind of interpreter. It uses formal mathematical methods to define how programs

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

execute and, step by step, transform a mathematical representation of the state of the computer executing the Algol program.

Hoare, who introduced the $\{P\}c\{Q\}$ notation for reasoning about programs, uses a different mathematical formalism. Rather than specifying how individual language constructs such as the assignment operator transform the state of the computer, he gives axioms about individual language constructs. For example, the Axiom of Assignment[67] states that, if predicate $p(f)$ holds for some f and the program executes an assignment $x := f$ that sets the value of the variable x to f, then a predicate $p(x)$ will hold afterwards. Using Hoare's notation, the axiom states that $\{p(f)\}x := f\{p(x)\}$. The method used by Hoare is quite different from that of McCarthy, but he relies on the same general approach of using a simple subset of the actual Algol language. He omits the tricky `goto` construct and acknowledges that this might be difficult, because the meaning of the command cannot be easily defined using the axiomatic approach. He, however, believes that other aspects of Algol will not cause any great difficulties. Hoare also shares McCarthy's overall optimism and believes that 'the practical advantages of program proving will eventually outweigh the difficulties'.

Scaling from a small subset of Algol to the full language proved to be a significant undertaking and required the development of new techniques for specifying programming language semantics. Peter Mosses, who worked on a formal definition of Algol as part of his PhD at the University of Oxford, followed a novel style developed by his PhD advisor Christopher Strachey. Rather than using mathematics to define an interpreter or provide axioms, his method assigns each statement of the language a meaning, independently of the rest of the program. The meanings are mathematical objects, specifically functions written using the lambda calculus formalism. The individual meanings can be composed to get the meaning of an entire Algol program. The 1974 definition by Mosses[68] covers almost the entire Algol language and uses a highly regular mathematical notation, which a reader with mathematical background (and sufficient dedication!) is expected to understand. The length of the document made Christopher P. Wadsworth, another PhD student from Oxford studying under Strachey's supervision, question the simplicity of Algol. In a letter to his advisor, he wrote about Mosses' definition of Algol:

> I must admit I still feel a little surprised it's as long as it is–I guess Algol 60 is just not nearly as 'well-behaved' as one tends to think it is.[69]

The case of the Algol language definition developed by Mosses reinforces the point that I made earlier about formal methods applied to programs. Despite being mathematical, they are nothing like normal mathematical texts. The report by Mosses consists of a 50-page-long definition, but no theorems. The complexity of the definition is rooted in the complexity of the programming language that it is modelling. Producing the definition is challenging not because one needs to come up with elegant mathematical tricks, but because one needs to produce a structure that can account for all the complexities of the language.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

The definition by Mosses, which was published as a technical report by the Oxford University Computing Laboratory in 1974 also illustrates that, by this time, the mathematical culture of programming had established a methodology for doing normal science. The report has only a brief 1-page introduction, which states that it models Algol, as described in the revised report, except that 'own' declarations are omitted. The author does not need to explain what this means and does not need to make any attempt to convince the reader that the model is correct, for example, by comparing the behaviour of the mathematical model with programs produced by a specific Algol compiler. The mathematical culture leaves implicit the fundamental assumption that the model is relevant to actual software. As we will see, both the complexity of the definition and the unclear relationship with actual software will be later identified as problematic by those not strongly committed to the mathematical culture.

Using the ordinary language of mathematics as a meta-language for complex programming language definitions is a limiting factor. Such a mathematical definition has to be written by hand and it has to be checked by other human readers. The computer scientists working on programming languages at the Vienna IBM laboratory hoped that they could develop a compiler for a programming language systematically from its definition through a series of step-wise refinements. For this, the meta-language should not be just the language of human-written mathematics, but instead another formal language that can be processed by a computer. To do this, the Vienna IBM team created a formal language known as Vienna Definition Language (VDL) for specifying the semantics of programs.

The IBM team used the newly developed VDL meta-language to give a series of definitions of Algol. The first was a 67-page definition from 1968[70] that followed the operational style used earlier by McCarthy. Another version developed four years later,[71] aimed to simplify the first one by handling jumps like `goto` in a different way. Finally, the definition that appeared a couple of years later, in 1978,[72] adopted the denotational approach used by Mosses earlier, aiming to produce 'an equally abstract but more readable definition' that is simple enough to fit in a book chapter.

The attempts to formalise Algol illustrate the variety of work done within the mathematical culture of programming, as well as influences of other cultures. Approaches that formalise only a subset of Algol generally aim at explaining interesting aspects of the language to a human. This kind of work is close to traditional mathematics in that it is primarily focused on other human readers. Much of contemporary academic work on programming language design follows this style. A well-chosen language subset makes it easy to illustrate interesting aspects of design choices or programming language features. It also typically limits the scope to an elegant mathematical object. The obvious limitation is that such formal definition cannot be used to provide any guarantees about the full language.

Approaches that aim to formalise the full programming language depart from traditional mathematical practice. They inevitably involve long and complex definitions that are shaped by practical programming concerns rather than mathematical elegance. The methods of defining formal semantics of a full language also often involve the

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

development of new software tools such as VDL built in the Vienna IBM laboratory. We might speculate that the influence of engineering culture of programming was stronger in an industrial research laboratory than at the University of Oxford where a similar definition used a more traditional mathematical notation.[73]

2.6 The Minority Report

For the mathematical culture of programming, the 1960s were a period of active development. The first stand-alone university departments of computer science started to appear, the ACM published its first curriculum recommendation in 1968 and universities started to offer undergraduate courses and degrees in computer science. Academic computer scientists were busy developing new algorithms, programming languages and mathematical tools for reasoning about them.

The efforts to formalise Algol were underway, but the language was still also under development. Soon after the publication of Algol 60, the Algol working group started discussing the requirements for the next version of the language. Proposals included more powerful data-structuring mechanisms that I return to in Chapter 5, as well as a plethora of other ideas, such as a more unified syntax, obtained by removing the distinction between statements and expressions, the ability for the language to manipulate its own programs, different mechanisms for handling I/O, as well as new ways of structuring the formalism used in the report that defined the language.[74]

In 1965, the Algol working group made a decision to adopt a proposal for a completely new design of the language, rather than one based on gradual improvements in Algol 60, which was the preferred choice earlier. The new design aimed at an even greater mathematical elegance and uniformity, but many of the working group members saw it as obscure, incomprehensible and overly complex. The design was eventually published as Algol 68, but the publication also included a 'Minority Report' written by working group members who disagreed with the final design. The dissidents felt that the language description methodology used in the project had failed.[75]

The authors of the Minority Report argued that the problems faced by programmers have changed significantly over the last decade and that programmers find themselves 'faced with tasks of completely different and still growing scope and scale'. In an insightful analysis of the Algol dissidents' thinking, computing historian Thomas Haigh pointed out[76] that the group was gradually becoming less interested in the problem of the 'expression [of programs] in some language' and more interested in the problem of 'reliable creation of programs to perform specified tasks'. In other words, they started to see 'languages as tools designed to support or enforce good practice in the design of complex programs'. This new way of thinking presents a shift from the mathematical culture of programming to the engineering culture of programming that was emerging at the time. The Algol dissidents went on to establish a working group on programming methodology (IFIP WG 2.3), which reflected this

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003



Figure 2.6 Douglas McIlroy presenting a lecture on software components at the NATO 1968 conference. (Source: Photograph courtesy of Robert McClure and Brian Randell)[78]

new focus, but was also organised in a much less formal way. A later collection of articles published by the authors[77] shows that the group kept a strong mathematical emphasis, but was gradually interested in broader software engineering issues, architectural metaphors for software development, program structuring and the activity of programming.

Many of the Algol dissidents were also involved in an event that has been widely regarded as a turning point in the history of computing. The event was the NATO Software Engineering conference, which took place in October 1968 in Garmisch, Germany (Figure 2.6). The conference is often linked with the broader industry crisis and the hopes to turn 'the black art of programming' into a 'science of software engineering',[79] although as convincingly argued by Haigh, the purported impact of the conference is hard to 'square with the actual historical record'. In reality, the legacy of the meeting has largely been shaped by the Algol dissidents, many of whom presented their work at the conference and who also served as editors of the final, highly quotable, conference report.[80]

The conference brought together some of the Algol dissidents, but also a broader computing community from both academia and industry, yet there was a heavy emphasis on those working on complex and academically interesting software systems such as compilers and operating systems. The phrase 'software engineering' was novel at the time and was deliberately chosen as being provocative. It implied the need to move, both from the question of programming languages to the question of software production, as well as the need to move from hacker practices to a development process based on theoretical and practical foundations, following the model of established branches of engineering.

Many of the participants left the Garmisch conference with a sense of excitement. Dijkstra recalled that the meeting was a success 'largely because most of the people

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

present were sufficiently high in their local hierarchies that they could afford to be honest, and were'. The common recognition that there was a software crisis was the most exciting aspect for all the participants. To capitalise on the excitement the organisers immediately planned a follow-up conference that was held a year later in Rome. The organisers hoped to move from the recognition of the software crisis to concrete proposals for addressing it. This did not turn out as well as hoped. In the words of the attendees, the conference just 'never clicked' and they 'left with an enormous sensation of disillusionment'.[81]

The two conferences brought together different cultures of programming. This did not prove to be an issue in 1968. The conference was focused on what is wrong with current methods and everyone was able to contribute from their own perspective. There was enough common understanding among the participants to accept points made by others, even if they felt those were not the most fundamental issues. The aim of the follow-up event was to discuss specific issues in detail and, possibly, come to agreements. This proved to be more difficult. As the editors of the second conference report acknowledge 'a lack of communication between different sections of the participants became a dominant feature'[82] of the second conference. Thinking about different cultures of programming offers a useful perspective for understanding this lack of communication.

The organisers wanted to focus on technical rather than managerial problems. By doing so, they already excluded the issues that mattered to the managerial culture of programming, which saw structuring of work and team management as the primary way towards more satisfactory results. Even among the more technical participants, there were widely different perspectives. The discussion about proofs of program correctness was of interest only to the mathematical culture of programming. Time-sharing systems, which I will discuss in Chapter 6, were seen as a powerful new approach by the hackers, but as a tool that teaches sloppy habits by more mathematically inclined participants. The Algol dissidents and like-minded attendees, who were starting to establish the engineering culture of programming, were primarily interested in the development of new ways of structuring programs and in practical tools to simplify programming, but not in management practices or purely mathematical puzzles. The case that perhaps best demonstrates the disagreements between the different cultures is that of structured programming.

2.7 Go to Statement Considered Harmful

The development of new proof techniques, like the axiomatic method based on Hoare triples, came hand in hand with the effort to write programs in a way that makes reasoning about them easier. This included both formal reasoning, but also reasoning in the engineering sense of being able to convince oneself that the program behaves as one intends. The primary construct in Algol that made such reasoning difficult was the `goto` statement. This is a low-level command that is close to how the machine operates and is similar to the `EnS` instruction from EDSAC that I talked about earlier.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

It unconditionally transfers the control, that is, the position in the program that is currently being executed, to another location that is defined by the argument passed to `goto`. To see the construct in action, we can compare our earlier factorial program with a version written using `goto` instead of `while`:

```
s = 1; i = 1;                    s = 1; i = 1;
while i < n do          L1       if i = n then goto L2
  i = i + 1;                     i = i + 1;
  s = s × i;                     s = s × i;
end                              goto L1;
print(s);               L2 :     print(s);
```

The two programs work in exactly the same way. The only difference is that the program on the left uses the `while` statement, whereas the version on the right implements the logic of the looping using `goto`. When $i = n$, the program on the right jumps to the end of the program to print the result. Otherwise it continues to run the sequence of instructions to update the values of i and s and then jumps back to the label $L1$ to again check if it is done.

Let's now consider how the use of `goto` complicates reasoning about programs. In the version on the left, the code has a structure that makes it easier to see that the block nested inside the `while` block will run repeatedly in a loop. If we want to reason about the code formally, we know that we need to understand the pre-conditions and post-conditions of the body of the loop and then compose those with what we know about the code before and after the loop. In the version on the right, the program is just a flat sequence of instructions. This breaks compositionality, because you cannot analyse the behaviour of `goto` $L1$ without looking at the rest of the program and locating the instructions at location $L1$.

A program structured using `while` is easier to study formally, but it is also easier to understand for a human reader. While the mathematicians were primarily concerned with formal reasoning, the readability of programs started to be of importance to the emerging engineering culture of programming that emphasised good practices in program design. Meanwhile, hackers may oppose constructs such as `while`, because the compiler needed to do more work in producing machine instructions and hackers thought it would produce code that is not as efficient as when using hand-written jumps.

Despite some opposition, there was an agreement among the proponents of the mathematical and engineering cultures that writing code in a structured way was preferable. The most vocal advocate of avoiding `goto` became Edsger Dijkstra, who made this position clear in his letter 'Go to statement considered harmful'.[83] Dijkstra explains the issues with reasoning about programs that include `goto` and observes, with his typical eloquence, that 'the quality of programmers is a decreasing function of the density of `goto` statements in the programs they produce'. Following the publication of the letter, Dijkstra also introduced the term *structured programming* for the style of programming that avoids `goto` in a working paper presented at the NATO 1969 Software Engineering conference in Rome.[84]

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

Dijkstra's letter sparked a debate in the computer science community. Most of those joining the debate accepted that `goto` should not be overused. Some argued that new programming language should not include it at all, while others gave various arguments in favour of it and examples where it is useful. Some also pointed out that any program can be rewritten without the use of `goto`. The canonical reference for this claim, which became known as the *Structured Programming Theorem* became the 1966 paper by Böhm and Jacopini[85] that Dijkstra cited to make this point. The style of the paper is very technical and it uses a somewhat different formulation of the problem and has been 'apparently more often cited than read in detail'.[86] Nevertheless, the theorem was a useful weapon for the proponents of structured programming. As later recalled by one of those who referred to it:

> Us converts waved this interesting bit of news under the noses of the unreconstructed assembly-language programmers who kept trotting forth twisty bits of logic and saying, 'I betcha can't structure this.'[87]

The quote is worth reading carefully, because it gives a number of hints about different cultures of programming. The reference to *assembly-language* programmers suggests that those who opposed structured programming preferred more direct access to the machine, a trait of the hacker culture of programming. Hackers were also concerned about efficiency, which was another argument in favour of `goto` in the debate. It is also interesting to see that a purely theoretical paper, which discusses normalisation procedure for flow diagrams, provided such valuable argument to the proponents of structured programming. The paper has no explicit mention of `goto` or, indeed, programming languages. Even Dijkstra quotes it cautiously, saying that it 'seems to have proved the superfluousness of `goto`'. He also adds that the mechanical translation used in the paper is 'not to be recommended', because it would produce a result that is likely much less readable than the original unstructured code. Yet, theoretical results are deemed as a prime form of knowledge in the mathematical culture of programming and so the result served as a strong argument.[88]

Regardless of the debate, the structured programming approach was soon widely known and adopted, not just by those members of the mathematical culture of programming who were interested in proofs about programs, but also by the broader programming community. As we will see next, the idea also captured the imagination of programming managers, who were less concerned with reasoning about programs and more with control over complexity, budgets and workforce.[89]

2.8 Chief Programmer Teams

Edsger Dijkstra, who introduced the term *structured programming* cannot be pigeonholed into a single culture of programming. He was one of the Algol dissidents and was hoping to find a sound body of knowledge for the discipline of programming. His work was rooted in mathematical methods, yet he was not using mathematics to solve mathematical puzzles, but to improve applied programming. His writing on structured

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

programming illustrates many aspects of what I refer to as the engineering culture of programming. Structured programming is presented as a way that helps the engineer, with their inherent limitations, to do their job better. There is also a strong emphasis on individual responsibility. One can almost imagine that using structured programming when possible would be a requirement in the Code of Ethics that all software engineers should adhere to. This engineering perspective on structured programming remains well aligned with the mathematical view that I discussed above. Structured programming simplifies formal reasoning, which is what mathematicians care about, but it also supports informal human reasoning that an engineer needs to undertake.

However, the idea of structured programming also influenced the management of software projects. Managers saw structured programming as a systematic approach where you divide the overall problem into smaller sub-problems that can be solved independently by less skilled programmers. This was appealing to the managers in the 1970s as the top-down software development approach resembled the hierarchical top-down structure of large corporations at the time.[90]

The success story of the management approach to structured programming was an information retrieval system developed by IBM for the *New York Times*.[91] According to its developers, the system was completed several times faster than comparable systems and ran with almost no errors. The programming followed a top-down methodology inspired by structured programming that the authors called Chief Programmer Team (CPT). In CPT, a single 'chief programmer' was responsible for the overall system design and was supported by a group of assistants who were assigned simpler programming tasks that did not require a full understanding of the system. The chief programmer designs the top-level program structure and directs a team of supporting programmers who solve specific well-defined problems as required by the chief programmer. The hierarchical structure in the program provided by structured programming is thus reflected in the structure of the development team, an idea that would take a new turn in the 2020s.[92]

The originator of the term *structured programming* Edsger Dijkstra never accepted the managerial interpretation of the idea. He was horrified that programming started to turn from an intellectual discipline to an industrial activity with the prevailing American 'management philosophy aiming at making companies as independent as possible of the competence of their employees'.[93] According to Dijkstra, managerial software engineering methods aim for an anti-intellectualism characterised by 'How to program if you cannot'.[94] Despite the disputes, the different takes on the idea often appeared together. We can see this in the December 1973 issue of the *Datamation* magazine,[95] which was dedicated to structured programming. It included the engineering account of `GOTO`-less programming, an article on the Chief Programmer Team methodology, as well as an overview of the structured programming theorem and a light-hearted article proposing to replace the universally hated `GOTO` construct with a 'revolutionary' `COMEFROM` feature.[96]

The concept of structured programming and the disagreements that it provoked illustrate a common pattern of interactions between cultures of programming. Structured programming undeniably benefited multiple different cultures. To the engineering

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

culture, it offered a useful tool for structuring programs so that they are easier to read. It supported compositional reasoning about programs that was appealing to the mathematical culture. The managerial culture used structured programming as a mechanism for structuring the workforce, rather than programs. Yet, as illustrated by Dijkstra's remarks, the proponents of different cultures held conflicting perspectives on the concepts they shared. Those perspectives were not incommensurable, but they were rooted in conflicting assumptions and led to contentious debates.

The cases of programming languages and structured programming are both instructive. Programming languages appeared thanks to a productive meeting of managerial, hacker and mathematical cultures of programming. Programming languages serve as boundary objects that the different cultures could share, but that is interpreted differently by each of them. Programming languages also enable further communication and exchange of ideas between different cultures.[97] This is exactly what happened with structured programming. It appeared as a way of writing code using loops and other structured constructs instead of `goto`. This shaped mathematical formalisations of Algol which sometimes omitted `goto`, seeing it as inelegant and an unnecessary complication. Structured programming then also led to the managerial idea of structuring teams to match the structure of code.

In the case of programming languages and structured programming, the collaboration between the different cultures of programming was enabled by having a shared technical concept, which can be implemented and has an existence of its own.[98] When dealing with a more abstract idea, direct collaboration is harder. The existence of multiple cultures of programming is beneficial in that, when a particular development reaches a dead end, another culture of programming may adopt an idea and find a different way forward. In a way, this is what happened in response to the difficulties with manually writing proofs of program correctness at the end of the 1970s.

2.9 Program Proofs and Social Processes

The opening keynote speaker at the 1968 NATO Software Engineering conference was a prominent computer scientist, Alan Perlis, who had received the Turing Award in 1966 'for his influence in the area of advanced programming techniques and compiler construction'.[99] Perlis was a mathematician by training, although with a clear interest in computers. His PhD thesis from 1950 describes an algorithm for solving integral equations. It does so from a mathematical perspective, but with computing applications in mind.

Despite his mathematical background, Perlis' keynote speech and comments at the NATO conference made it clear that he did not see programs as just another mathematical entity. In the opening keynote, he says that the conference is concerned with objects which 'depend in only very weak ways on the laws of physics' and whose 'structure depends as much on the social laws governing their usage as on their internal constraints'.[100]

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

Perlis also attended the follow-up 1969 conference and participated in a discussion on software correctness. The conference report contrasts Hoare's conviction that programmers should convince themseves that programs work by inductive mathematical proofs with Perlis' scpeticism about the method and a belief that suitably selected test cases are often sufficient. In response, the report editors counterposed Dijkstra's famous remark that 'testing shows the presence, not the absence of bugs'.[101]

Some five years later, Perlis was sitting in a seminar at Yale University. The speaker was Richard De Millo, who was visiting his collaborator Richard Lipton. De Millo was presenting his research on verifying programs and included an extensive proof of one of the algorithms. Perlis raised his hand and asked 'Why does it take 20 pages to prove a program that is obviously correct?'[102]

The question pointed to a fundamental difference between proofs of mathematical theorems and proofs of computer programs – a difference that De Millo and Lipton, also trained mathematicians working as computer scientists, soon recognised. The question prompted a collaboration that resulted in a 1977 paper 'Social Processes and Proofs of Theorems and Programs',[103] which triggered a controversy in the computer science circles. In the paper, the three authors argue that program verification is bound to fail because of the nature of the proofs that it demands.

What a mathematical proof is may appear simple at first glance. In logic, proofs are finite sequences of propositions, each of which is an axiom, an assumption or follows from earlier propositions by a rule of inference. In practice, proofs are much richer. They use a mix of natural and formal language and they are written to convince another, adequately qualified, colleague. Mathematical proofs are central to mathematical practice. They are taught at universities, shared with colleagues and discussed in whiteboard sessions where mathematicians marvel at the elegance of a proof and clever tricks that make it work. If they trust a proof, they believe that it could, in principle, be written as a sequence of propositions, but they never bother writing it in such formal way.

In programming, proofs are bound to be secondary. Mathematical proofs are a part of an interlinked network of concepts and can be reused to prove other theorems. Mathematical theorems are written to be proved, whereas computer programs are written to be executed. A proof about a program exists merely as a certification that a program matches its specification and it does not play a role as part of a broader network of concepts. Most proofs of program correctness are also quite dull. They typically have a very repetitive structure. Rather than using ingenious tricks, they enumerate a large number of fairly obvious cases.

The point made by De Millo, Lipton and Perlis is that mathematicians gain confidence about mathematical proofs thanks to the rich informal social processes of actual mathematics. Since no comparable social processes can take place among those who work on proving programs correct, formal mathematical approach to programming can never actually eliminate program errors. Although the authors focused merely on the confidence that we can have in the correctness of programs, De Millo was also aware of the idea by philosopher of science, Imre Lakatos,[104] that mathematical knowledge is built through a process of proofs and refutations. The authors knew that if computer

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

scientists treat proofs merely as certifications, they lose an invaluable force that shapes knowledge in mathematics.

Although De Millo, Lipton and Perlis were all mathematicians by training and inclination, they did not uncritically accept the assumption of the mathematical culture of programming that programs are mathematical entities like any other. They believed that programs can be studied using mathematical methods, but accepted that their structure is sufficiently different from ordinary mathematical theorems. Their paper was first presented at the ACM Symposium on Principles of Programming Languages (POPL) conference and a revised version was later published in *Communications of the ACM*.[105] The responses to the two publications show that questioning formal proofs about programs touches on a topic that different cultures treat very differently. Many of the positive letters published in a follow-up issue of the *Communications of the ACM* came from readers working in industry. One reader thanked authors for their wisdom, fairness, style, rigour, and wit while another reader thanked the editors for giving him the first article he enjoyed since joining the ACM.[106]

De Millo later reflected[107] that the three of them likely became spokespeople for computer industry practitioners who felt that academics were pushing solutions that were not going to work for them. To one manager, the article explained why he 'should not look for development of formal proof of the "correctness" of our programs' while another reader argued that he 'cannot recall a single instance in which a proof of a program's correctness would have been useful'. He argued that actual bugs are easy to discover through testing; more serious errors are errors in specification that would not be caught by proofs. As a better approach, the reader points to a review of software engineering techniques, including the Chief Programmer Team approach.

Given their background, De Millo, Lipton and Perlis cannot be clearly associated with a single culture of programming. All three of them have strong mathematical backgrounds, but their view of programs and programming seems more aligned with one that I associate with the engineering culture of programming. They recognise the irreducible complexity of computer programs as well as fundamental human limitations, both of which are typically ignored by the mathematical culture of programming. They also contributed to or advocated for the use of testing in programming: Perlis in his 1968 keynote and De Millo in his subsequent research on mutation testing. Their critical paper was certainly appreciated by those favouring the emerging engineering perspective, but also those in favour of more rigid managerial method.

The proponents of the mathematical culture produced multiple responses that criticised De Millo, Lipton and Perlis. In a letter to the editor, Leslie Lamport pointed out that the recent work of Floyd, discussed earlier in this chapter, 'taught us that a program is a mathematical object' and likens computer scientists before Floyd to 'geometers before Euclid'. Lamport believes that a 'theorem either can or cannot be derived from a set of axioms' and that their correctness is not decided by a social process. In the letter, he also criticises the practice of *Communications of the ACM* of publishing algorithms without correctness proofs in the 'Algorithms' column and notes that the published 'Certifications' provide only a flawed social process. Edsger Dijkstra went even further and labelled the article 'A political pamphlet from the Middle Ages'.[108] Dijkstra does

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

not disagree that communication with colleagues is an essential component of the mathematical culture, but he argues that the authors 'give a complete caricature of program verification' and that how to prove program properties more elegantly is the subject of lively interchange in the field. Dijkstra's response thus does not defend a purely formal view of proofs as that of Lamport. His perspective is more appreciative of the human side, but he believes that human programmers should aspire to mathematical rigour and elegance.

Many historians have written about the debate triggered by the De Millo, Lipton and Perlis paper and tried to assess its significance. Donald MacKenzie[109] suggested that it likely contributed to the decrease of US government funding for formal verification, but computer scientists committed to the mathematical culture who responded to the paper typically disregarded the arguments it made. Yet, looking at the follow-up developments, it seems that the paper captured the issue that the mathematical culture was, perhaps only unconsciously and rather reluctantly, aware of. If the mathematical culture insisted on using conventional mathematical proofs, it seems likely that it would soon reach a dead end. Real-world software systems were simply too complex for this.

I suggest that this is a case where the pluralism of programming with its multiple cultures came to the rescue. Rather than stagnating and eventually falling out of interest, the work on program verification received new impetuses, coming from two other cultures. The resulting two developments were not a direct response to the critique, but they both address the lack of social processes surrounding program proofs. The managerial Cleanroom methodology explicitly creates a new social process, while the engineering work on mechanised proofs replaces the need for human checking with machine checking.

2.10 Computing and Cultures of Proving

The first forays into proofs of program correctness, actioned in the mathematical culture in the 1960s, focused on simple programs and algorithms. This early work understood proofs as largely formal proofs that are actioned and checked by humans. The two examples involving the factorial function, which I discussed earlier, illustrate the approach. They were both done 'by hand' and consisted of a sequence of fairly direct applications of axioms and derivation rules. However, other ways of thinking about proofs about computer programs started to appear in the 1970s and gained prominence in the 1980s.[110]

Some of those working in the industry were aware of the developments arising from the mathematical culture of programming and attempted to adapt them to assist with large scale software development. We already saw one example. The Chief Programmer Team (CPT) methodology adapted the idea of structured programming, shifting focus from structuring programs to structuring teams. The methodology was developed by Harlan Mills, who had a background in mathematics and worked as a research fellow at IBM. With a PhD in game theory, but an industry job, Mills was perhaps the best person to import ideas from the mathematical culture into the world of applied

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

software development. Indeed, Mills soon realised that CPT only addressed one part of the problem. The quality of the resulting software still depended primarily on the skills of the chief programmer. Making sure that a software system or a software component correctly implements its specification thus became an even more important issue. As Mills later put it, clearly showing his appreciation for mathematical methods, 'software engineering without mathematical verification is no more than a buzzword'.[111]

To address the challenge of quality control, Mills proposed an extension of the CPT methodology that he called Cleanroom Software Development.[112] The approach took the mathematical idea of proving programs correct, but adapted it to the managerial context. It also took further an analogy with the surgical team that Mills used when designing the Chief Programmer Team structure. Like the software development team in CPT, a surgical team has a single lead surgeon, supported by a group of assistants. As the 'Cleanroom' name suggests, the new methodology adds a controlled environment, akin to the operating theatre used by surgeons. In the Cleanroom methodology, systems are built by following a detailed specification. A programmer is required to review the conformance to the specification rigorously, first to convince themselves and then to convince other team members. 'It is obvious' is an acceptable proof if everyone agrees. The methodology departs from the earlier understanding of proofs in the mathematical culture in that it admits rigorous but informal proofs, but it shares the property that the process is fundamentally human. The required degree of rigour is, however, a managerial choice and to ensure correctness of the final product, the Cleanroom methodology employs statistical quality control, measuring 'reliability over a probability distribution of usage scenarios'.

The development of the Cleanroom methodology was not a reaction to the De Millo, Lipton and Perlis critique, but it inadvertently addresses the issue that the critique raises, that is, the lack of social processes required for checking the correctness of a proof. Convincing colleagues that a software component matches its specification in the Cleanroom methodology may be as tedious as constructing a formal mathematical proof about non-trivial algorithm. However, the social structure needed to make the proof trustworthy is recreated through business organisation. The enlightened managers who choose to follow the Cleanroom method recognise that the time spent while reviewing tedious conformance to the specification will save time and money later and will require their employees to rigorously do so.

A different answer to the problem of social processes and program proofs was suggested long before the critique by De Millo, Lipton and Perlis. In the 1963 manifesto of the Algol research programme, John McCarthy wrote that 'one should prove that [a program] meets its specifications, and this proof should be checked by a computer program'.[113] McCarthy himself published a paper on computer programs for checking mathematical proofs and such proof checkers later became known as mechanical theorem provers.[114] The word prover might be misleading. A mechanical theorem prover is not expected to derive a proof automatically. This was, and remains to be, an unrealistic expectation. It is merely used to check a proof written by a human in a formal and often exasperatingly detailed way. In other words, 'a mechanical theorem prover does not act as an oracle that certifies bewildering arguments for inscrutable reasons but

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

as an implacable sceptic that insists on all assumptions being stated and all claims justified'.[115]

In the 1950s and 1960s, however, many believed that computers would be able to do more than to just check proofs written by humans. The optimism about Artificial Intelligence fuelled belief that computers would soon be able to prove complex theorems automatically. The earliest work in this area focused on classic mathematical theorems. The Logic Theorist, created by Allen Newell and Herbert Simon in the 1950s,[116] used heuristics to automatically derive mathematical proofs and they were able to prove 38 of the first 52 theorems in Chapter 2 of the *Principia Mathematica*. The work on provers for mathematical theorems prepared the ground for research on theorem provers working with proofs about programs. In 1971, Robert S. Boyer and J. Strother Moore in Edinburgh developed a system later known as the Boyer–Moore theorem prover, which was designed to automatically prove properties of programs in a variant of LISP. This prover was not able to complete non-trivial proofs without a stroke of luck or detailed human guidance, but it was a pioneering system that inspired follow-up research. This includes the ACL2 prover, which was developed in the 1990s as an 'industry strength'[117] version of the Boyer–Moore prover. I mentioned ACL2 in Chapter 1, because a conference on this system featured a talk about verifying financial systems with a reference to the Knight-Capital bug.

Another influential theorem prover designed for checking proofs about programs in the early 1970s was the LCF prover introduced by Robin Milner. The LCF was motivated by the aim to prove the correctness of programs such as compilers.[118] To simplify construction of LCF proofs, Milner's group developed a meta-language (ML), which could be used to write procedures that construct proofs. The idea is that humans should not have to write all steps of the proof. They can, instead, write programs that generate the steps. The programs to generate proofs are, in turn, checked to be correct using types. I will explain how exactly this is done when discussing types in Chapter 5, but the powerful combination gave rise to a family of interactive theorem provers that are still popular today. The ML language soon took a new form as a stand-alone programming language that influenced modern languages including OCaml and F#.

Systems that would automatically prove the correctness of a given program still remain elusive today, but work on theorem provers and other program verification tools remains an active field. On the one hand, semi-automatic mechanical theorem provers are able to do more work automatically. On the other hand, fully automatic tools are able to detect an increasing number of potential bugs. In both cases, the developments rely as much on theoretical innovations as on large-scale engineering efforts. A crucial theoretical development that enabled numerous modern verification tools is separation logic, which appeared in the early 2000s and extended Hoare triples with a mechanism for reasoning about data structures.[119] Practical tools that implement the idea, such as the Infer Static Analyser developed at Facebook, are complex systems. For example, the authors of Infer describe it as a 'general analysis framework which is an interface to the modular analysis engine which can be used by other kinds of program analyses' with instantiations 'for security, concurrency and in other domains'.[120]

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

The critique of proofs about programs by De Millo, Lipton and Perlis came at an interesting point in the history of the mathematical culture of programming. It received a heated response, perhaps because it questioned the central assumption of the culture that computer programs are mathematical entities like any other. At the same time, many were soon about to realise that formal proofs about programs done by hand were not going to scale up to prove interesting properties of large software systems. Donald MacKenzie uses the term cultures of proving to characterise the different approaches to program proofs.[121] The early mathematical proofs about programs were manual and formal. The Cleanroom methodology utilises proofs that are manual and rigorous, but informal. Finally, theorem provers result in proofs that are formal, but done semi-automatically. As also pointed out by MacKenzie, the different cultures of proving 'mostly coexist peacefully', although they may sometimes clash about 'presuppositions and preferences'.

The two developments discussed in this section are interesting, because they can be also seen as the results of interaction between different cultures of programming. The Cleanroom methodology takes the mathematical idea of rigorous program analysis. It keeps the mathematical rigour, leaves out the full formality, but surrounds it with a managerial team structure. The fact that it was developed by a trained mathematician working in industry is also telling. By contrast, the typical approach of the engineering culture is to develop a new tool or a method to assist programmers. Mechanised theorem provers can be seen as the result of a meeting between the mathematical and engineering culture of programming.

Mechanised proofs also quietly transformed what is considered as a proof. Mechanised proofs are no longer the complex human constructs that mathematicians know as proofs. They are computer representations of sequences of propositions where the prover can derive each proposition from earlier sentences, axioms or assumptions. This notion is close to the notion of proof in logic, with the caveat that we are now talking about computer representations. The mathematical culture of programming gradually adopted mechanised proofs as the hallmark of mathematical computer science research. In doing so, it began diverging from its pure mathematical origins where many mathematicians remain cautious about the use of computers in proofs.[122]

The lack of social process and the complexity of definitions were not the only challenges for proofs of programs that the mathematical culture of programming had to face. Another difference between proofs about programs and proofs in mathematics led to an equally heated debate in the mathematical culture of programming a decade later.

2.11 Fundamental Limits of Program Proofs

By the end of the 1980s, the techniques and tools of the mathematical culture became powerful enough to prove properties of non-trivial systems. This included both computer programs and hardware devices that were increasingly also modelled formally. By this point, the use of mechanical theorem provers became commonplace. What may

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

still have looked like an influence of the engineering culture a decade earlier became fully integrated into the mathematical culture. A subsequent version of the Boyer–Moore theorem prover mentioned above was, for example, used for verifying properties of a small multitasking operating system kernel, a low-level assembly language and aspects of a microcoded CPU.[123]

The adoption of mechanical theorem provers and their use for proving properties of software and even hardware with direct real world effects did not trigger an immediate reflection on the nature of program proofs. Many still saw a program as a mathematical entity and a program proof as a formal mathematical proof like any other. Yet, some practitioners started to realise that there are limits to what one can prove about a program that runs on a real computer and interacts with its environment. A large verification effort that illustrates this was the Software Implemented Fault Tolerance (SIFT) system, which was a flight control system designed for critical functions, such as the 'control of a dynamically unstable aircraft, designed for high energy efficiency'.[124] Boyer and Moore, who worked on the verification of a small but critical part of the system, pointed out that a proof could only be completed if they also had a precise specification of how the underlying microprocessor executes instructions and also the timing of such execution. However, the timing of execution is a property of the real hardware that a typical mathematical model of a microprocessor is likely to ignore. In some cases, such details may not even be deterministic. The timing can, for example, depend on the temperature. The issue is that programs are not just mathematical entities. They are also executed on real hardware.

A similar critical reflection was written by Brian Cantwell Smith, who worked at Xerox PARC. I will discuss his work on programming systems in Chapter 3, but Smith was also a founder of Computer Professionals for Social Responsibility, an organisation opposing the use of computers in warfare. Smith[125] starts with a story of the American Ballistic Missile Early-Warning System (Figure 2.7) that mistook the rising moon for a ballistic missile attack in 1960 and asks whether such a system, if it was formally verified, should be trusted with launching a counter-attack. According to Smith, the answer is no, because 'there are inherent limitations to what can be proved about computers and computer programs'. The core of his argument is that computer programs always work with a model of the real world, which inevitably ignores some aspects of reality. Computer programs 'are not, as some theoreticians seem to suppose, pure mathematical abstractions, living in a pure detached heaven'. They may ignore aspects of the real world, such as timing or the moon, and they trigger actions in the real world that escape the abstractions of the model.

Despite such limitations, Smith still finds formal verification valuable, just for a different reason. A proof of correctness is really a proof of the compatibility between two formal objects: a program and a specification. This is still useful, because a program and a specification are typically expressed in a different way. For example, a specification for a refrigerator will say that it is a device which maintains an internal temperature between 3 and 6 degrees. A program controlling the refrigerator has to specify how exactly this is done. Proving that a program satisfies a specification is

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003



Figure 2.7 Command post for NORAD (North American Aerospace Defense Command) operations, including the Command's surveillance and warning sensors around the globe, taken about 1982. (Source: National Archives, Still Pictures Branch, RG 306-PSE, box 79)

thus a useful added assurance, even if it cannot guarantee anything about an actual refrigerator.

The observation by Boyer and Moore was published as part of a 600-page long interim report submitted to NASA,[126] which was the project funder, while the critical reflection by Smith was initially published as a technical report. Consequently, neither of the two triggered a heated debate in the community. The controversy came in a later paper by a philosopher James Fetzer, which was published by the *Communications of the ACM*.[127] Fetzer became aware of the critique of program proofs by De Millo, Lipton and Perlis. He thought that the lack of social processes was not a major issue, but believed that there is a more fundamental problem, the one that Boyer, Moore and Smith all alluded to earlier. Although both programs and theorems about them are syntactic structures, a program is not a pure mathematical entity. The meaning of a program is that it controls a physical machine. Consequently, as Fetzer argued in more philosophical terms, the very idea of program verification is nonsensical, because it confuses mathematical a priori knowledge with empirical a posteriori knowledge.

According to Fetzer, program verification can mean two things depending on how we treat the nature of the machine that executes the program. If we see the machine as an abstract entity then program verification is a legitimate mathematical activity, but the proof is about a mathematical model, rather than the actual running programs. If we see the machine as an empirical description of a physical system, then we obtain

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

possibly useful, more or less reliable, empirical knowledge about a program, but not a mathematical proof.

Fetzer's critique provoked a fierce response from the verification community. In a letter to the editors of *Communications of the ACM*,[128] 10 prominent computer scientists claimed that his paper 'is not a serious scientific analysis of the nature of verification' and accused the editors of *Communications of the ACM* for 'abrogat[ing] their responsibility, to both the ACM membership and to the public at large' by publishing 'the ill-informed, irresponsible, and dangerous article'. The response does not, in fact, object to the main point raised by Fetzer. It argues that Fetzer attacks a parody of formal verification when he assumes that 'the purpose of program verification is to provide an absolute guarantee of correctness with respect to the execution of a program on computer hardware' and the authors give a number of references 'in which the limits of proofs are carefully drawn'.

One of the references cited in the letter was a current technical report by Avra Cohn, who was very well aware of this issue. Cohn worked at the University of Cambridge and was a part of a team formally verifying the VIPER microprocessor. VIPER was developed at the Royal Signals and Radar Establishment, a research establishment within the UK Ministry of Defence. It was advertised as 'the first commercially available microprocessor with both a formal specification and a proof that the chip conforms to it'.[129] To Cohn, this advertisement was dangerously misleading. In her reflections on the work, she carefully points out that 'a device can be described in a formal way, and the description verified; but …there is no way to assure the accuracy of the description. Indeed, any description is bound to be inaccurate in some respects, since it cannot be hoped to mirror an entire physical situation'.[130] In other words, even if we formally verify a system at all its levels, ranging from the low-level chip to high-level programming language, we are still only working with abstract models. By no means does this make formal verification useless. It just means that it should not have a unique status among other methods for building correct systems.

As with the critique by De Millo, Lipton and Perlis, the paper by Fetzer identified an issue that was, perhaps reluctantly or subconsciously recognised by at least some members of the mathematical culture of programming. According to one commenter who, nevertheless, accuses Fetzer of doing 'a disservice to the cause of the advancement of the science of programming',[131] the idea that no deductive argument can guarantee that a system will work perfectly is a fact obvious to practically everyone.

The likely reason for the raging controversy caused by the critique is that Fetzer questioned the idea that programs are mathematical entities, an assumption that is at the core of the mathematical culture of programming. In the author's response published in a later issue of the *Communications of the ACM*, Fetzer tries to support his argument that the limitations are not obvious to practically everyone in the community. He does this by quoting a number of key papers in the field, such as Hoare, who states that 'programming is an exact science in that all the properties of a program and all the consequences of executing it can, in principle, be found out from the text of the program itself by means of purely deductive reasoning'.[132] As Fetzer points out, such phrasing

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

is clearly in contrast with the idea that limitations of formal verification are obvious to most computer scientists.

I believe that there are two possible explanations of this discrepancy and that both are a part of the truth. First, the ideas about program proofs in the mathematical culture have evolved over the 20 years since Hoare's pioneering work, quoted by Fetzer, appeared. Mechanical theorem provers that draw from the engineering culture did not just provide new tools, but they also imported a more engineering-oriented thinking into the mathematical culture of programming. Second, the idea that programs are mathematical entities was and remains an accepted, but idealised assumption. It provides justification for the formal approach, but does not shape how it is practically done. As such, it is more a rhetorical device that is likely often used without elaborate consideration.

2.12 Proofs for Machines and Proofs for Humans

The two contentious debates that I just discussed share a number of characteristics. In both cases, they point to an issue that is related to core assumptions about what programs are. They also make explicit an issue that those working within the mathematical culture do not publicly acknowledge and, perhaps, are not even consciously aware of. At the same time, both critiques came at a time when the mathematical culture was already transforming how it works. In both cases, the transformation was adopting approaches from another culture of programming. In the case of De Millo, Lipton and Perlis, two ways of replacing the usual social processes of mathematical proofs appeared. The short-lived Cleanroom methodology created an explicit social process through a managerial structure. The now widely accepted engineering-inspired approach based on mechanised theorem proving replaces human checking with machine checking.

In the case of Fetzer, the complications with proving properties of software that controls physical systems were already becoming clear. A few observers started writing about those problems before Fetzer, although in a more reserved way. At the same time, in the 1980s, two subcultures were already appearing within the mathematical culture. Those who wanted to produce as complete proofs as possible treated the problem of program proofs increasingly as an engineering issue. Those who wanted to gain pure mathematical insights treated the problem as a formal one. The latter required working with minimal, formally tractable models and it became the cornerstone of programming language theory, whereas the former became a mainstream approach to software verification.

The first development started with the use of mechanical theorem provers, which are essential for ensuring that all cases in a long and tedious proof about complex systems are covered. A proof constructed using a mechanical theorem prover is really a part of the program. Just like ordinary mathematical proofs are written to convince another human that it would be, in principle, possible to construct a finite sequence of propositions representing the formal proof, a mechanised proof is essentially a program

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

that instructs a machine to construct a computer representation of such a sequence of propositions. Over time, the specification of a correctness property and its proof displaced the program itself as the main object that needs to be created. In modern theorem provers, the programmer only needs to write their proofs and the executable code of a corresponding program is generated based on the source code of the proof. A good recent example using this method is the formally verified CompCert compiler,[133] where a single program represents both the compiler and its correctness proof. In such systems, the proof becomes a technical artifact that is clearly distinct from a formal deductive mathematical proof.

The second development remained more faithful to traditional mathematics. It uses severely simplified mathematical models of programming languages or hardware devices. It is clear that proofs about such models do not provide guarantees about the reality. However, it makes it possible to reason about formal mathematical programs using standard mathematical methods, such as a hand-written proof on a blackboard. This way of working does not guarantee correctness of an actual system that is modelled, but the model may reveal tricky corner-cases that are equally tricky for the real implementation. A failure to prove correctness often points to an issue that also exists in the actual system.

The idea of using severely simplified mathematical models in proofs about programs and programming languages appeared well before the 1980s. It dates back to McCarthy's 1964 Microalgol, which is 'a very small subset of Algol' where 'all the difficult aspects of Algol are eliminated'.[134] Although the work on small mathematical models is rooted in the same mathematical culture as the work resulting in complex mechanised proofs, the language and presentation used in talking about them has gradually became more distinct, possibly highlighting the fact that the object of study is a mathematical object rather than an actual program or a programming language.

The development of two subcultures with two kinds of proof was not a response to Fetzer's critique. It was clearly already happening when the critique was published. Yet, it shows that the issue at the heart of Fetzer's critique is something that has been shaping the mathematical culture. Again, the developments arguably benefited from the pluralism of the field of programming. One of the two directions was influenced by the engineering culture, whereas the other adopted a more purist mathematical approach.

2.13 Mathematisation of Programming

To a contemporary computer scientist, the general idea of using mathematical methods for the analysis of programs and their correctness seems unproblematic. Their only lament is usually that the use of mathematical methods in software industry is not more commonplace. Many academic papers and grant applications that focus on developing new mathematical theories of programs and new tools motivate the work by claiming that only rigorous mathematical approach to programming can guarantee that software will work correctly. History shows that this is a simplistic view. On the one hand, the

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

view faces many limitations and the very nature of mathematical methods changes in response. On the other hand, mathematical perspective often reshapes programming in ways that go well beyond checking program correctness.

Despite its many early links to mathematics, programming started as a craft learned through practice. The search for better ways of programming meant different things to different people. Hackers were looking for better tools to control the machine, managers were looking for better ways of organising software production and mathematicians started to think about programming in terms of algorithms. The three cultures came together in the late 1950s and collectively established the idea of a programming language as an independent entity.

The birth of the idea of a programming language did not lead to a single unified way of thinking about programming. Programming was a pluralistic discipline before the 1950s and remains a pluralistic discipline today. The different cultures that I identify in this book each evolve, but they keep a surprisingly stable set of core assumptions and methods. For example, the hacker culture that we encountered in the early days of programming emphasises thorough understanding of and direct engagement with computers. The culture did not disappear once programming languages and methodologies inspired by structured programming gave programming a more rigid structure. As we will see in Chapter 3, the hacker culture quickly came back to prominence once computers became more interactive and it was again possible to work with a machine directly and acquire craft-like skills.

The story of the mathematisation of programming is centred around the mathematical culture of programming and its interactions with other cultures. Those often provide a useful impetus for moving forward in a new direction. The birth of programming languages was motivated by managerial needs and relied on previous hacker knowledge. Yet, it gave the mathematical culture a new way of looking at programs and made it possible to treat them as formal entities written in a formal language. The development of structured programming was motivated by the aim of writing better programs and is perhaps best aligned with the engineering culture, but it was also closely connected with the development of compositional reasoning methods in the mathematical culture. The idea also inspired new ways of managing the software development process.

Another kind of interaction occurs when one culture borrows general ideas or methods from another culture. The two times this happened in the story told here, the borrowing took place at a time of controversial debates, sparked by the questioning of basic assumptions of the mathematical culture. When De Millo, Lipton and Perlis pointed out that proofs of programs were different from mathematical proofs and lacked the necessary social processes, the mathematical culture was already in the process of adopting mechanical theorem provers that ready to replace human checking with machine checking. I suggest that this move resulted from the borrowing of methods from the engineering culture. The fact that the developments happened at the same time as the controversy may be just coincidence, but it is also possible that both were a reaction to problems in the mathematical culture that were slowly emerging on the horizon. Traditional mathematical ways of writing proofs worked for small programs, but it would not work for more complex software. This triggered reflection in the

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

community and it also motivated its members to look for ideas outside of their own culture.

The history of the mathematisation of programming and the controversies that we encountered in this chapter has been told in detail in a number of excellent accounts that I have been drawing on in this chapter.[135] What is new in my retelling of the story is the focus on pluralism within the discipline. If we look at how concepts like programming language, structured programming or mechanised proofs came to be, we can see that this is a product of rich interactions between different groups of individuals who have different assumptions and different methods of working.[136] Although different cultures of programming do not share assumptions and use different methods, they do not form incommensurable bodies of knowledge. They can exchange ideas through individuals who sometimes pass between multiple cultures, but they can also exchange ideas through shared technical concepts such as programming languages, programming constructs and programming practices.

Notes

1 Reflecting on programming in the 1950s, Dijkstra (1972) asks '[Where] was the sound body of knowledge that could support it as an intellectually respectable discipline?'
2 The establishment of algorithm as a basic concept of computer science is documented by Ceruzzi (1988) and further discussed by Ensmenger (2012), while Mahoney (1992) discusses the search for the mathematical foundations of computing.
3 Ensmenger (2012) provides a detailed account of the 1950s labour crisis.
4 The roots and effects of gender biases in computing have been documented in various contexts by Hicks (2010, 2017), Ensmenger (2010) and Abbate (2012).
5 The external form allowed programmers to use relative addressing (Wilkes et al., 1951), while interpretive routines made it possible to use pseudo-instructions, e.g., for working with floating point numbers (Campbell-Kelly, 1980).
6 Mahoney (1997).
7 See the work of De Mol and Bullynck (2018) for critical reflection on how the historical story of computer science has been constructed.
8 We return to the media-theoretic perspective in Chapter 3. The perspective seeing compilation as a translation process is documented by Nofre et al. (2014).
9 Ensmenger (2012); Priestley (2011).
10 Leroy (2009).
11 The theorem refers to the result by Böhm and Jacopini (1966). As noted by Harel (1980), the result in the paper is of a more technical nature and has been misinterpreted in the same way as done here by *Pythagoras*.
12 MacKenzie (2004).
13 See DeMillo et al. (1977). A detailed account of the history of program proofs can be found in the work of MacKenzie (2004).
14 We return to this topic when discussing the 'Cleanroom' methodology later in this chapter.
15 The history of ENIAC programming has been documented by historians, especially Haigh et al. (2016), and also by Bartik (2013) herself. There are disputes about some aspects of the history of the early ENIAC programming work. I generally follow the first-hand account of Jean Jennings Bartik, but Haigh et al. (2016) provide a more neutral perspective.
16 The story of six of them has been told by Kleiman (2022).
17 https://ftp.arl.army.mil/ftp/historic-computers/.
18 Ensmenger (2012).

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

19 John Backus, quoted by Ensmenger (2012).
20 Priestley (2011).
21 Wilkes (1985).
22 Ensmenger (2012).
23 Polanyi (1958) mentions common law as a system that supports tacit knowledge. This bears some similarity to the HAKMEM report (Beeler et al., 1972) that I mentioned as an example of a document resulting from the hacker culture, which is a large collection of often very specific cases.
24 Documented by Ensmenger (2012).
25 Dijkstra (1972).
26 Ensmenger (2012).
27 Knuth (1968).
28 Knuth (1968).
29 Ensmenger (2012).
30 Here, the field of programming follows the example of physics, whose preference for theory over experimental work has been documented by Hacking (1983).
31 Many of those clashes and struggles are documented by Ensmenger (2012).
32 The early history of programming systems has been documented by Priestley (2011). The birth of programming languages is told by Nofre et al. (2014), in a paper whose title inspired the heading of this section.
33 Univac (1957).
34 www.computerhistory.org/collections/catalog/102646140.
35 Nofre et al. (2014).
36 Backus et al. (1954).
37 Cited in Nofre et al. (2014).
38 Nofre et al. (2014).
39 Quoted by Nofre et al. (2014).
40 The preliminary (International Algebraic Language) report by Perlis and Samelson (1958) and the final report (Algol 60) by Backus et al. (1960) are the primary sources. The development of Algol has been documented in a first-hand account edited by Perlis (1978) and Naur (1978).
41 Star and Griesemer (1989).
42 Perlis (1978).
43 First described by Bosak et al. (1962). First-hand account of the history has been written by Sammet (1978).
44 First-hand account of the history has been written by McCarthy (1978).
45 Priestley (2017).
46 Backus et al. (1960).
47 Backus et al. (1963).
48 Priestley (2011) refers to research paradigms in the sense of Kuhn (1962), although with some caution, because the Algol paradigm exists in parallel with other ways of thinking and participants can exchange ideas with others. This is what I attempt to capture with the notion of 'culture of programming' in this book. Many of the pioneering works of the Algol paradigm can be found in a valuable collection by Colburn et al. (2012).
49 McCarthy (1963).
50 Ensmenger (2012).
51 Habermann (1969).
52 Naur (1966).
53 Floyd (1967).
54 Hoare (1969).
55 A first-hand account of the history can be found in Hoare (1981).

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

56 Hoare (1971).
57 McCarthy (1963).
58 Hoare (1971).
59 One cannot avoid an analogy with the seventeenth-century reports of experiments conducted by the members of the Royal Society, which required a confirmation of a trustworthy eyewitness, as discussed for example by Shapin and Schaffer (2011).
60 I am indebted to Mark Priestley for this observation, as well as for another example of engineering influence, which is the notion of 'language maintenance', i.e., the long-term support for the evolution of a language as an entity separate from a particular compiler.
61 Conger (1962).
62 The story can be found primarilly in their PhD thesis (Astarte, 2019), but their earlier publications are also invaluable sources (Jones and Astarte, 2016; Astarte and Jones, 2018).
63 Backus et al. (1963).
64 Backus et al. (1963).
65 Lakatos (1976).
66 McCarthy (1964).
67 Hoare (1969).
68 Mosses (1974).
69 Christopher P. Wadsworth, cited in Astarte and Jones (2018).
70 Lauer (1968).
71 Allen et al. (1972).
72 Henhapl and Jones (1978).
73 As reported by Astarte and Jones (2018), Mosses was working on a tool that would generate compilers from formal language specifications. His Algol description was written in a style used by the tool, but he never actually used the Algol description as input for his system.
74 Lindsey (1996a).
75 Hoare (1981).
76 Haigh (2010).
77 Gries (1978).
78 http://homepages.cs.ncl.ac.uk/brian.randell/NATO/N1968/index.html.
79 Ensmenger (2012).
80 Haigh (2010).
81 Pelaez Valdez (1988).
82 Buxton et al. (1970).
83 Dijkstra (1968).
84 The working paper appeared in Buxton et al. (1970) and in a longer, privately circulated version Dijkstra (1970). Author's later reflections on the history appear in Dijkstra (2002).
85 Böhm and Jacopini (1966).
86 Harel (1980).
87 Plauger (1993).
88 See also historical reflections on the `goto` statement by Leavenworth (1972).
89 Ensmenger (2012).
90 Ensmenger (2012).
91 Reported in the *Datamation* magazine by Baker and Mills (1973).
92 The Team Topologies model by Skelton et al. (2019) can be seen as a new take on the very same idea.
93 Dijkstra (1980).
94 Dijkstra (1988).
95 McCracken (1973).
96 Discussed by Slayton (2013).
97 We can thus perhaps also view programming languages as trading zones, in the sense introduced by Galison (1997).

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003

98 Concrete technical concepts play a role similar to experiments in physics which, can also 'have a life of their own.' Hacking (1983).
99 https://amturing.acm.org/award_winners/perlis_0132439.cfm.
100 Naur et al. (1969).
101 Buxton et al. (1970).
102 Documented by MacKenzie (2004).
103 DeMillo et al. (1977).
104 Lakatos (1976).
105 DeMillo et al. (1977) and De Millo et al. (1979).
106 Discussed by MacKenzie (2004).
107 Quoted by MacKenzie (2004).
108 Dijkstra (1978).
109 MacKenzie (2004).
110 The different ways of thinking about proofs are documented by MacKenzie (2005), which also inspired the title of this section.
111 Mills et al. (1987).
112 Introduced by Dyer and Mills (1981). The methodology has been extensively discussed by MacKenzie (2004).
113 McCarthy (1963).
114 McCarthy (1962).
115 Rushby and Von Henke (1993), quoted byMacKenzie (2004).
116 Newell and Simon (1956).
117 Moore (2019).
118 The system was first presented by Milner (1972); work on proofs is discussed later in Milner (1979).
119 A first-hand account on the development has been written by Pym et al. (2019).
120 https://fbinfer.com/docs/about-Infer.
121 MacKenzie (2005).
122 For mathematician's reflections on proofs, see for example the collection by Gold and Simons (2008).
123 The applications are discussed by Boyer and Moore (1988).
124 Goldberg et al. (1984), disucssed by MacKenzie (2004).
125 Smith (1985).
126 Boyer and Moore (1983).
127 Fetzer (1988).
128 Ardis et al. (1989).
129 Quoted in MacKenzie (2004).
130 Cohn (1989).
131 Pleasant (1989).
132 Hoare (1969).
133 Leroy (2009).
134 McCarthy (1964).
135 Especially the history of early programming documented by Priestley (2011), computer personnel and power struggles by Ensmenger (2012), history of proofs about programs by MacKenzie (2004), history of software engineering as seen through military software by Slayton (2013) and the history of the programming discipline by Tedre (2014). The book edited by Colburn et al. (2012) collects many of the papers involved in critical reflections on formal program verification and proofs of programs.
136 I use the term *cultures of programming* to tell the story, but the concept is similar to the idea of *systems of practice* introduced by Chang (2012) when analysing the history of chemistry.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 12 Nov 2026 at 07:35:21, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009492379.003