3 Von Neumann's Draft, Electronic Brains, and Cognition

Many things have been written regarding computer programming—often, I believe, in problematic ways. To avoid getting lost in this abundant literature, it is important to start this chapter with an operational definition of computer programming on which I could work and eventually refine later. I shall then temporally define computer programming as the situated activity of inscribing numbered lists of instructions that can be executed by computer processors to organize the movement of bits and to modify given data in desired ways. This operational definition of computer programming puts aside other practices one may sometimes describe as "programming," such as "programming one's wedding" or "programming the clock of one's microwave."

If I place emphasis on the practical and situated aspect of computer programming in my operational definition, it is because important historical events have progressively set it aside. In this first section that draws on historical works on early electronic computing projects, we will see that once computer systems started to be presented as input-output instruments controlled by a central unit—following the successful dissemination of the so-called von Neumann architecture—the entangled sociotechnical relationships required to make these objects operate in meaningful ways had begun to be placed in the background. If electronic computing systems were, in practice, intricate and highly problematic sociotechnical processes, von Neumann's modelization made them appear as functional devices transforming inputs into outputs. The noninclusion of practices—hence their *invisibilization*—in the accounts of electronic computers further led to serious issues that suggested the first academic studies of computer programming in the 1950s.

A Report and Its Consequences

One cornerstone of what will progressively be called "von Neumann architecture" is the *First Draft of a Report on the EDVAC* that John von Neumann wrote in a hurry in 1945 to summarize the advancement of an audacious electronic computing system initiated during World War II at the Moore School of Electrical Engineering at the University of Pennsylvania. As I believe this report has had an important influence on the setting aside of the practical instantiations of computer systems, we first need to look at the history and dissemination of this document as well as the world it participated in enacting.

World War II: An Increasing Need for the Resolution of Differential Equations

An arbitrary point of departure could be President Franklin D. Roosevelt's radio broadcast on December 29, 1940, that publicly presented the United States as the main military supplier to the Allied war effort, therefore implying a significant increase in US military production spending.[1] Under the jurisdiction of the Army Ordnance Department (AOD), the design and industrial production of long-distance weapons were obvious topics for this war-oriented endeavor. Yet for every newly developed long-distance weapon, a complete and reliable *firing table* listing the appropriate elevations and azimuths for the reaching of any distant targets had to be calculated, printed, and distributed. Indeed, to have a chance to effectively reach targets with a minimum of rounds, every long-distance weapon had to be equipped with a booklet containing data for several thousand kinds of curved trajectories.[2] More battles, more weapons, and more distant shots: along with the mass production of weapons and the enrollment of soldiers capable of handling them, the US's entry into another world war in 1942 further implied an increasing need for the resolution of differential equations.

These practical mathematical operations—which can take the form of long iterative equations that require only addition, subtraction, multiplication, and division—were mainly conducted in the premises of the Ballistic Research Laboratory (BRL) at Aberdeen, Maryland, and at the Moore School of Electrical Engineering in Philadelphia. Hundreds of "human computers" (Grier 2005), mainly women (Light 1999), along with mechanical desk calculators and two costly refined versions of Vannevar Buch's differential

analyzer (Owens 1986)—an analogue machine that could compute mathematical equations[3]—worked intensely to print out ballistic missile firing tables. Assembling all of the assignable factors that affect the trajectories of a projectile shot from the barrel of a gun (gravity; the elevations of the gun; the shell's weight, diameter, and shape; the densities and temperatures of the air; the wind velocities, etc.)[4] and aligning them to define and solve messy differential equations[5] was a tedious process that involved intense training and military chains of command (Polachek 1997). But even this unprecedented ballistic calculating endeavor could not satisfy the computing needs of this wartime. Too much time was required to produce a complete table, and the backlog of work rapidly grew as the war intensified. As Campbell-Kelly et al. (2013, 68) put it:

> The lack of an effective calculating technology was thus a major bottleneck to the effective deployment of the multitude of newly developed weapons.

In 1942, drawing on the differential analyzer and on the pioneering work of John Vincent Atanasoff and Clifford Berry on electronic computing (Akera 2008, 82–102; Burks and Burks 1989) as well as on his own research on delay-line storage systems,[6] John Mauchly—an assistant professor at the Moore School—submitted a memorandum to the AOD that presented the construction of an electronic computer as a potential resource for faster and more reliable computation of ballistic equations (Mauchly [1942] 1982).[7] The memorandum first went unnoticed. But one year later, thanks to the lobbying of Herman Goldstine—a mathematician and influential member of the BRL—a meeting regarding the potential funding of an eighteen-thousand-vacuum-tube electronic computer was organized with the BRL's director. And despite the skepticism of influent members of the National Defense Research Committee (NDRC),[8] a $400,000 research contract was signed on April 9, 1943.[9] At this point, the construction of a computing system that could potentially solve large iterative equations at electronic speed and therefore accelerate the printing out of the firing tables required for long-distance weapons could begin. This project, initially called "Project PX," took the name of ENIAC for *Electronic Numerical Integrator and Computer*.

The need to quickly demonstrate technical feasibility forced Mauchly and John Presper Eckert—the chief engineer of the project—to make irreversible design decisions that soon appeared problematic (Campbell-Kelly

et al. 2013, 65–87). The biggest shortcoming was related to the new computing capabilities of the system: If delay-line storage could potentially make the system add, subtract, multiply, and divide electric translations of numbers at electronic speed, such storage prevented the system from being instructed via punched cards or paper tape. This common way of both temporally storing data and describing the logico-arithmetic operations that would compute them was well adapted for electromechanical devices, such as the Harvard Mark I that proceeded at three operations per second.[10] But an electronic machine such as the ENIAC that was supposed to perform five thousand operations per second could not possibly handle this kind of paper material. The solution that Eckert and Mauchly proposed was then to set up both data and instructions manually on the device by means of wires, mechanical switches, and dials. This choice led to two related impasses. First, it constrained the writable electronic storage of the device; more storage would have indeed required even bigger machinery, entangled wires, and unreliable vacuum tubes. Second, the work required to set up all the circuitry and controllers and start an iterative ballistic equation was extremely tedious; once the data and the instructions were laboriously defined and checked, the whole operating team needed to be briefed and synchronized to set up the messy circuitry (Campbell-Kelly et al. 2013, 73). Moreover, the passage from diagrams provided by the top engineers to the actual setup of the system by lower-ranked employees was by no means a smooth process—the diagrams were tedious to produce, hard to read, and error-prone, and the number of switches, wires, and resistors was quite confusing.[11]

Two important events made an alternative appear. The first is Eckert's work on mercury delay-line storage, which built upon his previous work on radar technology. By 1944, he became convinced that these items could be adapted to provide more compact, faster, and cheaper computing storage (Haigh, Priestley, and Rope 2016, 130–132). The second event is one of the most popular anecdotes of the history of computing: the visit of John von Neumann at the BRL in the summer of 1944. Contrary to Eckert, Mauchly, and even Goldstine, von Neumann was already an important scientific figure in 1944. Since the 1930s, he was at the forefront of mathematical logic, the branch of mathematics that focuses on formal systems and their abilities to evaluate the consistencies of statements. He was well aware of the works on computability by Alonzo Church and Alan Turing, with whom

he collaborated at Princeton.[12] As such, he was one of the few mathematicians who had a formal understanding of computation. Moreover, by 1944, he had already established the foundations of quantum mechanics as well as game theory. Compared with him and despite their breathtaking insights on electronic computing, Eckert and Mauchly were still provincial engineers. Von Neumann was part of another category: he was a scientific superstar of physics, logics, and mathematics, and he worked as a consultant on many classified scientific projects, with the more notable one certainly being the Manhattan Project.

Von Neumann's visit was part of a routine consulting trip to the BRL and therefore was not specifically related to the ENIAC project. In fact, as many members of the NDRC expressed defiance toward the ENIAC, von Neumann was not even aware of its existence. But when Goldstine mentioned the ENIAC project, von Neumann quickly showed interest:

> It is the summer of 1944. Herman Goldstine, standing on the platform of the railroad station at Aberdeen, recognizes John von Neumann. Goldstine approaches the great man and soon mentions the computer project that is underway in Philadelphia. Von Neumann, who is at this point deeply immersed in the Manhattan Project and is only too well aware of the urgent need of many wartime projects of rapid computations, makes a quick transition from polite chat to intense interest. Goldstine soon brings his new friend to see the project. (Haigh, Priestley, and Rope 2016, 132)

By the summer of 1944, it was accepted among Manhattan Project's scientific managers that a uniform contraction of two plutonium hemispheres could make the material volume reach critical mass and create, in turn, a nuclear explosion. Yet if von Neumann and his colleagues knew that the mathematics of this implosion would involve huge systems of partial differential equations, they were still struggling to find a way of defining them. And for several months, von Neumann had been seriously considering electronic computing for this specific prospect (Aspray 1990, 28–34; Goldstine [1972] 1980, 170–182).

After his first visit to the ENIAC, von Neumann quickly realized that even though the ENIAC was by far the most promising computing system he had seen so far, its limited storage capacity could by no means help define and solve the very complex partial differential equations related to the Manhattan Project.[13] Convinced that a new machine could overcome this impasse—notably by using Eckert's insights about mercury delay-line

storage—von Neumann helped design a new proposal for the construction of a post-ENIAC system. He moreover attended a crucial BRL board meeting where the new project was evaluated. His presence definitely helped with attaining the final approval of the project and its new funding of $105,000 by August 1944. The new hypothetical machine—whose design and construction would fall under the management of Eckert and Mauchly—was initially called "Project PY" before being renamed EDVAC for *Electronic Discrete Variable Automatic Computer*.

Different Layers of Involvement

The period between September 1944 and June 1945 is crucial for my adventurous story of the setting aside of computer programming practices. It was indeed during this short period of time that von Neumann proposed considering computer programs as input lists of instructions, hence surreptitiously invisibilizing the practices required to shape these lists. As this formal conception of electronic computing systems was not unanimously shared among the participants of both ENIAC and EDVAC projects, it is important at this point to understand the different layers of involvements in these two projects that were intimately overlapping. One could schematically divide them into three layers: the engineering staff, the operating team, and von Neumann himself.

The first layer of involvement included the engineering staff—headed by Mauchly, Eckert, Goldstine, and Arthur W. Burks—that was responsible for the logical, electronic, and electromechanical architectures and implementations of both the ENIAC and the EDVAC. The split of the ENIAC into different units, the functioning of its accumulators—crucial parts for making the system compute electric pulses—and the development and testing of mercury delay-line storage for the future EDVAC were part of the prerogatives of the engineering staff. It is difficult to see now the blurriness of this endeavor that was swimming in the unprecedented. But besides the systems' abilities to compute more or less complex differential equations, one crucial element the engineering staff had to conceive and make happen was a way to instruct these messy systems. In parallel to the enormous scientific and engineering problems of the different parts of the systems, the shaping of readable documents that could describe the operations required to make these systems *do* something was a real challenge: How, in the end, could an equation be put into an incredibly messy electronic system? In

the case of the ENIAC, the engineering staff—in fact, mostly Burks (Haigh, Priestley, and Rope 2016, 35–83)—progressively designed a workflow that could be summarized as such: assuming ballistic data and assignable factors had been adequately gathered and translated into a differential equation—which was already a problematic endeavor—the ENIAC's engineering staff would first have to transform this equation into a logical diagram; then into an electronic diagram that took into account the different unit as blocks; and then into another, bigger, diagram that took into account the inner constituents of each block. The end result of this tedious process—the final "panel diagram" drawn on large sheets of paper (Haigh, Priestley, and Rope 2016, 42)—was an incredible, yet necessary, mess.

This leads us to another layer that included the so-called operators—mainly women computers—who tried to make sense, correct, and eventually implement these diagrams into workable arrangements of switches, wires, and dials. Contrary to what the top engineers had initially thought, translating large panel diagrams into a workable configuration of switches and wires was not a trivial task. Errors in both the diagrams and the configurations of switches were frequent—without mentioning the fragility of the resistors—and this empirical "programming" process implied constant exchanges between high-level design in the office and low-level implementations in the hangar (Light 1999, 472; Haigh, Priestley, and Rope 2016, 74–83). Both engineers and operators were engaged in a laborious process to have ENIAC and, to a lesser extent, EDVAC produce meaningful results, and these computing systems were considered heterogeneous processes that indistinctly mixed problematic technical components, interpersonal relationships, mathematical modeling, and transformative practices.

Next to these two layers of involvement was von Neumann who certainly constituted a layer on his own. First, contrary to Mauchly, Eckert, Burks, and even Goldstine, he was well aware of recent works in mathematical logic and, in that sense, was prone to formalizing models of computation. Second, von Neumann was very interested in mathematical neurology and was well aware of the analogy between logical calculus and the brain as proposed by McCulloch and Pitts in 1943 (more on this later). This further made him consider computing systems as *electronic brains* that could more or less intelligently transform inputs into outputs (Haigh, Priestley, and Rope 2016, 141–142; von Neumann 2012). Third, if he was truly involved in the early design of the EDVAC, his point of view was that

of a consultant, constantly on the move from one laboratory to another. He attended meetings—the famous "Meetings with von Neumann" (Stern 1981, 74)—and read reports and letters from the top managers of the ENIAC and EDVAC but was not part of the mundane tedious practices at the Moore School (Stern 1981, 70–80; Haigh, Priestley, and Rope 2016, 132–140). He was thus parallel to, but not wholly a part of, the everyday practices in the hangars of the Moore School. Finally, being deemed one of the greatest scientific figures of the time—which he certainly was—his visits were real trials that required preparation and cleaning efforts. If he visited the hangars of the Moore School several times, he mainly saw the *results* of messy setup processes, not the processes themselves. A lot was indeed at stake: at that time, the electronic computing projects of the Moore School were not considered serious endeavors among many important applied mathematicians at MIT, Harvard, or Bell Labs—notably Vannevar Buch, Howard Aiken, and George Stibitz (Stern 1981). Taking care of von Neumann's support was crucial as he gave legitimacy to the EDVAC project and even to the whole school.

All of these elements certainly contributed to shaping von Neumann's particular view on the EDVAC. In the spring of 1945, while the engineering and operating layers had to consider this post-ENIAC computing system as a set of problematic relations encompassing the definition of equations, the adequate design of fragile electromechanical units, and back-and-forth movements between hangars and offices, von Neumann could consider it as a more or less functional object whose inner relationships could be modeled.

Despite many feuds over the paternity of what has later been fallaciously called "the notion of stored program,"[14] it is clear now for historians of technology that the intricate relationships among these three layers of involvement in the EDVAC project collectively led to the design decision of storing both data and instructions as pulses in mercury delay lines (Campbell-Kelly et al. 2013, 72–87; Haigh, Priestley, and Rope 2016, 129–152). After several board meetings between September 1944 and March 1945, the top engineers and von Neumann agreed that, if organized correctly, the new storage capabilities of mercury delay lines could be used to temporally conserve not only numerical data but also the description of in-built arithmetical and logical operations that will later compute them. This initial characteristic of the future EDVAC further suggested, to varying degrees, the possibility

of paper or magnetic-tape *documents* whose contents could be loaded, read, and processed at electronic speed by the device, without the intervention of a human being.

For the engineers and operators deeply involved in the ENIAC-EDVAC projects, the notion of lists of instructions that could automatically instruct the system was rather disconnected from their daily experiences of unreadable panel diagrams, electronic circuitry, and messy setup processes of switches and wires. To them, the differentiation between the computing system and its instructions hardly made sense: in practice, an electronic computing system was part of a broader sociotechnical process encompassing the definition of equations, the writing of diagrams, the adequate design of fragile electromechanical units, back-and-forth movements between hangars and offices, etc. To paraphrase Michel Callon (1999) when he talked about Air France, for these two layers of involvement, it was not an electronic calculator that could eventually compute an equation but a whole arrangement of engineers, operators, and artifacts in constant relationship.

The vision von Neumann had for both the ENIAC and EDVAC projects was very different: as he was constantly on the move, attending meetings and reading reports, he had a rather disembodied view of these systems. This process of disembodiment that often affects top managers was well described by Katherine Hayles (1999) when she compared the points of view of Warren McCulloch—the famous neurologist—and Miss Freed—his secretary—on the notion of "information":

> Thinking of her [Miss Freed], I am reminded of Dorothy Smith's suggestion that men of a certain class are prone to decontextualization and reification because they are in a position to command the labors of others. "Take a letter, Miss Freed," the man says. Miss Freed comes in. She gets a lovely smile. The man speaks, and she writes on her stenography pad (or perhaps on her stenography typewriter). The man leaves. He has a plane to catch, a meeting to attend. When he returns, the letter is on his desk, awaiting his signature. From his point of view, what has happened? He speaks, giving commands or dictating words, and things happen. A woman comes in, marks are inscribed onto paper, letters appear, conferences are arranged, books are published. Taken out of context, his words fly, by themselves, into books. The full burden of the labor that makes these things happen is for him only an abstraction, a resource diverted from other possible uses, because he is not the one performing the labor. (Hayles 1999, 82–83)

Hayles's powerful proposition is extendable to the case that interests us here: contrary to Eckert, Mauchly, Burks, and the operating team, von Neumann

was not the one performing the labor. Whereas the engineering and operating teams were entangled in the headache of making the ENIAC and EDVAC do meaningful things, von Neumann was entangled in the different headache of providing relevant insights—notably in terms of formalization—to military projects located all around the United States. To a certain extent, this position, alongside his interest in contemporary neurology and his exceptional logical and mathematical insights, certainly helped von Neumann write a document about the implications of storing both data and instructions as pulses in mercury delay lines. Provided as a summary of the discussions among the EDVAC team between the summer of 1944 and the spring of 1945, he wrote the *First Draft of a Report on the EDVAC* ([1945] 1993) that, for the first time, modeled the logical architecture of a hypothetical machine that would store both the data and the instructions required to compute them. Unaware of, and not concerned with, its laborious instantiations within the Moore School, von Neumann presented the EDVAC as a system of interacting "organs" whose relationships could by themselves transform inputs into outputs. And despite the skepticism of Eckert and Mauchly about presenting their project with floating terms, such as "neurons," "memory," "inputs," and "outputs"—and eventually their fierce resentment to see that their names were never mentioned in the document[15]—thirty-one copies of the report were printed and distributed among the US computing-related war projects in June 1945.

Proofs of Concept and the Circulation of the Input-Output Model

The many lawsuits and patent-related issues around the *First Draft* are not important for my story. What matters at this point is the surreptitious shift that occurred and persistently stayed within the computing community: Whereas computing systems were, in practice, sociotechnical *processes* that could ultimately—perhaps—produce meaningful results, the formalism of the *First Draft* surreptitiously presented them as brain-like *objects* that could automatically transform inputs into outputs. And if these high-level insights were surely important to sum up the confidential work that had been undertaken at the Moore School during the war and share it with other laboratories, they also contributed to separating computing systems from the practices required to make them operate. The *First Draft* presented the architecture of a functioning computing machine and thus put aside the actions required to make this machine function. The translation operations from equations

to logical diagrams, the specific configurations of electric circuitry and logic gates, the corrections of the diagrams from inaccurate electronic circulation of pulses; all of these sociotechnical operations were taken for granted in the *First Draft* to formalize the EDVAC at the logical level. Layers of involvement were relative layers of silence (Star and Strauss 1999); by expressing the point of view of the consultant who built on the results of intricate endeavors, the "list of the orders" (the programs) and the "device" (the computer) started to be considered two different entities instead of one entangled process.

But were the instructions really absent from the computing system as presented in the *First Draft*? Yes and no. The story is more intricate than that. In fact, the *First Draft* defined for the first time a quite complete set of instructions that, according to the formal definition of the system, could make the hypothetical machine compute every problem expressible in its formalism (von Neumann [1943] 1993, 39–43). But similarly to Turing's seminal paper on computable numbers (Turing 1937), von Neumann's set of instructions was integrally part of his formal system: the system constituted the set of all sets of instructions it could potentially compute. The benefits of this formalization were huge as it allowed the existence of all the infinite combinations of instructions. Yet, the surreptitious drawback was to consider these combinations as nonproblematic realizations of potentialities instead of costly actualizations of collective heterogeneous processes. While making a universal machine do something in particular was, and is, very different from formalizing such a universal machine, both practices were progressively considered equivalent.[16]

The diffusion of von Neumann's architecture as presented in the *First Draft* was not immediate. At the end of the war, several computing systems coexisted in an environment of mutual ignorance—most projects were classified during the war—and persistent suspicion—the Nazi threat was soon replaced with the communist (or capitalist) threat. During the conferences and workshops of the *Moore School Series* that took place in summer 1946, the logical design of the EDVAC was, for example, very little discussed as it was still classified. Nonetheless, several copies of the *First Draft* progressively started to circulate outside of the US defense services and laboratories, notably in Britain, where a small postwar research community could build on massive, yet extremely secret, code-breaking computing projects (Abbate 2012, 34–35; Campbell-Kelly et al. 2013, 83–84).

Contrary to Cold War–oriented American research projects, postwar British projects had no important funding as most of the UK government's money was being invested in the reconstruction of the devastated infrastructures. This forced British scientific managers to design rather small prototypes that could quickly show promising results. In June 1948, inspired by von Neumann's architecture as presented in the *First Draft*, Max Newman and Frederic Williams from the University of Manchester provided a first minimal proof of concept that the cathode-ray tube storage system could indeed be used to store instructions and data for computation at electronic speed in a desired, yet fastidious, way. One year later, Maurice Wilkes from the University of Cambridge—who also obtained a version of the *First Draft* and participated in the *Moore School Series* in 1946—successfully led the construction of an electronic digital computer with a mercury delay-line storage that he called the EDSAC for *Electronic Delay Storage Automatic Calculator*. Largely due to the programming efforts of Wilkes's PhD student David Wheeler (Richards 2005), the EDSAC could load data and instructions punched on a ribbon of paper and print the squares of the first one hundred positive integers. These two successful experiences participated in rendering electromechanical relays and differential analyzers obsolete in the emerging field of computer science research. But more importantly for the present story, these two successful experiments also participated in the diffusion of von Neumann's functional definition of electronic computing systems as input-output devices controlled by a central organ. As it ended up working, the model, and its encapsulated metaphors, were considered accurate.

At the beginning of 1950s, when IBM started to redefine computers as data-processing systems for businesses and administrations, von Neumann's definition of computing system further expanded. As cited in Haigh, Priestley, and Rope (2016, 240), an IBM paper written by Walker Thomas asserts, for example, that "all stored-program digital computers have four basic elements: the memory or storage element, the arithmetic element, the control element, and the terminal equipment or input-output element" (Thomas 1953, 1245). More generally, the broader inclusion of computing systems within *commercial arrangements* (Callon 2017) participated in the dissemination of their functional definition. It seems indeed that, to create new markets, intricate and very costly computing systems had better be presented as devices that automatically transform inputs into outputs rather than artefacts requiring a whole infrastructure to operate adequately. The

noninclusion of the sociotechnical interactions and practices required to make computers compute seems, then, to have participated in their expansions in commercial, scientific, and military spheres (Campbell-Kelly et al. 2013, 97–117). But the putting aside of programming practices from the definition of computers further led to numerous issues related to the ad hoc labor required to make them function.

The Psychology of Programming (And Its Limits)

The problem with practice is that it is necessary to do things: essence is existence and existence is action (Deleuze 1995). And as soon as electronic computing systems started to be presented as input-output functional devices controlled by a central organ, the efforts required to make them function in desired ways quickly stood out: it was extremely tedious to make the devices do meaningful things. These intelligent electronic brains were, in practice, dull as dishwater. But rather than casting doubts on the input-output framework of the *First Draft* and considering it formally brilliant but empirically inaccurate, the blame was soon casted on the individuals responsible for the design of computer's inputs. In short, if one could not make electronic brains operate, it was because one did not manage to give them the inputs they deserved. What was soon called the "psychology of programming" tried, and tries, to understand *why* individuals interact so laboriously with electronic computers.

This emphasis on the individual first led to *aptitude tests* in the 1950s that aimed at selecting the appropriate candidates for programming jobs in a time of workforce scarcity. By the late 1970s, entangled dynamics that made Western software industry shift from scientific craft to gender-connoted engineering supported the launching of *behavioral studies* that typically consisted of programming tests whose relative results were attributed to controlled parameters. A decade later, the contested results of these behavioral tests as well as theoretical debates within the discipline of psychology led to *cognitive studies* of programming. Cognitive scientists put aside the notion of parameters as proposed by behaviorists to focus on the mental models that programmers should develop to construct efficient programs. As we shall see, these research endeavors framed programming in ways that prevented them from inquiring into what programmers do, thus perpetuating the invisibilization of their day-to-day work.

Personnel Selection and Aptitude Tests

By the end of the 1940s, simultaneous to the completion of the first electronic computing systems that the von Neumann architecture inspired, the problem of the actual handling of these systems arose: these automatons appeared to be highly heteronomous. This practical issue quickly arose in the universities hosting the first electronic computers. As Maurice Wilkes wrote in his memoirs about the EDSAC:

> By June 1949 people had begun to realize that it was not so easy to get programs right as at one time appeared. I well remember when this realization first came on me with full force. The EDSAC was on the top floor of the building and the tape-punching and editing equipment one floor below on a gallery that ran round the room in which the differential analyzer was installed. I was trying to get working my first non-trivial program, which was one for the numerical integration of Airy's differential equation. It was on one of my journeys between the EDSAC room and the punching equipment that "hesitating at the angles of stairs" the realization 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 my own programs. (Wilkes 1985, 145)

Although the EDSAC theoretically included all possible programs, the actualization of these programs within specific situations was the main practical issue. And this became obvious to Wilke once he was directly involved in trying to make the functional device function.

In the industry, the heteronomous aspect of electronic computing systems also quickly stood up. A first example is the controversies surrounding the UNIVAC—an abbreviation for *Universal Automatic Computer*—an electronic computing system that Eckert and Mauchly developed after they left the Moore School in 1946 to launch their own company (which Remington Rand soon acquired). The potential of the UNIVAC gained a general audience when a whole programming team—which John Mauchly headed—made it run a statistical program that accurately predicted the results of 1952 American presidential election. This marketing move, whose costs were carefully unmentioned, further expanded the image of a functional electronic brain receiving inputs and producing clever outputs. But when General Electric acquired a UNIVAC computer in 1954, it quickly realized the gap between the presentation of the system and its actual enactment: it was simply impossible to make this functional system function. And it was only after two years and the hiring of a whole new programming team that a basic set of accounting applications could start producing some meaningful

results (Campbell-Kelly 2003, 25–30). IBM faced similar problems with its computing system 701. The promises of smooth automation quickly faced the down-to-earth reality of practice: the first users of IBM 701—notably Boeing, General Motors, and the National Security Agency (Smith 1983)—had to hire whole teams specifically dedicated to making the system do useful things.[17]

US defense agencies were confronted with the same issue. After the explosion of the first Soviet atomic bomb in August 1949, the United States appeared dangerously vulnerable; the existing air defense system and its slow manual gathering and processing of radar data could by no means detect nuclear bombers early enough to organize counter operations of interceptor aircrafts. This threat—and many other entangled elements that are far beyond the scope of this chapter—led to the development of a prototype computer-based system capable of processing radar data in real time.[18] The promising results of the prototype further suggested in 1954 the realization of a nationwide defense system of high-speed data-processing systems—called *Semi-Automatic Ground Environment* (SAGE).[19] The US Air Force contacted many contractors to industrially develop this system of systems, with IBM being awarded the development of the 250 tons AN/FSQ-7 electronic computers.[20] But none of these renowned institutions—among them IBM, General Electric, Bell Labs, and MIT—accepted the development of the lists of instructions that would make such powerful computers usable. Almost by default, the $20 million contract was awarded to the RAND Corporation, a nonprofit (but nonphilanthropic) governmental organization created in 1948 that operated as a research division for the US Air Force. RAND had already been involved in the previous development of the SAGE project, but its team of twenty-five programmers was obviously far too small for the new programming task. So by 1956, RAND started an important recruiting campaign all around the country to find individuals who could successfully pursue the task of programming.

In this early Cold War period, the challenge for RAND was then to recruit a lot of programming staff in a short period of time. And to equip this massive personnel selection imperative, psychologists from RAND's *System Development Division* started to develop tests whose quantitative results could positively correlate with future programming aptitudes. Largely inspired by the Thurstone Primary Mental Abilities Test,[21] these aptitude tests—although criticized within RAND itself (Rowan 1956)—soon became

the main basis for the selection of new programmers as they allowed crucial time savings while being based on the statistically driven discipline of psychometrics. The intensive use of aptitude tests helped RAND to rapidly increase its pool of programmers, so much so that its *System Development Division* was soon incorporated into a separate organization, the *System Development Corporation* (SDC). As early as 1959, the SDC had "more than 700 programmers working on SAGE, and more than 1,400 people supporting them. … This was reckoned to be half of the entire programming manpower of the United States" (Campbell-Kelly 2003, 39). But besides enabling RAND/SDC to engage more confidently in the SAGE project, aptitude tests also had an important effect on the very conception of programming work. Although the main goal of these tests was to support a quick and nationwide personnel selection, they also contributed to framing programming as a set of abstract intellectual operations that can be measured using proxies.

The regime of aptitude testing as initiated by the SDC quickly spread throughout the industry, notably prompting IBM to develop its own questionnaire in 1959 to support its similarly important recruitment needs. Well in line with the computer-brain parallel inherited from the seminal period of electronic computing, the IBM Programming Aptitude Test (PAT) typically asked job candidates to figure out analogies between forms, continue lists of numbers, and solve arithmetic problems (see figure 3.1). Though the correlation between candidates' scores to aptitude tests and their future work performances was a matter of debate, aptitude tests quickly became mainstream recruiting tools for companies and administrations that purchased electronic computers during the 1960s. As Ensmenger (2012, 64) noted: "By 1962, an estimated 80 percent of all businesses used some form of aptitude test when hiring programmers, and half of these used IBM PAT." The massive distribution and use of these tests among the emerging computing industry further constricted the framing of programming practices as measurable innate intellectual abilities.

Supposed Crisis and Behavioral Studies

By framing programming as an activity requiring personal intuitive qualities, aptitude tests have somewhat worked against gendered discriminations related to unequal access to university degrees. As Abbate (2012, 52) noted: "A woman who had never had the chance to earn a college degree—or who had been steered into a nontechnical major—could walk into a job

PART III (Cont'd)

13. During his first three years, a salesman sold 90%, 105%, and 120%, respectively, of his yearly sales quota which remained the same each year. If his sales totaled $252,000 for the three years, how much were his sales below quota during his first year?

    (a) $800 (b) $2,400 (c) $8,000
    (d) $12,000 (e) $16,000

14. In a large office, 2/3 of the staff can neither type nor take shorthand. However, 1/4 of the staff can type and 1/6 can take shorthand. What proportion of people in the office can do both?

    (a) 1/12 (b) 5/36 (c) 1/4
    (d) 5/12 (e) 7/12

15. A company invests $80,000 of its employee pension fund in 4% and 5% bonds and receives $3,360 in interest for the first year. What amount did the company have invested in 5% bonds?

    (a) $12,800 (b) $16,000 (c) $32,000
    (d) $64,000 (e) $67,200

16. A company made a net profit of 15% of sales. Total operating expense were $488,000. What was the total amount of sales?

    (a) $361,250 (b) $440,000 (c) $450,000
    (d) $488,750 (e) $500,000

17. An IBM Sorting Machine processes 1,000 cards per minute. However, 20% is deducted to allow for card handling time by the operator. A given job requires 5,000 cards to be put through the machine 5 times and 9,000 cards to be put through 7 times. How long will it take?

    (a) 1 hr. 10 min. (b) 1 hr. 28 min. (c) 1 hr. 45 min.
    (d) 1 hr. 50 min. (e) 2 hrs. 10 min.

Figure 3.1

Sample of the 1959 IBM Programmer Aptitude Test. In this part of the test, the participant is asked to answer problems in arithmetic reasoning. *Source:* Reproduced by the author from a scanned 1959 IBM Programmer Aptitude Test by J. L. Hughes and W. J. McNamara. Courtesy of IBM.

interview, take a test, and instantly acquire credibility as a future programmer." From its inception, computer programming, unlike the vast majority of skilled technical professions in the United States, has involved women workers, some of whom had already taken part to computing projects during the war.

However, like most Western professional environments in the late 1950s, the nascent computing industry was fueled by pervasive stereotypes, often preventing women programmers from occupying upper managerial positions and encouraging them to do relational customer care work. These gender dynamics should not be overlooked as they help to understand the rapid, and often underappreciated, development of ingenious software equipment. Due to their unique position within the computer-related professional worlds—both expert practitioners and, often, representatives toward clients—women, given their rather small percentage within the industry, actively contributed to innovations aimed at making programming easier for experts and novices alike. The most notorious example is certainly Grace Murray Hopper, head of programming for UNIVAC, who developed the first compiler—a program that translates other programs into machine code[22]—in 1951 before designing the business programming language B-0 (renamed FLOW-MATIC) in 1955. But many other women actively took part to software innovations throughout the 1950s and 1960s, though often in the shadow of more visible male managers. Among these important figures are Adele Mildred Koss and Nora Moser who developed widely used code for data editing in the mid-1950s; Lois Haibt who was responsible for flow analysis of the FORTRAN high-level programming language; and Mary Hawes, Jean Sammet, and Gertrude Tierney who were at the forefront of the common business-oriented language (COBOL) project in the late 1950s (Abbate 2012, 79–81).

From the mid-1960s onward, refinements over compilers and high-level programming languages, which had often come from women, were added to the impressive tenfold increase in computing power (Mody 2017, 47–77). This combination of new promising software and hardware infrastructures prompted large iconic computer manufacturers to start building increasingly complex programs, such as operating systems and massive business applications. The resounding failures of some of these highly visible projects, like the IBM project System 360,[23] soon gave rise to a sense of uncertainty among commentators at the time, some of whom used the evocative

expression of "software crisis" (Naur and Randell 1969, 70–73). Historians of computing have expressed doubts about the reality of this software crisis as precise inquiries have shown that, apart from some highly visible and nonstandard projects, software production in the late 1960s was generally on time and on budget (Campbell-Kelly 2003, 94). But the crisis rhetoric, which also fed on an exaggerated but popular discourse on software production costs,[24] nonetheless had tangible effects on the industry to the point of changing its overall direction and identity.

When compared with the related discipline of microelectronics, programming has long suffered from a lack of credibility and prestige. Despite significant advances throughout the 1950s and the 1960s, actors taking part to software production were often accorded a lower status within Western computing research and industry. This was true for women programmers since they were working in a technical environment. But it was also true for men programmers since they were working in a field that included women. Under this lens, the crisis rhetoric that took hold at the end of the 1960s—feeding on iconic failures that were not representative of the state of the industry—provided an opportunity to *reinvent* programming as something more valuable according to the criteria of the time (Ensmenger 2010, 195–222). This may be one of the reasons why the positively connoted term "engineering" started to spread and operate as a line of sight, notably via the efforts of the 1968 North Atlantic Treaty Organization (NATO) conferences entitled "Software Engineering" and the setting up of professional organizations and academic journals such as the Institute of Electrical and Electronics Engineers' *IEEE Transactions on Software Engineering* (1975) and the Association for Computing Machinery's *ACM Software Engineering Notes* (1976). Though contested by eminent figures who considered that software production was already rigorous and systematic, this complex process of disciplinary relabeling was supported by many programmers—women and men—who saw the title of engineer as an opportunity to improve their work conditions. However, as Abbate (2012, 104) pointed out: "An unintended consequence of this move may have been to make programming and computer science less inviting to women, helping to explain the historical puzzle of why women took a leading role in the first wave of software improvements but become much less visible in the software engineering era."

This stated desire to make software production take the path of engineering—considered the solution to a supposed crisis that itself built on

a gendered undervaluation of programming work—has rubbed off on the academic analysis of programming. Parallel to this disciplinary reorientation, a line of positivist research claiming behaviorist tradition began to take an interest in programming work in the early 1970s. For these researchers, the analytical focus should shift: instead of defining the inherent *skills* required for programming and design aptitude tests, scholars should rather try to extract the *parameters that induce the best programming performances* and propose ways to improve software production. The introduction and dissemination of high-level programming languages as well as the multiplication of academic curricula in computer science highly participated in establishing this new line of inquiry. With programming languages such as FORTRAN or COBOL that did not depend on the specificities and brands of computers, behavioral psychologists along with computer scientists became able to design programming *tests* in controlled environments. Moreover, the multiplication of academic curricula in computer science provided relatively diverse *populations* (e.g., undergrads, graduates, faculty members) that could pass these programming tests. These two elements made possible the design of experiments that ranked different sets of parameters (age, experience, design aids) according to the results they assumedly produced (see figure 3.2).

This framework led to numerous tests on debugging performances (e.g., Bloom 1980; Denelesky and McKee 1974; Sackman, Erikson, and Grant 1968; Weinberg 1971, 122–189; Wolfe 1971), design aid performances (e.g., Blaiwes 1974; Brooke and Duncan, 1980a, 1980b; Kammann 1975; Mayer 1976; Shneiderman et al. 1977; Weinberg 1971, 205–281; Wright and Reid 1973), and logical statement performances[25] (e.g., Dunsmore and Gannon 1979; Gannon 1976; Green 1977; Lucas and Kaplan 1976; Sime, Green, and Guest 1973; Sime, Arblaster, and Green 1977; Sime, Green, and Guest 1977; Sheppard et al. 1979; Weissman 1974). But despite their systematic aspect, these studies suffered from the obviousness of their results, for as explained by Curtis (1988), without formally being engaged in behavioral experiments, software contractors were already aware that, for example, experienced programmers produced better results than inexperienced ones did, or that design aids such as flowcharts or documentation were helpful tools for the practice of programming. These general and redundant facts did not help programmers to better design lists of instructions. By the 1980s, the increasingly powerful computing systems remained terribly

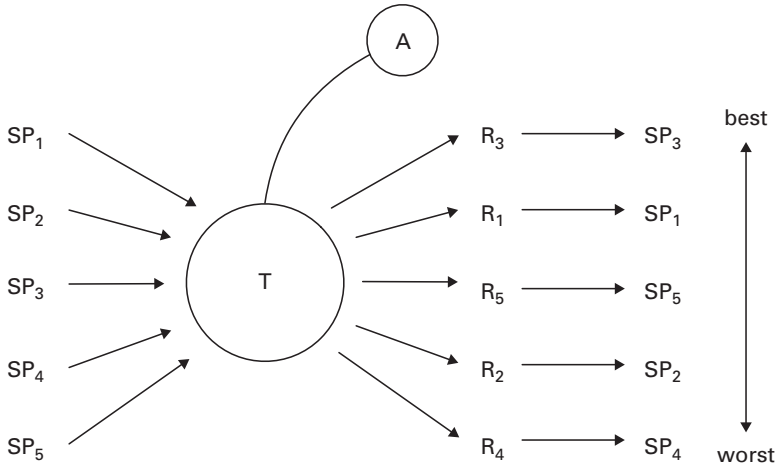


Figure 3.2
Schematic of behavioral studies of computer programming. Let us assume a programming test T, the test's best answers A, and five sets of parameters $SP_{1,\ldots,5}$. SP_1 could, for example, gather the parameters "unexperimented, male, with flowcharts"; SP_2 could, for example, gather the parameters "experienced, female, without flowcharts," and so on. Once all SPs have passed T, the results Rs of each SP allow the ranking of all SPs from best to worst. In this example, R_3 (the results of SP_3) made SP_3 be considered the best set of parameters. Inversely, R_4 (the results of SP_4) made SP_4 be considered the worst set of parameters.

difficult to operate, be they instructed by software engineers working in more and more malely connoted environments.

The Cognitive Turn

By the end of the 1970s, the behavioral standpoint began to be criticized from inside the psychological field. To more and more *cognitive psychologists*, sometimes working in artificial intelligence departments, it seemed that the obviousness of behavioral studies' results was function of a methodological flaw, with many of the ranked sets of parameters gathering important individual variations of results. According to several cognitive researchers, the unit of analysis of behavioral studies was erroneous; since many results' disparities existed within the same sets of parameters, the ranking of these sets was simply senseless (Brooks 1977, 1980; Curtis 1981; Curtis et al. 1989; Moher and Schneider 1981). The solution that these cognitivists proposed to account for what they called "individual differences" was then to dive

inside the individuals' head to better understand the *cognitive processes* and *mental models* underlying the formation of computer programs.

The strong relationships between the notions of "program" and "cognition" also participated in making the study of computer programming attractive to cognitive scientists. As Ormerod (1990, 63–64) put it:

> The fields of cognition and programming are related in three main ways. First, cognitive psychology is based on a "computational metaphor," in which the human mind is seen as a kind of information processor similar to a computer. Secondly, cognitive psychology offers methods for examining the processes underlying performance in computing tasks. Thirdly, programming is a well-defined task, and there are an increasing number of programmers, which makes it an ideal task in which to study cognitive process in a real-world domain.

These three elements—the assumed-fundamental similarity between cognition and computer programs, the growing population of programmers, and the available methods that could be used to study this population—greatly contributed to making cognitive scientists consider computer programming as a fruitful topic of inquiry. Moreover, investing in a topic that behaviorists failed to understand was also seen as an opportunity to demonstrate the superiority of cognitivist approaches. To a certain extent, the aim was also to show that behaviors were a function of mental processes:

> [Behaviorists] attempt to establish the validity of various parameters for describing programming behavior, rather than attempting to specify underlining processes which determine these parameters. (Brooks 1977, 740)

The ambition was then to describe the mental processes that lead to *good* programming performances and eventually use these mental processes to train or select *better* programmers. The methodology of cognitive studies was, most of the time, not radically different from that of behavioral studies on programming, though. Specific programming tests were proposed to different individuals, often computer science students or faculty members. The responses, comments (oral or written), and metadata (number of key strokes, time spent on the problem, etc.) of the individuals were then analyzed according to the rights answers of the test as well as based on general cognitive models of human understanding that the computational metaphor of the mind has inspired (especially the models of Newell and Simon [1972] and, later, Anderson [1983]). From this confrontation among results, comments, and general models of cognition, different mental models specific

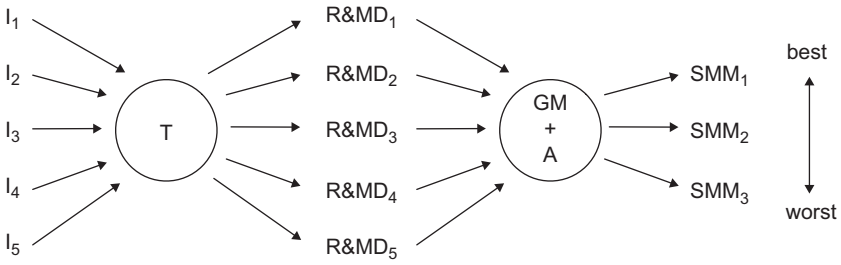


Figure 3.3
Schematic of cognitive studies of computer programming. Let us assume a programming test T, the test's best answers A, five individuals $I_{1,\ldots,5}$, and a general model of cognition GM. Once all Is have passed T, the corresponding results Rs and metadata MD (for example, comments from I on T) are gathered together to form five R&MDs. All R&MDs are then evaluated and compared according to A and GM. At the end of this confrontation, specific mental models (SMMs) are proposed and ranked from best to worst according to their assumed ability to produce the best programming results.

to the task of computer programming were inferred, classified, and ranked according to their performances (see figure 3.3).

This research pattern on computer programming led to numerous studies proposing mental models for solving abstract problems (e.g., Adelson 1981; Brooks 1977; Carroll, Thomas, and Malhotra 1980; Jeffries et al. 1981; Pennington 1987; Shneiderman and Mayer 1979) and developing programming competencies (e.g., Barfield 1986; Coombs, Gibson, and Alty 1982; McKeithen et al. 1981; Soloway 1986; Vessey 1989; Wiedenbeck 1985). Due, in part, to their mitigated results—as admitted by Draper (1992), the numerous mental models proposed by cognitivists did not significantly contribute to better programming performances—cognitive studies have later reintegrated behaviorist considerations (e.g., controlled sets of parameters) to acquire the hybrid and management-centered form they have today (Capretz 2014; Ahmed, Capretz, and Campbell 2012; Ahmed et al. 2012; Cruz, da Silva, and Capretz 2015).

Limits

From the 1950s up to today, computer scientists, engineers, and psychologists have deployed important efforts in the study of computer programming. From aptitude tests to cognitive studies, these scholars have spent

a fair amount of time and energy trying to understand what is going on when someone is programming. They certainly did their best, as we all do. Yet I think one can nonetheless express some critiques of, or at least reservations about, some of their methods and conceptual habits regarding the study of programming activity.

Aptitude tests certainly constituted useful recruiting tools in the confusing days of early electronic computing. In this sense, they surely helped counterbalance the unkeepable promises of electronic brains, themselves deriving—I suggest—from the dissemination of von Neumann's functional depiction of electronic computers and its setting aside of programming practices. Moreover, the weight of aptitude tests' results has also constituted resources for women wishing to pursue careers in programming, and some of these women have devised crucial software innovations. Yet as central as they might have been for the development of computing, aptitude tests suffer from a flaw that prevents them from properly analyzing the actions taking part in computer programming: they test candidates on what electronic computers should supposedly do (e.g., sorting numbers, solving equations) but not on the skills required to make computers do these things. They mix up premises and consequences: if the results of computer programming can potentially be evaluated in terms of computing and sorting capabilities, the way in which these results are achieved may require other units of analysis.

Behavioral studies suffer from a similar flaw that keeps them away from computer programming actions. By analyzing the relationships between sets of parameters and programming performances, behaviorist studies put the practices of programming into a black box. In these studies, the practices of programmers do not matter: only the practices' *conditions* (reduced to contextual parameters) and *consequences* (reduced to quantities of errors) are considered. One may object that this nonconsideration of practices is precisely what defines behaviorism as a scientific paradigm, its goal being to predict consequences (behaviors) from initial conditions (Watson 1930), an aim that well echoed the engineerization of software production in the 1970s. It is true that this way of looking at things can be very powerful, especially for the study of complex processes that include many entities, such as traffic flows (Daganzo 1995, 2002), migrations (Jennions and Møller 2003), or cells' behaviors (Collins et al. 2005). But inscribing numbered lists of symbols is a process that does not need any drastic reduction: a programming situation involves only one, two, perhaps three individuals whose actions

can be accounted for without any insurmountable difficulties. For the study of such a process that engages few entities whose actions are slow enough to be accounted for, no need a priori exists to ignore what is happening in situation.

For cognitive studies, the story is more intricate. They are certainly right to criticize behavioral studies for putting into black boxes what precisely needs to be accounted for. Yet the solution cognitivists propose to better understand computer programming leads to an impasse we now need to consider.

As Ormerod (1990, 63) put it, "cognitive psychology is based on a 'computational metaphor' in which the human mind is seen as a kind of information processor similar to a computer." From this theoretical standpoint, cognition refers to the reasoning and planning models the mind uses to transform emotional and perceptual input information into outputs that take the form of thoughts or bodily movements. Similarly to a computer—or rather, similarly to *one specific and problematic image of computers*—the human mind "runs" mental models on inputs to produce outputs. The systematic study of the complex mental models that the mind uses to transform inputs into outputs is the very purpose of cognitive studies. Scientific methods of investigation, such as the one presented in figure 3.3, can be used for this specific prospect.

When cognitive science deals with topics such as literature (Zunshine 2015), religion (Barrett 2007), or even chimpanzees' preferences for cooked foods (Warneken and Rosati 2015), its foundations usually hold on: complex mental models describing how the mind processes input information in terms of logical and arithmetic statements to produce physical or mental behaviors can be proposed and compared without obvious contradictions. But as soon as cognitive science deals with computer programming, a short circuit appears that challenges the whole edifice: the cognitive explanation of the constitution of computer programs is tautological as the very notion of cognition already requires constituted computer programs.

To better understand this tricky problem, let us consider once again the computational metaphor of the mind. According to this metaphor, the mind "runs" models—or programs—on inputs to produce outputs. In that sense, the mind looks like a computer as described by von Neumann in the *First Draft*: input data are stored in memory where lists of logical and arithmetic instructions transform them into output. But as we saw in the

previous sections, von Neumann's presentation of computers was functional in the sense that it did not take into consideration the elements required to make a computer function. In this image of the computer that reflects von Neumann's very specific position and status, the elements required to assemble the actual transformative lists of instructions—or programs—that command the functioning of an electronic computer's circuitry have already been gathered.

From here, an important flaw of cognitive studies on computer programming starts to appear: as the studies rely on an image of the computer that already includes constituted computer programs, these cognitive studies are not in a position to inquire into what constitutes computer programs. In fact, the cognitive studies are in a situation where they can mainly propose circular explanations of programming: if there are (computer) programs, it is because there are (mental) programs. Programs explain programs: a perfect tautology.

As long as cognitive science stays away from the study of computer programming, its foundations hold on: mental programs can serve as explicative tools for observed behaviors. But as soon as cognitive science considers computer programming, its limits appear: cognition and programs are of the same kind. Thunder in the night! Cognition, as inspired by the computational metaphor of the mind, works as a stumbling stone to the analysis of computer programming practices as its fundamental units of analysis are assembled programs. In such a constricted *epistemic culture* (Knorr-Cetina 1999), the in situ analysis of courses of action cannot but be omitted, despite their active participation in the constitution of the collective computerized world. This is an unfortunate situation that even the bravest propositions in human-computer interaction (HCI) have not been able to modify substantially (e.g., Flor and Hutchins 1991; Hollan, Hutchins, and Kirsh 2000). Is there a way to conceptually dis-constrict the empirical study of computer programming?

Putting Cognition Back to Its Place

Most academic attempts to better understand computer programming seem to have annoying flaws: aptitude tests mix up premises and consequences, behavioral studies put actions into black boxes, and cognitive studies are stuck in tautological explanations. If we want to consider computer programming

as accountable practices, it seems that we need to distance ourselves from these brave but problematic endeavors.

Yet, provided that our critics are relevant, we are at this point still unable to propose any alternative. Do the actions of programmers not have a *cognitive* aspect? Do programmers not use their *minds* to computationally solve complex *problems*? The confusion between cognition and computer programs may well derive from a misleading history of computers—as I tried to suggest—its capacity to establish itself as a generalized habit commands respect. How can we not present empirical studies of computer programming practices as silly reductions? How can we justify the desire to account for, and thus make visible, the *courses of action* of computer programming, these practices that are obligatory passage points of any computerization project?

Fortunately, contemporary work in philosophy has managed to fill in the gap that has separated cognition from practices, intelligent minds from dull actions. It is thanks to these inspiring studies that we will become able to consider programming as a practice without totally turning our back on the notion of cognition. To do so, I will first need to quickly reconsider the idea that computers were designed in the image of the human brain and mind. As we already saw—though partially—this idea is relevant only in retrospect: what has concretely happened is far more intricate. I will then reconsider the philosophical frame that encloses cognition as a computational process. Finally, following contemporary works in the philosophy of perception, I will examine a definition of cognition that preserves important aspects of how we make sense of the things that surround us while reconnecting it to practices and actions. By positing the centrality of agency in cognitive processes, this *enactive conception of cognition* will further help us empirically consider what is happening during computer programming episodes.

A Reduction Process

The computational metaphor of the mind forces cognitivists to use programs to explain the formation of programs. The results of programming processes—programs—are thus used to explain programming processes. It is not easy to find another example of such an explicative error: it is like explaining rain with water, chicken poultry with the chicken dance … But how did things end up this way? How did programs end up constituting the fundamental base of cognition, thus participating in the invisibilization of computer programming practices?

The main argument that justifies the computational metaphor of the mind is that "computers were designed in the image of the human" (Simon and Kaplan 1989, quoted in Hutchins 1995, 356). According to this view that spread in the 1960s in reaction to the behavioral paradigm (Fodor 1975, 1987; Putnam [1961] 1980), how the human brain works inspired the design of computers, and this can, in turn, provide a clearer view on how we think. Turing is generally considered the father of this argument, with the Universal Machine he imagined in his 1937 paper "On Computable Numbers" being able to simulate any mechanism describable in its formalism. According to this line of thought, it was Turing's self-conscious introspection that allowed him to define a device capable of any computation as he was looking "at what a mathematician does in the course of solving mathematical problems and distilling this process to its essentials" (Pylyshyn 1989, 54). Turing's demonstration would then lead to the first electronic computers, such as the ENIAC and the EDVAC, whose depiction as giant brains appears legitimate as how we think inspired these computers in the first place.

In line with the recent work of Simon Penny (2017), I assume that this conception of the origins of computers is incorrect. As soon as one considers simultaneously the process by which Turing's thought experiment was reduced to an image of the brain *and* the process by which the EDVAC was reduced to an input/output device controlled by a central organ, one realizes that the relationship between computers and the human brain points to the other direction: the human brain was designed in a very specific image of the computer that already included all possible programs.

Let us start with Turing as he is often considered the father of the computational metaphor of the mind. It is true that Turing compared "a man in the process of computing a real number" with a "machine which is only capable of a finite number of conditions" (Turing 1937, 231). Yet his image of human computation was not limited to what is happening inside the head: it also included hands, eyes, paper, notes, and sets of rules defined by others in different times and locations. As Hutchins put it: "The mathematician or logician was [for Turing] materially interacting with a material world" (Hutchins 1995, 361). By modeling the properties of this sociomaterial arrangement into an abstract machine, Turing could distinguish between computable and noncomputable numbers, hence showing that Hilbert's *Entscheidungsproblem* was not solvable. His results had an immense

impact on the mathematics of his time as they suggested a class of numbers calculable by finite means. But the theoretical machine he invented to define this class of numbers was by no means designed *only* in the image of the human brain; it was a theoretical device that expressed the sociomaterial process enabling the computation of real numbers.

What participated in reducing Turing's theoretical device to an expression of a mental process was the work of McCulloch and Pitts on neurons. In their 1943 paper entitled "A logical Calculus of the Ideas Immanent in Nervous Activity," McCulloch and Pitts built upon Carnap's (1937) propositional logic and a simplified conception of neurons as all-or-none firing entities to propose a formal model of mind and brain. In their paper, neurons are considered units that process input signals sent from sensory organs or from other neurons. In turn, the outputs of this neural processing feed other neurons or are sent back to sensory organs. The novelty of McCulloch and Pitts's approach is that, thanks to their simplified conception of neurons, the input signals that are processed by neurons can be represented as propositions or, as Gödel (1931) previously demonstrated, as numbers.[26] From that point, their model could consider configurations of neural networks as logical operators processing input signals from sensory organs and outputting *different* signals back to sensory organs. This way to consider the brain as a huge network of neural networks able to express the laws of propositional calculus on binary signals allowed McCulloch and Pitts to hypothetically consider the brain as a Turing machine capable of computing numerical propositions (McCulloch and Pitts [1943] 1990, 113). Even though they did not mathematically prove their claim and recognized that their model was computationally less powerful than Turing's model, they nonetheless infused the conception of mind as the result of the brain's computational processes (Piccinini 2004).

At first, McCulloch and Pitts's paper remained unnoticed (Lettvin 1989, 17). It was only when von Neumann used some of their propositions in his 1945 *First Draft* (von Neumann [1945] 1993, 5–11) that the equivalence between computers and the human mind started to take off. As we saw earlier, von Neumann had a very specific view on the EDVAC: his position as a famous consultant who mainly sees the clean results of laborious material processes allowed him to reduce the EDVAC as an input-output device. Once separated from its instantiation within the hangars of the Moore School of Electric Engineering, the EDVAC, and especially the

ENIAC, effectively *looked like* a brain as conceived by McCulloch and Pitts. From that point, the reduction process could go on: von Neumann could use McCulloch and Pitts' reductions of neurons and of the Turing machine to present his own reductive view on the EDVAC. However, it is important to remember that von Neumann's goal was by no means to present the EDVAC in a realistic way: the main goal of the *First Draft* was to formalize a model for an electronic computing system that could inspire other laboratories without revealing too many classified elements about the EDVAC project. All of these intricate reasons (von Neumann's position, wartime, von Neumann's interest in mathematical biology) made the EDVAC appear in the *First Draft* as an input-output device controlled by a central organ whose configuration of networks of neurons could express the laws of propositional calculus.

As we saw earlier, after World War II, the *First Draft* and the modelization of electronic computers it encapsulates began to circulate in academic spheres. In parallel, this conception of computers as giant electronic brains fitted well with their broader inclusion in commercial arrangements: these very costly systems had better be presented as functional brains automatically transforming inputs into outputs rather than intricate artifacts requiring great care, maintenance, and an entire dedicated infrastructure. Hence there were issues related to their operationalization as the buyers of the first electronic computers—the Air Force, Boeing, General Motors (Smith 1983)—had to select, hire, and train and eventually fire, reselect, rehire, and retrain whole operating teams. But despite these initial failures, the conception of computers as electronic brains held on, well supported, to be fair, by Turing's (1950) paper "Computing Machinery and Intelligence," the 1953 inaugural conferences on artificial intelligence at Dartmouth College (Crevier 1993), Ashby's book on the neural origin of behavior (Ashby 1952), and von Neumann's posthumous book *The Computer and the Brain* ([1958] 2012). Instead of crumbling, the conception of computers as electronic brains started to concretize to the point that it even supported a radical critique of behaviorism in the field of psychology. Progressively, the mind became the product of the brain's computation of nervous inputs. The argument appeared indeed indubitable: as human behaviors are the results of (computational) cognitive processes, psychology should rather describe the composition of these cognitive processes—a real tour de force whose consequences we still experience today.

But this colossus of the computational metaphor of the mind has feet of clay. As soon as one inquires sociohistorically into the process by which brains and computers have been put into equivalence, one sees that the foundations of the argument are shaky; a cascade of reductions, as well as their distribution, surreptitiously ended up presenting the computer as an image of the brain. Historically, it was first the reduction of the Turing machine as an expression of mental processes, then the reduction of neurons as on/off entities, then the reduction of the EDVAC as an input-output device controlled by a central organ, then the distribution of this view through academic networks and commercial arrangements that *allowed* computers to be considered as deriving from the brain. It is the collusion of all of these *translations* (Latour 2005), along with many others, that made computers appear as the consequences of the brain's structure.

Important authors have finely documented how computer-brain equivalences contributed, for better or worse, to structuring Western subjectivities throughout the Cold War period (e.g., Dupuy 1994; Edwards 1996; Mirowski 2002). For what interests me here, the main problem of the conception of computers as an image of the brain is that its correlated conception of cognition as computation contributed to further invisibilizing the courses of actions taking part in computer programming. According to the computational metaphor of the mind, the brain *is* the set of all the combinations of neural networks—or logic circuits[27]—that allow the computation of signals. The brain may choose one specific combination of neural networks for the computation of each signal, but the combination itself is already assembled. As a consequence, the study of how combinations of neural networks are assembled and put together to compute specific signals—as it is the case when someone is programming—cannot occur as it would imply to go beyond what constitutes the brain. Cognitive studies may involve inquiring about *which* program the brain uses for the computation of a specific input, but the way this program was assembled remains out of reach: it was already there, ready to be applied to the task at hand. In short, similarly to von Neumann's view on the EDVAC but with far less engineering applications, the brain as conceived by the computational metaphor of the mind *selects the appropriate mental program from the infinite library of all possible programs*. But as this library is precisely what constitutes the brain, it soon becomes senseless to inquire into how each program was concretely assembled.

The cognitivist view on computers as designed in the image of the brain seems then to be the product of at least three reductions: (1) neurons as on/off firing entities, (2) the Turing machine as an expression of mental events, and (3) the EDVAC as an input/output device controlled by a central organ. The further distribution of this view on computers through academic, commercial, and cultural networks further legitimatized the conception of cognition as computation. But this cognitive computation was a holistic one that implied the possibility of all specific computations: the brain progressively appeared as the set of all potential instruction sets, hence preventing inquiries into the constitution of actual instruction sets. The tautological impasse of cognitive science when it deals with computer programming seems, then, to be deriving from a delusive history of the computer. The ones who inherit from a nonempirical history of electronic computers might consider cognition as computation and programming as a mental process. Yet the ones who inherit from an empirical history of the constitution of electronic computing systems and who pay attention to translation processes and distributive networks have no other choice but to consider cognition differently. But how?

The Classical Sandwich and Its Consequences

We now have a clearer—yet still sketchy—idea of the formation of the computational metaphor of the mind. An oriented "double-click" history (Latour 2013, 93) of electronic computers that did not pay attention to the small translations that occurred at the beginning of the electronic computing area enabled cognitive scientists—among others—to retroactively consider computers as deriving from the very structure of the brain. But historically, what has happened is far more intricate: McCulloch and Pitts's work on neurons and von Neumann's view on the EDVAC echoed each other to progressively form a powerful yet problematic depiction of computers as giant electronic brains. This depiction further legitimized the computational metaphor of the mind—also coined *computationalism*—that yet paralyzed the analysis of the constitution of actual computer programs since the set of all potential programs constituted the brain's fundamental structure. At this point of the chapter, then, to definitively turn our back on computationalism and propose an alternative definition of cognition that could enable us to consider the task of computer programming as a

practical activity, we need to look more precisely at the metaphysics of this computational standpoint.

If computationalism in cognitive science derives from a quite recent nonempirical history of computers, its metaphysics surely belongs to a philosophical lineage that goes back at least to Aristotle (Dreyfus 1992). Susan Hurley (2002) usefully coined the term "classical sandwich" to summarize the metaphysics of this lineage—also referred to as "cognitivism"—that considers perception, cognition, and agency as distinct capacities. For the supporters of the classical sandwich, human perception first grasps an input from the "real" world and translates it to the mind (or brain). In the case of computationalism, this perceptual input takes the shape of nervous pulses that can be expressed as numerical values. Cognition, then, "works with this perceptual input, uses it to form a representation of how things are in the subject's environment and, through reasoning and planning that is appropriately informed by the subject's projects and desires, arrives at a specification of what the subject should do with or in her current environment" (Ward and Stapleton 2012, 94). In the case of computationalism, the cognitive step implies the selection and application of a mental model—or mental program—that outputs a different numerical value to the nervous system. Finally, agency is considered the output of both perception and cognition processes and takes the form of bodily movements instructed by nervous pulses.

This conception of cognition as "stuck" in between perception and action as meat in a sandwich has many consequences. It first establishes a sharp distinction between the mind and the world. Two realms are then created: the realm of "extended things" that are said to be material and the realm of "thinking things" that are said to be abstract and immaterial.[28] If matter thrones in the realm of "extended things" by allowing substance and quantities, mind thrones in the realm of "thinking things" by allowing thoughts and knowledge.

Despite the ontological abyss between them, the realms of "thinking things" and "extended things" need to interact: after all, we, as individuals, are part of the world and need to deal with it. But a sheet of paper cannot go through the mind, a mountain is too big to be thought, a spoken sentence has no matter: some transformation has to occur to make these things possible for the mind to process. How, then, can we connect both "extended"

and "thinking" realms? The notions of *representation* (without hyphen) and *symbols* have progressively been introduced to keep the model viable. For the mind to keep in touch with the world of "real things," it needs to work with *representations* of real things. Because these representations happen in the head and refer to extended things, they are usually called *mental representations of things*.

Mental representations of things need to have at least two properties. They first need a *form* on which the mind could operate. This form may vary according to different theories among cognitivism. For the computational metaphor of the mind, this form takes, for example, the shape of electric nervous pulses that the senses acquire and that are then routed to the brain. The second property that mental representations of things require is *meaning*; that is, the distinctive trace of what representations refer to in the real world. Both properties depend on each other: a form has a meaning, and a meaning needs a form. The notion of *symbol* is often used to gather both the half-material and semantic aspects of the mental representations of things. In this respect, cognition, as considered by the proponents of the classical sandwich, processes symbolic representations of things that the senses offer in their interactions with the real world. The result of this processing is, then, another representation of things—a statement *about* things—that further instructs bodily movements and behaviors.

The processing of symbolic representations of things does not always lead to accurate statements about things. Some malfunctions can happen either at the level of the senses that badly translate real things or at the level of the mind that fails to interpret the symbols. In both cases, the whole process would lead to an inaccurate, or *wrong*, statement about things. These errors are not desirable as they would instruct inadequate behaviors at the end of the cognitive process. It is therefore extremely important for cognition to make *true* statements. If cognition does not manage to establish adequate correspondences between our minds and the world, our behaviors will be badly instructed. Conversely, by properly acquiring *knowledge* about the real world, cognition can make us behave adequately.

I assume that the symbolic representational thesis that derives from cognition as considered by the classical sandwich leads to two related issues. The first issue deals with the amalgam between *knowledge* and *reality* it creates, hence refusing giving any ontological weight to entities whose trajectories are different from scientific facts. The second issue deals with the

thesis's incapacity to consider practices *in the wild*, with most of the models that take symbolic representational thesis to the letter failing the test of ecological validation.

Let us start with the first issue, certainly the most difficult. We saw that, according to cognitivism, the *adaequatio rei et intellectus* serves as the measure of valid statements and behaviors. For example, if I say "the sun is rising," I make an invalid statement and thus behave wrongly because what I say does not refer adequately to the real event. Within my cognitive process, something went wrong: in this case, my senses that made me believe that the sun was moving in the sky probably deceived me. In reality, thanks to other mental processes that are better than mine, we know as a matter of fact that it is the earth that rotates around the sun; some "scientific minds"—in this case, Copernicus and Galileo, among others—managed indeed to adequately process symbolic representations to provide a true statement about the relations between the sun and the earth, a relation that the laws of Reason can demonstrate. My statement and behavior can still be considered a joke or some form of sloppy habit: what I say/do is not *true* and therefore does not really *count*.

The problem of this line of thought that only gives credit to scientific facts is that it is grounded on a very unempirical conception of science. Indeed, as STS authors have demonstrated for almost fifty years, many material networks are required to construct scientific facts (Knorr-Cetina 1981; Lynch 1985; Latour and Woolgar 1986; Collins 1992). Laboratories, experiments, equipment, colleagues, funding, skills, academic papers: all of these elements are necessary to laboriously construct the "chains of reference" that give access to remote entities (Latour 1999b). In order to know, we need equipment and collaboration. Moreover, as soon as one inquires into science in the making instead of ready-made science, one sees that both the knowing mind and the known thing start to exist only at the very end of practical scientific processes. When everything is in place, when the chains of reference are strong enough, when there are no more controversies, I am becoming able to look at the majestic Californian sunrise and meditate about the power of habits that makes me go against the most rigorous fact: the earth is rotating. Thanks to numerous scientific networks that were put in place during the sixteenth and seventeenth centuries, I *gain* access to such—poor—meditation. Symmetrically, when everything is in place, when the chains of reference are strong enough, the sun *gains* its status of

known thing as one part of its existence—its relative immobility—is indeed being captured through scientific work and the maintenance of chains of reference. In short, what others have done and made durable enables me to think directly about the objective qualities of the sun. As soon as I can follow solidified scientific networks that gather observations, instruments, experiments, academic papers, conferences, and educational books, I *become* a knowing mind, and the sun *becomes* a known object. Cognitivism started at the wrong end: the possibility of scientific knowledge starts with practices and ends with known objects and knowing minds. As Latour (2013, 80) summarized it:

> A knowing mind and a known thing are not at all what would be linked through a mysterious viaduct by the activity of knowledge; they are the progressive result of the extension of chains of reference.

One result of this relocalization of scientific truth within the networks allowing its production, diffusion, and maintenance is that reality is not the sole province of scientific knowledge anymore: other entities that go through different paths to come into existence can also be considered *real*. Legal decisions (McGee 2015), technical artifacts (Simondon 2017), fictional characters (Greimas 1983), emotions (Nathan and Zajde 2012), or religious icons (Cobb 2006): even though these entities do not require the same type of networks as scientific facts in order to emerge, they can also be considered *real* since the world is no longer reduced to sole facts. As soon as the dichotomy between knowledge and mind is considered one consequence of chains of reference, as soon as *what is happening* is distinguished from *what is known*, there is space for many varieties of existents. By disamalgamating reality and knowledge, the universe of the real world can be replaced with the multiverse of performative beings (James 1909)—an ontological feast, a breath of fresh air.

Besides its problematic propensity to posit correspondence between things and minds as the supreme judge of what counts as real, another problem of cognitivism—or *computationalism*, or *computational metaphor of the mind*; at this point, all of these terms are equivalent—is its mitigated results when it comes to support so-called expert systems (Star 1989; Forsythe 2002).

A first example concerns what Haugeland (1989) called "Good Old Fashioned Artificial Intelligence" (GOFAI), an important research paradigm in

artificial intelligence that endeavored to design intelligent digital systems from the mid-1950s to the late 1980s. Although the complex algorithms implied in GOFAI's computational conception of the mind soon appeared very effective for the design of computer programs capable of complex tasks, such as playing chess or checkers, these algorithms symmetrically appeared very problematic for tasks as simple as finding a way outside a room without running into its wall (Malafouris 2004). The extreme difficulty for expert systems to reproduce very basic human tasks started to cast doubts on computationalism, especially since cybernetics—an cousin view on intelligence that emphasizes "negative feedback" (Bowker 1993; Pickering 2011)—effectively managed to reproduce such tasks without any reference to symbolic representation. As Malafouris (2004, 54–55) put it:

> When the first such autonomous devices (*machina speculatrix*) were constructed by Grey Walter, they had nothing to do with complex algorithms and representational inputs. Their kinship was with W. Ross Ashby' Homeostat and Norbert Wiener's cybernetic feedback … On the basis of a very simple electromechanical circuitry, the so-called 'turtles' were capable of producing emergent properties and behavior patterns that could not be determined by any of their system components, effecting in practice a cybernetic transgression of the mind-body divide.

Another practical limit of computationalism when applied to computer systems is the so-called frame problem (Dennet 1984; Pylyshyn 1987). The frame problem is "the problem of generating behaviour that is appropriately and selectively geared to the most contextually relevant aspects of their situation, and ignoring the multitude of irrelevant information that might be counterproductively transduced, processed and factored into the planning and guidance of behaviour" (Ward and Stapleton 2012, 95). How could a brain—or a computer—adequately select the inputs relevant for the situation at hand, process them, and then instruct adequate behaviors? Sports is, in this respect, an illuminating example: within the mess of a cricket stadium, how could a batter process the right input in a very short amount of time and behave adequately (Sutton 2007)? By what magic is a tennis player's brain capable of selecting the conspicuous input, processing it, and—eventually—instructing adequate behaviors on the fly (Iacoboni 2001)? To date, the only satisfactory computational answer to the frame problem, at least with regard to perceptual search tasks, is to consider it NP-complete, thus recognizing it should be addressed by using heuristics and approximations (Tsotsos 1988, 1990).[29]

Finally, the entire field of HCI can be considered an expression of the limits of computationalism as it is precisely because human cognition is not equivalent to computers' cognition that innovative interfaces need to be imagined and designed (Card, Moran, and Newell 1986). One famous example came from Suchman (1987) when she inquired into how users interacted with Xerox 8200 copier: as the design of Xerox's artifact included an equivalence between computers' cognition and human cognition, interacting with the artifact was a highly counterintuitive experience, even for those who designed it. Computationalism made Xerox designers forget about important features of human cognition, such as the importance of action and "situatedness" for many sense-making endeavors (Suchman 2006, 15). Besides refusing giving any ontological weight to nonscientific entities, computationalism thus also appears to restrain the development of intelligent computational systems intended to interact with humans.

Enactive Cognition

Despite its impressive stranglehold on Western thought, cognitivism has been fiercely criticized for quite a long time.[30] For the sake of this part II—whose main goal is, remember, to document the practices of computer programming because they are nowadays central to the constitution of algorithms—I will deal only with one line of criticisms recently labeled "enactive conception of cognition" (Ward and Stapleton 2012). This reframing of human cognition as a local attempt to engage *with* the world is here crucial as it will—finally!—enable us to consider programming in the light of situated experiences.

Broadly speaking, proponents of enactive cognition consider that agency drives cognition (Varela, Thompson, and Rosch 1991). Whereas cognitivism considers action as the output of the internal processing of symbolic representations about the "real world," enactivism considers action as a relational co-constituent of the world (Thompson 2005). The shift in perspective is thus total: it is as if one were speaking two different languages. Whereas cognitivism deals with an ideal world that is being accessed indirectly via representations that, in turn, instruct agency, enactivism deals with a becoming environment of transformative actions (Di Paolo 2005). Whereas cognitivism considers cognition as computation, enactivism considers cognition as adaptive interactions with the environment whose properties are offered to and modified through the actions of the cognizer. For

enactivism, the features of the environment with which we try to couple are then not fixed nor independent: they are continuously provided as well as specified based on our ability to attune with the environment.

With enactivism, the cognitivist separations among perception, cognition, and agency are blurred. Perception is no longer separated from cognition because cognizing is precisely about perceiving the takes that the environment provides: "The *affordances* of the environment are what it *offers* the animal, what it *provides* or *furnishes*, for either good or ill" (Gibson 1986, cited in Ward and Stapleton 2012, 93). Moreover, cognition does not need to be stuck in between perception and agency, processing inputs on representations to instructively define actions: for enactivism, the cognizer's effective actions both participate in, and are functions of, the takes that the sensible situation provides (Noë 2004; Ward, Roberts, and Clark 2011). Finally, agency cannot be considered the final product of a well or badly informed cognition process because direct perception itself is also part of agency: the way we perceive grips also depends on our capacities to grasp them. But the environment does not structure our capacity to perceive either; actions also modify the environment's properties and affordances, thus allowing a new and always surprising "dance of agency" (Pickering 1995). Perceptions suggest actions that, in turn, suggest new perceptions. From take to take, as far as we can perceive: this is what enactive cognition is all about.

This very minimal view on cognition that considers it "simply" as our capability to grasp the affordances of local environments has many consequences. First, enactivism implies that cognition (and therefore, to a certain extent, perception) is *embodied* in the sense that "the categories about the kind and structure of perception and cognition are constrained and shaped by facts about the kind of bodily agents we are" (Ward and Stapleton 2012, 98). Notions such as "up," "down," "left," and "right" are not anymore necessarily features of a "real" extended world: they are contingent effects of our bodily features that suggest a spatially arrayed environment. We experience the world through a body system that supports our perceptual apparatus (Clark 1998; Gallagher 2005; Haugeland 2000). Cognition is therefore multiple: to a certain extent, each body cognizes in its own way by engaging itself differently with its environment.

Second, enactivism implies that cognition is *affective* in the sense that "the form of openness to the world characteristic of cognition essentially depends on a grasp of the affordances and impediments the environment

offers to the cognizer with respect to the cognizer's goal, interest and projects" (Ward and Stapleton 2012, 99). Evaluation and desires thus appear crucial for a cognitive process to occur: no affects, no intelligence (Ratcliffe 2009, 2010). "Care" is something we take; what "shows up" concerns us. Again, it does not mean that our inner desires structure what we may perceive and grasp; our cognitive efforts also suggest desires to grasp the takes our environment suggests.

Third, enactivism considers that cognition can sometimes be extended: nonbiological elements, if properly embodied, can surely modify the boundaries of affective perceptions (Clark and Chalmers 1998). It does not mean that every nonbiological item would increase our capability to grasp affordances: some artifacts are, of course, constraining ongoing desires (hence suggesting new ones). But at any rate, the combinations of human and nonhuman apparatus, the *association* of biological and nonbiological substrates fully participate in the cognitive process and should therefore also be taken into account.

The fourth consequence of enactivism is the sudden disappearance of the frame problem. Indeed, although this problem constitutes a serious drawback for cognitivism by preventing it from understanding—and thus from implementing—the initial selection of the relevant input for the task at hand, enactive cognition avoids it by positing *framing* as part of cognition. Inputs are not thrown at cognizers anymore; their embodied, affective, and, eventually, extended perception tries to grasp the takes that the situations at hand propose. Cricket batters are trained, equipped, and concerned with the ball they want to hit; tennis players inhabit the ball they are about to smash. In short, whereas cognitivism deals with procedural *classifications*, enactivism deals with bodily and affective *intuitions* (Dreyfus 1998).

The fifth consequence is the capacity to consider a wide variety of existents. This consequence is as subtle as it is important. We saw that one deleterious propensity of cognitivism was to amalgamate truth (or knowledge) and reality: what counts as real for cognitivism is a behavior that derives from a true statement about the real world. Cognition is, then, considered the process by which we know the world and—hopefully—act accordingly. The picture is very different for enactivism. As enactive cognition is about interacting with the surrounding environment, grasping the takes it offers and therefore participating in its reconfiguration, knowledge can be considered as an eventual, very specific, and very delightful by-product of

cognitive processes. Cognition surely helps scientists to align inscriptions and construct chains of reference according to the veridiction mode of the scientific institution; however, cognition also helps writers to create fictional characters, lawyers to define legal means, or devout followers to be altered via renewed yet faithful messages. In short, by distinguishing knowledge and cognition—cognizers do not *know* the world but *interact* with it, hence participating in its reconfiguration—enactivism places the emphasis on our local attempts to couple with what surrounds us and reconfigure it, hence sometimes creating new existing entities.

Finally, enactivism makes the notions of symbols and representations useless for cognitive activities. Indeed, since the world is now a local environment whose properties are constantly modified by our attempts to couple with it, no need exists to posit an extra step of mental representations supported by symbols. For enactivism, there may be symbols—in the sense that a take offered by the environment may create a connection with many takes situated elsewhere or co-constructed at another time—but agency is always first. When I see the hammer and sickle on a red flag on a street of Vientiane, Laos, I surely grasp a symbol but only by virtue of the connections this take is making with many other takes I was able to grasp in past situations: TV documentaries about the Soviet revolution, school manuals, movies, and so on. In that sense, a symbol becomes a network of many solidified takes. Similarly, some takes may re-present other takes, but these re-presentations are always takes in the first place. For example, I may grasp a romantic re-presentation of a landscape at the second floor of Zürich's *Kunsthaus*, but this re-presentation is a take that the museum environment has suggested in the first place. This take may derive from another take—a pastoral view from some country hill in the late eighteenth century—but, at least at the cognitive level, it is a take I am grasping at the museum in the first place.

To sum up, enactive cognition starts with agency; affective and embodied *actions* are considered our way of engaging with the surrounding environment. This environment is not considered a preexisting realm; it is a collection of situations offering takes we may grasp to configure other take-offering situations. From this minimal standpoint, cognition infiltrates every situation without constituting the only ingredient of what exists. Scientists surely need to cognize to conduct experiments in their laboratories; lawyers for sure need to cognize to define legal means in their offices;

programmers surely need to cognize to produce numbered lists of instructions capable of making computers compute in desired ways; yet facts, legal decision, or programs cannot be reduced to cognitive activities as they end up constituting existents that populate the world. With enactive cognition, the emphasis is made on the interactions among local situations, bodies, and capabilities that, in turn, participate in the formation of what is *existing*, computer programs included. Cognition, then, appears crucial as it provides grips but also remains very limited as it is constantly overflowed: there is always something more than cognition. May computer programming be considered as part of this *more*. This could make it finally appear in all its subtleties.

The Constitution of Algorithms

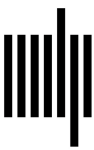
Ground-Truthing, Programming, Formulating

By: Florian Jaton

Citation:

The Constitution of Algorithms: Ground-Truthing, Programming, Formulating
By: Florian Jaton
DOI: 10.7551/mitpress/12517.001.0001
ISBN (electronic): 9780262363235
Publisher: The MIT Press
Published: 2021

The open access edition of this book was made possible by generous funding and support from Arcadia – a charitable fund of Lisbet Rausing and Peter Baldwin



The MIT Press

The open access edition of this book was made possible by generous funding from Arcadia—a charitable fund of Lisbet Rausing and Peter Baldwin.

This book was set in Stone Serif and Stone Sans by Westchester Publishing Services.

Library of Congress Cataloging-in-Publication Data

Names: Jaton, Florian, author. | Bowker, Geoffrey C., writer of foreword.
Title: The constitution of algorithms : ground-truthing, programming, formulating / Florian Jaton ; foreword by Geoffrey C. Bowker.
Description: Cambridge, Massachusetts : The MIT Press, [2020] | Series: Inside technology | Includes bibliographical references and index.
Identifiers: LCCN 2020028166 | ISBN 9780262542142 (paperback)
Subjects: LCSH: Algorithms--Case studies. | Computer programming--Case studies. | Algorithms--Social aspects. | Mathematics--Philosophy.
Classification: LCC QA9.58 .J38 2020 | DDC 518/.1--dc23
LC record available at https://lccn.loc.gov/2020028166