Computing Cultures: Knowledges and Practices (1940–1990)

Edited by

Arianna Borrelli and Helena Durnová

meson press

[1]

Synthetic Machines and Practical Languages: Masking the Computer in the 1950s

Mark Priestley

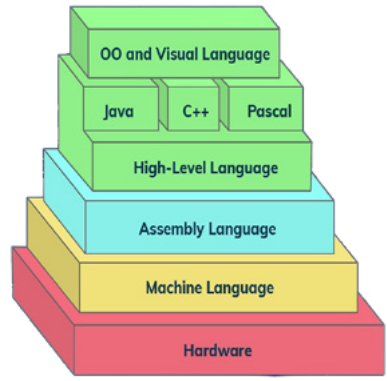
In the 1950s, programming was seen as an arcane practice and a likely bottleneck in the use of the new high-speed automatic computers. One proposal to solve this problem was to enable scientists and others to use computers directly in their work. To make this feasible, improved notations were designed to mediate between users and the machine. These notations are often understood purely linguistically, as ways to describe the domain-specific knowledge and techniques of a particular field in a machine-processible way. However, these improved notations also interpreted the computer for the non-specialized user. Using a new notation gave users access to a "virtual" or "synthetic" computational device often simpler or better tailored to a specific application area than the available physical machine. This chapter examines the synthetic machines of the 1950s, with a focus on those developed in connection with MIT's Whirlwind

machine, and shows how programming languages bridged apparently incompatible areas of knowledge, allowing users to grasp some of the ideas underlying the new computers and simultaneously express at least some of their domain-specific knowledge and problems in a computational form. In conclusion, it is suggested that this history challenges the simple hardware/software dichotomy which continues to shape much thinking about the computer itself.

The automatic digital computer was, from the moment of its appearance in the 1940s as a novel tool for scientific calculation, understood as a linguistic agent, an interpretation clearly visible in the common description of programming as the preparation of problems in "the language the machine can understand" (Burks, Goldstine, and von Neumann 1946, preface). This attribution of agency placed computers in a long, if subaltern, tradition of thought about "active machinery" defined in opposition to the idea of "brute mechanism" relying solely on the transfer of momentum (Riskin 2016). Understood as language users, computers shared the imaginative space occupied by Karel Čapek's robots, for whom a crucial boundary was the one across which orders were transmitted (Čapek 1920, 66). Even before it was a writing machine (Dick 2013), the computer was an obeying machine: Bell Labs mathematician George Stibitz defined a computer's "native intelligence" as "its ability to understand and obey a large number of orders" (Stibitz 1943), while Alan Turing, commenting on the Automatic Computing Engine's (ACE) versatility, suggested that "we should consider the machine as doing something quite simple, namely carrying out orders given to it in a standard form which it is able to understand" (Turing [1946]).

Language was initially seen as an organic part of a computer and programming as a communicative act between human and machinic agents, but throughout the 1950s, as computers became more widely available, this collaboration (or confrontation) was increasingly mediated by notations that were easier for humans to use and which the computer itself could translate into its "native language," a practice known as "automatic coding."[1] By the end of the

1 The metaphor of translation was widely used, despite the fact that translation between natural languages is nothing like the mechanical meaning-preserving transformations carried out by programming language compilers. There may be a connection here with the parallel project of machine translation, itself informed by cryptographic experience during the Second World War, for which see Léon in this volume.



[Fig. 1]: Typical contemporary representation of programming language hierarchies

decade, as the intermediate processing grew in sophistication, some of these notations had come to be described as "programming languages." Expected benefits from the use of mediating notations included reductions in the cost, staffing, and time required for program preparation, greater modifiability and portability of programs, and increased versatility of computers.

Nofre, Priestley and Alberts (2014) describe this process as a transformation from technology to language, driven by the need for computer installations to preserve their investment in software by ensuring interoperability among suites of incompatible machines. They thematize independence from the machine, noting in particular how universalizing projects such as Algol aimed to produce notations that were abstract and formal like the languages of mathematics and logic. In this narrative, notation evolves away from the machine, from machine code through symbolic assembly languages to higher-level programming languages, and the resulting hierarchical vision is still a widespread way of thinking about programming languages (see Figure 1). Language types are thought to represent specific "levels of abstraction" and the process that allows one to move down the hierarchy and eventually use the invisible machine lurking at the bottom of the stack is understood as one of translation.

Many researchers in the 1950s shared this vision. But automatic coding was also understood as a process that worked on the machines themselves, a technique for transforming an awkward and hard-to-use computer into one that was more accessible. These improved computers had no distinct material existence but possessed different logical characteristics from the underlying physical machines. Such spectral devices are now often referred to as "virtual machines," but to avoid confusion we will adopt a phrase coined at the time and refer to the computers conjured up by automatic coding systems as "synthetic machines" (Backus 1959). Massachusetts Institute of Technology (MIT) programmer Frank Heart made early use of this notion of synthesis,

describing automatic coding as "a class of methods which synthesize programmed digital computers 'within' actual digital computers" (Heart 1954).

In a recent discussion of the origins of computer science, Nofre has drawn attention to a related phenomenon, describing how "the conceptualization of the activity of computing became disentangled from the computer itself, a conceptual shift that was closely intertwined with *a process of dematerialization of the very notion of the computer*" (Nofre 2023, emphasis added). He focuses on the work of university staff anxious to escape the "commercial capture" of the computer and to establish a research role in computing distinct from the research and development carried out by commercial organizations such as IBM (International Business Machines). However, a differently motivated group inhabited the space between the universities and commercial computer developers, namely computer programmers and users concerned above all with making easy and economic use of the new machines in real-world applications. For this group, "dematerializing" the computer was not a way of making the physical machine disappear but rather of improving access to it.

This chapter traces the development of the discourse on synthetic machines as it evolved throughout the 1950s in parallel with linguistic accounts of programming. An early symposium paper described three automatic coding systems developed for MIT's Whirlwind computer (Adams and Laning 1954): these make convenient type specimens for a preliminary classification of synthetic machines and are described in sections 2 to 4. Sections 5 and 6 then consider how the idea of synthetic machines persisted, albeit sometimes in an attenuated and implicit way, as attention increasingly turned to the development of programming languages. But we begin by considering the basic socio-economic driver of this activity, namely the need to make programming easier.

1. Users as Programmers

As early as 1946 it was recognized that programming was a potential bottleneck in the use of electronic digital computers. In a lecture that summer, John Mauchly, principal designer of the ENIAC (Electronic Numerical Integrator and Computer), explained that a basic economic issue posed by the new machines was the incompatibility between extremely high speeds of computation and the time-consuming task of program preparation (Mauchly 1946a). Mauchly further noted that coding was the only aspect of computer use where substantial automation appeared not to be possible, but he hoped that well-designed instruction codes would lead to appreciable reductions in cost. A similar concern was expressed three years later at the initial meeting of the UK government's advisory committee on computers, when computation expert Douglas Hartree commented that "at least in the immediate future, it would

be difficult to find enough programme material to keep [100] machines in operation" (DSIR 1949).

One strategy to address this problem was to turn users into programmers. Rather than relying exclusively on a specialized cadre of machine operators, scientists and engineers would write their own programs with advice and guidance being provided by experts where necessary. This approach was pioneered by Maurice Wilkes's group at the University of Cambridge's Mathematical Laboratory, who organized a summer school on program design in September 1950, only a year after their new electronic computer, the EDSAC (Electronic Delay Storage Automatic Calculator), had become operational. The prospectus for the course described the "large contribution" that computers could make "to the solution of problems in mathematics, physics, engineering and other subjects," and while acknowledging the specialized nature of programming, stated that "the subject will be treated throughout from the point of view of the mathematical user" (UML 1950). Historian Doron Swade writes that the EDSAC approach created "the user as a new class of practitioner" (Swade 2011), and one former user described the laboratory as a "disparate" yet "close-knit" community of "mathematicians, crystallographers, molecular biologists, theoretical chemists, physicists, astronomers and statisticians" developing programs related to their individual research (Barron 2011).

These developments had a profound influence on the Whirlwind group at MIT, similarly situated in an elite techno-scientific institution and surrounded by scientists and engineers eager to get their hands on any new computational technology. Although Whirlwind was largely dedicated to specific military projects, a portion of its time was allocated without charge for general use by MIT staff and students (Adams 1952), and in the summer of 1948 Alan Perlis and John Carr were employed as research assistants to work alongside Charles Adams developing coding technique. Carr was subsequently awarded a Fulbright scholarship to study in France and on his journey home spent some time in Cambridge, returning to MIT filled with enthusiasm for what he had learned there (Wilkes 1985, 177). As Perlis recalled:

> [Carr] showed us something called a symbolic code that he had seen at Cambridge … Charlie Adams was the head of our programming group and we turned to each other [and said] "That's the way to do business!" and from that followed immediately invitations to the British to send us your rich, your intelligent, and so forth. (Perlis 1983, min. 17:47)

Wilkes himself visited MIT in August 1950 and discussed the EDSAC approach with Adams. Immediately after the meeting, Adams reported that Wilkes's "philosophy is both a confirmation and an extension of ideas" independently developed at MIT, noting in particular that experience with EDSAC confirmed the feasibility of "allowing users to learn coding by actually doing their own

problems" (WW M-1092). He listed three key technical points—fully automatic loading of programs from tape with no machine set-up required, the use of standard conventions and a subroutine library, and time-effective methods of trouble-shooting—that were also emphasized by Wilkes in a report on the Cambridge summer school (Wilkes 1950). While in the USA Wilkes gave two talks on "Operational experience on the EDSAC" at a meeting of the Association for Computing Machinery (ACM) in September. Adams described the meeting as "exceptionally stimulating" and noted that he had made personal contacts that had "encouraged work [at MIT] towards a practical, convenient, fully automatic means for handling coding details and computer set-up" (WW M-1096). Rounding off these early interactions, the Whirlwind group soon received a copy of the long and detailed internal report on EDSAC programming that was the basis for the famous textbook published the following year by Wilkes, David Wheeler, and Stanley Gill (WW M-1119).

As Whirlwind became increasingly reliable, Adams' applications group began to think how it could best be used as a research tool. They aimed:

> to show the feasibility of using a fast, general-purpose computer to solve many relatively small-scale scientific and engineering problems (where a single problem may require only a minute or two of machine time once a day or once a month). (WW SR-25, 28)

To ensure productive use of the machine's time under such a regime, "efficient methods of using the machine and efficient methods of locating troubles in programs" were essential, and Adams' group adopted and adapted the EDSAC framework, developing input and conversion programs that performed a similar job to EDSAC's Initial Orders, an extensive subroutine library, and a variety of tools for debugging. The end result of this work was the "Comprehensive System of Service Routines" (CSSR), which became available to Whirlwind users towards the end of 1952 (WW E-516).

These developments were bolstered by ongoing personal contacts. David Wheeler, creator of EDSAC's Initial Orders, visited Adams in September 1951 to discuss "programming techniques" (WW SR-27), and attendees at a significant informal discussion at a conference in December included Wilkes, Wheeler and John Bennett from Cambridge, and Adams, Carr and Jack Gilmore from MIT. Another discussant, perceiving the similarity between the two approaches, referred to a common "Wilkes–Adams philosophy" of machine organization (AIEE 1952, 113–14). Wilkes visited MIT after the conference and discussed Adams' plans for CSSR, suggesting that EDSAC's operations manager Eric Mutch be invited to MIT (Wilkes 1985, 180). Mutch duly spent the summer of 1952 at MIT contributing to the development and documentation of CSSR, while Wheeler taught on MIT's summer course (WW M-1546) and Wilkes visited briefly following an ACM meeting in Toronto (Wilkes 1985, 180). In 1953, Stanley

Gill spent the summer at MIT en route to the University of Illinois, teaching alongside Wilkes on that year's summer course (WW SR-35). He returned in the summer of 1954 and helped Adams deliver summer courses in Advanced Coding Techniques and Business Applications (WW SR-39).

Adams concluded that making programming simple enough for novices to learn with ease "essentially solves the difficult problem of where one finds a staff large enough to keep a high-speed computer busy on small-scale problems" (Adams 1952). Knowledge and experience gleaned from the EDSAC group's experience had, as Adams acknowledged, been pivotal in orienting the approach that Whirlwind's applications group took to programming after 1950. As they gained experience, however, MIT staff began to adapt and extend the EDSAC approach in various ways, as the following sections illustrate.

2. Interpretive Subroutines

By the end of 1952, then, Whirlwind was equipped with a programming environment similar in scope to EDSAC's and similarly centered on an extensive subroutine library providing easily reusable code for a range of common tasks. Both environments included a number of so-called interpretive subroutines often characterized as providing a computer with new synthetic machinery. This section describes a common use of interpretive subroutines, namely to provide programmers with a range of different number systems, an area of functionality referred to in CSSR as "programmed arithmetic."

One of a computer's most fundamental physical characteristics is its word length, which determines the quantum of storage directly manipulable by the machine's basic operations and the range of numbers that it can handle at the highest available speed. The word length on early computers represented a compromise between the need for numerical precision and the desire to simplify machine construction, and it was widely recognized that some applications would require higher levels of precision than that provided by the hardware. An early program to test Mersenne primes run at Manchester, for example, involved the manipulation of integers of several hundred digits (Newman 1949). Increased precision was provided by using more than one machine word to store a single number, the trade-off being that additional coding was required to implement basic arithmetic operations on these numbers, making programming more complex and programs longer and hence slower.

This was a pressing issue for Whirlwind whose intended use in real-time flight simulation mandated a particularly high speed of operation. Arguing that the "short computing sequences" involved in simulation did not require great precision, its designers had chosen a 16-bit word length with a fixed decimal point

(WW SR-5, 10–1). General scientific and engineering applications would require higher precision, however, and by the end of 1947 it had been decided to provide subroutines for operations on numbers stored in two consecutive words. Special orders were included in the code in the hope that coding with these "multiple-length" subroutines would be "no more complicated than for single-length numbers" (WW SR-3). Taking the "point of view of an individual coding a problem" (WW M-221), the aim was to allow the multiple-length operations to be used "as if they were built into the computer" (WW M-111), even at the cost of longer computing times. To a greater extent than other machines, then, general-purpose use of Whirlwind was predicated on the existence of coding techniques to circumvent hardware limitations. This work became Adams' responsibility, and by March 1948 he had drafted a set of subroutines to enable the execution of "any desired operation … on double-length numbers" (WW SR-6).

In the summer of 1948, John Carr worked largely on what he called a "two-register" coding method (WW SR-10). As Whirlwind was a fixed-point machine, all stored numbers had an absolute value less than one and to compute with numbers outside this range programmers had to explicitly code the relevant scaling factors, a time-consuming and error-prone process. So-called floating-point methods, such as Carr's two-register method, eased this burden by storing the scale factor as part of the number itself and automating the process of scaling. On fixed-point machines, floating-point arithmetic was provided by subroutines and involved the usual trade-off between ease of programming and speed of computation. Carr noted that his method could be extended to allow simple coding of programs that carried out computations on entities such as complex numbers, vectors, and matrices.

In the following year, Adams' group worked mostly on the coding of specific problems, as a result of which a definitive new order code was proposed for Whirlwind. The automatic subroutine orders were eliminated and Adams revised the existing subroutines for double-length and floating-point arithmetic (WW M-887). Investigations on making the use of subroutines as easy and efficient as possible were also carried out (WW E-329). As Whirlwind approached usability at the end of 1950, however, the contacts with the EDSAC group described above led to a change of technique and so-called interpretive subroutines became central to the provision of programmed arithmetic.

John Bennett, an Australian research student working in Cambridge, developed some interpretive subroutines for EDSAC around 1949 (Campbell-Kelly 1980). Bennett's immediate goal was to reduce the space required to store instructions so that longer programs could be written:

> a considerable saving can be effected in the space occupied by orders by using a special "order" code, of which each "order" specifies a sequence

> of machine orders, and by packing two such "orders" into a single storage location … An interpretive subroutine is required to interpret "orders" expressed and packed in this form. (Wilkes, Wheeler, and Gill 1951, 162–63)

Interpretive subroutines read and decoded "pseudo-orders," which were not part of a machine's actual order code. Execution of a pseudo-order involved the execution of multiple machine orders and possibly the invocation of other, non-interpretive, subroutines. Saving space was thus traded off against the increase in run time caused by having the pseudo-orders decoded by software rather than hardware. EDSAC programmers soon began to use interpretive subroutines in a more general way, however, to extend or even replace a machine's order code with new instructions, a practice that Bennett later described as a "fortuitous byproduct" of the technique (Bennett 1990).

Work on "floating-point and extra-precision interpretive subroutines" for Whirlwind started in February 1951 and that June it was reported that:

> Interpretive subroutines capable of interpreting a fairly general code of instructions, resembling quite strongly the ordinary Whirlwind I order code, have been written for performing several different forms of multiple-length arithmetic. (WW SR-26)

Enlargement of Whirlwind's memory at the end of the year made this work more urgent:

> The recent availability of 1024 registers of electrostatic storage has, for the first time, made feasible the undertaking of extensive calculations which require either extra precision … or floating point or both. (WW SR-28)

Throughout the early part of 1952 work continued on a variety of schemes. From the user's point of view, this "programmed arithmetic" appeared to alter Whirlwind's basic characteristics. As applications group programmer John Frankovich noted about a triple-precision interpretive subroutine he had written:

> Programmers using this subroutine can assume that Whirlwind is a floating binary point computer which performs arithmetic and logical operations upon numbers greater than or equal to 2^{-64} and less than 2^{+64} with 39 binary digit precision. (WW M-1517)

As well as altering machine characteristics, interpretive subroutines could compensate for missing hardware. The utility of the index registers added to the Manchester machine in 1949 had been widely noted and they were frequently simulated on other machines:

> The purpose of the early systems was to provide synthetic machines which had floating-point operations and often index registers (B-tubes), since the real machines did not. (Backus 1959, 234)

The features provided by early interpretive subroutines were thought of as extensions to the physical machine and programmers used them by switching between native and interpreted code as necessary. But no matter how similar they were to the built-in orders, a group of pseudo-orders had to be prefixed by a call to the subroutine that would interpret them, a process that required a certain level of care and sophistication on the part of the programmer. Interpretive subroutines were soon developed which removed this requirement, however, replacing all the native code with interpreted code and defining complete and self-contained synthetic machines.

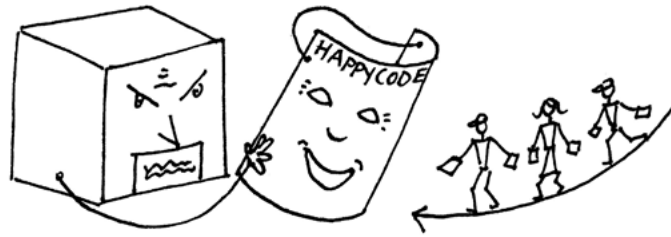
3. Synthetic Machines

MIT ran a two-week course on "Digital Computers and Their Applications" in July 1952 (WW SR-31, 17). The course was oversubscribed, attracting 95 attendees from a wide range of government and industrial organizations. Students gained practical experience of Whirlwind programming using a simplified version of the order code and special service routines that made data input and error detection easier. For the 1953 edition of the course, the applications group went one better and developed a new computer specifically designed to help students write, operate, and debug programs within the time constraints of the course. This new computer was not a physical machine, however:

> The MIT Summer Session Computer, hereafter referred to as the SS Computer, does not exist as an assembly of electronic apparatus; rather its realization is achieved by appropriate conversion, assembly, and interpretive routines operating in the Whirlwind I Computer. (WW SR-35, 47)

The SS computer had its own instruction code, supported by an input routine to translate alphanumeric tapes of programs into binary form and post mortem and error diagnosis routines designed to help inexperienced programmers locate errors quickly. Development started in June 1953, and after an intense programming effort by Frankovich, Gill, and Arnold Siegel the SS computer was ready for use on the course that August. Although some errors were uncovered, it was reported that "the overall operation of the simulated computer was quite satisfactory" (WW M-2415).

The SS computer's instruction code was translated by an interpretive routine running on Whirlwind. Unlike existing interpretive routines that extended the computer's built-in order code with commands for specific features like floating-point arithmetic, the SS computer was realized by a single interpretive

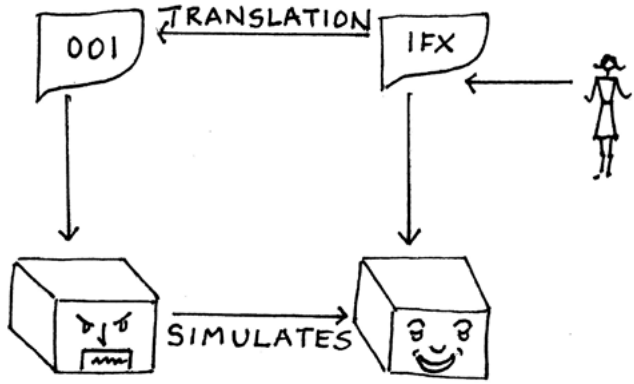


[Fig. 2] Pseudocode as a mask (after Hopper [MIT 1954])

routine which supported a complete and self-contained order code, "thus simulating a computer with characteristics quite different from those of [Whirlwind]" (WW M-2497). In place of Whirlwind's 16-bit registers, the SS computer's memory consisted of 28-bit words with a 4-bit tag which could store either fixed-point integers or floating-point numbers, and to enable easy modification of addresses within instructions it had seven "counters," or index registers, a hardware feature that was entirely absent from Whirlwind. The SS computer paradoxically allowed students to use Whirlwind precisely by concealing it from them.

In the fall semester the SS computer was used on two undergraduate courses, proving to be a "very satisfactory aid" and apparently achieving its goal of making programming easier: it was reported that more than half of SS programs, all written by beginner programmers, "have given correct results on the first or second run" (WW SR-36). But this benefit came with an associated cost, as interpreted SS programs were estimated to run approximately 10 times slower than those written in Whirlwind's native code. Low execution speed was not a critical factor in student programs but probably restricted wider use of the system, although it was used in early 1954 on a project to analyze data obtained from blast furnaces (WW M-2812, 7). The SS computer code written for this project was soon rewritten using standard CSSR routines (WW M-2870, 13), but the SS computer was used as the basis for the Single Address Computer, one of two new synthetic machines developed for a business computing course offered in the summer of 1954 (WW SR-39). The other, the Three Address Computer, provided students with access to a completely different type of machine, being "designed to simulate an actual medium speed drum computer" (WW M-2900).

By 1954 interest in automatic coding was gaining momentum, and in May the US Navy's Office of Naval Research (ONR), which had funded much computing research (including the Whirlwind project) since the war, organized a symposium bringing together researchers in the field (ONR 1954). The ONR also sponsored a further course offered at MIT that summer, on "Advanced Programming Techniques," organized by Adams, Gill, and Whirlwind



[Fig. 3] Interaction of the linguistic and machinic aspects of automatic programming

programmer Donn Combelic (MIT 1954). Attendees at the MIT course included Grace Hopper, who had chaired the ONR symposium, and John Backus, leader of IBM's new Fortran project.

At the MIT course, Gill described what he called the "language aspect" of automatic coding, pointing out that programmers could write programs using "pseudocodes" that were different from the instruction code of the machine and that required translation into the machine's own code before they could run. In a cartoon on the front cover of the notes from the course (Figure 2), Hopper imagined this situation from the users' point of view, depicting the pseudocode as a mask transforming a threatening and unapproachable computer into a more programmer-friendly machine.

As the example of the SS computer illustrates, such masked machines were discussed and experienced by users as if they were machines in their own right. Figure 3 expands Hopper's cartoon by making explicit the relations of translation and simulation that supported the use of synthetic machines. The diagram shows how a pseudocode could not only provide a new way to program a given computer but could also, by defining a new synthetic machine, be experienced as providing a replacement for that computer.

A few years later, Backus expanded on this idea in the context of programming languages. He defined a "synthetic machine" as one "equipped to accept statements of procedures in a language other than the 'hardware language,'" and suggested that such machines could be evaluated in the same way as "real machines," by assessing their programmability and speed of execution (Backus 1959). The consistency of the programmers' experience is guaranteed by the fact that the diagram of Figure 3 is expected to commute, in

the sense of category theory: the effect of executing a pseudocode operation on the synthetic machine is the same as the effect of running the translated operation on the simulation.

Interpretive routines were not the only way in which synthetic machines could be created. Frank Heart drew attention to the variety of practice while talking about the increasingly widespread phenomenon of synthetic computers:

> Subprogramming, conversion routines, and executive routines are discussed as members of a class of methods which synthesize programmed digital computers "within" actual digital computers. Ease of programming, training, and design testing are pointed out as major reasons for turning to such procedures. (Heart 1954)

Synthetic computers were rife in the 1950s, as Hopper observed in her concluding remarks at the ONR symposium:

> Miss Moser has said that "any reasonably good compiler is a pseudo-computer" … Thus a "second-level" of computers is being "built." Computers with a wide range of instructions, easy to program, and able to be shipped by mail are being created to exploit basic hardware. (ONR 1954, 148)

Some were "imaginary machines" with no physical counterpart, often conceived and developed like the SS computer to make programming simpler. For example, once the University of Michigan's digital automatic computer MIDAC had been installed in 1953, instructors found that the difficulty of scaling, instruction modification, and binary representation meant that in the time available students only managed to complete trivial programs. "To alleviate these problems it was decided that a new computer was needed: one designed to make programming easier" (Perkins 1956). The resulting "pseudo-computer," EASIAC (EASy Instruction Automatic Computer), was implemented in the same way as the SS computer: source code was translated by an input routine into an intermediate form that was then interpreted to execute the program, and a "mistake printout program" provided diagnostic information to assist in debugging.

Other developers of synthetic machines aimed to replicate physical machines. In 1958, the ACM published lists of 20–30 such efforts, many involving IBM machines as the company delivered a series of increasingly capable computers (ACM 1958a, 1958b). An older machine could be simulated on its more powerful successor as a way of preserving investment in software written for the older machine, or for other reasons. For example, the US Council for Economic and Industry Research announced a program for the IBM 704 that simulated the less powerful but widely used IBM 650. The synthetic machine was deemed valuable because "it can serve to alleviate a temporary

overload of 650 facilities, run programs with excessive storage requirements or act in many ways as a buffer for various emergency conditions" (ACM 1958a). But it could also make sense to simulate a new computer on one less powerful, for example to develop and test software for a promised new machine so that it could be put into service as soon as it was delivered. This had been the motivation behind one of the earliest applications of interpretive subroutines, in Manchester:

> We are building a new machine, the code of which differs from the code of the machine now in existence … when the new machine is finished, several months will have to be spent in getting the new routines right. We are trying to avoid this waste of time by developing a scheme which will enable us to test the new programmes on the old machine. (Bennett, Prinz, and Woods 1952)

4. Synthesizing New Kinds of Computer

Designers often aimed to make the new orders provided by interpretive subroutines and synthetic machines resemble those of existing codes. In a project run by the General Electric Company, for example,

> every effort was made to keep the coding treatment of suboperation instructions as close as possible to that of [IBM] 702 instructions so that from the programmer's standpoint there would be no difference between the sixty pseudo operations made available by SCRIPT and the thirty-seven "built in" 702 commands. (Thompson 1956, 9)

A more radical proposal was to use forms of notation other than computer code, for example by allowing programming to take place directly with mathematical formulas. As early as 1949, Mauchly had proposed an interpretive "program which would accept algebraic equations as ordinarily written, which program would perform the indicated operations" (Remington Rand 1952). As implemented for UNIVAC (UNIVersal Automatic Computer) in 1952, this "Short Code" still required algebraic symbols to be hand-coded as numbers, but the third MIT system discussed at the 1954 ONR symposium allowed equations in recognizably mathematical notation to be presented directly to Whirlwind for solution.

J. Halcombe Laning, a mathematician working in MIT's Instrumentation Laboratory, visited Whirlwind in late 1951, returning early in 1952 with a proposal for a program to solve differential equations using the Runge–Kutta method. This was accepted by Adams's applications group and coded with the assistance of John Carr, using around five hours of Whirlwind runtime. In April, Laning was joined by Neil Zierler, also of the Instrumentation Lab, and the project expanded to encompass a range of programs, including a

pseudo-random number generator. Laning's original plans expanded in this context and at the end of the year it was reported that:

> An interpretive program has been written to read in and solve sets of algebraic equations of a certain class, each problem being presented to the machine as a flexo-tape in essentially ordinary mathematical notation. This program has at present partial provision for the handling of more elaborate problems; e.g. iterations and the computing of special functions. (WW SR-32, 26)

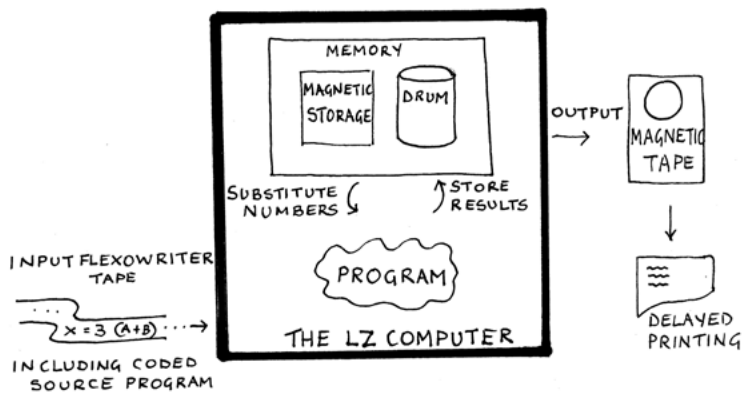
The keyboard on the Flexowriter, the teleprinter used to prepare Whirlwind's input, contained a number of mathematical symbols, including parentheses, symbols for arithmetical operations, and exponents, thus allowing complex equations such as $d = a\,(b + c)^2/(c - a)\,a^{-3}$ to be punched directly onto tape (Laning and Zierler 1954, 6). Equations had to be expressed as strict sequences of characters, but this format nevertheless marked a great gain in readability and usability over systems such as Short Code.

Work on the interpretive program continued throughout 1953. In November it was used on another project to compute a Fourier transform, and on December 1, Laning described it at an internal MIT seminar as an "interpretive program for mathematical equations" that

> will accept algebraic equations, differential equations, etc., expressed on Flexowriter punched paper tape in ordinary mathematical notation (within certain limits imposed by the Flexowriter) and automatically provide the desired solutions. Additional features such as the automatic computation of special functions and the automatic solution of differential equations are being included. (WW SR-36, 17)

An internal report on the program was issued in January 1954 and it was used for a variety of test applications, but never gained wide usage (Laning and Zierler 1954). As with the SS computer, reports regularly stressed that programs "operated satisfactorily on their first trial" (e.g. WW M-2699), with the suggestion that the system made programming easier, but it is likely that the slow running of programs compared with other Whirlwind programming systems (a slowdown factor of 10 was often cited) hindered its wide acceptance. It was used on at least one other project during 1954, and at the end of the year Laning began an effort to rewrite it. Substantial amounts of time were dedicated to this, but by early 1955 Laning had been forced to abandon the project through pressure of other work.

The Laning and Zierler program is famous as an early, if not the first, example of a system that might be described as a programming language rather than simply an autocode. This is understandable, as its input notation looks significantly different from, say, the code of the SS computer. But there are



[Fig. 4] The structure of the synthetic machine underlying the Laning and Zierler system

also similarities between the two systems: both were implemented by interpretive routines that blocked access to Whirlwind's native code and provided the user with a synthetic machine significantly different from Whirlwind.

Laning and Zierler's report reflected both sides of the machine–language duality. While the title described the system in linguistic terms as "a program for translation of mathematical equations," the text stated that "the effect of our program is to create a computer within a computer." Laning and Zierler did not explicitly describe the synthetic machine underlying their system but its high-level structure can be inferred from their text (see Figure 4). The LZ computer, as we might call it, differed substantially from synthetic machines such as the SS computer, which shared with their host machines a familiar computer architecture organized around a single, addressable memory and were programmed with instructions consisting of operation codes and addresses.

The memory of the LZ computer held only floating-point numbers. These were not stored in addressable memory locations but, following normal mathematical practice, were understood to be the values of variables such as the *a*, *b*, *c* and *d* appearing in the example equation above. The memory was divided into two areas: "magnetic storage" and "the drum." By default, variables were stored in magnetic storage, but could be explicitly placed on the drum by means of an ASSIGN instruction.

The input Flexowriter tape contained a sequence of equations interspersed with control instructions; we will refer to this form of the program as the LZ code. Equations and instructions could be numbered to provide targets for jump instructions, and the sequence ended with a STOP instruction. Equations had the form <*variable*> = <*expression*>, and the expression on the right-hand

side was regarded as "a set of computational operations to be performed." The number computed by these operations was "stored in the registers assigned to the letter that appears on the left hand side." A PRINT statement allowed the values of variables to be recorded on magnetic tape for later printing in "equational form," for example, *x = +.314157 + 01*.

Although referred to as an interpretive program, the input routine of Laning and Zierler's system translated each equation into a block of code that was then stored on Whirlwind's drum. This code used the CSSR routines, and when an equation had to be evaluated the corresponding code was read from the drum and interpreted in the usual way (Adams and Laning 1954, 64). However, neither this code nor the original equations were stored in the memory of the synthetic LZ computer, which held only numbers. The LZ computer therefore broke with one of the fundamental design principles of the modern computer, that of storing data and code in a single shared memory.

The LZ system thus differed from a purely interpretive system in that the LZ code, the equational form of a program, was *absent* when the program was being run: Whirlwind stored only the translated form of the program, while the memory of the LZ computer held, as described above, only numbers. However, the LZ code was at the same time *present* to the programmer and the LZ computer and used by both as a way of understanding the run-time behavior of the program. The programmer naturally imagined sequential execution of the code within the computer, each step involving the evaluation of an equation or the carrying out of an operation, and the system supported this interpretation in various ways. For example, the locations at which error conditions like overflow occurred were reported with reference to the LZ code rather than its translated form. The diagram in Figure 4 depicts this anomalous situation by showing the LZ code as an amorphous and unlocatable entity within the LZ computer but separate from its memory.

One consequence of the absence of the LZ code was that programmers could not change it while the program was running. And yet, the ability to modify a program at run time was widely recognized to be a key feature on which the utility of high-speed automatic computers rested. A form of substitution gave the LZ computer an analogous capability. Before evaluating an equation or executing an instruction, the synthetic machine substituted the variables appearing in the LZ code with their values, rounding the floating-point numbers retrieved from memory to integers when they were needed as index variables or to denote line numbers in the program. The equations and instructions of the LZ code were therefore not changed at run time, but before execution they were transformed by substitution into a form that contained no variables.

In the organization of its memory and the way it handled code, then, the LZ computer radically differed from contemporaneous hardware. Memory was not structured as a sequence of addressable words and numbers, and instructions were not stored in the same space. Numbers were stored in named variables, but the way in which instructions were stored remained undefined. The LZ code provided no way of directly accessing instructions, and these were therefore modified not by explicit programming but by a process of substitution taking place before execution. The "computer within a computer" that Laning and Zierler designed was therefore significantly different from Whirlwind, altering and in some ways restricting the familiar "von Neumann" design. Even though they were not explicit about its properties, the LZ system enabled Whirlwind to simulate a novel computer design.

Users of the LZ computer could imagine, design, and debug their programs purely in terms of the LZ code, even though that code was not directly involved in the execution of a program. In this sense, programmers worked directly with the synthetic machine rather than with Whirlwind. Nevertheless, some aspects of the underlying system remained visible, such as the floating-point number format inherited from CSSR's programmed arithmetic. The distinction in the LZ computer's memory between the magnetic store and the drum was of course a direct reflection of Whirlwind's hardware and gave rise to some apparently arbitrary restrictions in the LZ input language, such as a limitation on the number of variables that could be stored in the magnetic store. There were also indirect effects on programs, such as run-time properties arising from the speed difference between magnetic storage and the drum.

In the historiography of computing Laning and Zierler's system is usually presented as a linguistic innovation, but this was predicated upon a parallel innovation in machine design, as described in this section. This interplay between language and synthetic machine continued to play an important role in the development of increasingly sophisticated programming languages, as the following section illustrates.

5. Programming Languages

The emergence of programming languages in the second half of the 1950s has been seen as a move away from the machine towards a purely linguistic notion of programming, an interpretation visible in diagrams showing a linguistic hierarchy built on top of the physical computing machine (see Figure 1). The examples of the SS computer and the Laning and Zierler system suggest that the development of automatic programming was not simply a smooth progression upward through such a hierarchy, however. Both systems were motivated by a desire to move away from particular features of Whirlwind and both presented ideas about what a more suitable computer would be. They

did not represent a move away from the machine *in general*, but rather moves away from a *particular* machine or machine design in search of machines and notations better suited for specific purposes.

While the SS computer was programmed with a conventional machine code, Laning and Zierler's work emphasized the search for a more human-facing programming notation. Although it was not widely used at MIT it served as an inspiration for subsequent projects, as Hopper observed in 1956:

> The presentation by Charles Adams of a description of Laning and Zierler's system of algebraic pseudocoding for the Whirlwind computer led to the development of Boeing's BACAIC for the 701, FORTRAN for the 704, AT-3 for the UNIVAC, and Purdue System for the Datatron and indicated the need for far more effort in the area of algebraic translators. (ONR 1956, 2)

IBM's Fortran project was started in early 1954, and the ONR symposium in May brought together researchers from MIT and IBM. Fortran project members John Backus and Harlan Herrick gave a talk on IBM's earlier Speedcoding system and listened to Adams briefly describe the LZ system. Shortly after the meeting Backus wrote to Laning, who arranged a demonstration of his system for the IBM group in June. Backus and other members of the Fortran group were also present at MIT's summer course in August, where the LZ system was discussed in more detail. Reflecting on this episode, Backus noted that for many years he had believed that Fortran had been directly inspired by the LZ system. Although he stepped back from this position somewhat, it is clear that in 1954 the IBM researchers were fully aware of Laning and Zierler's work, and there are many similarities between the two projects (Backus 1980, 129–30).

Although the Fortran project emphasized the definition of the language and its translation into efficient machine code, early documentation described Fortran as a system which "in effect transforms the 704 into a machine with which communication can be made in a language more concise and more familiar than the 704 language itself" (IBM 1956, 2). Like the LZ system, Fortran assumed a synthetic machine whose language "closely resembl[ed] the ordinary language of mathematics." The system translated a "source program" written in this language into a machine language program. In broad outline, the implicit Fortran machine was very similar to the LZ computer described above: numerical data was held in variables rather than explicitly addressed memory locations, programs were written as sequences of numbered statements that were not accessible for modification at runtime, and the variables appearing in statements were substituted with their current values before execution.

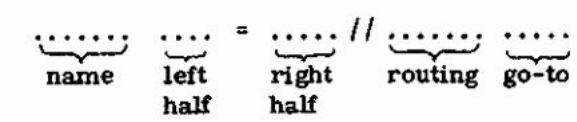
COMIT, a development led by Victor Yngve of MIT's Mechanical Translation (MT) group, provides a contrasting and less familiar example of a programming language developed in the mid-1950s. Like Laning and Backus, Yngve wanted to allow a group of non-specialists, in this case linguists rather than scientists and engineers, to do their own programming. The emerging requirements of computational linguistics, however, led to a very different language and machine model.

Yngve described COMIT as a solution to the problem of allowing non-experts to use the computer:

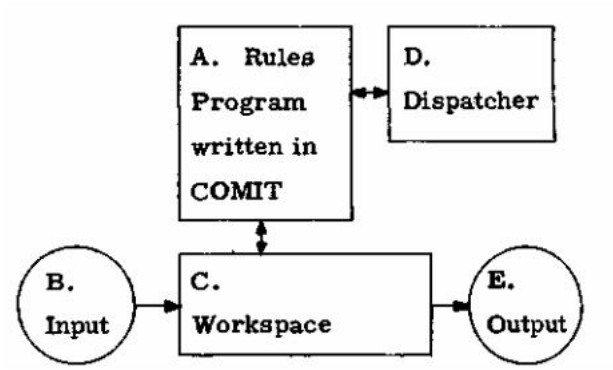
> It has been the custom for the linguist who wanted to try out a certain approach to mechanical translation to ask an expert programmer to program his material rather than to learn the art of programming himself … The linguist has not become aware of the full power of the machine, and the programmer, not being a linguist, has not been able to use his special knowledge of the machine with full effectiveness on linguistic problems.
>
> The solution offered here to these difficulties is an automatic programming system. The linguist writes the results of his research in a notation or language called COMIT, which has been specially devised to fill his needs. The programmer writes a conversion program or compiler capable of converting anything written in this notation into a program that can be run on the computer. Thus the expense, time, and effort needed to separately program each linguistic approach is saved, and, even more important, the linguist is given direct access to the machine. He becomes more fully aware of its potentialities, and his research is greatly facilitated. (Yngve 1958, 25)

To a first approximation, COMIT can be described as an attempt to take at face value Noam Chomsky's description of a grammar as "a device … for producing the sentences of [a] language" (Chomsky 1957, 11). Yngve had criticized the prevailing "word-for-word" approach to MT, arguing that adequate translation could only be obtained by taking into account the structure of the sentence being translated (Yngve 1955). In his view translation was better modeled as a two-stage process of *decoding* the input sentence into its constituent structure and then *encoding* this structure in the target language. However, he noted that progress towards a "sentence-for-sentence" approach was hindered by the lack of an adequate account of sentence structure.

In 1955, the MIT group appointed four new staff, including Chomsky, and Yngve came to believe that Chomsky's ideas about grammar could provide a account of sentence structure adequate to ground a new approach to MT. By June 1957, Chomsky's work and the arrival of MIT's new IBM 704 computer had inspired Yngve and his colleague Hugh Matthews to begin sketching plans for a programming notation for linguists. They started working on a



[Fig. 5a] "The five sections of a rule in COMIT."



[Fig. 5b] "How a COMIT program works in the computer" (Yngve 1958)

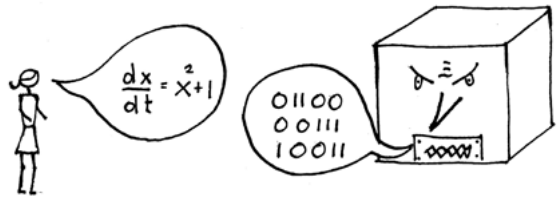
"construction routine," or program for generating sentences, but in collaboration with the programming staff of MIT's Computation Center (including Fortran veteran Sheldon Best) soon redirected their efforts towards creating a more general-purpose programming language, COMIT. Yngve published the first full description of the language in early 1958, but its implementation was not complete until 1961.

Chomsky described grammars as sets of rules capable of generating all the sentences of a language. Phrase structure rules generated what Chomsky called the "kernel sentences" of the language by rewriting strings of symbols to produce new strings:

> [w]e interpret each rule X → Y ... as the instruction "rewrite X as Y." (Chomsky 1957, 26)

Transformations operated on kernel sentences to produce more complex sentences. Transformations were also expressed as rules, but in order to determine when a transformation was applicable it was necessary to have some knowledge of the phrase structure of the kernel sentence. Transformations were therefore more complex and harder to apply than the simple production rules that described phrase structure.

Yngve and Matthews strove to organize COMIT around a single type of rule adequate to express both the phrase structure and the transformational structure of the sentences of natural language. It turned out that to make Chomsky's rules suitable for practical tasks in computational linguistics, such



[Fig. 6a] The linguistic gap between programmer and computer.

[Fig. 6b] Mediated interaction between programmer and computer

as the production of specific sentences, it was necessary to add a lot of detail and control information to the basic production rules. In their final form, COMIT rules comprised five distinct sections, as shown in Figure 5a.

As with the LZ system and Fortran, COMIT programmers could think of themselves as coding for a dedicated COMIT machine rather than for the 704 itself. Yngve described the structure of the synthetic COMIT machine in the diagram shown in Figure 5b.

The workspace was the memory holding the linguistic material being worked on. Unlike the memory of conventional computers, it was not subdivided into addressable locations. Instead, it held a single long string that was transformed by the application of rules. Like the LZ computer, the COMIT machine held its program—a set of rules—separately from the workspace in an undefined space that was not accessible to the program itself. The dispatcher held detailed information about the control flow required in the program.

With COMIT, Yngve adopted the strategy that had proved successful with algebraic and formula translation languages: notation familiar to a group of domain experts was transformed into a machine-processible form while at the same time the computer was transformed into a machine that could directly interpret the new notation and thereby be more accessible to those experts. The following section examines this double transformation from a more general perspective.

6. Programming in the Trading Zone

By the early 1960s the idea of programming languages was well established and developments were in hand for a wide range of application areas and purposes. This section considers what was distinctive about the way programming languages were thought to solve the problem of using digital computers.

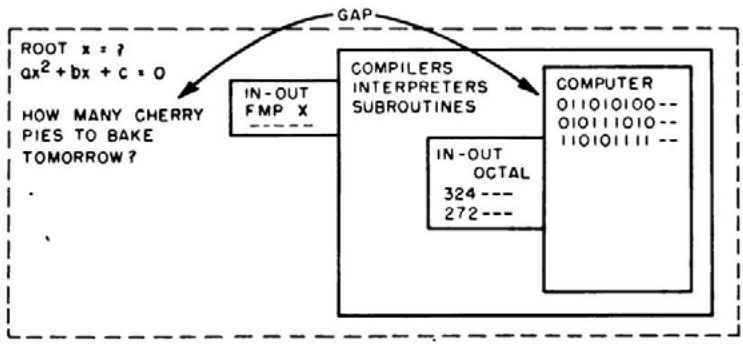
The problem was often posed in terms of a *gap* between human and machine caused by the incompatibility of human and machine language (Figure 6a), a gap that initially could only be filled by "hard, complicated, routine, clerical human labor" to translate often informally understood computations into machine code (Brown and Carr 1954, 84).

Anticipating Peter Galison's well-known metaphor, Joseph Wegstein, a programmer at the US National Bureau of Standards, colourfully described the gap as a trading zone:

> Sending a problem into a computer nowadays is like sending an expedition to Africa to trade with the natives. It has to be complete with the missionaries to translate and possibly convert the natives. If the missionaries speak only French and the native tongue then we must speak French but if the missionaries speak English too then everything is all right. (Wegstein 1956, 6)

The details of Wegstein's analogy do not really withstand scrutiny, but as the decade progressed the gap developed into a productive transactional zone linking humans and machines. As with commercial trading zones, it was gradually bridged by the development of new technological and social *matériel* and the evolution of new means of communication. Wegstein depicted the technical innovations as a cantilevered structure of systems software reaching out across the gap from the machine towards the programmer (Figure 7).

The automatic programming systems of the early to mid-1950s mostly aimed at crossing the gap only in this direction, producing computer-based coding systems and notations that were more accessible to humans than purely binary code. But as the decade progressed, the relationship between programming languages and application areas was increasingly understood as involving more than simply making the computer easier to use. This transition is implicit in comments made by Backus in 1958 which echo Yngve's description of COMIT as a language that linguists would find easy to use in their research:



[Fig. 7] Bridging the gap (Wegstein 1956)

> Automatic programming techniques seek to make it easy to do a desired job on a computer. This is accomplished by providing a language in which the statement of the desired process is natural and concise, together with some means of causing a machine to carry out the stated process. (Backus 1959, 233)

The important point here is the idea that programming languages enable problems and processes to be expressed in notations that can be mechanically interpreted, in contrast to automatic programming systems which were designed primarily to make machines easier to use. The direction of travel across the gap has been reversed: instead of the reconfiguration of the machine, emphasis is increasingly given to the redescription of problems. RAND programmer William Orchard-Hays highlighted this point, writing that a feature of programming languages that distinguished them from other automatic coding systems was "that 'languages' develop around applications" (Orchard-Hays 1961, 295), and a similar thought was expressed by Bert Green, a programmer in MIT's Lincoln Laboratory:

> Digital computers have outgrown the problems for which they were designed … Complex mathematics, business records and inventories, control of man-machine systems, and "intelligent" automatic systems are all jobs for digital computers today … It has become necessary to develop new programming languages that are particularly suited to the more complicated problems. (Green 1961, 3)

The role of programming languages in human–machine interaction is illustrated in Figure 6b which, in contrast to Wegstein's diagram, highlights the medial nature of programming languages as opposed to autocodes and also the double movement involved in bridging the gap: as well as transforming the computer by the provision of system software "extend[ing] the computer's reach across the gap towards the problem," programming languages extended

the problem towards the computer. In the context of the trading zone metaphor, programming languages can be understood as creoles: Fortran neither reflected the full complexity of mathematical notation nor allowed the fine control over computational procedure that machine code did, but it did facilitate communication and, most importantly, allow trade in the form of productive use of the computer to take place.

In general, expressing a particular application in a programming language also transformed it by causing it to be reformulated in an explicit, computable form. Computerization therefore led to epistemic innovation, as Harry Larson, editor of a special computer issue of the *Proceedings of the IRE* (Institute of Radio Engineers), noted: "when computers are applied to a field of activity outside the fields that primarily employ mathematics and logic in their theories, *that field undergoes greater formalization*" (Larson 1961, 5, emphasis in original).

As Larson pointed out, increased formalization was not particularly visible in the specific field of scientific computation, likely for the reason that decades of research in numerical analysis had already reduced large areas of mathematics to a fully computable form—and indeed automated it with both analog and digital devices. Although revisions of technique within numerical mathematics were needed to deal with the new machines' patterns of computation, the worldview of applied mathematicians was largely unchanged by the advent of computers.

As illustrated above, however, the situation was very different in other fields such as linguistics. Although the point is somewhat controversial, it seems clear that the Chomskian revolution of the mid-1950s was at least partly informed by Chomsky's awareness of formal logic, computability theory, and the practical development of computers (Golumbia 2009). Yngve and Matthews transformed Chomsky's rather simple and abstract rules into a notation suitable for automation and the application of Chomsky's ideas to tasks other than grammatical analysis.

Programming languages were not viewed solely as intermediate notations, however, but also, like the simpler autocoding systems, as systems which could materially transform the computer. Backus's comments on programming languages quoted above continued as follows:

> The synthetic machines provided by [systems like Fortran] range in speed from virtually the speed of the corresponding real machine to perhaps one-half of that speed … Considerable effort was devoted to making the speed of the synthetic FORTRAN machine as close as possible to that of the IBM 704, its real counterpart. (Backus 1959, 234)

Discussing the origins of the Common Business-Oriented Language, COBOL, Ben Allen made a related observation, commenting that a programming

language represents "a particular *agreement* on what a digital computing machine can or should be" (Allen 2018, 25, emphasis in original). We can add that a complementary move is also involved, namely an agreement on how a particular application area should be rendered computable. Programming notations are Janus-faced, simultaneously taking up a specific perspective on the computing machine itself and also on a given application domain, a point made very early on by John Mauchly:

> The particular instruction code which is selected for the machine will depend in part upon the kinds of components available for building the machine and in part upon the emphasis we place upon convenience in handling certain classes of problems. (Mauchly 1946b)

Mauchly went on to note that "there will always remain some kinds of problems for which a special machine will be better suited" than an "all-purpose machine." In effect, the development of programming languages took Mauchly at his word, with each language postulating the existence of a synthetic machine suited for a particular application area.

7. Masking the Machine

This chapter has argued that much early work on improving programming *notation* was simultaneously understood as a way of providing programmers with better *machines*, most of which were realized through simulation rather than as distinct physical computers. These synthetic machines can be provisionally classified as follows:

1. those that alter or extend the hardware and order code of an underlying computer (see section 2);
2. those that replace an underlying computer with a simulation of a complete machine of the same type, but with different physical characteristics (see section 3);
3. those that replace an underlying computer with a simulation of a machine and order code of a significantly different type (see section 4).

This complicates the simple interpretation, given for example by Gill in 1954, of higher-level notations as purely linguistic constructs transformed into the "native language" of the computer through a process of translation. As Gill himself explained to a notable visitor to Cambridge, the computer was not singular in the machine rooms of the early 1950s:

> And we were, I remember, enthusing to Claude Shannon about the horizons opened up by the idea of an interpretive routine. And I remember saying to Claude, "It's, it's as if we had several computers here. We have the EDSAC and then we have the floating-point computer designed by Tony Brooker which happens to run on EDSAC." (Gill 1972, 44)

At the start of the 1960s, translation and simulation were understood to be two sides of the same coin:

> Building the translating routine for [an automatic programming] language is simply the problem of constructing a routine which makes a given physical sequential machine imitate the behaviour of a nonexistent, but still sequential, machine whose language is the particular automatic coding language. (Elbourn and Ware 1962, 1060)

The interplay between these processes, depicted above in Figure 3, was described in more formal terms by Arthur Burks:

> Automatic programming also involves simulation. Let U_c be a computer that operates with a machine language inconvenient for the programmer to use. The programmer uses his more convenient programmer's language. It is theoretically possible to build a machine which will understand the programmer's language directly; call this hypothetical computer M_p. Let P_t be the program (written in the language of machine U_c) which translates from the programmer's language to the machine language of U_c. Then U_c, operating under the direction of P_t, will compute the same results as M_p. That is, U_c with P_t simulates M_p. (Burks 1966, 14–15)

As Burks suggests, languages and machines occur in pairs. A programming language can be thought of as a dyad consisting of a notation and a synthetic machine on which that notation might be executed.[2]

It is striking, however, that as the 1950s progressed synthetic machines were less fully and explicitly described than the corresponding notations, a tendency that became ever more prominent in the 1960s. Elbourn and Ware offered an explanation for this. Designed and built as "number crunchers," the aboriginal theory of computers was based upon decades of research into numerical analysis. But as a greater range of applications were programmed, computers came to be understood more generally as "symbol crunchers," or information-processing machines. This called into question many existing ideas and raised the question of what the basic theory of the machines might be, if not mathematical:

> Language … is the best framework we have found for understanding what an automatic computer is functionally and what it can do … a most important area of research for advancing the uses of computers is the study of language in the abstract. (Elbourn and Ware 1962, 1062)

2 The contrast between this dyadic view and the purely linguistic view perhaps foreshadows two major approaches to programming language semantics that emerged from the 1960s onward. While denotational semantics constructed an abstract domain of mathematical meanings for programs considered purely as syntactic entities, operational approaches specified a language's meaning in terms of the behavior of an abstract machine.

The study of language in the abstract and, as compiler technology became increasingly sophisticated, an emphasis on the purely linguistic relationship of translation, appears to have drawn attention away from the complementary study of machines and simulation.

Elbourn and Ware's comments reinforce the thesis that the move towards a purely linguistic account was driven by the desire to construct a theoretical model of the computer as an object of university-based research (Nofre 2023). But the work described in this chapter had a different and more practical motivation, namely the desire to make computer programming accessible to ever wider groups, firstly to workers in mathematical sciences who were not computer experts, and subsequently to scientists in other areas and to potential computer users in commerce and industry. Far from wanting to create abstract models of the physical devices, this approach made use of novel programming notations and languages as ways to facilitate access to the physical machines to solve very practical problems.

In a general account of the emergence of computer science, Stephanie Dick has written:

> From the heterogeneous assemblies of people that converged at early computer centers, several distinct computing communities emerged, each held together by a particular vision of how computers should be used … in order to create their communities, practitioners promoted a particular image of the computer and a corresponding identity for themselves, often disassociating the new technology from certain of its inaugural associations. (Dick 2015, 60–61)

Distinct communities emerged even within the relatively narrow field of programming research in the 1950s. This chapter has described how the vision of making programming easier, of providing enough material to keep the new machines busy, generated a view of programming and the relationship between languages and machines that was distinct from the more purely linguistic, universalizing approach that emerged from concerns with portability (Nofre, Priestley, and Alberts 2014). The distinction between the two communities is subtle, as both approaches drew upon ideas about language and many actors participated in both, but it can best be appreciated by keeping in mind the differing problems being addressed. To put it another way, in the course of the 1950s, two distinct visions of programming languages emerged, one emphasizing the creation of a single, ideal, and universal language, while the other, more pragmatic approach, viewed languages as mediating specific relationships between machines and users in given application areas.

The "disassociation," to use Dick's term, of programming from the physical machine is common to both approaches, and was noted early on:

> Indeed, to many [programmers], understanding a "machine" is in reality understanding one or more programming systems for the machine, with scarcely a thought for the processor [i.e. compiler] which vivifies the system, much less for the machine itself. (Orchard-Hays 1961)

What characterized the second approach was its explicit recognition of the importance of synthetic machines and the way which it recognized that programming languages brought machines and applications together by transforming both, and in conclusion we suggest that Hopper's suggestive imagining of a programming system as a mask captures a distinctive and flexible interpretation of programming notations which in fact rarely, if ever, allowed the underlying hardware to be completely forgotten.

At first sight, Hopper's cartoon can be read as a straightforward depiction of abstraction, with the physical computer hidden behind the mask labeled "HAPPYCODE." Yet this is belied by the structure of the image: the programmers are confronted not by an abstract interface, but by a machine which appears to be in the process of donning a mask. The mask is not an impermeable interface that completely obscures the machine, however, but rather something that allows it to present a different face to the world. In many situations, indeed, the function of a mask is not simply to hide something, but rather to conceal it with a view to enabling behavior and interactions that would not otherwise be possible. Surgical masks can enable the resumption of normal social behavior in a pandemic, masks worn at *bals masqués* or carnivals can license interactions that might otherwise be unacceptable, and shamanic masks are an important technique for enlarging the range of accessible spiritual or religious experiences.

It was rarely, if ever, possible to make the physical machine completely disappear, and indeed for some, like John McCarthy in the earliest stages of the development of Lisp, while the existence of an automatic programming system could shield novice programmers from the physical machine, access to it might be essential for expert use:

> The source language is mainly independent of the machine being used, except that the provisions for referring directly to machine registers and their parts, which we believe must be included in any powerful source language, have been worked out only for the IBM 704 … The conventions defining a duffers' system might contain such a statement in order to keep the duffers uncontaminated by any actual knowledge of the machine … Programming has not yet reached a state where all kinds of calculations can be described with no regard at all for the fact that the machine has a storage which is divided into numbered registers. (McCarthy 1957, 1, 10, 12)

In the work described in this chapter, the computer was concealed in order to make it more accessible, to allow complicated tasks to be programmed

with greater ease, and above all to enable experts from other fields to make productive use of the machine. The social dynamics powering this work were far more complex than a simple narrative of progressive abstraction would suggest, and the resulting ontology of synthetic machines complicates the familiar binary opposition between abstract and concrete.

Early versions of this work were presented at a session of the 26th ICHST conference, which took place online in 2021, and at a colloquium at the Technische Universität, Berlin, in 2022. I thank Liesbeth de Mol, Arianna Borrelli and Adrian Wüthrich for invitations to give these talks. The work on this chapter has been greatly stimulated by my participation in the ANR project "PROGRAMme" (ANR-17-CE38-0003-01), directed by Liesbeth de Mol. Thanks also to Karen Keogh for the hand-drawn figures.

Archival Sources

TNA: The UK National Archives, Kew, London.
WW: MIT Libraries, Project Whirlwind Collection (MC-0665).

Key to Project Whirlwind memoranda and reports cited:
- E-329: Adams, C., "Techniques for Using Standard Automatic Subroutines" (10 February 1950)
- E-516: Uchiyamada, H., "Comprehensive System of Service Routines" (17 December 1952)
- M-111: Everett, R. R., "Block Diagram Work" (6 October 1947)
- M-221: Bi-weekly report (23 January 1948)
- M-887: Bi-weekly report (5 August 1949)
- M-1092: Bi-weekly report (1 September 1950)
- M-1096: Bi-weekly report (15 September 1950)
- M-1119: Bi-weekly report (27 October 1950)
- M-1517: Bi-weekly report (6 June 1952)
- M-1546: Bi-weekly report (3 July 1952)
- M-2415: Bi-weekly report (20 September 1953)
- M-2497: Bi-weekly report (2 November 1953)
- M-2699: Bi-weekly report (21 February 1954)
- M-2812: Bi-weekly report (2 May 1954)
- M-2870: Progress Report (13 June 1954)
- M-2900: Progress Report (11 July 1954)
- SR-3: Summary Report 3 (December 1947)
- SR-5: Summary Report 5 (February 1948)
- SR-6: Summary Report 6 (March 1948)
- SR-10: Summary Report 10 (July 1948)
- SR-25: Summary Report 25 (March 1951)
- SR-26: Summary Report 26 (June 1951)
- SR-27: Summary Report 27 (September 1951)
- SR-28: Summary Report 28 (December 1951)
- SR-31: Summary Report 31 (September 1952)
- SR-32: Summary Report 32 (December 1952)
- SR-35: Summary Report 35 (September 1953)
- SR-36: Summary Report 36 (December 1953)
- SR-39: Summary Report 39 (September 1954)

References

ACM. 1958a. "Techniques Department: News." *Communications of the ACM* 1, no. 2.

———. 1958b. "Techniques Department." *Communications of the ACM* 1, no. 11.

Adams, Charles W. 1952. "Small Problems on Large Computers." *Proceedings of the 1952 ACM National Meeting (Pittsburgh)*: 99–102.

Adams, Charles W., and J. Holcombe Laning, Jr. 1954. "The M.I.T. Systems of Automatic Coding: Comprehensive, Summer Session, and Algebraic." In *Automatic Programming for Digital Computers*, edited by Office of Naval Research, Department of the Navy, Washington DC.

AIEE 1952. *Review of Electronic Digital Computers*: *Joint AIEE-IRE Conference, Philadelphia,* December 10–12,1951.

Allen, Ben. 2018. "Common Language: Business Programming Languages and the Legibility of Programming." *IEEE Annals of the History of Computing* 40, no. 2: 17–31.

Backus, John 1959. "Automatic Programming: Properties and Performance of FORTRAN Systems I and II." In *Mechanisation of Thought Processes*. National Physical Laboratory, Symposium No. 10.

———. 1980. "Programming in America in the 1950s." In *A History of Computing in the Twentieth Century*, edited by Nicholas Metropolis, Jack Howlett, and Gian-Carlo Rota, 125–35. Orlando: Academic Press.

Barron, David. 2011. "EDSAC: A Programmer Remembers." *The Computer Journal* 54, no. 1: 139–42.

Bennett, John M. 1990. "Early Computer Days in Britain and Australia—Some Autobiographical Snippets." *Annals of the History of Computing* 12, no. 4: 281–85.

Bennett, John M., Dietrich G. Prinz, and Mary Lee Woods. 1952. "Interpretative Sub-routines." *Proceedings of the ACM National Meeting (Toronto):* 81–87.

Brown, Jeremy H., and John W. Carr. 1954. "Automatic Programming and its Development on the MIDAC." In *Automatic Programming for Digital Computers*, edited by Office of Naval Research, Department of the Navy, Washington DC.

Burks, Arthur W. 1966. "Editor's introduction." In *Theory of Self-reproducing Automata*, edited by Arthur W. Burks, 1–28. Champaign: University of Illinois Press.

Burks, Arthur W., Herman H. Goldstine, and John von Neumann. 1946. *Preliminary Discussion of the Logical Design of an Electronic Computing Instrument*. Princeton: The Institute for Advanced Study.

Campbell-Kelly, Martin. 1980. "Programming the EDSAC: Early Programming Activity at the University of Cambridge." *Annals of the History of Computing* 2, no. 1: 7–36.

Čapek, Karel. 1920. *R.U.R. (Rossum's Universal Robots).* Page reference to the translation by Claudia Novak-Jones in P. Kussi, ed. 1990. *Toward the Radical Center: A Karel Čapek Reader.* North Haven: Catbird Press.

Chomsky, Noam. 1957. *Syntactic Structures*. Berlin and New York: Mouton de Gruyter.

Dick, Stephanie. 2013. "Machines Who Write." *IEEE Annals of the History of Computing* 35, no. 2: 88.

———. 2015. "Computer Science." In *A Companion to the History of American Science*, edited by Georgina M. Montgomery and Mark A. Largent, 55–68. Hoboken: Wiley-Blackwell.

DSIR. 1949. "Department of Scientific and Industrial Research, Advisory Committee on High-Speed Calculating Machines: Minutes of the First Meeting," October 31, 1949 (TNA DSIR 10/317).

Elbourn, Robert D., and Willis H. Ware. 1962. "The Evolution of Concepts and Language of Computing." *Proceedings of the IRE* 50, no. 5: 1059–66.

Gill, Stanley. 1972. *Oral History Interview with Henry S. Tropp*. National Museum of American History, Computer Oral History Collection.

Golumbia, David. 2009. *The Cultural Logic of Computation.* Cambridge, MA: Harvard University Press.

Green, Bert F. 1961. "Using Computers to Study Human Perception." *Educational and Psychological Measurement* 21, no. 1: 227–33.

Heart, Frank E. 1954. "The Programmed Synthesis of Digital Computers within Digital Computers." Paper presented at *ACM National Meeting (Ann Arbor)*, June 23–25, 1954. Abstract printed in *Computers and Automation* 4, no. 4: 23.

IBM. 1956. *FORTRAN Programmer's Reference Manual*. Applied Science Division and Programming Research Dept. International Business Machines Corporation, New York NY.

Laning, J. Holcombe, and Neil Zierler. 1954. "A Program for Translation of Mathematical Equations for WWI." MIT Instrumentation Laboratory Engineering Memorandum E-364.

Larson, Harry T. 1961. "The Computer Issue." *Proceedings of the Institute of Radio Engineers* 49, no. 1: 4–7.

Mauchly, John. 1946a. "Digital and Analog Computing Machines." In *The Moore School Lectures: Theory and Techniques for Design of Electronic Digital Computers*, edited by Martin Campbell-Kelly and Michael R. Williams, 25–40. CBI Reprint Series for the History of Computing, vol. 9. Cambridge, MA: MIT Press and Tomash Publishers.

———. 1946b. "Machine Design and Instruction Codes." In *The Moore School Lectures: Theory and Techniques for Design of Electronic Digital Computers*, edited by Martin Campbell-Kelly and Michael R. Williams, 453–61. CBI Reprint Series for the History of Computing, vol. 9. Cambridge, MA: MIT Press and Tomash Publishers.

McCarthy, John. 1957. "A Proposal for a Compiler." Internal MIT Memorandum to P. M. Morse, December 13, 1957.

MIT. 1954. *Digital Computers: Advanced Coding Techniques.* Course Documentation, Massachusetts Institute of Technology / Summer Session.

Newman, Maxwell H. A. 1949. "Some Routines Involving Large Integers." In *The Early British Computer Conferences*, edited by Michael R. Williams and Martin Campbell-Kelly. CBI Reprint Series for the History of Computing, vol. 14, 73–74. Cambridge, MA: MIT Press and Tomash Publishers.

Nofre, David. 2023. "'Content is Meaningless, and Structure is All-important': Defining the Nature of Computer Science in the Age of High Modernism, c. 1950 – c. 1965." *IEEE Annals of the History of Computing* 45, no. 2: 29–42.

Nofre, David, Mark Priestley, and Gerard Alberts. 2014. "When Technology Became Language: The Origins of the Linguistic Conception of Computing, 1950–1960." *Technology and Culture* 55, no. 1: 40–75.

ONR. 1954. *Automatic Programming for Digital Computers.* Office of Naval Research, Department of the Navy, Washington DC.

———. 1956. *Advanced Programming Methods for Digital Computers.* Office of Naval Research, Symposium Report ACR-15.

Orchard-Hays, William. 1961. "The Evolution of Programming Systems." *Proceedings of the Institute of Radio Engineers* 49, no. 1: 283–95.

Perkins, Robert. 1956. "EASIAC, a Pseudo-computer." *Journal of the ACM* 3, no. 2.

Perlis, Alan J. 1983. "The Evolution of Software." Video Recording of Lecture. Computer History Museum, catalogue number 102703091.

Remington Rand. 1952. *UNIVAC Short Code.* Remington Rand Inc., Eckert-Mauchly Division, Philadelphia, PA: October 24, 1952.

Riskin, Jessica. 2016. *The Restless Clock.* Chicago: University of Chicago Press.

Stibitz, George. 1943. "Capabilities of Relay Interpolator." In *A Statement Concerning the Future Availability of a New Computing Device.* Applied Mathematics Panel, NRDC, Note No. 7, November 12, 1943.

Swade, Doron. 2011. "Inventing the User: EDSAC in Context." *The Computer Journal* 54, no. 1: 143–47.

Thompson, Charles E. 1956. "Development of Common Language Automatic Programming Systems." In *Advanced Programming Methods for Digital Computers,* edited by Office of Naval Research, Symposium Report ACR-15.

Turing, Alan M. n.d. [1946]. "Proposed Electronic Calculator." TNA DSIR 10/385.

UML. 1950. "Summer School in Programme Design for Automatic Digital Computing Machines." University of Cambridge Mathematical Laboratory. TNA DSIR 10/319.

Wegstein, Joseph H. 1956. "Automatic Coding Principles." In *Advanced Programming Methods for Digital Computers,* edited by Office of Naval Research, Symposium Report ACR-15: 3–6.

Wilkes, Maurice V. 1950. "Automatic Computing." *Nature* 166, no. 4231: 942–43.

———. 1985. *Memoirs of a Computer Pioneer.* Cambridge, MA: MIT Press.

Wilkes, Maurice V., David J. Wheeler, and Stanley Gill. 1951. *The Preparation of Programs for an Electronic Digital Computer.* Cambridge, MA: Addison-Wesley.

Yngve, Victor. 1955. "Sentence-for-sentence Translation." *Mechanical Translation* 2, no. 2: 29–37.

Yngve, Victor. 1958. "A Programming Language for Machine Translation." *Mechanical Translation* 5, no. 1: 25–41.