9

The Style of Sources
Remarks on the Theory and History of Programming Languages

Wolfgang Hagen
Translated by Peter Krapp

"Le style c'est l'homme même"

—Buffon

"…the procedure…whose mastery exerts a decisive influence over the style and the quality of the work of a programmer."

—Wirth

"The principles of style, however, are applicable in all languages"

—Kerningham

"A style of programming is based on an idea (possibly speculative) of a 'calculator'…which is to work off the program"

—Stoyan

"Can we be liberated from the von-Neumann-style?"

—Backus

Let us begin with a thought experiment.

1. The Library of Modern Sources

Imagine a large library called "The Library of Modern Sources." What would the blueprint for such a source museum look like? We might arrange departments and divide them along the large groups of programming languages: procedural (FORTRAN, ALGOL, PASCAL and C), functional (LISP, ML or MIRANDA), declarative (LOGO or PROLOG), and a new department for the object-oriented (SMALLTALK, EIFFEL, C++); parallel and neuronal languages would be in development. However, as a minimal condition, all sources ever written must be available as code, plus the descriptions and sources of all compiler-, interpreter-, and assembler-codes that belong to each system, including all those texts, blueprints, tables and diagrams describing the machines that run those codes. We would collect everything belonging to the symbolic register of our project: everything written, all knowledge on each code ever put into signs and sketches. Would this be sufficient? Does the history of source code include only what has been registered symbolically? I am afraid our library would in the end also have to include the real machines themselves, plus running versions of all operating systems and development platforms. Otherwise the bulk of older code would remain incomprehensible. But does anyone have even a minimal number of computers that ever ran at their disposal? No. Our collection would at best document those codes that never actually ran on a machine.

With regret our thought-museum would have to declare at the entrance that the history of all computer source code, their "historia rerum gestarum" as the Roman historians put it, coincides with the "res gestae," with the events themselves. Thus our thought experiment is a logical impossibility.

2. An Archive of Source Codes

Not to speak of the problem of procuring the code, the source of the sources. Where are they kept? Of course I have to trust the archives of Big Blue and MIT, Xerox, Apple, Microsoft, of the U.S. Navy and U.S. Air Force, but I doubt whether they actually kept the source codes of the UNIVAC or the BINAC, of the DUAL and the SHACO machines at Los Alamos, or of the IBM 701 or the 704, or of the prototypes of these machines. We recognize a grave and fundamental problem of the archive. Between an abstract computer representing every artificial code and a real machine to be controlled there is at least one generational process of another machine. Some German computing manuals call that generic machine which compiles the source code into machine code the "translator." Concerning the language of abstract computers, the "source," the operation of the translator is like a crossing of that subterranean river that the Greeks identify with the realm of death. The transition from symbolic program-text to real machine-code kills the language which sets it in motion. Often enough, the transition is one from being (software) to nothingness (hardware). To try to describe the running machine, and we have to do this often enough, we resort to another symbolic machine, the "assembler." Its "dis-ensembling" debugging—a process one can set up even if the source is written in "higher" languages—discursifies the running program for us into another, new language which has little to do with the source code. This path offers only a literally ideal and deceptive continuity. In reality, there is a "breakpoint" between symbolic machines and their real runtime. We call that breakpoint the halt where the machine is still running and where we place our symbolic vocabulary in between—but we do not actually displace anything: a masked interrupt (i.e., a piece of software which is inherent in the machine itself) is tracing and debugging at this location of the last communication with the machine. Hardware description language (HDL) may show in a diagram or in a temporal logic design how this interrupt works. If such breakpoints do not clarify what happens in the pipelines of machine hardware, there is still one last remedy left: the "post mortem dump" or post mortem debugging. In the end, a conceptual last judgment. "But," as Hegel says in the *Phenomenology of Spirit*, "the life of Spirit is not the life that shrinks from death and keeps itself untouched by devastation, but rather the life that endures it and maintains itself in it."[1]

3. "There Is No Software"

At this stage, I want to point out a figure of secret idealism nesting in the thought of these symbolic connections, in the transitions and transformations of abstract machines of computer programs; a secret idealism that is so seductively obvious. You may find the same figure in a number of American philosophers of "electric language," such as Michael Heim or Jay David Bolter. They say computer programs and systems are a "demanding collection of programmed texts that interact with each other."[2] But in real computing machines, texts do not interact: electron diffusion and quantum mechanic tunnel effects run over all chips, n-million transistor cells in n^2 correlations. If you want to call this occurrences "interaction," then you have to face the current technology production treating such interactions as systemic barriers, as physical side-effects, distortions, etc.[3] Symbolic software programs and the real runtime actions of computing machines are not joined in the interference of a continued universality, as the Gutenberg Galaxy has taught us to expect, but differ discontinuously, often almost grotesquely. If this were not the case, we would not see the recurring waves of painful software crises on micro- and macro-levels. "If programming was a strictly deterministic process following firm rules," the Swiss computing scholar Niklaus Wirth writes laconically, "it would have been automatized long ago."[4] Or else there would be, as Kittler provocatively put it, "no software."

4. Genealogy of Computer Media

Let us return to the early days of the computer, when one could easily assert, "there is no software." We do not have to discuss the fact that computers are based on the mathematical model of the Turing machine, and that this model was widely known to American scientists since the late 1930s. But Turing did not write about "software" in 1937—he offered a negative proof showing that a general algorithm for the general solution of general mathematical problems cannot be proffered. It follows that whatever can be addressed in a finite description by an algorithm is positively calculable. Oswald Wiener's idea of the entire world as a universe of folded Turing machines has less to do with Turing himself than with the impact the massive development of digital storage media has made on us.[5] We may suspect that the number of bytes on all computers in the world has already surpassed the number of letters in all the books in the world. This entropic digression of storage media sends us to another mathematical model, equally responsible for the current media sea-change: namely Shannon's mathematical theory of communication developed in the 1940s. Were we to situate the computer medium as it appears to us today in an evolutionary model—as we will try for the sake of the argument—then this model would project the development of three overlapping evolutionary complexes onto a time axis, neither interchangeable nor initially corresponding. They are:

- the mathematical model of calculability
- the engineering technique of storage development and addressing
- the mathematics and physics of communications technology.

I hasten to add that the strangely nontransparent "terminus a quo" of our problem, namely the question of the origin of programming languages, could also be articulated in three sections:

1. a first approach of programming languages, in the early 1950s, which follows symbolic contiguities, but no mathematical mode
2. in the late 1950s, the counter-movement of mathematically oriented functional and declarative languages that must idealize the machine from which they abstract

3. in the early 1970s and again in the late 1980s, the incision of simulation, marking the entry of earlier media (writing, image, sound) in the "computer"—making it a medium in itself. This last step leads to the object-oriented languages that are defined as neither procedural nor functional.

This background for a sketch of the three large groups of programming styles cannot be reconstructed otherwise. Thus each theory of programming languages, like any media theory, observes one *a priori*, namely its own: it can only be written up as historical theory. I want to reconstruct the first of the three steps I recognize in the development of programming languages, and thus in the end show a bit more of what one may call style.

5. "Stilus" or Metonymic Style

There are few poetological and linguistic concepts that describe a fundamental property of language and are simultaneously an effect of that same property. "Style" is one of them. Latin etymology indicates that "stilus" originally denoted a sharp stake, used to break up soil in agriculture, in wartime in traps. Later it became the name for a tool for making marks in wax, made of wood, horn, or metal, one end sharp, the other flat to smoothen the slab.[6] "Stilus," therefore, is a tool to write and to erase the written as well, a writing/erasing tool. Hans Ulrich Gumbrecht noted the metonymy that this binary tool of contradiction commences in the first century BC. For as ancient Rome turned into an imperial monarchy and with the development of books and libraries the written document began to replace the politics of forensic orators, some hastened to invert the facts of the visible world on their writing pads. So "stilus" means not only pen, but also "the use of the pen . . . the practice of writing, the manner of the writer," and even the "language of the writer" itself, the "stilus artifex." This "feedback"-inversion of the "stilus" became the primary stylistic act of language. Gumbrecht quotes Cicero: "Vertit stilum in tabulis suis, quo facto causam omnem evertit suam."[7] "He inverts in his writing how he acts to destroy all his things." The reversion of the pen, "stilum vertere," now means to distinguish the written by writing. This is metonymy and thus is the thing—to use the "style" is the style, and vice versa. Can the written, in software, make the written unrecognizable?

Cicero's style relies on omission, setting up the economy of erasure for the greatest effect in speech. A differentiated conceptual history follows throughout all Latin, Scholastic and Renaissance rhetoric and poetics until the Enlightenment offers this motto: "style is the man himself." Style is supposed to be the bourgeois subject of speech and writing, the producing author liberating the free, but paid genius, from the formal prescriptions of past centuries. There are after all some few books in computer science that warn against such unfettered subjective style exercises, but this is merely stupidity on both sides. In their Kafka studies, both Friedrich Kittler and Bernhard Siegert demonstrate how in the end style is always derived from media effects, which with the rise of technical media exert their effect also on literature.[8]

In the first century BC, when "stilus" became metonymical and in the concept of style a scriptural tool became its own effect, the next media transition, a new media *a priori*, sets its shift in motion. For the arms and hands of Cicero, Sallust, Terence or Caesar rest less and less on papyrus and more and more on the *caudex*, the bound book of wax slabs, as well as the *codex* of vellum. Both *caudex* and *codex* complement the y-axis (of a script roll and continuity of text) with an x-axis of the page, and a z-axis, the number of pages. This brings writing into a three-dimensional order for the first time in history, and thus makes it accessible as script/text with the aid of numeric and other symbols: page numbers, paragraphs, indents, marginalia, sections, chapters, comma, hyphen, colon and semicolon. Now one can know of a writing that one does not see, but which is referred to as if it were visible, a *virtualizing* effect of the codex and the birth of style. With the rise of the printing press in the 15th century, all those codex-symbols, never incorporated in the standard alphabet, were incorporated into another code—that of mathematics.[9] And so we arrive at the symbolic basis of the style of programming.

6. The Signs of the First Programs

What was the first computer program and what were its first symbols? This is like asking: what were the first computers in history? It is well known that there are no satisfying answers. The MARK computers by Howard Aiken, Konrad Zuse's Z3, Turing's COLOSSUS and ACE-machine, and Mauchly-Eckert's ENIAC participate in the first developments. There is a ubiquity of beginnings, a dissipation of the mechanic start up of the computer.

6.1. Dissipative Evolution

The historical facts about the huge engineering boom in America in building calculators and computing machines after 1941 are well known enough. That technological evolution pivoted upon the need to calculate a growing number of "firing tables" for all possible flying objects and projectiles, and later the famous ENIAC addressed the so-called hydrodynamic Los Alamos Problem. The numeric calculation of shock wave equations of an H-bomb implosion was carried out on the ENIAC during three long months, beginning in October 1945.[10]

6.2. Zuse

The loner Zuse, in his sparse rooms in the German Experimental Institute for Aeronautics around 1940, cut off from the world of scientists and lost in the chaos of the National Socialist research administration, nevertheless deserves a good credit for important parts of the evolution of the computer—particularly with respect to the deep structure of developing programmable calculators, which cannot be restricted to the efforts of Turing or von Neumann, but go back to the mathematical problems of Hilbert, Ackermann and Frege. Zuse responds directly to them, as Turing and von Neumann did implicitly. Therefore we can assume that even without Turing, the debates about the axiomatics and foundations of mathematics in the 1920s could have led to the computer. Given what we know today, the difference between German and American computer development is not caused by a scientific gap (with mathematical and logical bases), but a due to the huge differences in military and industrial support. America and England had grown their military-industrial-academic complex continuously, arguing with what they thought the Germans were going to develop before all else and reinforcing their efforts by the massive investment in Los Alamos. In 1942, the "Manhattan Project" united 2,500 scientists in America and England, Goldstine, Mauchly, Teller and von Neumann, as well as Turing. Konrad Zuse, in contrast, worked partly on his own and partly for an organization pulled apart by the competing interests of the Wehrmacht, the SS, the Navy and the Airforce, misjudged and not institutionally recognized or supported. However in the end, his influence after the war would even extend to the conferences of ALGOL 58 and ALGOL 60, if only because the Z4 and Zuse's own programs were saved, and some of his developments were known to Rutishauser, who brought the knowledge and Zuse's name to the U.S.[11]

6.3. ENIAC

One always says very generally that John von Neumann introduced elementary concepts of the Turing machine into the computer boom dominated by the Americans. But we can specify the historical time and place, namely the ENIAC team between spring and fall 1944. Here von Neumann is on a team with Brainerd, Goldstine, Eckert, Mauchly, and Burks, and these team-mates share the claim to have invented sequentially stored programming, although it is usually associated only with von Neumann—all the more since this was an engineering advance that inevitably followed from the architecture of the ENIAC.[12]

6.3.1. Research Inputs With its 18,000 tubes, the ENIAC was the first computer of such size—the model for the admired post-war "electronic brains" and the first proof in the history of technology that switches of this magnitude were possible. Contemporaries also knew which major engineering trends helped develop this computer. Contributing to the hardware were:

- the electronics industry which had reached a peak boom in receivers and transmitters in its radio days
- the mechanical and electromechanical industry active in arms production
- robust 100KHz tubes from radar technology.

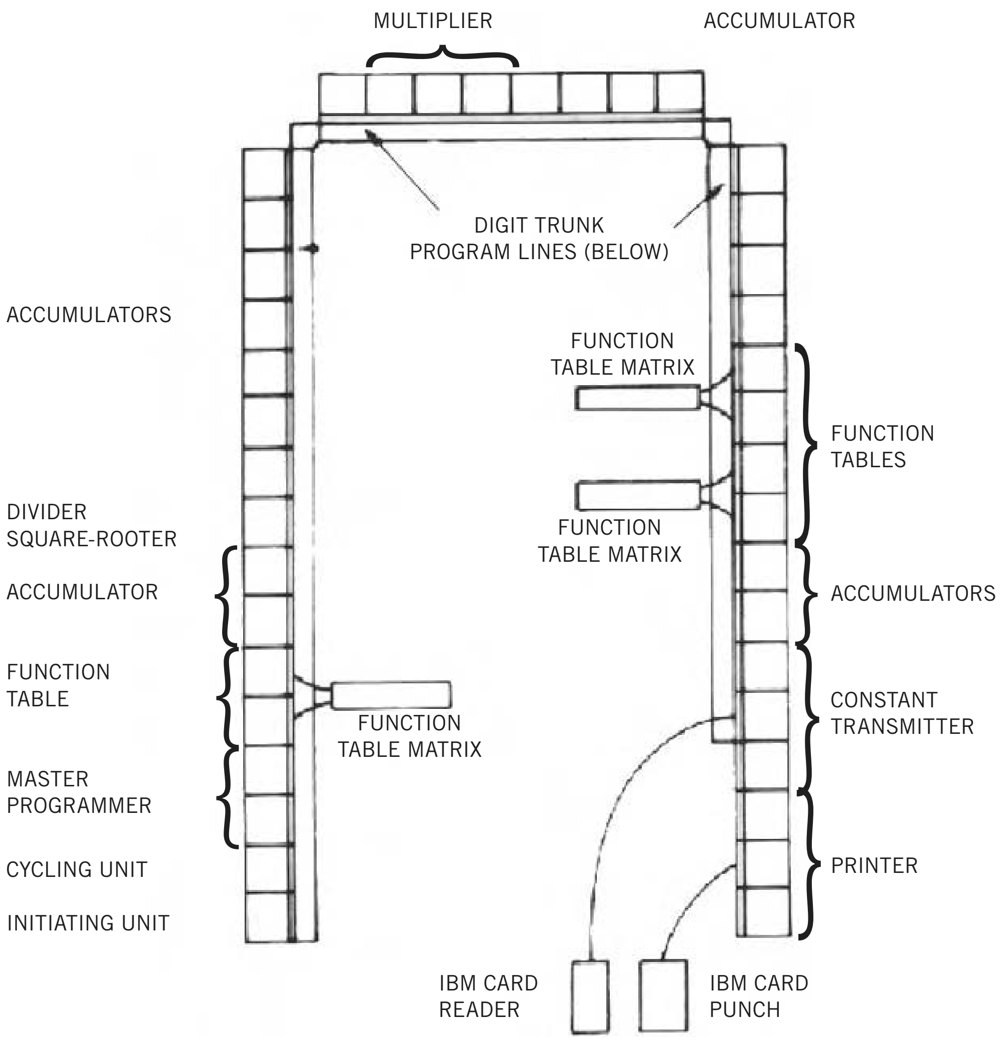

Figure 9.1 ENIAC layout.

Contributing to the computer architecture were:

- Vannevar Bush's differential analyzer as a model for machine organization,
- IBM switchboards for control, and experiences from Bell Labs and Aiken's MARK I.

All of these were the research input for ENIAC, which led inevitably to the concept of stored programs. ENIAC has two successor generations, the IAS, the WHIRLWIND (we will come back to this) to the IBM 700, which more or less directly leads into the present and to the more academically oriented product lines EDVAC, EDSAC and UNIVAC, which we will also encounter again.

6.3.2. Extension and Construction The ENIAC consists of four accumulators; one square rooter unit; one multiplication unit; three complex switchboards for matrix calculation; on the bottom left, the three control units and on the right, the punch-card input and output.[13] Each of these units had to be "programmed" first, that is wired together.

6.3.3. Programming The programming of ENIAC fell into two separate areas, numerical programming and, what the ENIAC-team called "programming proper."

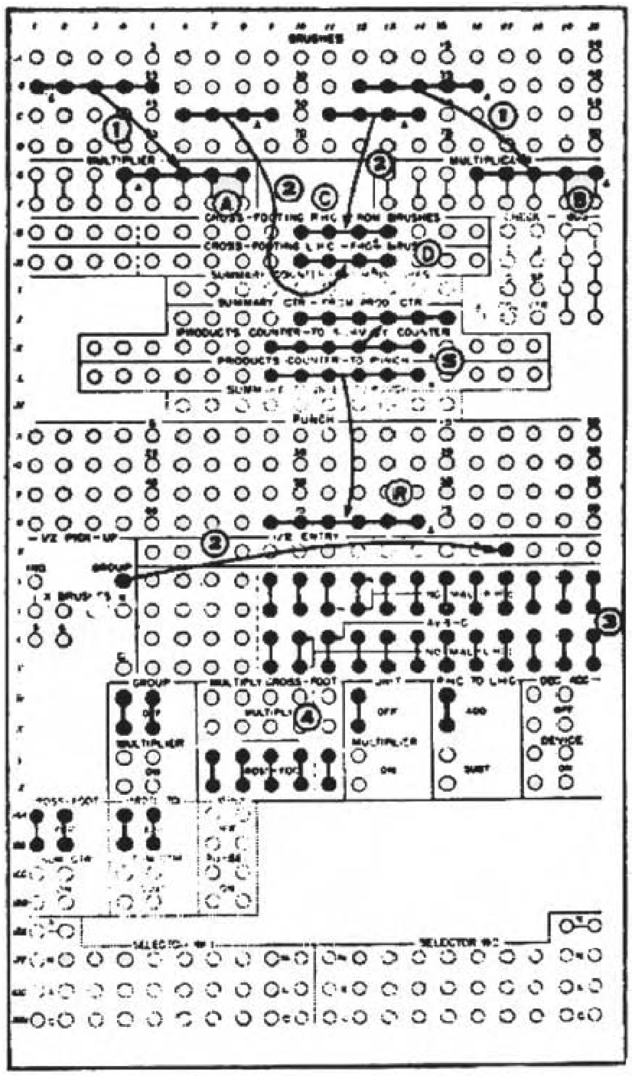
BRUSHES
MULTIPLIER
MULTIPLICAND
CROSS-FOOTING RHC ← ROW BRUSHES
CROSS-FOOTING LHC ← PROD BRUSH
SUMMARY CTR ← FROM PROD CTR
PRODUCTS COUNTER → TO SUMMARY COUNTER
PRODUCTS COUNTER → TO PUNCH
PUNCH
V2 PICK-UP
V2 ENTRY
GROUP
NORMAL
GROUP
MULTIPLY CROSS-FOOT
MULTIPLY
MULTIPLIER
OFF
ON
ADD
SUBT
DEVICE
SELECTOR NO. 1
SELECTOR NO. 2

Figure 9.2 Programming of an accumulator unit. Wiring diagram for IBM 601 plugboard.

6.3.3.1. Numerical Programming The illustration shows the "programming" of an accumulator unit as used in the ENIAC. On top you see the set calculation "a*b+c+d." Working with these interfaces was called numerical programming, as Arthur Burks reports, and depending on the formula calculated, it had to be done for all accumulator, square root, or function units separately. For the most part, this specialized labor was carried out by the "ENIAC girls." In order to keep track of the sequence of such numerical programming, block diagrams such as the one shown in Figure 9.3 were used.

The diagrams determined what had to happen in the various units of the ENIAC, which accumulator would calculate which part of the formula, etc. One might call it numerical programming assignment, a first kind of addressing, for the accumulators were nothing but the intelligent storage cells of the calculator. There was no generalized symbolic notation for their coding, i.e., a plan for their connection. In such process diagrams, we find the predecessors of the first programming routines, as well as one of the oldest representations of calculation. For the Greeks drew their geometric figures and their derivations in a "diagramma"—and this was also the name of a scale, since musical sequences were understood as derived from cosmic geometry. For the ENIAC, program sequences are noted in diagrams, as the graphic interface between the mathematics of the formula to be calculated and the electronic plan for their numeric solution. The concrete implementation of such diagrams was laconically called "programming proper"—as if nothing was easier.

6.3.3.2. Programming Proper Programming proper consisted in synchronizing the separate units with the digit trunk or data bus on the one hand, and with the seven parallel program lines on the other hand. To this end, there were further diagrams as shown in Figure 9.4.

Transmit and receive in the accumulator had to be connected by hand; the three program controls of each accumulator were connected to the bus of the program lines. This is basically the archetypal innovation of the ENIAC. The electromagnetic differential analyzer and Zuse's machine still had a central, motor-operated unit controlling the speed of the calculations, like the inventions of Schickart or Babbage. ENIAC replaced those mechanical impulses with the completely new concept of ten bus cables upon which a cycling unit transferred a complex parallel pulse which

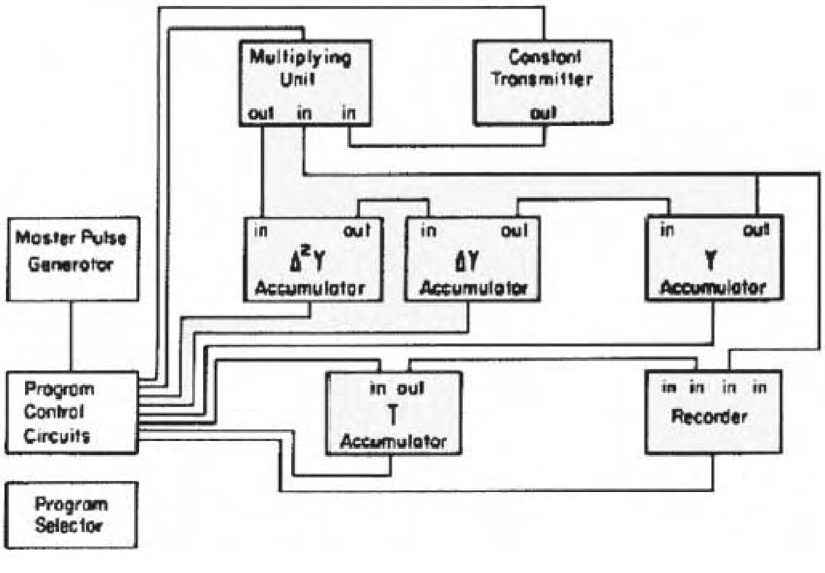

Figure 9.3 Block Diagram. Block diagram illustrating method of solution of $d^2\, y/dt^2 = ky$ or $\Delta^2 y = k(\Delta t)^2 y$.

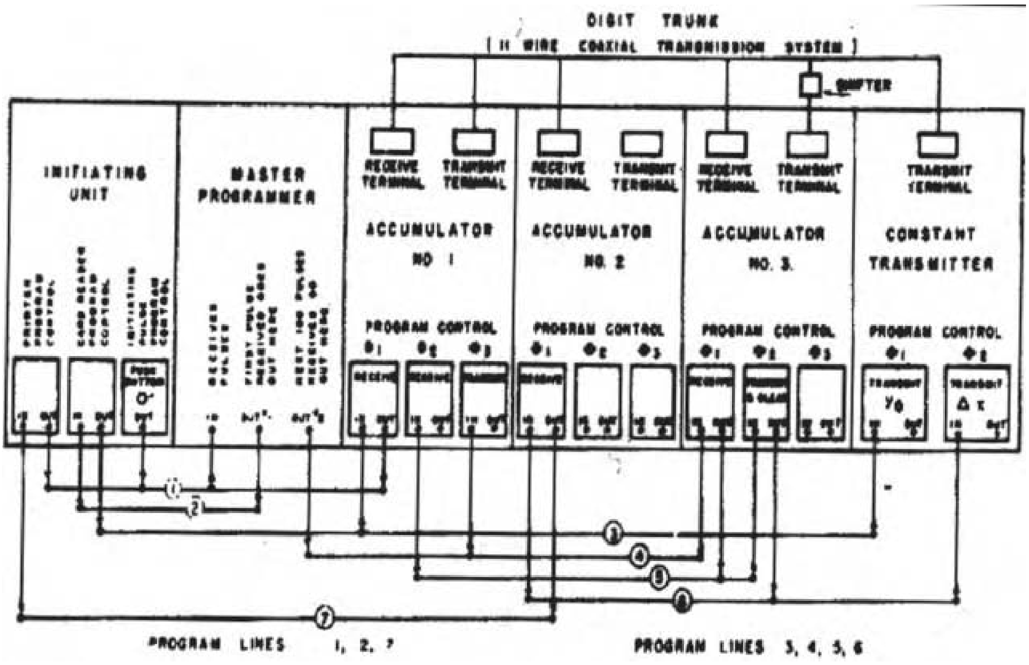

Figure 9.4 Simplified ENIAC program diagram (from Burke: 1980).

could be as fast as 3 micro-seconds, thus allowing switching at a veritable 3.3 KHz. The control of such a fast pulse had become possible with advances in radar technology. Once a program was set up, its running was static, with no recursions, inductions and conditional branching.[14]

6.3.4. Mercury Delay Lines It is obvious that the concept of stored programming developed from the tubes of the ENIAC and not from electromechanical calculators. The ENIAC provided a fast bus and its technology was robust enough to trigger impulse control-frequencies at 3.3 KHz running through a system the size of half a ballroom. This made it possible to approach mechanically impossible storage problems, by inventing all sorts of impulse-controlled delays. In the case of the ENIAC the team used the so-called "mercury delay lines" that Turing in his work on the English ACE-computer had used also. Pres Eckert had developed them in the spring of 1944 for radar units—another input from radar research into the computer.[15]

The "mercury delay line" was a slim tube filled with mercury, here two meters long, that could delay a pulsating ultrasound signal up to a millisecond. A deft switch allowed only ten tubes to refresh 1000 impulse bits in the space of a microsecond. The result was a phased electronic 1K-Bit storage unit for the cost of a few liters of mercury and ten tubes for ten dollars each. The mercury line was a revolutionary step: it reduced both the cost and the space required for memory at once by a factor of 100 to 1.

Between March and June 1945, the team built 32K of memory for data and program instructions for a new type of computer we now call the "von Neumann machine" by cascading 256 such mercury lines. The storage blocks required a phased bus architecture, the bus address logic condensed the distributed accumulators, multiplying units and square rooters onto one unit—the central processing unit (CPU)—and the whole system thus demanded a central control. Far from Turing's logic machines (which—as far as we know—was mentally not present in the ENIAC-Team and was definitely nothing von Neumann argued with), the synchronization of the described electronics architecture alone led to the legendary discrete sequentiality, giving up the generous parallel set-up the ENIAC still represented. Henceforth the guideline for the most successful machine type ever built by man was, in Burks' laconic words: "One thing at a time, down to the last bit!"[16] Single instruction, single data. In the arguments over the patent, John von Neumann would later admit that it was "practically impossible to list who was the apostle."[17] The innovation of binary storage and stored programming is anything but an invention or a defined patent. Moreover, the computer itself was not the end goal, neither for the engineers Mauchly and Eckert nor for the mathematician von Neumann.[18] In the competition between different architectures, the goal was to build another calculator that was able to solve nonlinear equations numerically, with great demands on rounding and induction. Nobody "invented" the computer as we know it now in the strict sense, nor did anybody want to invent it the way we know it now.

Von Neumann himself would soon go beyond this machine limited to mathematics that Turing had indeed described sufficiently, dedicating himself for another decade to the theory of automata and of machines that tolerate mistakes; von Neumann would write about cellular automata, matrix inversions, and neuronal learning. The scientific paradigm of the war years that give birth to the most important medium of the century is not the computer itself, but what we associate with Claude Shannon, namely thermodynamics and entropy as elementary principles of information theory. Read what Claude Shannon wrote on John von Neumann in the late sixties, and you see how information theory and automata research are interconnected.[19] For Shannon and von Neumann knew the problematic reliability of the "von Neumann machine" as well as any persons who ever had to deal with it. As Shannon writes:

> individual components must be built to extreme reliability, each wire must be properly connected, and each order in a program must be correct. A single error in components, wiring, or programming will typically lead to complete gibberish in the output. [...] The problem is analogous to that in communication theory where one wishes to construct codes for transmission of information for which the reliability of the entire code is high even though the reliability for the transmission of individual symbols is poor.[20]

Von Neumann's work on this topic concludes that reliable systems consisting of unreliable parts are possible if either the redundancy of parallel but similar components or their redundant connections are fantastically high.[21]

7. Von Neumann's Scores

After the question of the evolution of the "von Neumann machine," our interest turns to the question of how von Neumann programmed it. With the aid of Donald Knuth and Luis Pardo, I want to offer a provisional answer. Let us proceed with a rather useless little program written in ALGOL60 so as to not overly complicate the matter.

```
ALGOL 60:
begin INTEGER i; REAL ARRAY a[0:10];

REAL PROCEDURE F(t); REAL t; VALUE t;
   f := sqrt(abs(t)) + 5 + t3 ;

   for i := 0 step 1 until 10 do read(a[i]);
   for i := 10 step -1 until 0 do
       begin
      y := f(a[i]);
```

```
        if y > 400 then write(i,"value too large")
          else write(i,y);
       end;
    end.
```

The mere point of this program is that it contains the eleven-fold iteration of a formula that calculates the value of an eleven-digit array of REAL numbers in the inverse order of their input; if the result is > 400, the program issues a warning; if the result is smaller, it yields the numeric value. In von Neumann's notation, this program would look like Figure 9.5.

In three long pamphlets entitled "Planning and Coding Problems for an Electronic Computing Instrument" for the Army Research and Development Department, von Neumann explained this technique of programming. This is the stuff of advanced mathematics classes and I cannot go into it here. But I follow our colleague Jörg Pflüger in calling what you see here not a programming language but "planning"—on the one hand, a systematic planning diagram where the parts to be coded are simply entered into well defined boxes, and on the other hand a very open and repeatedly revised code notation which you do not see here, by which the people coding would execute what the different boxes of the diagram describe. The diagram itself is a loop of loops leading from i to e. I already referred you to the ancient Greek *diagramma* which also denotes a tonal system. Thus we may read this diagram more like a musical score than like a written notation of language.

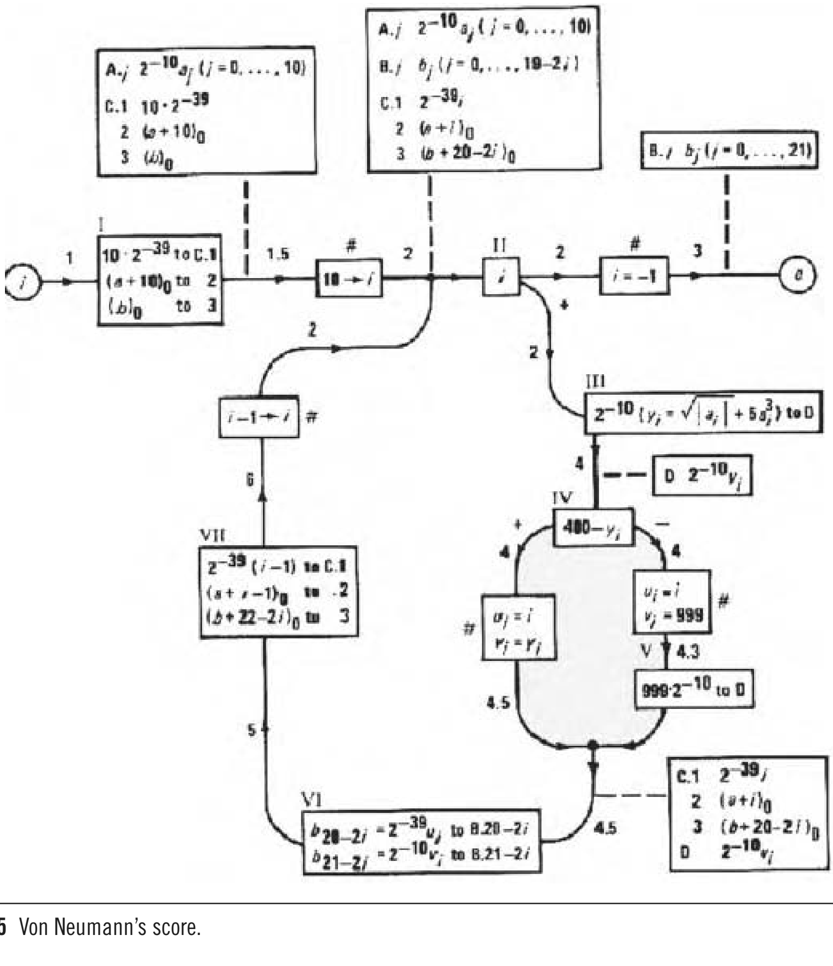

Figure 9.5 Von Neumann's score.

The rectangular boxes represent a mathematical orchestration of the work, while the parts and instrumentation are to a large extent free.

I have to limit myself to a few explanatory remarks. First you see that von Neumann leaves the scaling of the memory, the bitmapping of number types to the programmer. One had to get used to this. In the second box from the top on the left side for instance you read: “10*2 to the power of minus 39 to C.1”—which means 10 units of 40 bits each: set 10 40-bit words in storage area C.1. There are four types of boxes in this diagram:

1. “Operation boxes,” which have a roman numeral, as in the box III, where after the 10th bit of the 40-bit word the arithmetic unit inserts the result of the formula “root of absolut a plus 5 a to the power of 3” with the order to shift the value to point D in the storage.
2. “Alternative boxes,” also with roman numerals, which branch out depending on their prefix of the box value. For instance, in the straight line from i to e you see a box II inscribed with “i” which then branches to the lower right if i >+0, and to the left, in this case straight, if i is negative (smaller than zero).
3. “Substitution boxes,” marked by a cross outside and an arrow inside. They do not signify code instructions but tell the reader of the diagram only how the value of a diagram variable has to be addressed.
4. “Assertion boxes,” which simply give the value of variables, such as the last box on the straight line to e, i =−1, which means: the iteration is over.

Thus we can say that von Neumann’s flow charts represent a graphic programming language. They show certain validation properties and commutative elucidations that have nothing to do with coding, but tell the programmer what happens to the operative variables and memory values when they re-enter a coded block.

Let us leave it at these hints. I have addressed only the most obvious attributes of the flow chart diagram, but it may suffice to remember Jörg Pflüger’s statement: von Neumann had nothing like a concept of language in his mind. Even more, “planning and coding” does not raise the question whether it is possible to address a computer with a coherent symbolic notation. I show you this complicated flow chart to reveal how von Neumann suggested computers be addressed, namely by non-language means. Although they go together in even the most formal language, the planning score, which we see here keeps the four essential mechanisms separate: operation from alternation, alternation from substitution and substitution from assertion. There are no generative symbols, no substitutions of sign values, no literal or syntactic assertion, simply because the flow chart never even tries to furnish proof of its own correctness. This dissection of the structure of language is due to the changed levels, to the way the flow chart alternates concrete descriptions of operations with descriptions of description. In this way von Neumann prevents mathematical nonsense from creeping into his notation (think for instance of the simple mathematical nonsense in the C-command “x=x+1”). Let us be clear: the first manner in which von Neumann addresses the computer implements the conviction that programming does not require language. An egregious error by von Neumann, or shall we call it prudence?

Behind “planning and coding” we recognize a neat division of labor, if you will. The flow charts are reminiscent of what was in use before the von Neumann machine, and they can be read as a kind of perfection of the ENIAC diagrams. In this respect they also rely on programmers’ habits. And since we know that the accumulators and function tables were largely operated by the “ENIAC girls” only and that the box-logic of the flow charts is nothing but a permutation of ENIAC diagrams, then planning and coding, although not a language, still has a lot to do with men as architects and women as coders—with a diagrammatic chasm, if you will, that is so self-evident that it is too easily overlooked.

8. Priesthood and Revolution

Perhaps we should add that the flow chart technique of programming as developed by von Neumann and Goldstine has remained rather theoretical. It was not implemented regularly on any calculator but rather served as conceptual crutch. Since the published and much discussed concepts of the ENIAC team—the technology of binary stored programming—there have been engineering solutions: immense sums were invested all over the U.S. to build computers, but there was no clear concept of how they should be addressed. What, therefore, was programming in the early fifties? According to John Backus:

> Programming in the early 1950s was really fun. Much of its pleasure resulted from the absurd difficulties that "automatic calculators" created for their would-be users and the challenge this presented. The programmer had to be a resourceful inventor to adapt his problem to the idiosyncrasies of the computer: He had to fit his program and data into a tiny store, and overcome bizarre difficulties in getting information in and out of it, all while using a limited and often peculiar set of instructions. He had to employ every trick he could think of to make a program run at a speed that would justify the large cost of running it. And he had to do all this by his own ingenuity, for the only information he had was a problem and a machine manual. Virtually the only knowledge about general techniques was the notion of a subroutine and its calling sequence. [...] Programming in the early 1950s was a black art, a private arcane matter involving only a programmer, a problem, a computer, and perhaps a small library of subroutines and a primitive assembly program. Existing programs for similar problems were unreadable and hence could not be adapted to new uses. General programming principles were largely nonexistent. Thus each problem required a unique beginning at square one, and the success of a program depended primarily on the programmer's private techniques and invention. [...]
>
> Just as freewheeling westerners developed a chauvinistic pride in their frontiersmanship and a corresponding conservatism, so many programmers of the freewheeling 1950s began to regard themselves as members of a priesthood guarding skills and mysteries far too complex for ordinary mortals. [...] This attitude cooled the impetus for sophisticated programming aids. The priesthood wanted and got simple mechanical aids for the clerical drudgery which burdened them, but they regarded with hostility and derision more ambitious plans to make programming accessible to a larger population. To them, it was obviously a foolish and arrogant dream to imagine that any mechanical process could possibly perform the mysterious feats of invention required to write an efficient program. Only the priests could do that. They were thus unalterably opposed to those mad revolutionaries who wanted to make programming so easy that anyone could do it.[23]

You may know this passage from Backus' report on the founders' years of FORTRAN well enough so I do not have to comment at length. But it leads us in a few sentences, and this is my intention in quoting it, to another stage, namely computers as media as well as machines. This stage would take another two or three decades, yet it was already discernible: something has started to produce priesthoods, mysteries and revolutionaries, something of considerable factual and economic impact as well as of immense technical importance, and it was expressing itself already without having a language. Somehow, we may understand from Backus, an imperative is indicated in the vehemently growing world of computers in the fifties, an imperative that makes priests, pioneers and revolutionaries dance around a golden calf, and the author of all these metaphors is none other than John Backus, who wrote probably the most influential programming language in computer history, namely FORTRAN. He concludes his *"confiteor"* with the sentence, "I am the culprit."

Perhaps to the uninitiated, to those who are not computer scientists, this mystery which is no mystery continues. So I insert a clarifying remark: John Backus wrote this in 1980, after his third important intervention in the history of programming languages, namely the Turing lecture of 1979 entitled "Can we be liberated from the von Neumann style?" Thus he discredited all the attempts to define languages in the way he and others had done before and aimed in the direction of a mathematically founded functionality. But even these attempts didn't solve the problem he describes in his retrospection. For at the birthof his language—the language he was going to specify and whose style would dominate the medium for many decades, perhaps until today—there way no language yet, but only a linguistic climate, as it were, an imperative, a denial as well as a demand.

```
SHORTCODE (sugg. John W. Mauchly, by William F. Schmitt, 1950)
Memory "Variable" i = W0, t = T0, y = Y0
11 inputs are addressed to the following words: U0, T9, T8,..., T0
constant: Z0 = 000000000000
          ZI = 010000000051
          Z2 = 010000000052
          Z3 = 040000000053
          Z4 = $$$TOO$LARGE
          Z5 = 050000000051 [5.0]

"Equation number recall information" [Labels]: 0 = line 01, 1 = line 06, 2 = line 07
  Short Code:
   Equations                            Coded representation
————————————————————————————————————————————————————————————————
  00       i = 10                       00 00 00 W0 03 Z2
  01  0:   y = (sqrt abs t) + 5 cube t  T0 02 07 ZS 11 T0
  02                                    00 Y0 03 09 20 06
  03       y 400 if<=to 1               00 00 00 Y0 Z3 41
  04       i print, 'TOO LARGE'
           print-and-return             00 00 Z4 59 W0 58
  05       0 0 if=to 2                  00 00 00 Z0 Z0 72
  06  1:   i print, y print-and-return  00 00 Y0 59 W0 58
  07  2:   T0 U0 shift                  00 00 00 T0 U0 99
  08       i = i – 1                    00 W0 03 W0 01 Zl
  09       0 i if<=to 0                 00 00 00 Z0 W0 40
  10       stop                         00 00 00 00 ZZ 08

Code-Equivalents:

01 - 06 abs value ln (n+2)nd power      59 print and return carriage
02 ) 07 + 2n (n+2)nd root               7n if= to n
03 = 08 pause 4n ifsto n                99 cyclic shift of memory
04 / 09 (                               58 print and tab
                          Sn, Tn,..., Zn quantities
```

The first implemented programming concept, as claimed since 1977 without contradiction, was Short Code. It is a language of imperatives, which simply follows an imperative. This concept also stems from the efforts of the ENIAC team—not from the mathematical corner around Goldstine and von Neumann, but from the technical pragmatism of John W. Mauchly. In 1949 he suggested this simple algebraic interpreter language, which you see here implementing our stupid algorithm. You will observe that it is "readable"—line 00 initializes i=10, line 01 the label 0, and behind it the square root formula. The interpreter jumps back there until line 08 assumes the value "–1." Already quite classical and very "spaghetti"-like.

I do not want to detain you with details, although some aspects might be rather interesting. Short Code was implemented on a UNIVAC in 1950 and already offered the programmer an "electronic dictionary," as the *ACM* magazine put it. Each arithmetic operation had a short code, hence the name. You see some of these short codes in the lower section. Short code did not know arrays, yet it shifted stored words cyclically, for instance in line 07. The simplifications are obvious: a limited dictionary of computer operations is born describing immediately what is to happen to the operands. Evidently the seeds of a language.

"With Short Code," the Remington Rand Corp. announced, "every mathematical equation can be evaluated by the mere means of notation. There is a simple symbolic transformation of equations into code [...] the necessity for special programming is eliminated. In our comparisons of computing time we observed a speed increase of at least 50:1 in comparison to manual programming." After the hundredfold gain in memory by the von Neumann machine, it was now a matter of gaining human storage as to become constitutive of a language—that is to say, of gaining time. Lacan would be happy getting this: programming history defines *language* as an instance of gaining mere storage time. Further: "Short Code will demonstrate its power as a tool in mathematical research and as a checking device for some large-scale problems," which means that it will also be checking priesthoods in Backus' sense, by means of a "simple tool," which will test mathematical arcana.[24]

9. FORTAN and Mariner I

Short Code is mentioned only because of its being the first putative interpreter language in computer history. It stems from the ENIAC team, just as von Neumann's concepts did. Otherwise it could not plead for much historical significance, like so many developments from the early years, although it represents quite well what everybody was looking for some years later, namely an algebraically oriented programming language. Short Code disappeared from the scene because the Remington Rand UNIVAC was used only by a small number of scientists. The pivotal condition for the scientific solution of complex problems was still missing, namely communication. Look at Jean Sammet, Grace Hopper, John Backus or anybody else—the late forties and early fifties in America are determined by the absence of what could be called a discourse on the linguistic speech-based control over the new digital computers. There was no national or international exchange, hardly a conference, no periodical publication until 1954. This further supports the thesis that the computer as the von Neumann machine was not a consciously assigned goal of the research organizations and of the scientific world in the U.S.

This is why the path to FORTRAN, which we will briefly outline, had to follow from another large-scale military project of the now icecold Cold War, the WHIRLWIND computer built at MIT in seven(!) years between 1945 and 1952. Since 1951 it had become the backbone for the "Semiautomatic Ground Environment Air Defense System" (SAGE), the first large computer project of the Air Force and the Navy. One must never forget that this very first net of computers ever built in the history of mankind is to be seen at the same time as the great grandfather of today's Internet, even though it had yet to experience the "EMP" shock of the hydrogen bomb.[25] The WHIRLWIND-computer (commanding the SAGE-net) not only connected regular radar oscilloscopes (or television screens) with a computer for the first time on a regular basis, it not only featured such puzzling devices as "light guns" touching the goals on the screen directly (later, in the '70, disarmed to "cursors" and "mouses" by the PARC-kids in Palo Alto, getting from there right into the first Mac's surfaces...) and probably the first keyboard ever, but it also attracted a huge crowd of scientists interested in programming, because the military organization of research worked as well as the scientific one did not.

```
1            v|N = {input},
2            i = 0,
3     1      j = i + 1,
4            a|i = v|j,
5            i = j,
6            e = i – 10.5,
7            CP 1,
8            i = 10,
9     2      y = F^1(F^11(a|i)) + 5(a|i)^3,
10           e = y – 400,
11           CP 3,
12           z = 999,
13           PRINT i, z.
14           SP 4,
15    3      PRINT i, y.
16    4      i = i – 1,
17           e = –0.5 – i,
18           CP 2,
19           STOP
```

The language "style" which we see here was called "Laning/Zierler Algebraic Compiler" and was developed on and for the WHIRLWIND. The participants of the 1954 Office of Naval Research's first computer language conference were enthusiastic, although it does not live up to what is nowadays expected from a compiler.

A quick glance at how it works: the variables v and a are each indicated or subscribed by a vertical line to form an array, CP 1 or 3 in line 7 or 11 means a conditional jump order in the assembler style, namely, if the previous instruction yielded a negative result, jump to label 1 or 3. F to the power of 1 is the square root instruction; F to the power of 11 means an absolute instruction; lines 3 through 8 iterate in order to read the input v in to the variable a; lines 9 through 18 represent the core loop. After a few minutes, most of you will be able to read this Laning/Zierler Compiler well.

Now I want to quote what John Backus, the revolutionist, wrote in 1954. The Laning/Zierler Compiler immediately instigated the development of FORTRAN at IBM right after the conference in 1954 taking 18 man-years including the compiler. Backus writes, "a programmer may not be considered unreasonable if all he wants is formulas for the numerical solution of his problem, and perhaps a plan that shows him how his data are shifted from one storage hierarchy into another…No doubt, if he was to pursue this vigorously next week, he would be a psychiatric case, but perhaps next year he would be taken more seriously."[26]

Here is the result, less than a year later, in November 1954: the IBM FORmula TRANslation System, FORTRAN 0, developed under the auspices of John Backus:

```
1            DIMENSION A(11)
2            READ A
3     2      DO 3,8,11 J=1,11
4     3      I=11-J
5            Y = SQRT(ABS(A(I+1))) + 5*A(I+1)**3
6            IF (400. >=Y) 8,4
7     4      PRINT I,999.
8            GO TO 2
9     8      PRINT I,Y
10   11      STOP
```

> We think that FORTRAN offers a useful language for the formulation of problems which are fed into a machine solution... After one hour of instruction in FORTRAN, the average programmer will have completely understood the steps of writing a procedure in FORTRAN, and this without any further comments.[27]

I will not say anything new about FORTRAN; the DIMENSION declaration in line 1 of course demands an eleven-digit array of REAL numbers; line 2 reads them; line 3 shows the legendary DO statement in FORTRAN which in its 0-version denominates the start and end label, so in other words line 3 says: iterate from line with label 3 to line with label 8 until the variable J takes the value 11, then go to label 11 and end. The IF-statement was only binary then, but otherwise it is simple FORTRAN.

Here, in the language of the "lazy character," as Backus occasionally called himself, we also find the legendary programming error which made the Mariner I miss Venus by far (July 1962), and left its imprint on the software crisis of the 1970s: Instead of "DO 3 I = 1,3" the program read "DO 3 I = 1.3."[28] This mistake could not be recognized by the 18 man-year compiler project of IBM-FORTRAN whose speed still made Backus proud in 1980, because in the definition of FORTRAN spaces are not significant, owing to the fact that FORTRAN relied on punch cards where space has no significance. Consequently, because of a stop instead of a comma between 1 and 3, the result was interpreted as implicit declaration of the variable DO3I with a real value of 1.3, whereas it would have been correct to initiate a threefold iteration of all the program lines that we do not see here until the mark CONTINUE and the label 3. Needless to say, this is not just computer folklore.

10. Epilogue

The genesis of programming language styles up to and including FORTRAN is the genesis, as we know, of procedural and imperative languages. The fundamental weakness of these languages is something I do not have to impress upon computer scientists. And it is clear that declarative and functional languages are at least logically superior to the ones shown here, so I agree with Jörg Pflüger's thesis that the newer generation of object oriented languages represent an interesting synthesis of procedural structurability and functional logic design.

This paper presented a first draft discourse analysis containing a hypothesis on the development of programming languages. Strangely they do not stem from the recourse to a logical model, although the machines they control are explicitly based upon such a model. This remains a contradiction still to be resolved. The same goes for the question why those who developed the design of the machine, with full knowledge of the basic logical model, did not recognize that there is a demand for an effective and well-defined language in order to address that machine. My thesis is that for decades, the *arché*-structure of the von Neumann machine did not reveal that this machine would be more than a new calculator, more than a mighty tool for mental labor, namely a new communications medium. The development of FORTRAN demonstrates all too clearly how the communication-imperative was called on the machine from all sides. That imperative call obviously could not be detected in the *arché*-structure of the machine itself. It grew out of the Cold War, out of the economy, out of the organization of labor, perhaps out of the primitive numeric seduction the machines exerted, out of the numbers game, out of a game with digits, placeholders, *fort/da* mechanisms, and the whole quasi-linguistic *quid pro quo* of the interior structure of all these sources.

At any rate, communications media always have the structure of language, as we know since Freud. This side of and beyond explicitly spoken languages, they are characterized by the insistence of their inherent signifier, that is to say by contiguities and substitutions whose effects and traces can be visualized in graphs and diagrams, not in logical but in probabilistic and still unpredictable rules of generation.

Notes

1. Georg Wilhelm Friedrich Hegel, "Phänomenologie des Geistes" (1807), *Werkausgabe* vol. 3. (Frankfurt: Suhrkamp, 1970), 36. Georg Wilhelm Friedrich Hegel, *Phenomenology of Spirit.* (Oxford: Oxford University Press, 1977), 19.
2. Jay David Bolter, *Writing Space. The Computer, Hypertext and the History of Writing* (New Jersey: Lawrence Erlbaum, 1991), 9. See also Michael Heim, *Electric Language. A Philosophical Study of Word Processing* (New Haven: Yale 1987).
3. See Friedrich Kittler, "There is no software," *Draculas Vermächnis. Technische Schriften* (Leipzig: Reclam, 1993), 224–242, here: 242.
4. Niklaus Wirth, *Algorithmen und Datenstrukturen, Pascal* (Stuttgart: Teubner, 1983), 120.
5. Oswald Wiener, *Probleme der Künstlichen Intelligenz* (Berlin: Merve, 1988).
6. Alois Walde, *Lateinisches Etymologisches Wörterbuch* (Heidelberg: C. Winter, 1910), 738 and Hermann Menge ed., *Enzyklopädisches Wörterbuch der lateinischen und deutschen Sprache*, (Berlin: Langenscheidt, 1911), 716.
7. Hans Ulrich Gumbrecht, "Schwindende Stabilität der Wirklichkeit. Eine Geschichte des Stilbegriffs," edited by Hans Ulrich Gumbrecht and K.L. Pfeiffer, *Stil. Geschichten und Funktionen eines kulturwissenschaftlichen Diskurselements (*Frankfurt: Suhrkamp, 1986), 731.
8. Friedrich A. Kittler, "Im Telegrammstil," in Gumbrecht and K.L. Pfeiffer eds., *Stil.*, 358–370; Bernhard Siegert, *Relays. Literature as an Epoch of the Postal System* (Palo Alto: Stanford, 1999).
9. Jan Tschichold, *Was Jedermann vom Buchdruck wissen sollte. Ein Leitfaden für Drucksachen-Besteller (*Basel: Birkhäuser, 1949), and Florian Cajori, *A History of Mathematical Notations*, (Chicago: Open Court Publishing Company, 1928).
10. Arthur W. Burks, "From Eniac to the Stored-Program Computer," in *A History of Computing in the Twentieth Century,* edited by Nicholas Metropolis (New York: Academic Press, 1980), 311–344, here: 334. In the effort to avoid mistakes, the calculations took up immense amounts of memory. - "I have often been asked, 'How big was the ENIAC storage?' The answer is, Infinite. The punched-card output was not fast, but it was as big as you wished. Every card punched out could be read into the input again, and indeed that is how Metropolis and Frankel managed to handle cycle after cycle of the big problem from Los Alamos." Kathleen R. Mauchly, "John Mauchly's Early Years." *Annals Of The History Of Computing* 6 (1984), 547.
11. Nicholas Metropolis ed., *A History of Computing in the Twentieth Century* (New York: Academic Press, 1980), 522.
12. Court proceedings concerning the patent debate mark the end of the ENIAC team's collaboration. See Herman H. Goldstine, *The Computer from Pascal to Neumann (*Princeton: Princeton University Press, 1972).
13. As part of a virtual memory concept, stacks of punch-card inputs and outputs for the calculation of the Los Alamos problem, as Markly reports, would be fed by hand into the IBM card reader.
14. Arthur W. Burks allows in his description which I borrow here that the ENIAC was not simple and easy to program. "Programming proper" appeared more obscure and complicated to contemporaries than the central programming of the similarly built units of electromagnetic differential analyzers. For their firmly wired wheels and switches on the control panels one had advanced ideas of punch card control, also in Zuse, which would set the control panels by corresponding relays. The hundreds of inserts of the ENIAC which produced the wild cable trees visible on all photographs seemed less practicable for central or even stored programming.
15. Metropolis ed., *A History of Computing in the Twentieth Century*, 531.
16. Arthur W. Burks, "From Eniac to the Stored-Program Computer," in: *A History of Computing in the Twentieth Century*, edited by Nicholas Metropolis, ed., 311–344, here: 338.
17. T. Legendy and T. Szentivanyi eds., *Leben und Werk von John v. Neumann* (Mannheim-Wien-Zürich: Bibliographisches Institut, 1979), 57.
18. Friedrich Wilhelm Hagemeyer, a theoretical physicist from the old East Germany who earned his doctoral degree in sociology in the late 1970s with Wolf Lepenies, offers an exemplary idea of the complications that mark the genesis of scientific developments in the 1940s in his work on Claude Shannon's mathematical theory of communication. These complications in industrial research and development, civil technology, war economy and research, theoretical physics and mathematics result in a factual, autopoetic and planned research organism which he presents in his sociology of knowledge as an "evolutionary model of technological development" (Friedrich-Wilhelm Hagemeyer, *Die Entstehung von Informationskonzepten in der Nachrichtentechnik. Eine Fallstudie zur Theoriebildung in der Industrie- und Kriegsforschung.* Ph.D. thesis, Free University Berlin 1979, 8ff). In the case of Shannon's revolutionary work, it has the advantage of coming together in a relatively concise terminus ad quem.
19. H. Ulam, H.W. Kuhn, A.W. Tucker, and Claude E. Shannon: "John von Neumann," in *The Intellectual Migration: Europe to America 1930–1960,* edited by Donald Fleming and Bernard Bailyn (Cambridge: Harvard/Belknap, 1969), 235–269, here: 259ff.
20. H. Ulam, H.W. Kuhn, A.W. Tucker, and Claude E. Shannon: "John von Neumann," here: 256.
21. John von Neumann, "Probabilistic Logic and the Synthesis of Reliable Organisms from Unreliable Components," in *Automata Studies*, edited by Claude E. Shannon and John McCarthy, eds. (Princeton: Princeton University Press, 1956), 43–98.
23. John Backus, "Programming in America in the 1950s – Some Personal Impressions," in *A history of computing in the twentieth century,* edited by Metropolis, 125–135, here: 125–128.

24. Donald E. Knuth and Luis Trabb Pardo, "Early Development of Programming Languages," *Encyclopedia of Computer Science and Technology* (New York, Publisher, 1977): vol. 7, 419–493, here: 434. See C.A.R. Hoare, "Der neue Turmbau zu Babel. Rede zur Verleihung des Turing-Preises der Association for Computing Machinery, 1980," *Kursbuch* 75 (1980), 57–73, here: 61.
25. Wulf Halbach, "Virtualität und Ereignisse," in *Medien und Öffentlichkeit*, edited by Rudolf Maresch (München: Boer, 1996), 165–187.
26. Knuth and Pardo, "Early Development of Programming Languages," here: 456.
27. Knuth and Pardo, "Early Development of Programming Languages," here: 460.
28. G. J. Myers, "Software Reliability: Principles and Practices," John Wiley, 1976, here: 275.