Prom. Nr. 2186

DIGITAL COMPUTERS

On encoding logical-mathematical formulas using the machine itself during program conception

Dissertation
presented to
The Federal Technical University in Zurich
for obtaining the
Degree of doctor of the sciences
by
CORRADO BÖHM, electrical engineer EPUL
from Milan (Italy)

Supervisor: Prof. Dr. E. Stiefel
Co-supervisor: Prof. Dr. P. Bernays

BOLOGNA
COOPERATIVA TIPOGRAFICA AZZOGUIDI
1954

English translation 2016 by Peter Sestoft • Version 1.0

Prom. Nr. 2186

CALCULATRICES DIGITALES

DU DÉCHIFFRAGE DE FORMULES LOGICO-MATHÉMATIQUES PAR LA MACHINE MÊME DANS LA CONCEPTION DU PROGRAMME

THÈSE

PRÉSENTÉE

À L'ÉCOLE POLYTECHNIQUE FÉDÉRALE, ZURICH,

POUR L'OBTENTION DU

GRADE DE DOCTEUR ÈS SCIENCES MATHÉMATIQUES

PAR

CORRADO BÖHM, ing. électr. dipl. EPUL
de Milan (Italie)

Rapporteur : Prof. Dr. E. Stiefel
Co-rapporteur : Prof. Dr. P. Bernays

BOLOGNA
COOPERATIVA TIPOGRAFICA AZZOGUIDI
1954

Frontpage of the original.

Extract from the Annali di Matematica pure ed applicata
Series IV, Volume XXXVII (1954)

TO MY WIFE

Translator's remarks Corrado Böhm (born 17 January 1923) completed his PhD dissertation at ETH Zürich in late 1951 under the supervision of E. Stiefel and P. Bernays. This translation from the French is based on the 1954 version at http://e-collection.library.ethz.ch/eserv/eth:32719/eth-32719-02.pdf.

Böhm's PhD dissertation was completed shortly after David Wheeler's August 1951 Cambridge dissertation and so is probably only the second one in computer science. In addition to its historical interest, Böhm's dissertation is a stellar example of conceptual and notational economy. In just 46 pages, it presents (a) an abstract machine, quite faithful to contemporary real stored-program machines such as the IAS design from Princeton, the EDVAC in Philadelphia and the EDSAC in Cambridge, but with index instructions like the Manchester Mark 1; (b) a simple but complete programming language, including parenthesized arithmetic expressions; (c) a loader program similar to Wilkes's for the EDSAC; and (d) a compiler from the language to the abstract machine's instruction set, where — also a first — the compiler is written in the compiled language itself. Contribution (a) is in chapter 1; (b) in chapters 2 and 3; (c) in chapter 4; and (d) in chapters 5 through 7.

For historical context, the other doctoral students of Bernays are Julius Büchi, Haskell Curry, Erwin Engeler, Gerhard Gentzen and Saunders Mac Lane.

Böhm's programming language is very minimal; the only statement is assignment to a variable x (denoted $e \to x$) or to the memory address stored in variable x (denoted $e \to \downarrow x$). Since the program counter π is an assignable variable, this suffices to express jumps and switches.

The left-to-right assignment notation $e \to x$, which was customary until the mid-1950es, allows for a particularly elegant form of literate programming, with the (left-justified) natural language explanation to the left and the corresponding (right-justified) program code to the right. This is used to excellent effect in chapter 7 which consists of the entire commented compiler source code.

Böhm's early work does not seem to have received the attention it deserves, although it features prominently in Don Knuth's survey *The early development of programming languages* in Encyclopedia of Computer Science and Technology, volume 7, 1977, 419-493.

It is my hope that this English translation will make Corrado Böhm's dissertation accessible to a wider audience. I have tried to translate it sentence for sentence and to preserve connotations where possible. A more idiomatic English translation would have shorter sentences and fewer Latinate words, but I believe the present approach better preserves the original's atmosphere.

Everything in these pages (except the present page and the appendices) is Böhm's work, including the chapter numbers starting at 0, the notation, the footnotes and the terminology. However, I have translated *exploration*, *substitution* and *terme* into the more recognizable terms *load*, *store*, and *operand*, and fixed a few misprints.

Peter Sestoft
IT University of Copenhagen
4 May 2016

0. Introduction

0.1. Utility of automatic coding

Today one tends more and more to employ big digital computers because of the following characteristics:

0.11. The ability to execute a sequence of computations by following a *program* fixed beforehand.

0.12. A computation *speed* remarkably better (by several hundred times) than preceding electromechanical or manual calculators.

These two properties allow one to establish an analogy, from an organizational and economical viewpoint, between one of these machines and a computing office. From the same viewpoints we could outline a classification into three classes of work that one could submit to a programmable digital computer:

A) Computations having a character of *extreme urgency* and involving a very large amount of data: e.g. meteorological forecasts or analysis of electoral results.

B) Mass computations involving a very large number of operations that must be repeated many times, such as inversion of high-dimensional matrices.

C) Other computations not classifiable as neither *A*) nor *B*): e.g. integration of a differential equation, solution of a transcendental equation, etc.

For categories *A*) and *B*) the difficulty of the program and the *duration* of its preparation does not play a big role, because what is required of the program is to allow one to profit maximally from the speed of computation. By contrast, for category *C*) the time spent by the people preparing and checking the program may be much larger, by even another order of magnitude, than the time spent by the machine executing the computations of that same program (e.g. it may have the relation days/minutes). For category *C*) it is property 0.11 that is essential, i.e. the ease and flexibility of programming, as conceived in the machine design.

In the following we assume that the reader knows the principles of a digital computer and especially those of recently constructed computers [1], [2] ([1]). The work of programming may even have a direct influence on the overall duration of the computation: if, by accident, when one submits a program to the machine for the first time, the results of the computation do not agree with those predicted by a preceding control computation, and if one can exclude a machine malfunction, one is obliged to admit at least one of the following disturbing causes:

a) the program is erroneous from a logical-mathematical viewpoint.

b) the program has not been correctly recorded on the machine input media.

[1] The numbers in square brackets refer to the bibliography placed at the end of this study.

To pinpoint these errors, additional tests of the computation, often at reduced speed, become necessary, with a relatively large accompanying loss of usable machine time.

It would be better to try to avoid producing errors of any kind. An interesting method is the one that consists in the auxiliary automation grouped under the name "coding machines" ([2]). Another method that has given very good results is that adopted by Mr. WILKES [3], called "the method of subroutine libraries"; according to this method one seeks to compose each new program by opportune combinations of partial programs established in advance and of which duplicates are conserved in a kind of library. The checking of a new program is thus reduced to checking the instructions that connect the various *subroutines*.

On the subject of turning a mechanical computation into a program, one must consider that the human work rests on a double knowledge: on the one hand about the method to solve a given problem, and on the other hand about the general organization (functioning and structure) of the machine used. The human must thus transform each phase of the computation into a sequence of operations that can be executed by the machine.

But, to what extent could the machine even compute that sequence for itself, starting from certain formulas that express the method for solving the proposed problem? The answer is the principal goal of the present report. Below, for brevity we refer to this question as *automatic programming*.

The problem of automatic programming has been posed with a certain latitude to avoid immediately falling into the difficulties attached to the notion of a programmable computer. As also remarked by Mr. VON NEUMANN [4] we do not yet possess, today, a satisfactory theory of automata and in particular programmable calculating machines. To be able to formulate the question of automatic programming more rigorously than we have done here, in a later paragraph we shall use certain results due to Mr. TURING.

Similar question have by the way already been treated; here is a short survey on the topic:

In 1949 Mr. ZUSE [5] treated the formulation of the problem of recognizing whether a sequence of signs involving variables and algebraic operator symbols (see the examples in 4.2) do or do not possess a meaning ([3]). This question constitutes a part of the more general problem of automatic programming of a formula containing parentheses, and is the object of chapter 5 in the present work. By coincidence Mr. RUTISHAUSER has also been occupied by this latter question [6] but the principles of our two solutions are entirely different (see 5.1); these kinds of problems admit numerous ways to attain the same goal.

Before concluding these general remarks it seems useful to emphasize those of the results of our study that appear most susceptible to practical application:

0.13. The possibility of writing every program in the form of a sequence

[2]Such machines have been constructed by Mr. AIKEN and Mr. ZUSE.

[3]In the cited work Mr. ZUSE expresses the opinion that one could construct a new kind of machine that would be able to solve this problem among others. Since then we have succeeded in solving the same problem using a punch card tabulator (System Bull). This success has encouraged us to undertake the present attempt at synthesis on automatic programming.

of formulas adhering, as closely as possible, to the notational conventions followed by mathematicians, while excluding instructions that concern primarily the computer and its particularities.

0.14. The possibility of resolving the problem of automatic programming.

The coding is done exclusively with the use of the computer, in two consecutive phases: During the first phase, the support medium on which the program is recorded as a sequence of formulas is placed at the machine's input device and the computer executes a series of computations that produce the same program, but recorded as coded instructions ready to be interpreted and automatically executed later on. During the second phase the computer performs the computation, according to the coded instructions previously produced.

It is evident that the two possibilities mentioned facilitate the checking of a program by reducing it to the checking of formulas, and by eliminating transcription errors because no transcription takes place.

Furthermore, even if one abstracts away from any practical value of this research, the mere existence of these possibilities appear to have, in our opinion, a certain interest from the point of view of the theory of computing machines.

0.2. Specification of the category of digital machines considered

To avoid circular definitions of the terms: automatic computers, program, coded instructions, automatic programming etc., let us briefly make precise to which categories of machines these phrases refer.

0.21. Among the automatic digital computers we encounter first of all the office machines (manual or electromechanical) that contain some fixed *microprogram*; they can in effect automatically execute operation sequences to determine e.g. the product or quotient of two given numbers.

0.22. Punched card installations consist essentially of digital computers that may be deployed differently according to their previous programming. One varies the program by changing the electrical circuits of the machine, thus obtaining a corresponding variation in the sequence of arithmetic operations executed.

0.23. The most evolved computers differ from the preceding ones, from the programming point of view, first and foremost by the fact that the modification of electrical circuits that determine the operations is not effectuated by hand by the operator, but by means of conventional numbers or *coded instructions* defined beforehand, and once and for all, by the machine. After this description the reader easily recognizes that our study concerns only the latter category of machines. In any case, the difference between 0.21, 0.22 and 0.23, so evident once one thinks about the construction or use of a given machine, risks disappearing when one must use, as we shall do, very general logical properties of mechanical calculation. Thus the necessity of appealing to the notion of the next paragraph.

0.3. Using Turing's theory

Since 1937 Mr. TURING has created a logical theory that has allowed him, starting from a profound analysis of what a computation is, to define what is today called *mechanical calculation*. His definition follows from the description of a particular kind of automatic machine, suited for calculating sequences of digits representing numbers known in advance or having known properties.

His study uses this definition to quickly obtain certain results in mathematical logic. Some of his conclusions — which we report further below — also have a great interest for a possible future theory of computers, and they have influenced the advances in computer programming over the last ten years.

Mr. TURING has shown how the notion of mechanical computability of a number is fundamentally equivalent to the notion of existence of general methods for determining that very number. He has furthermore shown, starting from the hypothesis that one knows how to construct particular computers for calculating particular number series, that there exists a so-called *universal* computer that enjoys the following property: if one gives the universal computer the *description* (conventionally formalized) of the functioning of any *particular* computer, the former is able to *simulate* the behavior of the latter, i.e. compute in its place.

We would like to suppose — what seems quite plausible — that the most advanced computers are universal, in the sense specified by Mr. TURING. That allows us to formulate the following two working hypotheses, to which we will appeal later on:

0.31. Programmable computers of category 0.23 are, from a logical-mathematical viewpoint, mutually equivalent.

This hypothesis allows us to limit our study to a single type of computer, e.g. a *three address* computer, without fearing loss of generality.

0.32. The "program" admits, in universal computers, a double interpretation. The first is: "Description of a behavior of the computer". The second: "Description of a numeric method of computing".

This hypothesis is just a new wording of TURING's idea and justifies the search for a formalism apt to making this idea fully evident.

0.4. Summary

We shall first describe (chapter 1) the structure and organization of a computer like those already constructed. We can thus define explicitly a fundamental cyclic program that expresses the definition of "program" (see 0.32) in terms of this machine, before even specifying what coded instructions there should be. When this latter choice has been made, we demonstrate the universality of this computer and introduce some supplementary instructions that will be useful later. After having passed from the usual program notation to another notation (chapter 2), we justify the symbolism introduced by the possibility of writing any program in this formal language, building only on logical and algebraic notions (chapter 3). Moreover, we demonstrate (chapter 4) that one can make this formalism available to the computer using a teletypewriter as intermediary, by

establishing a two-way unambiguous correspondence between symbols and integer numbers. The formulas that represent a calculation program algebraically may contain parentheses (chapter 5) or polynomials of multiple variables on a normal form (chapter 6). In both cases we can make the computer itself, thanks to a fixed program independent of the particular nature of the formulas (chapter 7), execute the computations necessary to produce detailed coded instructions from the formulas expressing the intended numerical method. Finally (chapter 8) we have put in relief the logical redundancy of certain operations and recalled the arithmetization of propositional calculus.

1. Description of a three-address programmable computer

1.0. The machine that we are going to describe in outline is a programmable computer in principle. Consequently we leave aside all questions concerning the number representation (binary or decimal or a mixture) used in the computer and similarly all considerations concerning representation of fractions (point placement) and negative numbers.

Without loss of generality we can assume below that every number used in the computer is a non-negative integer of a fixed length (e.g. 14 digits), i.e., composed by a fixed number of decimal digits[4].

The double meaning of the numbers used.

Each integer thus quantified in general possesses a second meaning: That of a coded order fixed once and for all at the construction of the computer by a table of coded operations, or instruction code (see 1.3). For instance, the number zero, interpreted as an instruction, causes the computation to halt (STOP).

The number-numbers and the instruction-numbers, while having no distinguishing features, undergo different treatment thanks to the structure of what we call the computer's fundamental cyclic program (see 1.2).

1.1 The computer's organs (see figure 1)

I. An arithmetic unit UA where numbers originating from other machine organs (figure 1 connection 3 and 7) are subject to arithmetic operations (addition, subtraction, multiplication, etc.). From our point of view, it is convenient to think of all operations in the UA as binary, i.e. as characterized by mapping each ordered pair of numbers — called respectively the 1st and 2nd operand of the operation — to a 3rd number, or result.

II. An internal memory M composed of approximately 1000 cells each of which can hold a number of an instruction (of 14 decimal digits) for an arbitrary duration. To each cell is attached an order number or address. The internal memory enables at any time two categories of operations:

[4] For instance, the numbers 00000000000000 and 00000000000001 represent zero and one.

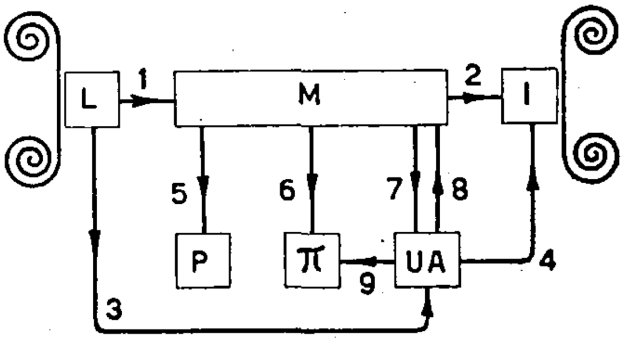

Figure 1: The computer's organs.

1.11. The store or transfer to a particular address, i.e. the act of transferring a number from a machine organ to the cell having a particular address, where the number previously held in that cell will be forgotten, i.e., replaced by the new one (see connections 1 and 8 in figure 1).

1.12. The load or transfer from a particular address, i.e. the act of transferring a copy of the number found in a given (but otherwise arbitrary) cell, to an organ of the computer (see connections 2, 5, 6, 7 in figure 1).

III. A reader or input organ L that allows to put into M (connection 1) or into UA (connection 3) numbers (or instructions) recorded on perforated paper tape or on magnetic tape.

IV. A printer or output organ I that allows to perforate or magnetically register on tape, at the machine's output, numbers originating from M (connection 2) or from UA (connection 4).

V. A register π with a capacity of four decimal digits. Connections 6 and 9 (fig. 1) represent, respectively, the loading of the four rightmost digits of a number contained at an arbitrary address of M, or contained in UA. The use of this register will be explained in 1.2.

VI. An organ of automatic control, or pilot, P whose purpose is to coordinate the action of all the organs previously mentioned, in such a way that the instructions belonging to a calculation program get executed correctly and in the right order. In particular, all the connections shown in fig. 1 representing the transfer of numbers internally in the computer are activated by the control at the appropriate moments. Overall, we can describe the action of the control simply as the ability to first interpret the number contained in a certain address obtained via a load operation (connection 5 in fig. 1) as a coded instruction, and the ability to then effect its execution.

1.2 Stationary functioning

After having described the computer's organs separately, we now seek to describe how a given computation runs in time, confining ourself to the stationary phase, i.e. excluding the moments at which the machine begins or finishes a

computation[5]. We must characterize the computer's degree of automation and explain the mechanism by which the presence of certain numbers in a region of the internal memory M leads to a corresponding course of computations. Thus we assume that at a certain moment of the stationary phase, coded instructions are stored in M in such a way that the majority of those that must be executed consecutively are at consecutive addresses.

The computation takes place, independently of the specific nature of the computations to perform, according to a fixed scheme with a periodic structure, in which the π register and the control P play an essential role.

This scheme comprises the following three successive phases:

1.21. The control loads from the address in M specified by the number contained in the π register.

1.22. The number contained in the π register is replaced by that obtained by incrementing it by one.

1.23. The control P, after having interpreted the number loaded in phase 1.21 as an instruction, effects its execution.

The next phase of the computation is formally identical to 1.21, the one that follows identical to 1.22, and so forth. The set 1.21, 1.22 and 1.23 constitute the fundamental cyclic program. This calculation schema is formal in the sense that it characterizes not just a programmable computer, but a class of programmable computers. In fact, we have not yet specified, apart from this schema, the nature of the arithmetic or logical operations described by the coded instructions. It is only after one has chosen a class of coded instructions and joined it with the fundamental cyclic program that the description of the principles of our computer is unambiguous.

Our intention is to demonstrate that if one codifies only the operations that we have mentioned in paragraph 1.1, that already defines a computer having the same flexibility as computers of category 0.23, i.e. universal computers.

In summary, the operations of that paragraph were the following:

1.10. Arithmetic operations (addition, subtraction, etc.)

1.11 and 1.12. Store and load operations, i.e. transfer operations, including exchanges with the external world (connections 1, 2, 3 and 4 of fig. 1), internal connections (6, 7, 8 and 9) ([6]) and the STOP operation.

To demonstrate this, we must first introduce some convenient notations related to all programs, i.e., define a code.

[5]The moment of starting a computation is chosen by the operator, who acts on the control by means of organs that we have not described, limiting outselves to the essential ones. Similarly, the moment at which the machine stops computing may depend — except in the case where it is caused by the program — on multiple circumstances foreign to the normal functioning of the machine.

[6]Connection 5 is not mentioned as it belongs to the fundamental cyclic program.

1.3. Three-address code

Every coded instruction is represented by a number N defined like this:

$$N = 10^{10} \cdot c_1 + 10^8 \cdot c_{op} + 10^4 \cdot c_2 + c_r$$

where c_1, c_2, c_r are integers < 1000 and c_{op} an integer < 100; i.e. the decimal structure of the number N is the following:

$$\underset{0}{\cdot}\ \underbrace{\cdot\cdot\cdot}_{c_1}\ \underbrace{\cdot\cdot}_{c_{op}}\ \underset{0}{\cdot}\ \underbrace{\cdot\cdot\cdot}_{c_2}\ \underset{0}{\cdot}\ \underbrace{\cdot\cdot\cdot}_{c_r} \quad \text{(in total 14 decimal digits)}$$

The meanings of c_1, c_{op}, c_2, c_r are fixed by the following rules:

c_1	is the address of the instruction's first operand.
c_{op}	is the number of the instruction's characteristic code.
c_2	is the address of the instruction's second operand.
c_r	is the address to which one must transfer the operation's result.

The c_{op} values are defined according to this table:

Symbol	c_{op}	**Result of the corresponding operation**
$+$	01	Sum (modulo 10^{14}) of the 1st and 2nd operand.
$\dot{-}$	03	Difference between 1st and 2nd operand, if positive; else 0 ([7]).
$\cdot$	02	Product (modulo 10^{14}) of 1st and 2nd operand.
:	04	The integer quotient 1st operand divided by 2nd operand.
$\div$	06	The absolute difference between 1st and 2nd operand.
mod	07	The remainder of 1st operand divided by 2nd operand.
$\cup$	08	The greater of the two operands.
$\cap$	09	The smaller of the two operands.

Thus the instruction

$$N' = 0123\,04\,0567\,0890$$

has the following meaning:

"The number held at address 123 should be divided by the number held at address 567; the integer part of the quotient must be transferred to address 890."

Repeating the general notation with letters instead of the numbered addresses, all transfer operations are represented by numbers having the following decimal structure:

$$T = 0c_1\,05\,0c_2\,0c_r$$

where 05 is the code number for transfer instructions and c_1, c_2, c_r have the same meaning as before; the instruction coded as T thus has the following meaning: "The number loaded from address c_2 is transferred to address c_r." Note that

[7] These modifications of the usual subtraction and division operations are dictated by the need to stay within the class of non-negative integers.

one can consider the transfer an operation with two operands, just neglecting the first operand address c_1; in reality it is an operation in one operand only.

Let us associate once and for all the address 000 with register π, and the same address 001 with both organs L and I (see 1.1) so that we can interpret the following three numbers:

$$\begin{aligned} J &= 0c_1\,05\,0c_2\,0000 \\ L &= 0c_1\,05\,0001\,0c_r \\ I &= 0c_1\,05\,0c_2\,0001 \end{aligned}$$

as follows, respectively:

J) "The rightmost four digits of the number with address c_2 must be transferred to register π".

L) "The number on the input tape L must be transferred to address c_r and the input tape must advance to the next number written on it".

I) "The number loaded from address c_2 must be written to the output tape I and the tape must advance to the next place".

One recognizes in the three instructions J, L and I the preparation for an unconditional jump [2, page 8], a read order, and a print order or output.

Finally, let us interpret the number

$$S = 0000\,00\,0000\,0000$$

as STOP. Below, we accept a further reason for stopping: when at a given moment in time the contents of the π register is 0000 (^{8}).

In summary, we have represented every coded machine instruction described by an integer N having a fixed decimal structure: by specializing the numeric value of certain decimal positions have defined multiple subclasses of operations that we here name by the letters previously introduced in this paragraph: N' (as example of an instruction having an arithmetic character), T, J, L, I and S (instructions having a logical character).

1.4 Universality of the described computer

To demonstrate that the machine described has a flexibility comparable to that of computers recently constructed, it obviously suffices to show that every instruction coded for the latter can be expressed using instructions of the former. We shall limit ourselves to the most interesting case, i.e. to that of conditional commands[9] (C.C.), saving the discussion of instructions related to the devices called respectively "i-register" and "B-tube" for a later paragraph.

[8] If in the computer the digit zero is represented by the absence of a pulse, the second cause is a consequence of the first one, as one can easily verify by following the mechanism of the fundamental cyclic program.

[9] In English: conditional call (see also [2, page 45]).

The C.C. most often used is described as follows, assuming that it is placed at address n:

C.C.1. "If the number stored at address a is positive, execute the instruction whose address is x, otherwise execute the instruction at address $n+1$." With the numbers a, x and n known in advance, the purpose of this instruction is obviously to be able to automatically influence the succession of operations, or in general the entire course of the computation from a given point, as a function of the results obtained until that very point. We are going to show the possibility of encoding a similar C.C., which is a slight generalization of C.C.1, namely:

C.C.2. "If the number contained at address a is positive, execute the instruction whose address is x, otherwise the one whose address is y."

Proof. Suppose first that one can transfer to address c_2 the number u defined like this:

$$u = \begin{cases} x & \text{if } (a) > 0 \\ y & \text{if } (a) = 0 \end{cases}$$

where (a) represents the number held at address a.

Now the execution of a J instruction immediately brings us to the goal. Here as elsewhere, we can calculate u by the following relation:

$$u = [1 \cap (a)] \cdot x + [1 \dot{-} (a)] \cdot y \tag{1.41}$$

which completes the proof. (For $\cap$ and $\dot{-}$, see 1.3). As illustration, let us write the complete sequence of coded instructions in a particular case. Let, for example,

$$x = 301, \quad y = 302, \quad a = 002, \quad u = 300$$

and let the number 1 be stored e.g. at address 007, whereas x and y are respectively at addresses 025 and 026. Then the following sequence of instruction codes

$$\begin{array}{rcll} n_1 & = & 0007\,09\,0002\,0028 & \text{Compute } 1 \cap (a) \\ n_2 & = & 0007\,03\,0002\,0029 & \text{Compute } 1 \dot{-} (a) \\ n_3 & = & 0028\,02\,0025\,0030 & \text{Compute } [1 \cap (a)] \cdot x \\ n_4 & = & 0029\,02\,0026\,0031 & \text{Compute } [1 \dot{-} (a)] \cdot y \\ n_5 & = & 0030\,01\,0031\,0300 & \text{Compute } u \\ J & = & 0000\,05\,0300\,0000 & \text{Unconditional jump} \end{array} \tag{1.42}$$

is equivalent to C.C.2.

1.5 Role of the B-tube and the i-register

In [8] it is shown by examples how the devices called B-tube or i-register facilitate "address computations" thanks to supplementary instructions related to them. Their superfluity from a purely logical point of view is likewise noted [8, page 51]; but it is almost obvious that nothing prevents (even in the computer we just described) the program execution from modifying the encoded instructions by adding, e.g. to an address that is part of an instruction, an integer (mod

1000), before the instruction itself gets executed. This is, roughly, the effect of the B-tube.

1.6 Iterated store and load operations

If one reflects on the practical utility of these auxiliary instructions for address calculations, one realizes that they play a certain role in the solution of the problem of automatic coding by the method of subroutine libraries. In fact, thanks to these auxiliary instructions, the instructions making up the subroutine undergo only very weak modifications according to the zone of internal memory (more precisely, the addresses) in which they are placed.

Since our ultimate intention is to express the coded instructions by formulas that are invariant relative to the internal memory addresses, we shall add some mechanisms to the machine that permit what we call, in brief:

1.61 "iterated stores" and
1.62 "iterated loads", i.e.

(see again 1.11 and 1.12) operations defined like this:

1.61. Transfer of a number (placed in a particular organ of the computer) to the address indicated by the three rightmost digits contained at a given address.

1.62. Transfer to a particular organ of the computer of a duplicate of the number held at the address indicated by the three rightmost digits of the number held at a given address.

The description of the mechanisms that perform these operations fall outside the scope of this study; but it is useful to show how one could incorporate the "iterated" operations in the three address numeric code previously described.

Recall that the decimal structure of any of the instructions described was:

$$N = 0c_1\, c_{op}\, 0c_2\, 0c_r$$

and note that

1st The execution of this instruction implies, in general, two loads (at addresses c_1 and c_2) and one store (at address c_r).

2nd The digit immediately to the left of the first digit of c_1, c_2 and c_r is zero.

Let us distinguish the *iterated* load of c_1 or c_2 or the *iterated* store of c_r from the corresponding *simple* operations by replacing, respecticely, the 1st, 7th and 11th digit from the left of N by unity, where these replacement may be simultaneous or not, as the case may be.

Example: The instruction number

$$n' = 1997\,04\,1996\,1995$$

thus has the following meaning:

"The number contained at the address equal to the number (mod 1000) contained at address 997 must be divided by the number contained at the address equal to the number (mod 1000) contained at address 996; the integer part of the quotient must be transferred to the address equal to the number (mod 1000) contained at address 995."

Obviously the number n' is invariant relative to any change of the addresses of the division's three operands, whereas the number N' (see 1.3) is not; one understands thus the interest that instructions such as n' may have for the automatic coding.

2. Changing the notation system

Our final goal is to demonstrate the possibility of solving the problems of automatic coding with the help of a computer following the principles described in the preceding chapter. To simplify this task we propose a purely formal change of notation for the three-address instruction numbers.

Let us consider a completely arbitrary instruction among those that we have previously introduced. It will have the following decimal structure:

$$\varepsilon_1 c_1\, c_{op}\, \varepsilon_2 c_2\, \varepsilon_r c_r$$

where c_1, c_2, c_{op} and c_r have the well-known meaning and each ε_i $(i = 1, 2, r)$ represents one of the digits 0 or 1. Now in the following let us:

2.1. Purely and simply delete ε_i if $\varepsilon_i = 0$, and replace every $\varepsilon_i = 1$ by the symbol $\downarrow$.

2.2. Always interpose the *transfer symbol* $\rightarrow$ between c_2 and c_r (or $\downarrow c_r$).

2.3. Replace c_{op} by the corresponding operation symbol (see the table in 1.3) but leaving out the nonessential part $\varepsilon_1 c_1 c_{op}$ when $c_{op} = 05$.

2.4. Systematically replace the adresses 000 and 001 by the symbols π and ?, and the 52 addresses from 002 to 053 by the letters $a, b, \ldots, z, A, B, \ldots, Z$ in alphabetical order.

2.5. Use letters with indexes or non-Latin characters to indicate numeric addresses > 053; in particular, Ω for an address that always contains the number 0 (STOP), for instance address 998.

2.6. Interpret a number written in italics as indication of an address that contains that number, for instance, *123* represents some address that contains the number 123 (^{10}).

Let us use these conventions to write in the new notation the instruction

[10] The corresponding traditional notation is $\langle 123 \rangle$ (see e.g. [6, page 4]).

numbers encountered in the preceding pages:

N'	$123 : 567 \to 890$	(convention 2.2, 2.3)
T	$c_2 \to c_r$	(convention 2.2, 2.3)
J	$c_2 \to \pi$	(convention 2.2, 2.3, 2.4)
L	$? \to c_r$	(convention 2.2, 2.3, 2.4)
I	$c_2 \to ?$	(convention 2.2, 2.3, 2.4)
Instr. equiv. to S	$\Omega \to \pi$	(convention 2.2, 2.3, 2.4, 2.5)
Instr. equiv. to S	$0 \to \pi$	(convention 2.2, 2.3, 2.4, 2.6)
n'	$\downarrow 997 : \downarrow 996 \to \downarrow 995$	(convention 2.1, 2.2, 2.3)
$N(^{11})$	$c_1\ op\ c_2 \to c_r$	(convention 2.1, 2.2, 2.3)

The short coded program equivalent to a C.C.2 (see (1.42)) now takes this form:

(2.7)
$$\begin{array}{ll} n_1 & 1 \cap a \to A \\ n_2 & 1 \dot{-} a \to B \\ n_3 & A \cdot x \to C \\ n_4 & B \cdot y \to D \\ n_5 & C + D \to u \\ J & u \to \pi \end{array}$$

1ST COMMENT. The correspondence between the notational system we are used to and the one we have just proposed is one-to-one. The inverse transformation is effectuated by deleting the symbol $\to$ (and inserting $c_{op} = 05$ only if the first operand and the operation symbol are missing) and replacing each letter or operation symbol by the code number according to the established conventions: one must write 1 or 0 before each address according as the symbol $\downarrow$ is present or absent. We have called $\to$ the transfer symbol because it replaces operation 05; nevertheless the real function of this symbol is to make it evident that in the execution of $c_1\ op\ c_2 \to c_r$, the addresses c_1 and c_2 undergo a different treatment than does c_r. In fact the two former are the objects of *load* operations (for which we have not designated any symbol) whereas the latter is the object of a *store* instruction.

2ND COMMENT. The introduction of the "iterated store and load" operations in the three address code could lead to be belief that double iteration of these instructions would constitute a new problem and make a new coded instruction or a new symbol necessary. But they do not, as we can show by a relatively simple application of the proposed symbolism. In fact, the following "doubly iterated store" operation: "Transfer a number from address x to the address contained at the address contained at address A" is expressed in two lines by

$$\begin{array}{rcl} \downarrow A & \to & P \\ x & \to & \downarrow P \end{array}$$

Similarly, the inverse operation which is a kind of "doubly iterated load":

$$\begin{array}{rcl} \downarrow A & \to & P \\ \downarrow P & \to & x \end{array}$$

[11] Here and below, "*op*" stands for any operation symbol arbitrarily chosen among the coded operations.

3. Justification of the symbolism introducted

3.1 Potential for an abstract program formulation

In chapter 1 we have demonstrated the universality of the computer described. That is equivalent to a confirmation that every description of calculation methods can be expressed with the help of a suitable sequence of the symbols $a, b, \ldots, z, A, B, \ldots, Z, \Omega, ?, \rightarrow, \downarrow, +, \dot{-}, \ldots, \cup, \cap$, where the meaning of the symbols is that defined in the preceding chapter.

But our intention is to show that to use the symbols correctly, it is not necessary to go back in each case to their operational meaning with respect to the computer. The same symbols — and that is what constitutes the practical advantage of the notation — may be subject to a second interpretation of an entirely logical and algebraic nature ([12]). This interpretation is not directly linked to any notion of a machine; rather it is associated with that of a "description of a numeric method for the solution of a given problem", a phrase that in the following we shall replace with the shorter "description".

What follows obviously has a heuristic character and could as well serve as definition of what we understand, today, by "description".

Every *description* implies that one must have numeric *data* and *exhaustive information* about the arithmetic operations to be carried out on the data, so as to obtain intermediate values and the final result of the computation.

Every *description* is formed by an alternation of formulas and phrases; the former usually indicating the sequence of operations and the latter indicating the variants, iterations, or in general the auxiliary conditions influencing the actual course of the computation. For instance, there are *descriptions* that use iterative methods where the number of iterations is unknown *a priori*, and descriptions in which certain intermediate results necessary for the rest of the computation must satisfy auxiliary conditions (being positive, numbers being real, random choice, etc.).

We thus allow that for each *description*, one can state *a priori* the definitive sequence of arithmetic operations, or at least the criteria of choice, i.e. the conditions that must be tested at a later time, during the computation, to uniquely determine the course of the computations.

As a consequence of abstaining from a higher order of complexity than that, it is possible to decompose every *description* into a sequence of formulas — of the logical-mathematical kind — by applying a system of conventions concerning in particular:

3.2. The decomposition of every *description* into a certain number of similar groups, each composed of three parts.

3.3. A further decomposition of the three parts of each group into formulas whose operations are at most binary.

[12] One can already glimpse this result by comparing formula (1.41) with that obtained by replacing, everywhere in the first five lines of the formulas (2.7), the sign $\rightarrow$ with $=$, and eliminating A, B, C and D, considered as algebraic quantities, from the five equations thus obtained.

3.2. Decomposition into groups. Graphical interpretation

Any *description* is decomposable into different groups having a similar structure. Let us call these groups $A, B, C, \ldots, K, \ldots$. Any group K comprises the following three parts:

- An indication that it belongs to group K.
- A chronological indication of arithmetic operations and the quantities on which they operate.
- An indication of the next group to choose (the choice being in general dependent on certain conditions).

It is convenient to call the first group A; similarly let us call Ω the group characterized by the fact that it has no operations nor any indication of the next group to choose; in this case it evidently is an indication that the *description* is finished. The decomposition that we have just proposed has the advantage that it makes the order of the groups inessential, so that they can be permuted arbitrarily without affecting the intelligibility of the *description*.

Below we shall also make use of a graphical interpretation of such a decomposition by means of a *graph* of a very general type [9]. This is a generalization of the "structure diagram" [10] that we easily obtain in the following fashion: to each group one has a corresponding *point* denoted by the same letter. Henceforth let K and L be two arbitrary points or groups. These are joined by the oriented segment $\vec{KL}$ if and only if, in the third part of group K, L is included among the next groups to choose from. The graph drawing is complete when it contains all the links included in the *description*. Each iterative process is represented in the graph by a circle or by a loop (see fig. 2, page 26). To each group containing a choice among n alternatives correspond a point with at least n branches in the graph (see fig. 5, page 42, point D where $n = 11$).

3.3. Conventions concerning formula writing

First recall that since the beginning of chapter 1 we have limited ourselves to considering only integral non-negative numbers.

The given quantities and the intermediate or definitive results will be designated by letters chosen at will, but when possible in lowercase letters. Numeric constants will be written in italics and the arithmetic operations will be represented by the usual symbols (see table in 1.3).

Quantities provided with indexes may also be represented directly as we shall show, if one renounces (in general) the possibility of letting the indexes range over the same set of values that they range over in the description. In any case, this reservation is not too onerous because, in the worst case, it is a matter of transforming the set of values assumed by each index by adding a constant number, known *a priori*. The final conventions for the translation of phrases into formulas can be condensed using the following "dictionary":

Phrase	*Symbol*	
"Becomes" or "is made equal to ..."	$\rightarrow$	([13])
"A known number"	$?$	
"The group to choose next"	π	
"The quantity with index i"	$\downarrow i$	
"The group that begins here"	π'	
"No group"	Ω or 0.	

The most important applications become:

Phrase or formula	*Formula*	*Verbal translation*
Let a and b be given numbers	$? \rightarrow a$	Make a equal to a given number
	$? \rightarrow b$	Make b equal to a given number
x is a result	$x \rightarrow ?$	x becomes a known number
The next group to choose is C	$C \rightarrow \pi$	C becomes the next group to choose
Do not choose any next group	$\Omega \rightarrow \pi$	No group is next
	$0 \rightarrow \pi$	
Let r be the product of a and b	$a \cdot b \rightarrow r$	Make r equal to $a \cdot b$
If $m = 1$ the next group is S; if $m = 0$ there is no next group	$m \cdot S \rightarrow \pi$	$m \cdot S$ becomes the next group to choose
Let the new number h be 1	$1 \rightarrow h$	h becomes 1
Henceforth let us call m what was previously $m + 1$. Or, increment m by unity.	$m + 1 \rightarrow m$	m becomes $m + 1$
Here begins group K ([14])	$\pi' \rightarrow K$	The group that begins here is made equal to K

The dictionary and the sample of formulas that we have presented should suffice to translate any description into formulas, if in addition one takes into account these three rules of "syntax":

3.31. *Rule of explicitation.* In a "well-formed" sequence of formulas each letter ([15]) appearing to the left of the sign $\rightarrow$ has previously ([16]) appeared at

[13] Our definition of the sign $\rightarrow$ with respect to the computer (see 2.2) corresponds to Mr. Von Neumann's [10]. On the other hand, the logical definition given above comprises the meaning given by Mr. Zuse [5] to the sign $\rangle =$ "ergibt" as well as that given by Mr. Rutishauser [6] to the sign $\hookrightarrow$. Rather than use three different symbols [6] to describe what in the machine is achieved by a single operation (i.e. a store), we have preferred to slightly broaden the logical definition of the sign $\rightarrow$.

[14] We shall systematically translate the first part of each group like this.

[15] Certain letters are exceptions to this rule; notably Ω (trivially) and π', and the letters that represent groups of the description. These exceptions are the object of a discussion in chapter 4.

[16] Here "previously" means "in a preceding formula of the same group or in a formula be-

least once to the right of the same sign; to the right of the sign $\rightarrow$ there is never more than one letter.

3.32. *Rule of index transformation.* If in the description there is a quantity indexed by i where i varies e.g. from 0 to n, one must replace i by the index $i + z$ where $z \geq 100$ is a fixed number. If more than one quantity depends on the same index i, the notation $\downarrow i$ (see above) cannot suffice: this advertises the fact that one must in general introduce as many indices (differing from each other by constants) as quantities; finally the sets of numbers traversed by the different indices must have no element in common and each index must be an integer number with three digits (which implies in any case that $n < 900$).

3.33. *Rule of arithmetization of condition of choice of the following group.* The need for unambiguous execution over time requires us to assume that the criteria for choosing among groups $A, B, \ldots, K, \ldots$ can be always expressed like this: "If the condition C_A is satisfied one must choose A as the next group, if the condition C_B is satisfied one must choose B as the next group, ..., if the condition C_K is satisfied one must choose K ...", where the conditions $C_A, C_B, \ldots, C_K, \ldots$ constitute strict alternatives in the sense that one and only one of them can be satisfied at the same time.

Suppose now that the groups $A, B, \ldots, K, \ldots$ are distinct positive integers and that to each condition C_K there is a corresponding nonnegative integer e_K such that for each group:

$$e_K = 0 \text{ if and only if } C_K \text{ holds}$$

If we denote by S the group to choose, it is easy to see that the rule we seek can be expressed by the formula

$$S = (1 \dot{-} e_A) \cdot A + (1 \dot{-} e_B) \cdot B + \cdots + (1 \dot{-} e_K) \cdot K + \cdots$$

One can consider S the scalar product of two vectors in an n-dimensional space (n being the number of terms in alternative) where the vectors have components $(1 \dot{-} e_A, 1 \dot{-} e_B, \ldots, 1 \dot{-} e_K, \ldots)$ and $(A, B, \ldots, K, \ldots)$ respectively. In any case, since $1 \dot{-} e$ is 0 or 1 according as $e > 0$ or $e = 0$, it follows that the first vector is a unity vector and belongs to a basis: thus one has $S = A, B, \ldots, K, \ldots$ according as $0 = e_A, e_B, \ldots, e_K, \ldots$; QED ([17]).

3.4. Two examples of description

1) Euclid's algorithm: Search for the greatest common divisor m of two given numbers (see figure 2).

---

longing to a group such that a traversal of the graph starting from group A in the direction of the arrows will encounter that group first".

[17] In the case of an alternative with just two terms one finds a formula analogous to (1.41) by using the identity $1 \dot{-} (1 \dot{-} e) = 1 \cap e$ (e is integral and ≥ 0).

A	Let a and b be two given numbers. Let M be the greatest and m the least of these.	$\pi' \to A$ $? \to a$ $? \to b$ $a \cup b \to M$ $a \cap b \to m$ $B \to \pi$
B	Let r be the remainder of division of M by m. If $r = 0$ continue at C, otherwise at D. ([18])	$\pi' \to B$ $M \bmod m \to r$ $\{[(1 \dot{-} r) \cdot C] + [(1 \cap r) \cdot D]\} \to \pi$
C	m is the result. Stop.	$\pi' \to C$ $m \to ?$ $\Omega \to \pi$
D	Rename m as M, and r as m; continue at B.	$\pi' \to D$ $m \to M$ $r \to m$ $B \to \pi$

2) Generalization to n numbers Computing the greatest common divisor of n given numbers (see figure 3).

The description is much more complicated than the preceding one, not just because $n > 2$ but especially because it must be valid for any value of n (in practice we assume $n \leq 800$). The envisaged method can be summarized in few words: given a known n, let $(a_n, a_{n-1}, \ldots, a_i, \ldots, a_2, a_1)$ be the set of given numbers. Now let m be the smallest of the a_i and r_i be the remainder of the division of a_i by m. Consider the set of $h \leq n$ numbers $(r_1, r_2, \ldots, r_k, \ldots, r_h = m)$ obtained from the r_i while adding m and discarding all those remainders that are zero. Continue the calculation while treating the r_k as previously the a_i, and so on, until the moment when $h = 1$. Then m is the number sought.

We will now write the detailed description of the mechanical calculation. One will note how much of the description just sketched is implicit, even from a logical or algebraic point of view; furthermore one notes that the sets of numbers traversed by the indices i and k cannot, in this particular case, be disjoint since at a given instant the r_k literally take the place of the a_i. The number z (see convention 3.32) has the value 100 below.

[18] The expression to the left of the $\to$ sign, while being comprehensible, does not obey convention 3.3 on page 20. (One ought to write five lines as in (2.7)). Since the entire chapter 5 is dedicated to demonstrating that an expression with parentheses can be given directly to the computer, we hope the reader will pardon us this foresight for the benefit of clarity.

A Let n be given and put $i = n + 1$.

$$\pi' \to A$$
$$? \to n$$
$$n + 101 \to i$$
$$B \to \pi$$

B Decrement i; let a_i be given. If $i = 1$ continue with C, else repeat B.

$$\pi' \to B$$
$$i - 1 \to i$$
$$? \to \downarrow i$$
$$[(1 \dot{-} (i \dot{-} 101)) \cdot C] + [(1 \cap (i \dot{-} 101)) \cdot B] \to \pi$$

C h is initially equal to n. Let m be a_i.

$$\pi' \to C$$
$$100 + n \to h$$
$$\downarrow i \to m$$
$$D \to \pi$$

D Increment i. Let m be the smallest of a_i and m. If $i = h$ continue with E, else repeat D.

$$\pi' \to D$$
$$i + 1 \to i$$
$$\downarrow i \cap m \to m$$
$$\{[(1 \dot{-} (h \dot{-} i)) \cdot E] + [(1 \cap (h \dot{-} i)) \cdot D]\} \to \pi$$

E Put $i = 0$ and $k = 1$.

$$\pi' \to E$$
$$100 \to i$$
$$101 \to k$$
$$F \to \pi$$

F Increment i. Compute the remainder r of the division of a_i by m. If $r = 0$ continue at G, else at H.

$$\pi' \to F$$
$$i + 1 \to i$$
$$\downarrow i \bmod m \to r$$
$$\{[(1 \dot{-} r) \cdot G] + [(1 \cap r) \cdot H]\} \to \pi$$

G If $i = h$ continue at I, else restart at F.

$$\pi' \to G$$
$$\{[(1 \dot{-} (h \dot{-} i)) \cdot I] + [(1 \cap (h \dot{-} i)) \cdot F]\} \to \pi$$

H Put the value of r at index k. Increment k. Restart at G.

$$\pi' \to H$$
$$r \to \downarrow k$$
$$k + 1 \to k$$
$$G \to \pi$$

I Make k equal to h. Put $m = r_h$. Put $i = 1$ and make m equal to r_1. If $h = 1$ continue at L; else at D and rename the $(a_1, a_2, \ldots, a_i, \ldots, a_h)$ as $(r_1, r_2, \ldots, r_k, \ldots, r_h)$.

$$\pi' \to I$$
$$k \to h$$
$$m \to \downarrow h$$
$$101 \to i$$
$$\downarrow i \to m$$
$$\{[(1 \dot{-} (h \dot{-} 101)) \cdot L] + [(1 \cap (h \dot{-} 101)) \cdot D]\} \to \pi$$

L m is the result. Stop.

$$\pi' \to L$$
$$m \to ?$$
$$\Omega \to \pi$$

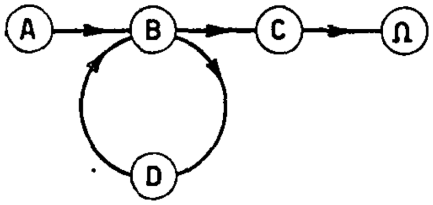

Figure 2: Graph of program for Euclid's algorithm ($n = 2$).

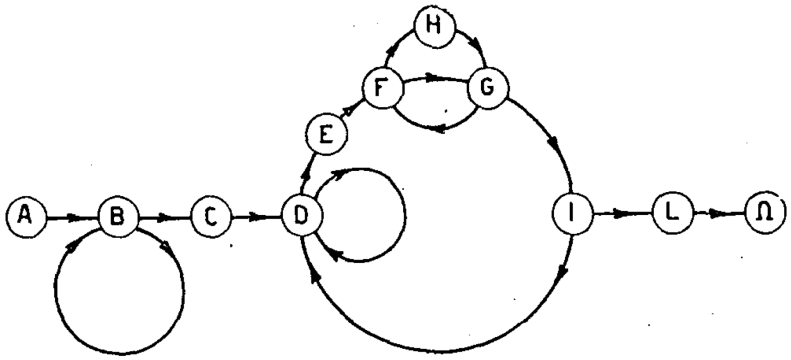

Figure 3: Graph of program for Euclid's algorithm (arbitrary n).

4. Principle of automatic programming

4.1. Input program

Let us try to answer the following question:

After having expressed the description of a numerical method for solving a given problem in the notation whose formal rules were established in the preceding chapter, let us perform the inverse of the notational transformation given in chapter 2. We thus obtain a sequence of instruction numbers.

By which procedure can this sequence of numbers be placed in the computer's internal memory so as to constitute the final calculation program for solving the given problem?

Note that it is only if this procedure exists and is general that we can claim to have solved (at least in principle) the problem of automatic programming. In this case, effectively, every program would be expressible by formulas that obviously do not keep count of the memory addresses at which instruction numbers must be stored.

Moreover, since the formulas can be established without reference to the particular properties of the computer used, apart from it being three-address

([19]) and universal, one would demonstrate at the same time the program's independence of the particular computer that must execute the computations.

But the reply to our question is relatively simple, if one fixes once and for all, as we shall do, the role played by the address π' and the addresses $A, B, \ldots, K, \ldots$ etc. as used in instruction numbers of the type $\pi' \to A$ etc., that we shall call "immediate instructions". We shall represent π' by a specific address, e.g. the address 999.

Let us suppose that the input tape of the computer contains, one after the other, N instruction numbers belonging to some program. Let $100 \leq \phi \leq 970$ be some address of the memory such that none of the N following addresses, in increasing order, coincide with any value that the program's indexes may assume. To begin with the contents of address 999 must be $\phi + 1$.

We imagine that at addresses 970 and the following ones are written the short input program described below, whose role is this:

Every instruction number that enters the machine is analysed: if it is an "immediate instruction" it is executed instantly; otherwise the instruction numbers are placed at consecutive addresses starting from $\phi + 1$.

α	The next instruction number on the tape is called a; if it is zero, stop; else continue at β. ([20])	$\pi'' \to \alpha$ $? \to a$ $(1 \dot{-} a) \cdot \Omega + (1 \cap a) \cdot \beta \to \pi$
β	Let b be the number formed by the 4 last digits of a. If the 5th–10th digits of a are 050999 continue at γ, else at δ.	$\pi'' \to \beta$ $a \bmod 10000 \to b$ $a \dot{-} b \to c$ $c + 9490010000 \to f$ $f \bmod 10000000000 \to d$ $(1 \dot{-} d) \cdot \gamma + (1 \cap d) \cdot \delta \to \pi$
γ	The number in π' must be transferred to the address designated by b; continue at α.	$\pi'' \to \gamma$ $\pi' \to \downarrow b$ $\alpha \to \pi$
δ	The number a must be transferred to the address designated by π'. Increment π'. Restart at α.	$\pi'' \to \delta$ $a \to \downarrow \pi'$ $\pi' + 1 \to \pi'$ $\alpha \to \pi$

By means of the device just described, the numbers written to addresses $A, B, \ldots, K, \ldots$ before the start of the computation will be precisely the addresses where the first instruction of the corresponding group is stored: this fact guarantees the correct execution of the program during *unconditional jumps* (e.g. $K \to \pi$) and even *conditional* ones.

[19] This restriction, however, is not indispensable for the developments that follow.

[20] For reasons of clarity, in this description we have written π'' instead of π' at the beginning of all groups (which are named by Greek letters).

4.2. Realization of the principle of automatic programming

Since our goals for "wholesale" programming have been reasonably attained, let us concentrate on the more restrained problem of making the formalism directly accessible to the computer. It is about constructing a kind of coding machine. In any case it suffices to have a teletypewriter with keys, each carrying one of the following indications:

Variables $a, b, \ldots, z, A, B, \ldots, Z, \downarrow a, ([21]) \downarrow b, \ldots, \downarrow z, \downarrow A, \downarrow B, \ldots, \downarrow Z, \pi, \pi', ?, \Omega.$
Operations $+, \cdot, \dot{-}, :, \to, \div, \mod, \cup, \cap.$
Parentheses $(,)$.

When pressing a key, the corresponding symbol is printed on a sheet of paper and at the same time recorded on punched paper tape or magnetic tape. To each key press corresponds a unique integral number fixed once and for all. Thus for each program there is a corresponding sequence of numbers. We consider this sequence the given data, and consider how to obtain another sequence of numbers, namely the sequence of instruction numbers for the given program. This is an arithmetic problem that can be solved by the computer itself, by applying a program for "automatic programming". On the other hand, since we must in any case have a program for encoding a formula of type $a \; op \; b \to c$, we seek to extract the maximal profit from this circumstance. We shall demonstrate in the following two chapters that one can automatically encode formulas of the type

$$(((a + b) \cdot d) : d) \to x$$

i.e. formed by an arbitrary (but meaningful) sequence of symbols comprising variables, operation signs, and parentheses. Similarly we shall automatically encode formulas of the type

$$a \dot{-} b \cdot c + d : f \cdot g \to x \qquad \text{where by } d : f \cdot g \text{ one understands } d \cdot f^{-1} \cdot g$$

i.e. representing a polynomial of multiple variables, in normal form, each monomial formed by a product of factors having an exponent equal to $+1$ or -1.

Moreover, we shall include in these programs the check that the symbol sequence of each formula has a meaning ([22]), to be able to report automatically whether, e.g., the two operation signs $+$: appear immediately after one another.

4.3. Correspondence between symbols and numbers

Between the symbol S and the number c that must be recorded on the input tape by the teletypewriter, we choose the following correspondence:

$$c = 5 \cdot k(S) + r$$

[21] To simplify the presentation, here and below we treat each $\downarrow a$, $\downarrow b$, etc., as if it were a single symbol and not two grouped together.

[22] Regarding the balance of parentheses and the succession of two symbols, see also [5].

and put, as we did already in 2.4:

k= 0,1,2,3,. . . ,27,28,. . . ,52,53,998,999,1002,1003,. . . ,1026,1027,. . . ,1052,1053
if
S= π,?,a,b,. . . , z, A,. . . , Y, Z, Ω, π', $\downarrow a$, $\downarrow b$, . . . , $\downarrow z$, $\downarrow A$, . . . , $\downarrow Y$, $\downarrow Z$

Furthermore,

$$k = c_{op} = 1, 2, 3, 4, 5, 6, 7, 8, 9,$$

if respectively

$$S = +, \cdot, \dot{-}, :, \rightarrow, \div, \text{mod}, \cup, \cap,$$

and

$$k = 6 \qquad \text{if} \qquad S = (,).$$

To obtain a one-to-one correspondence between c and S we encode the *kind* r of a symbol in this manner:

r	=	0,	1,	2,	3,	4
if						
S	=	),	(,	$\rightarrow$,	variable,	operation

4.4. Automatic encoding of a sequence of binary operations

Let us undertake to automatically encode some arbitrary program, e.g. the following sequence of symbols (see also 2.7):

$$h \cap a \rightarrow A \quad h \dot{-} a \rightarrow B \quad A \cdot x \rightarrow C \quad B \cdot y \rightarrow D \quad C + D \rightarrow u \quad u \rightarrow \pi$$

so as to obtain the sequence of numbers denoted respectively in (1.42):

$$n_1, n_2, n_3, n_4, n_5, J$$

To reduce this first example of automatic programming to the essentials, let us abstract away the part of the program related to checking that the symbol sequence has a meaning. Also assume once and for all that

$$m_1 = 10^{10}, m_2 = 10^8, m_3 = 10^4$$

α	Examine the first symbol c. If $c = 0$ stop, else continue at β.	$\pi' \to \alpha$ $? \to c$ $c : 5 \to C$ $\{(1 \dot{-} c) \cdot \Omega + (1 \cap c) \cdot \beta\} \to \pi$
β	Examine the second symbol d. If it is $\to$, continue at γ, else at δ.	$\pi' \to \beta$ $? \to d$ $d : 5 \to D$ $[(1 \dot{-} (D \div 5)) \cdot \gamma] + [(1 \cap (D \div 5)) \cdot \delta] \to \pi$
γ	Let f be the third symbol. Calculate the instruction number (which is of type T, J, L or I) and start over at α.	$\pi' \to \gamma$ $? \to f$ $f : 5 \to F$ $5 \cdot m_2 + C \cdot m_3 + F \to ?$ $\alpha \to \pi$
δ	Examine the third symbol g and the fifth symbol j. Calculate the instruction number and start over at α.	$\pi' \to \delta$ $? \to g$ $? \to j$ $? \to j$ $g : 5 \to G$ $j : 5 \to J$ $C \cdot m_1 + D \cdot m_2 + F \cdot m_3 + J \to ?$ $\alpha \to \pi$

5. Automatic encoding of formulas with parentheses

5.1. A first solution to the problem. Discussion

For this problem Mr. RUTISHAUSER has already developed an iterative solution method based on the successive lowering of the level of parentheses [6]. In his formulas one distinguishes round parentheses as the innermost (level 1), square brackets (parentheses level 2), curly braces (parentheses level 3), etc.

One obtains an iterative solution (from the program's point of view) to the problem if one encodes first the operations contained in round parentheses, replacing such a parenthesis by the address of its intermediate result and lowering by 1 the level of the remaining parentheses. One repeats this process as many times as indicated by the maximal level of parentheses.

The brevity and simplicity of the program constitute the principal advantages of this method, which has nevertheless some drawbacks:

5.11. The necessity to input a complete formula to the computer before the encoding work can begin. In other words, all the data of the problem fill the internal memory at the same time, with corresponding reduction in the computing capacity of the digital machine.

5.12. The necessity to fully explore the same formula multiple times just to extract the few facts relevant to each iteration leads to a waste of time, observable in particular on slower machines.

5.13. The necessity to use different symbols for parentheses at different levels. Consequently, if one wants to use a keyboard to input the symbols, it must have keys with opening and closing parentheses for each level until a certain maximum, beyond which a formula cannot be input to the machine.

We have used, by contrast, a method for solving the problem in which the symbols constituting the formula are input one after the other, in the order in which they are written; following this method the computer begins the encoding when the second symbol has been input.

The capacity of the internal memory necessary for the data may thus be reduced to two consecutive symbols instead of the entire formula ([23]). The data are used as one goes along and there is no waste of time for retrieving them.

Lastly, we have found a method with which it is unnecessary to input the level of each parenthesis to the machine, the machine being able to automatically deduce a hierarchy between the parentheses from the structure of the formulas to which they belong.

5.2. Conventions to observe when writing formulas

5.21. All operations are binary operations.

5.22. From the point of view of introducing parentheses, the operations are treated indiscriminately as non-associative operations; e.g. one writes:

$$\begin{array}{rll} ((a+b)+c)\to x & \text{instead of} & a+b+c\to x \\ ((a\cdot b)+c)\to y & \text{instead of} & a\cdot b+c\to y \\ (((a+b)\cdot(c+d))\cdot(e\dot{-}f))\to z & \text{instead of} & (a+b)\cdot(c+d)\cdot(e\dot{-}f)\to z \end{array}$$

5.23. Every formula containing parentheses must begin with an opening parenthesis ([24]).

It is evident that with these conventions the nature of the operations play no role for the rules about placement of parentheses.

5.3. Solution principle

Given a formula of the form

(5.31) $$(((a\ op_1\ b)\ op_2\ (c\ op_3\ d))\ op_4\ ((f^0\ op_5\ g)\ op_6\ h))\to x$$

we must construct a sequence of instruction numbers corresponding, in the order shown below, to the formulas

[23] In the description that follows this possibility has not been exploited, because it implies a change to the input program, which we do not want to revisit.

[24] The two conventions (5.23) and (6.11) together facilitate the fusion of programs corresponding to the problems of chapters 5 and 6 into one.

$$
\begin{array}{ll}
 & f^0\ op_5\ g \ \rightarrow x_6 \\
 & x_6\ op_6\ h \ \rightarrow x_5 \\
 & c\ op_3\ d \ \rightarrow x_4 \\
(5.32) & a\ op_1\ b \ \rightarrow x_3 \\
 & x_3\ op_2\ x_4 \rightarrow x_2 \\
 & x_2\ op_4\ x_5 \rightarrow x_1 \\
 & x_1 \ \rightarrow\ x
\end{array}
$$

These formulas are obtained by successively eliminating parentheses starting from the innermost ones. The x_f are intermediate results of the computation. We note that the indices k, added to operations to distinguish them, are chosen in (5.31) in increasing order from the left. On the other hand, the indices f, which serve only to distinguish the intermediate results, may be chosen arbitrarily. However, we have chosen them so that they form a decreasing progression when the formulas are written in their final order (5.32). This numbering is justified by the fact that the f may be calculated by recurrence starting from the left in any formula of type (5.31), as a function of the arrangement of parentheses, as we shall show later.

5.4. Parenthesis numbering by the function $F(n)$

Let n be the be level of a parenthesis and $r(n)$ its "nature", defined by

$$
r(n) = \begin{cases} 0 & \text{if the } n\text{th parenthesis is closing} \\ 1 & \text{if the } n\text{th parenthesis is opening} \end{cases}
$$

Now define a function $F(n)$ by this recurrence:

$$
(5.41)\quad F(0) = 0 \qquad F(n) = \begin{cases} p+1 \text{ where } p = \sum_{i=1}^{n-1} r(i) & \text{if } r(n) = 1 \\ F\{Min[F^{-1}(F(n-1))] - 1\} & \text{if } r(n) = 0\ ([25]) \end{cases}
$$

In the chosen example (5.31) one would have

		(	(	(	)	(	)	)	(	(	)	)	)
n	=	1	2	3	4	5	6	7	8	9	10	11	12
$r(n)$	=	1	1	1	0	1	0	0	1	1	0	0	0
$F(n)$	=	1	2	3	2	4	2	1	5	6	5	1	0

The function $F(n)$ can furthermore be represented graphically as in figure 4 (still for the same example).

This figure is constructed from the parenthesis structure by linking with a closed curve corresponding parentheses at the beginning and end of the same expression. The successive numbering, from left to right, of the regions of the plane delimited by these curves is realized precisely by the function F (0 corresponds to the region containing the point at infinity).

[25] This formula reads as follows: if the nth parenthesis is closing, the value of $F(n)$ is that of $F(h-1)$ where h is the smallest argument such that $F(h)$ has the same value as $F(n-1)$.

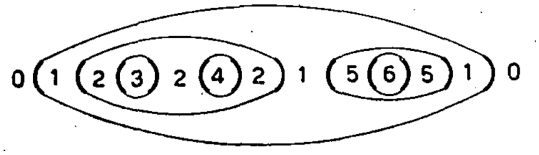

Figure 4: Parenthesis level nesting.

Now, if we want to automatically encode the formula (5.31) by exploring it from left to right, we cannot avoid successively obtaining these formulas:

$$
\begin{array}{ll}
a\ op_1\ b & \rightarrow x_3 \\
x_3\ op_2\ x_4 & \rightarrow x_2 \\
c\ op_3\ d & \rightarrow x_4 \\
x_2\ op_4\ x_5 & \rightarrow x_1 \\
f^0\ op_5\ g & \rightarrow x_6 \\
x_6\ op_6\ h & \rightarrow x_5 \\
x_1 & \rightarrow \ x
\end{array}
$$

which coincide with those of (5.32), except for their order. The indices f (of the x_f) thus constitute our principal unknowns. However, the relation between the index f and the function $F(n)$ is very simple, as shown by the following table.

	First operand	Second operand	Result
$)_n\ op\ \cdots$	$f = F(n-1)$	— —	— —
$)\ op\ (_n$	— —	$f = F(n)$	$f = F(n-1)$
$\cdots\ op\ \cdots)_n$	— —	— —	$f = F(n-1)$

The index n characterizes the last parenthesis input to the machine. The contents of the table can be summarized like this:

The value of f is the level of the region (figure 4) where we find the operation with this index.

5.5. Explicit calculation of $F(n)$

Since the second formula (5.41) is written in implicit form, let us transform it suitably. Note that if one puts, when $r(n) = 1$,

$$X(n) = F(n) = p + 1 \qquad \text{and} \qquad Y_{X(n)} = F(n-1)$$

it follows easily from (5.41) for the case where $r(n) = 0$ that

$$X(n) = F(n) = Y_{X(n-1)}$$

The explicit calculation of $F(n)$ thus becomes:

If $r(n) = 1$:		
Increment p	$p + 1 \to p$	
Old $F(n)$ is renamed $F(n-1)$	$X \to Y$	(see group E in chapter 7)
New $F(n)$ equals p (up to a constant m)	$m + p \to X$	
Put $Y = F(n-1)$ at index $X = F(n)$	$Y \to \downarrow X$	
If $r(n) = 0$:		
Old $F(n)$ is renamed $F(n-1)$	$X \to Y$	
New $F(n)$ is the quantity Y_X whose index is $X = F(n-1)$	$\downarrow X \to X$	(see group F in chapter 7)

5.6. The decision table

To allow for the automatic construction of a sequence similar to (5.32) from a formula of type (5.31), the program that we are going to devise must be flexible enough to account for any structure of the given symbol sequence. In principle, the decision taken by the computer after reading the ith symbol could depend on all the symbols read until that moment. But in fact, it turns out that apart from decisions related to parentheses, the decisions to be made by the computer depend directly only on the nature of the two last symbols read. This much simplifies the program that becomes iterative, while however requiring us to account for 25 possible variants, since each of the two symbols may be independently one of the following:), (, →, variable, operation. Many of these variants constitute a check ([26]). In the table below, each absurd variant provokes a STOP. The others carry an indication of the group to choose next.

(5.61) *Nature s of the second-last symbol* / *Nature r of the last symbol*

s \ r	)	(	→	var.	oper.
)	$G \to \pi$	$\Omega \to \pi$	$J \to \pi$	$\Omega \to \pi$	$T \to \pi$
(	$\Omega \to \pi$	$C \to \pi$	$\Omega \to \pi$	$P \to \pi$	$\Omega \to \pi$
→	$\Omega \to \pi$	$\Omega \to \pi$	$\Omega \to \pi$	$Q \to \pi$	$\Omega \to \pi$
var.	$H \to \pi$	$\Omega \to \pi$	$\Omega \to \pi$	$\Omega \to \pi$	$W \to \pi$
oper.	$\Omega \to \pi$	$I \to \pi$	$\Omega \to \pi$	$R \to \pi$	$\Omega \to \pi$

5.7. Auxiliary quantities

The complete description of the program consists of the groups $A, B, \ldots, T, U, W$ of chapter 7. The $\downarrow t, \downarrow t+1, \downarrow t+2, \ldots, \downarrow t+24$ are addresses reserved for containing the instructions corresponding to the elements $(r, s) \approx 5 \cdot r + s$ $(r, s = 0, 1, \ldots, 4)$ of the table (5.61).

[26] The check (see also [5]) concerns the immediate succession of two symbols, the balancing of parentheses ($F(n) = 0$ only at the end of the formula) and the 5.2 conventions. In the following we do all these checks except for check 5.22 whose realization would have considerably complicated the program. However, this last check can be carried out automatically, thanks to the following numerical interpretation: "The 5.22 convention is respected if the function $F(n)$ assumes the same value at most three times".

For the temporary recording of the instruction numbers in the memory, before their output in decreasing order (see group N), we have set aside the addresses $\downarrow m, \downarrow m+1, \downarrow m+2, \ldots$ etc., i.e. the same addresses that previously served to record the $F(n)$.

The groups $A', B', \ldots, M', N'$, on the other hand, belong to the program for automatic programming studied in the next chapter.

6. Automatic programming for polynomials in normal form

The prohibition (5.22) against using the associate property of product and sum when writing formulas containing parentheses, requires the use of more parentheses than usual. To reduce this inconvenience, we want to allow for a second way to write the formulas, in addition to the (5.31) model. Although we limit ourselves from now on to the four arithmetic operations, in principle nothing prevents us from envisioning normalized formulas containing other elementary operations. Let us make the convention, as in elementary algebra, to establish priority levels between the four operations, taken two and two:

$$(+, \dot{-}) < (\cdot, :)$$

where the sign $<$ here represents the expression "binds less than".

Thus a formula such as

$$a + b \cdot c + d : f \cdot g \to x$$

acquires a meaning and we are in a position to save some parentheses.

6.1. Conventions for writing formulas

6.11. If the first symbol is not an opening parenthesis (in which case one would be back to the problem of chapter 5) one must not use any parenthesis in the entire formula.

6.12. The only operations used are

$$\to, +, \dot{-}, \cdot, : \qquad \text{i.e. } k(op) \leq 5$$

6.2. Solution principle

Let us examine four prototypical polynomial expressions and the corresponding solutions. It is easy to see that it suffices to use an essentially iterative program where only two letters are used to designate the intermediate results: the letter X for the results of a multiplication or division ($\vdots$) and the letter S for the results of an addition or subtraction ($\pm$).

(6.2)

Given formula	*Resulting instruction numbers*	*Group designation*
$a_1 \vdots_1 b_1 \vdots_2 c_1 \to d_1$	$a_1 \to X$	G'
	$X \vdots_1 b_1 \to X$	H'
	$X \vdots_2 c_1 \to X$	H'
	$S + X \to S$	I'
	$S \to d_1$	N'
$a_2 \pm_1 b_2 \vdots_1 c_2 \to d_2$	$a_2 \to S$	F'
	$b_2 \to X$	M'
	$X \vdots_1 c_2 \to X$	H'
	$S \pm_1 X \to S$	I'
	$S \to d_2$	N'
$a_3 \pm_1 b_3 \pm_2 c_3 \to d_3$	$a_3 \to S$	F'
	$S \pm_1 b_3 \to S$	J'
	$S \pm_2 c_3 \to S$	J'
	$S \to d_3$	N'
$a_4 \vdots_1 b_4 \pm_1 c_4 \to d_4$	$a_4 \to X$	G'
	$X \vdots_1 b_4 \to X$	H'
	$S + X \to S$	I'
	$S \pm_1 c_4 \to S$	J'
	$S \to d_4$	N'

The parts of the program related to the groups

$$F', G', H', I', J', M', N'$$

are iterative with respect to the sequence of two consecutive operation symbols (separated by a variable, obviously), as indicated in the following table:

Second-last symbol read \ *Last symbol read*	$\vdots$	$\pm$	$\to$
$\vdots$	H'	$H'I'$	$H'I'N'$
$\pm$	M'	J'	$J'N'$

The first group of two symbols is treated separately according to this schema:

(The first symbol is a variable) \ *Second symbol*	$\to$	$\pm$	$\vdots$
	D'	F'	G'

One will note that in each program (6.2) there are two or three instructions not needed if the encoding was made by a human operator. Our method, however, becomes more and more advantageous as the length of formulas grows, since there is only roughly one superfluous instruction at the beginning and the end of each polynomial and one in each monomial.

6.3. Checking and auxiliary quantities

For an sequence of symbols to have a meaning, operation symbols must alternate with variable symbols.

In the following we shall use some auxiliary quantities called g_i $(i = 1, 2, \ldots, 6)$ that are incomplete instruction numbers, i.e. having zeroes instead of the symbols in boldface:

$$\begin{array}{rcr} g_1 & \approx & X\ \mathbf{op}\ \mathbf{V} \to X \\ g_2 & \approx & S\ \mathbf{op}\ X \to S \\ g_3 & \approx & \mathbf{V} \to X \end{array} \qquad \begin{array}{rcr} g_4 & \approx & S\ \mathbf{op}\ \mathbf{V} \to S \\ g_5 & \approx & S \to \mathbf{V} \\ g_6 & \approx & \mathbf{V} \to S \end{array}$$

The part not in boldface is constant, independent of the variables and the operations of the expression to encode. For instance, one has

$$g_1 = m_1 \cdot k(X) + k(X) = 10^{10} \cdot 51 + 51 = 0051\ 00\ 0000\ 0051$$

Remark: We do not need a special symbol to mark the end of a sequence of formulas. It suffices to use the closing parenthesis symbol, thanks to group A of the program which will automatically stop the computer.

7. Detailed automatic programming program

A	Analyse the first symbol c of a formula. If it is a) or a $\to$ or an operator symbol then STOP. If it is a variable, continue at A'; if it is a (continue at B.	$\pi' \to A$ $? \to c$ $c \bmod 5 \to r$ $[(r+1) \bmod 2] \cdot \Omega + [[1 \dot{-} (r \div 3)] \cdot A'] +$ $+[[1 \dot{-} (r \div 1)] \cdot B] \to \pi$
B	Put $Y = 0$ (up to a constant m), $p = 1$ and compute $F(1)$ (see 5.4). Continue at C.	$\pi' \to B$ $m \to Y$ $1 \to p$ $m + p \to X$ $Y \to \downarrow X$ $C \to \pi$
C	Analyse the next symbol; let s be the nature of the preceding symbol and r that of the current one. If it is a (continue at E. If it is a) continue at F, else at D.	$\pi' \to C$ $r \to s$ $? \to c$ $c : 5 \to k$ $c \bmod 5 \to r$ $[1 \cap (r \dot{-} 1)] \cdot D + (1 \cap r) \cdot E + (1 \dot{-} r) \cdot F \to \pi$

D	The nature of the two last symbols contribute to determining the next group to choose (see table (5.61)).	$\pi' \to D$ $5 \cdot r + s + t \to \pi$
E	Determine the value of function $F(n)$ for an opening parenthesis (see explanation in 5.5). Continue at D.	$\pi' \to E$ $p + 1 \to p$ $X \to Y$ $m + p \to X$ $Y \to \downarrow X$ $D \to \pi$
F	Determine the value of function $F(n)$ for a closing parenthesis (see explanation in 5.5). Continue at D.	$\pi' \to F$ $X \to Y$ $\downarrow X \to X$ $D \to \pi$
G	If the number of closing parentheses already counted exceeds that of opening parentheses then STOP, else continue at C.	$\pi' \to G$ $[[1 \dot{-} (Y \div m)] \cdot \Omega] + [[1 \cap (Y \div m)] \cdot C] \to \pi$
H	Determine the last 4 digits (11th–14th) of the instruction number and save it at an address computed from the level of the last parentheses analysed. Continue at C.	$\pi' \to H$ $n + Y \to \downarrow Y$ $C \to \pi$
I	Determine the last 8 digits (7th–14th) of the instruction number and save it as under H. Continue at C.	$\pi' \to I$ $m_3 \cdot X + Y + n \to \downarrow Y$ $C \to \pi$

J If the number of closing parentheses is not equal to that of opening parentheses then STOP, else continue at M.

$$\pi' \to J$$
$$[1 \cap (X \div m)] \cdot \Omega + [1 \dot{-} (X \div m)] \cdot M \to \pi$$

M The formula's last instruction number (which is a store) is calculated up to the 11th digit, inclusive. The temporary address of last instruction number is called L. Continue at N.

$$\pi' \to M$$
$$5 \cdot m_2 + m_3 \cdot Y \to n$$
$$m + p \to L$$
$$N \to \pi$$

N The number at address L is recorded on the output tape. Decrement L. Restart at N until all the calculated numbers have been output ($L = 0 = p$ up to the constant m). Continue at C.

$$\pi' \to N$$
$$\downarrow L \to ?$$
$$L \dot{-} 1 \to L$$
$$[1 \dot{-} (L \dot{-} m)] \cdot C + [1 \cap (L \dot{-} m)] \cdot N \to \pi$$

P Determine the 4 first digits of the instruction number; continue at C.

$$\pi' \to P$$
$$k \cdot m_1 \to n$$
$$C \to \pi$$

Q The last instruction number of the formula with parentheses is completed until its 14th digit and recorded on the output tape. Continue at A.

$$\pi' \to Q$$
$$k + n \to ?$$
$$A \to \pi$$

R Determine the 7th–11th digits of the instruction number. Continue at C.

$$\pi' \to R$$
$$k \cdot m_3 + n \to n$$
$$C \to \pi$$

T Check the balancing of parentheses as under G. If the balance is correct, continue at U.

$$\pi' \to T$$
$$[[1 \dot{-} (Y \div m)] \cdot \Omega] + [[1 \cap (Y \div m)] \cdot U] \to \pi$$

U Determine the first 6 digits of the instruction number. Continue at C.

$$\pi' \to U$$
$$Y \cdot m_1 + k \cdot m_2 \to n$$
$$C \to \pi$$

A' The second symbol of a formula without parentheses is analysed. If it is a variable, a parenthesis or an operation other than $+, \dot{-}, \cdot, :, \to$ then STOP. If it is $\to$ continue at D'; if it is $+$ or $\dot{-}$ continue at F'; if it is $\cdot$ or $:$ continue at G'.

$$\pi' \to A'$$
$$c : 5 \to h$$
$$? \to c$$
$$c : 5 \to k$$
$$c \bmod 5 \to r$$
$$[[1 \cap (k \dot{-} 5)] + (1 \dot{-} (k \dot{-} 5)) \cdot (1 \dot{-} (r \div 3))] \cdot$$
$$\cdot \Omega + [1 \dot{-} (r \div 2)] \cdot D' + [1 \dot{-} (r \div 4)] \cdot$$
$$\cdot [k \bmod 2] \cdot F' + [1 \dot{-} (r \div 4)] \cdot$$
$$\cdot [(k + 1) \bmod 2] \cdot G' \to \pi$$

B' Another symbol is analysed. If it is not a variable then STOP. If the second last symbol analysed is $\to$ then continue at N', else at C'.

$$\pi' \to B'$$
$$k \to j$$
$$r \to q$$
$$? \to c$$
$$c : 5 \to k$$
$$c \bmod 5 \to r$$
$$[1 \cap (r \div 3)] \cdot \Omega + [1 \dot{-} (r \div 3)] \cdot$$
$$\cdot \{[1 \dot{-} (q \div 2)] \cdot N' + [1 \cap (q \div 2)] \cdot C'\} \to \pi$$

C' Another symbol is analysed. If it is a parenthesis or a variable or an operation other than $+, \dot{-}, \cdot, :, \to$ then STOP. If the second last symbol is $\cdot$ or $:$ continue at H'. If the second last is $+$ or $\dot{-}$ continue at M', else at J'.

$$\pi' \to C'$$
$$k \to h$$
$$? \to c$$
$$c : 5 \to k$$
$$c \bmod 5 \to r$$
$$\{[1 \cap (k \dot{-} 5)] + [1 \dot{-} (k \dot{-} 5)] \cdot [1 \dot{-} (r \div 3)]\} \cdot$$
$$\cdot \Omega + [(1 + j) \bmod 2] \cdot H' + (j \bmod 2) \cdot$$
$$\cdot \{[(k + 1) \bmod 2] \cdot M' + (k \bmod 2) \cdot J'\} \to \pi$$

D'	The third symbol is analysed. If it is not a variable then STOP, else continue at E'.	$\pi' \to D'$ $? \to c$ $c : 5 \to k$ $c \bmod 5 \to r$ $[1 \cap (r \div 3)] \cdot \Omega + [1 \dot{-} (r \div 3)] \cdot E' \to \pi$
E'	Computing the instruction number for a processed formula that is a store. Continue at A.	$\pi' \to E'$ $h \cdot m_3 + k \to ?$ $A \to \pi$
F'	Determination and output of the first instruction number of type g_6. Continue at B'.	$\pi' \to F'$ $h \cdot m_2 + g_6 \to ?$ $B' \to \pi$
G'	Determination and output of the first instruction number which is $0 \to S$ and of the second instruction number which is of type g_3. Continue at B'.	$\pi' \to G'$ $998 \cdot m_3 + g_6 \to ?$ $h \cdot m_3 + g_3 \to ?$ $1 \to g$ $B' \to \pi$
H'	Determination and output of an instruction number of type g_1. If the last symbol analysed is $\cdot$ or : continue at B'; else at I'.	$\pi' \to H'$ $g_1 + j \cdot m_2 + h \cdot m_3 \to ?$ $(k \bmod 2) \cdot I' + [(k+1) \bmod 2] \cdot B' \to \pi$
I'	Determination and output of an instruction number of type g_2. Continue at B'.	$\pi' \to I'$ $g \cdot m_2 + g_2 \to ?$ $B' \to \pi$
J'	Determination and output of an instruction number of type g_4. Continue at B'.	$\pi' \to J'$ $j \cdot m_2 + h \cdot m_3 + g_4 \to ?$ $B' \to \pi$

M'	Determination and output of an instruction number of type g_3. Continue at B'.	$\pi' \to M'$ $h \cdot m_3 + g_3 \to ?$ $B' \to \pi$
N'	Determination and output of the last instruction number of the formula without parentheses (which is a store of type g_5). Continue at A.	$\pi' \to N'$ $k + g_5 \to ?$ $A \to \pi$

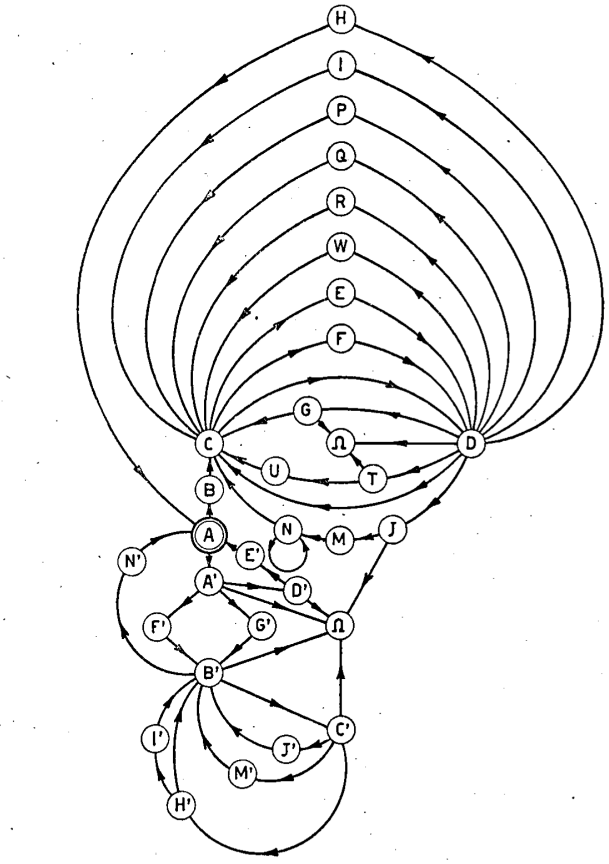

Figure 5: Graph of the program for automatic programming.

8. Relations to logic

Among the instructions that we have chosen in 1.3, there is a certain logical redundancy. For instance, each of the four operations $\dot{-}, \div, \cup, \cap$ could be expressed in terms of a single one of these and addition, as shown by these reductions:

$$\begin{array}{rcl} a \div b & \equiv & (a \dot{-} b) + (b \dot{-} a) \\ a \cap b & \equiv & a \dot{-} (a \dot{-} b) \\ a \cup b & \equiv & b + (a \dot{-} b) \end{array}$$

Likewise, the operations $\cdot$ and : could easily be realized by subroutines comprising only the operations $+$ and $\dot{-}$. On the other hand, negative and fractional numbers could also be represented, in multiple ways even, by pairs of non-negative integers, so the fact that we have limited ourselves to the latter does not cause any loss of generality.

In the program treated in the preceding chapter, and in particular at the end of groups A', B' and C', the choice criteria are quite complicated. To facilitate the writing of formulas we have often appealed to certain one-to-one correspondences that one can establish between the fundamental logical operations of *negation*, *conjunction* and *disjunction* on the one hand, and the operations $\dot{-}$, $+$ and $\cdot$ on the other.

In fact, if one designates by P and Q some propositions, by p and q non-negative integer values, and by r' and s numbers that can have only the values 0 and 1, we can establish the following table (where the logical notations are those of [11]):

P, Q true	$p, q = 0$	$r', s = 1$
P, Q false	$p, q \neq 0$	$r', s = 0$
$\overline{P}$	$1 \dot{-} p$	$1 \dot{-} r'$
$\overline{\overline{P}}$	$1 \dot{-} (1 \dot{-} p) = p$	$r' = 1 \cap p$
$P \& Q$	$p + q$	$r' \cdot s$
$P \vee Q$	$p \cdot q$	$1 \cap (r' + s)$

To demonstrate an application of the correspondence between the first and third column of the preceding table, let us consider, e.g. the choice criteria at the end of group B', chapter 7.

Let us name the coefficients of Ω, N' and C' respectively ω, n' and c'.

The first criterion requires that $\omega = 1$ if and only if $r \neq 3$, from which we get:

$$\omega = 1 \cap (r \div 3)$$

The second criterion requires that $n' = 1$ *if not* $r \neq 3$ *and* at the same time *not* $q \neq 2$, from which we get:

$$n' = [1 \dot{-} (r \div 3)] \cdot [1 \dot{-} (q \div 2)]$$

The third criterion requires that $c' = 1$ *if not* $r \neq 3$ *and* at the same time $q \neq 2$, from which we get:

$$c' = [1 \dot{-} (r \div 3)] \cdot [1 \cap (q \div 2)]$$

Bibliography

[1] E. C. Berkeley, *Giant Brains or Machines that Think*, John Wiley & Sons, New York, (1949).

[2] H. Rutishauser, A. Speiser, E. Stiefel, *Programmgesteuerte digitale Rechengeräte (elektronische Rechenmaschinen)*, Mitt. Nr. 2, Inst. f. ang. Math., ETH Zurich, (1951).

[3] M. V. Wilkes, *Report on the Preparation of Programmes for EDSAC and the use of library of sub-routines*, University Math Laboratory Cambridge, (Sept. 1950).

[4] J. Von Neumann, *The general and logical theory of automata*, p. 1–41 of: *Cerebral Mechanisms in Behavior.* The Hixon Symposium, John Wiley & Sons, New York, (1951).

[5] K. Zuse, *Über den allgemeinen Plankalkül als Mittel zur Formulierung schematisch-kombinativer Aufgaben*, Arch. Math., 1, 441–449, (1948-49).

[6] H. Rutishauser, *Über automatische Rechenplanfertigung bei Programmgesteurten Rechenmaschinen*, Mitt. Nr. 3, Inst. f. ang. Math., ETH Zurich, (1952).

[7] A. M. Turing, *Computable numbers, with an application to the Entscheidungsproblem*, Proc. London Math. Soc., 2, s. 42, (1937), p. 230–265.

[8] A. P. Speiser, *Entwurf eines elektronischen Rechengerätes*, Mitt. Nr. 1, Inst. f. ang. Math., ETH Zurich, (1950).

[9] D. Koenig, *Theorie der endlichen und unendlichen Graphen*, Leipzig, (1936).

[10] H. H. Goldstine, J. Von Neumann, *Planning and Coding for an Electronic Computing Instrument*, 3 vol., Institute for Advanced Study, Princeton, N. Y., (1947-48).

[11] D. Hilbert, W. Ackermann, *Grundzüge der theoretischen Logik*, Springer-V., Berlin, (1949).

Summaries in Italian, English and German

(The original's summaries in Italian, English and German have been left out).

Curriculum vitae

Corrado Böhm, Italian citizen, born 17 January 1923. In 1941 he obtained his secondary school degree from the scientific high school "Vittorio Veneto" in Milan.

Having entered the École Polytechnique de l'Université de Lausanne (Switzerland) in 1942 he left it in 1946 with a degree in electrical engineering.

He got employed in 1947 at the École Polytechnique Federale de Zurich as assistant to Mr. R. Dubs (professor in hydraulics and hydraulic machines) for three semesters and for three more as assistant to Mr. E. Stiefel (professor of geometry and head of the department of applied mathematics).

During this period, while deepening his knowledge of mathematics, he followed a specialization course at IBM on punched card machines. Subsequently, he familiarized himself with the similar Bull machines.

In 1949 he was sent to Neukirchen (Germany) to study on location the relay machine constructed by Mr. Zuse, a machine that was adopted by the École Polytechnique itself.

Using these machines he solved also various mathematical problems: Tabulation of functions, computations of differences, computations of residues, multiplication of the elements of finite groups, etc.

Translator's appendices

A.1. Notes on Böhm's compiler

A program to be compiled is structured as a set of labelled groups (section 3.2), each consisting of a sequence of assignment formulas. The last instruction of a group is always an assignment to the program counter π (and an assignment to π can appear only as the last formula in a group). A "group" is what is today called a basic block: it can be entered only at its beginning and can be exited only at the end. The first formula in a group K is an assignment of the form $K \to \pi'$, which serves only to name the group at load-time, thus called an "immediate instruction" (see below); it is not executed at run-time. The group labels S and V are not used in the chapter 7 compiler since they are used in the translation of polynomials (chapter 6). Similarly, X and Y are used in the tabulation of F (section 5.5).

The compiler translates one assignment formula at a time, and works in a single pass without backpatching of jumps to those labels (group names) not yet encountered. This is possible because all jumps in the generated code are indirect, via the code addresses stored under group labels $A, B, \ldots$ during run-time. Defining these variables at load-time is one of the loader's jobs; see the next section and Figure 6.

The compiler in chapter 7 does not handle the operations $\div, \cup, \cap$ because they can be encoded using the other operations as shown in chapter 8.

The machine and the language do not support numeric constants, so all constants must be loaded from a "constants prefix" of the input tape into suitable memory addresses before the program is run; see below. Also, a symbolic program to be compiled must be preprocessed to replace each pseudo-constant such as *1* (written in italics as per section 3.3) by a memory address such as 997.

A.2. Notes on Böhm's loader

The group labels used in the compiler's (symbolic) input code and its (numeric) output code become defined only at load-time, when the loader's group β in section 4.1 recognizes a group header $K \to \pi'$: the condition "If the 5th–10th digits of a are 050999" is a numerical test for the instruction in a being a group header, that is, having form $\pi' \to K$ for some K, since 05 is the instruction code for assignment and π' is at address 999. Subsequently the loader's group γ stores the next instruction address in the variable at address K.

The loader or "input program" in section 4.1 serves the same purpose as Wilkes's "initial orders" for the Cambridge EDSAC, but does not have to perform jump address patching because all jumps are indirect.

The pseudo-code for the loader uses the constants *1*, *10000*, *9490010000* and *10000000000* so these must be allocated to memory addresses, for instance 997 through 994. They must be read from a constants prefix of the input tape, before the numeric program instructions to be loaded. To read the constants prefix the loader code should start with a loop that reads pairs (a, n) from the

input tape and stores constant n at address a, until it reads the address $a = 0$. Using label ρ for this constant prefix reading loop it can be written like this, (ab)using the Ω location 998 to store the address a:

$$\begin{aligned} \pi'' &\to \rho \\ ? &\to \Omega \\ ? &\to \downarrow \Omega \\ \rho &\to \pi \end{aligned}$$

When $\Omega = a = 0$ the second-last statement becomes a jump to n, since the program counter π is in memory address 0. If we let the constants prefix end with two lines containing 0 and α, the loop will terminate with a jump to the loader's first group α as desired. Also, this will leave 0 in Ω as per Böhm's conventions.

The pseudo-code for the loader given by Böhm in section 4.1 uses label variables α, β, γ, δ, the variable π'' and the $\downarrow \pi'$ combination, none of which can be represented in Böhm's section 4.3 character encoding. But in any case, the loader must exist in absolute numeric code form; relying on the loader to load itself would lead to infinite regress. We may decide to store the values of α, β, γ, δ and ρ (which are group code addresses) in memory addresses 993 through 989 and read their absolute values from the constants prefix using the ρ constant reader already given above.

To accommodate the loader's constant prefix reader and make room for the constants, the loader's first instruction (at address ρ) must go into address 966, with α in 969. This is akin to Böhm's suggestion (section 4.1) that α is at address 970, but he may have had a different detailed arrangement in mind.

B. An implementation of Böhm's compiler and loader

We have made a simple implementation in Java of Böhm's machine, and his compiler and loader, as well as a separate program to convert a source program in Unicode into a stream of the character codes (section 4.3) accepted by Böhm's compiler. See Figure 6.

It will be made available at http://www.itu.dk/people/sestoft/boehmthesis/ We distinguish these types of text files:

- `*.cb`: Böhm absolute numeric code file, one decimal instruction number per line. The first line contains the memory address of the first instruction; each following line contains a 14-digit instruction number. This is the raw format read and executed by the Machine.java simulator.

- `*.cbs`: Böhm symbolic code file, in Unicode, using the conventions of the dissertation. This format is input to the Converter.java preprocessor.

- `*.cbc`: Böhm symbolic character file, one character code per line, using the character encoding described in section 4.3. This format is output by the Converter.java preprocessor, and input by the compiler (chapter 7).

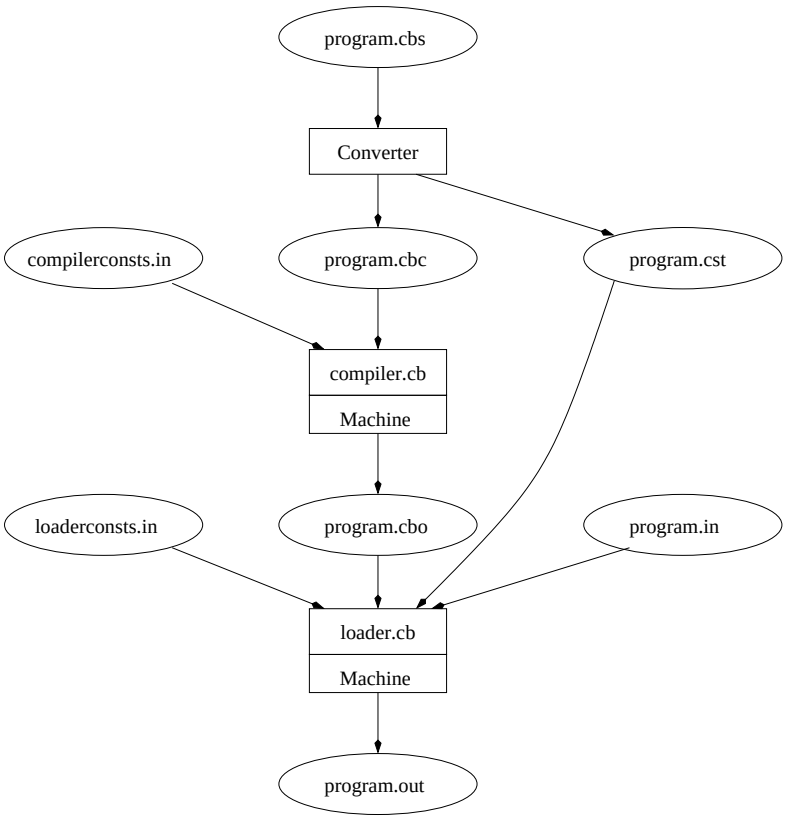

Figure 6: A source program program.cbs (in symbolic relative code) is first converted to a sequence of character encodings program.cbc (according to the conventions of section 4.3) and also a "constants prefix" tape program.cst containing the program's constants. Then the character sequence is compiled, using Böhm's compiler in chapter 7, into a sequence of relative numeric instruction codes in program.cbo. Then this program is loaded into memory, using Böhm's loader program in section 4.1, and finally the loaded program is executed. The compilerconsts.in and loaderconsts.in files contain all numeric constants needed by the programs; these are loaded into memory cells (by suitable header code, see Section) before the program proper executes. All file names used here are inventions of the translator; when Böhm wrote his dissertation there were no file systems and no file names, only input and output tapes.

- `*.cbo`: Böhm relative numeric code file, one instruction or "immediate instruction" (group header) per line, using the conventions of the dissertation. This is the format output by the Boehm compiler and input by the Boehm loader.
- `*.cst`: Constant prefix file containing pairs (a, n) of a numeric constant n and its address a in memory.
- `*.in`: Input file, possibly prefixed by the program's constant prefix file.