Table of Contents

translation by Bill Faught

Introduction

For the numerical treatment of a mathematical problem on program-directed digital computers, the machine code, that is, the collection of commands for the execution of the individual calculation steps, must either be fed to the machine through a punched tape, or else already be available in core memory.

It is known that machine code generation, that is, the determination of such machine code for a particular problem, is often a considerable task, and in extreme cases a significant part of the whole expenditure can fall to the preparation task. Thus tools have been constructed for the simplification of code generation, such as the "Coding Machine" for the Mark III (See [1] 1/). K. Zuse made far-reaching beginnings in the automation of code generation in his "Allgemeinen Plankalkül" (General code-calculator) [7], which, however, also required the construction of special hardware.

On the other hand, the author of this work acquired the conviction long ago that it must be possible, owing to their versatility, to utilize program-directed computers themselves as code generators. This would therefore mean that one would not only solve numerical problems with these calculating machines, but also "calculate" machine code.

Although the methods to be described could also be implemented on already existing machines, it is advantageous for the formalization of the procedure to use as a basis for our discussion the following account of a particular, although only projected machine, whose mathematical characteristics, as required in the following, are given in §1.

1/ Square brackets refer to the Bibliography.

The concepts and conventions used in this work will not particularly be explained further; rather, reference should be made to previously published literature, in particular [4] and [5].

Finally, it should be mentioned that the author presented certain parts of this work at the GaMM Conference in Freiburg in Breisgau (see also [9]).

1. Brief Description of a Program-directed Computer [2]

1.1 Number system, number storage

The representation of numbers in the decimal system in scientific notation is provided for: $x = a \times 10^b$. The exponent b can vary between -31 and +31. The factor a , which satisfies the additional condition $|a| < 10$, has 9 decimal places after the decimal point; the single digits are dual encoded (after Aiken [3]). Thus a number requires 48 bits altogether for its representation, namely one for the sign, 40 for the factor a , 6 for the exponent b and one more for a special sign (see §1.4).

It should be stressed here that $|a|$ can also be smaller than 1 , for in this work, as in BARK [8], no normalization[4] follows subtraction; instead, normalization occurs immediately before multiplication and division. This fact makes it possible among other things to determine the nearest integer to a given number x through the addition of $0.000\ 000\ 000 \times 10^9$.

To receive intermediate results, a memory with 1000 memory cells (numbered from 0 to 999) is provided. In reference to this memory, several abbreviations will be used, namely:

---

[2] This description is drawn from a project which is currently being completed at the Institute for Applied Mathematics of the ETH under the direction of Dr. E. Stiefel.

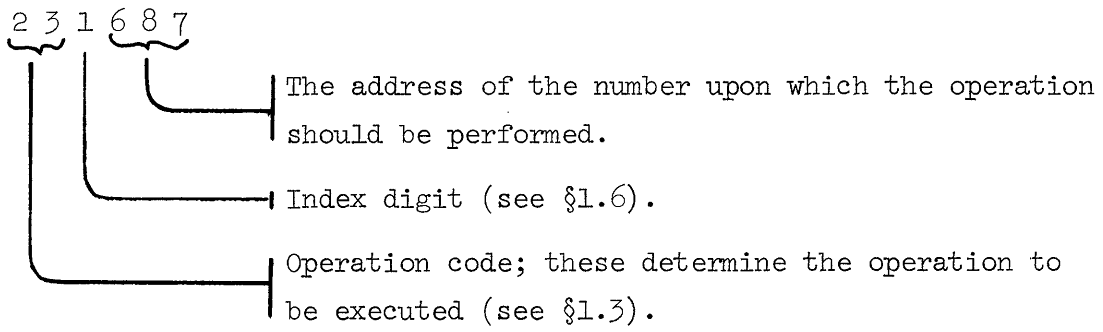
[3] The digit z is represented by the dual-number $3+z+3 \times \mathrm{sgn}(z-4.5)$ (see also [1], [4]).

[4] By normalization, as in most other machines of this type, we mean the customary decimal point alignment of a and a corresponding reduction of b so that $1 \leq |a| < 10$.

(n)	for the number in memory cell n
<x>	for the address of the number x , that is, for the number of the cell in which x is stored
$x \to n$	for: "Store the number x in the cell n ."

1.2 Instructions

The planned machine is a one-address machine; every command is represented by a 6-place number, whose digits have the following meaning:



The address of the number upon which the operation should be performed.

Index digit (see §1.6).

Operation code; these determine the operation to be executed (see §1.3).

The instructions are stored in the machine in a special instruction memory with 1000 memory cells which can store 2000 one-address instructions (2 instructions per cell). The instructions are executed in the order in which they are stored; only through the so-called jump instruction (see §1.5) is the normal execution sequence interrupted.

1.3 The organization of the arithmetic unit

When the machine is stopped or between 2 arithmetic operations a number is always in the so-called operand register Op which can later be used, according to circumstances, as an augend, minuend, multiplicand, or dividend.[5]

For the execution of an arithmetic operation, the second operand is transferred from the memory to the arithmetic unit by the corresponding command and the designated operation immediately executed. The result itself goes again to Op, from where it can be stored into memory anytime; it so doing however, Op is not obliterated. A special command An serves to insert a new first operand in Op and destroy the old contents (see the table on the following page). Through this organization we especially avoid the necessity of making one of the two factors of a multiplication available through a special command.

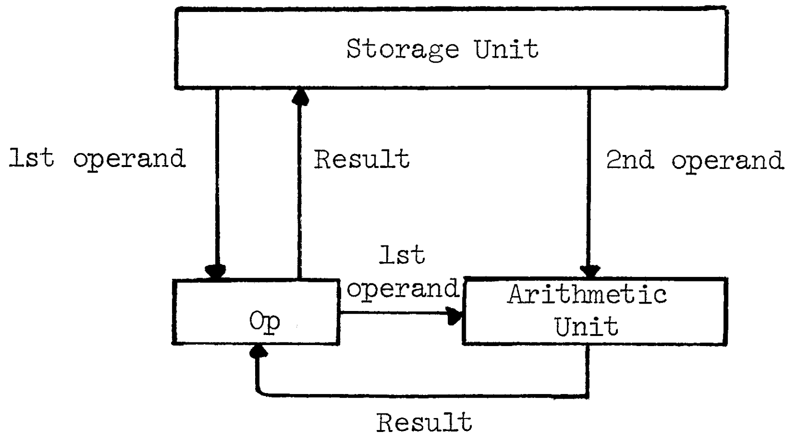


Figure 1: Schematic representation of the arithmetic unit

[5] This kind of organization can be found essentially in calculating machine Z4 by K. Zuse, but otherwise does not appear to be too common.

List of the arithmetic and memory commands

Notation of the operation	Opcode	Address	Effect of the operation	Time in ms
'void op'	00	irrelevant		40
A	01	n	(n) $\rightarrow$ Op	40
+	02	n	(Op)+(n) $\rightarrow$ Op	40
-	03	n	(Op)-(n) $\rightarrow$ Op	40
x	06	n	(Op)x(n) $\rightarrow$ Op	160
/	07	n	(Op)/(n)	560
Maj	08	n	Max[(Op),(n)] $\rightarrow$ Op	40
Min	09	n	Min[(Op),(n)] $\rightarrow$ Op	40
Sgn	10	irrelevant	Sgn(Op) $\rightarrow$ Op	40
$\|x\|$	11	irrelevant	$\|$(Op)$\|$ $\rightarrow$ Op	40
-1	12	irrelevamt	-(Op) $\rightarrow$ Op	40
N	19	irrelevant	(Op) is normalized	40
S	20	n	(Op) $\rightarrow$ n	40

1.4 Input and output; the number Q

The insertion of the initial values of a problem in the machine results from a punched tape, which is hand-punched beforehand. On this punched tape the numbers to be inserted are connected together according to mathematical principles by grouping (so-called blocks). The individual blocks can be of arbitrary and mixed lengths. The end of such a block, and with it the automatic beginning of the next, is characterized by a special symbol Q , which is also considered a number and is stored with the data in the machine. One such Q stands at the beginning of the punched tape; furthermore one denotes the end itself appropriately by two or more successive Q's . Thus, a number tape appears somewhat like the following:

```
Q..........Q......Q...        ...Q..........QQ
 1st block  2nd block           last block
```

This number Q can be stored and one can also calculate with it: The result of any operation is exactly Q when at least one of the operands is Q . One distinguishes Q from the other numbers by the 48-th bit, which in all normal numbers is zero and in Q is 1 .

If a computer tape is placed in one of the read stations of the computer, the tape runs automatically up to the next Q , so that the following number is ready to be read; the first Q does not go into the machine. The actual reading of the number and transmission itself into the arithmetic unit results from an appropriate command, which we won't need in the following.

One also inserts the machine language program on a punched tape into the machine (so-called instruction tape) before beginning execution; in this case the machine always reads 2 instructions together to fill one program memory cell.

It is the goal of this work, as already mentioned, to use the machine to calculate machine code; for this it is useful to punch the calculated instructions at once. To do this is a special command:

LB : Punch from the number $a \times 10^b$ in Op, without regard to the exponent, the last 6 digits in a .

For further instructions for input and output of numbers, instructions to punch all 48 bits of a number, to accept numbers from a keyboard, and to print results with a printer connected to the computer are naturally provided.

1.5 The conditional instructions

Because a one-address instruction requires only 6 digits, one can store 2 instructions in every memory cell. For the address of an instruction, we simply designate its cell number in the instruction memory and then distinguish the two instructions in the same cell n as instruction $n,0$ and $n,5$ (left and right instruction).

As previously mentioned, the instructions are normally taken for execution in the order in which they are stored, e.g. 105,0 - 105,5 - 106,0 - 106,5 - etc. Certain instructions -- the so-called jump instructions -- cause a disruption of the normal sequence:

Notation of the operation	Operation-code	Address	Effect of the instruction
C	35	n	Unconditional jump: Perform for the next instruction the instruction n,0 , without destroying Op.
Cc	36	n	Conditional jump: Handle as for C if (Op) is positive, otherwise proceed as usual to the next instruction.
CQ	37	n	Handle as for C if (Op) = Q , otherwise proceed as usual to the next instruction.
Co	38	n	Handle as for C if (Op) = 0 , otherwise proceed as usual to the next instruction.

The instruction n,0 which is first executed after a jump is called the <u>target instruction</u> of the jump; after jumping the instructions should be executed in the normal sequence again until the next jump instruction.

1.6 The arithmetic of instructions

Besides the 1000 memory cells for numbers and instructions, the machine possesses another 9 registers for storing three-digit unsigned integers, namely the so-called index registers IR_k $(k = 1,\dots,9)$. These index registers have the following function: When the index digit in an instruction equals k , $k \neq 0$, the contents of index register k are added to the address of the instruction, modulo 1000, before the instruction is executed.[6] For example, if IR_4 contains the number 993, then the instruction 01 4 586 does not load Op from cell 586, but from cell 579 (= 586 + 993 mod 1000).

To load an index register a special instruction is used:

$$SI_k \ (= 21\ 0\ 00k) \qquad (k = 1,2,\ldots,9) \ ,$$

causing the last 3 digits of a in the number $a \times 10^b$ stored in Op to be transferred into index register k. Through a preceding addition of 10^9 one can cause the storage of just the ones-, tens-, and hundreds-digits from (Op) into IR_k; moreover, negative numbers appear in so-called tens-complement representation.

The index registers therefore make it possible to summon a cell from memory whose address was previously calculated by the machine itself. One can also insert the index digit in a jump instruction, i.e., 35 4 586. The target instruction of this jump is then not 586,0 but $586{,}0 + (IR_4)$.

In addition, the following instruction is used in conjunction with the index registers:

$$BZ\ k \ (=22\ 0\ 00k) \qquad (k = 1,2,\ldots,9) \ .$$

With this instruction the current setting of the so-called <u>sequence counter</u> -- i.e., the address of the current instruction + 1 -- is stored

[6] Such an arrangement was previously proposed by T. Kilburn [2] (so-called "B-Tube"). In comparison, in other projects, in particular the Mark III, the contents of an index register is not added to the address of an instruction, but only inserted in its place. The additive effect of the index registers produces important advantages.

in index register k , without chainging Op. One uses this instruction when one wants to jump at a place in a program (which is not a priori known) to a subroutine and later wants to return to the same place to continue execution.

1.7 Constants in the program

It is often convenient to introduce integers into the computation (especially addresses) with the hlep of the following instruction, instead of storing them before the beginning of execution:

Z n : Clear the register Op and insert the three-digit number n .

If the index digit in this instruction is different from 0 , and equals k , then the number $n + (IR_k)$ will be transferred to Op accordingly. In particular, this can serve to load a number from an index register.

2. The Calculation of Machine Code

We are dealing with calculating the machine code corresponding to a given formula (parenthesized expression) such as

$$[A_1 : (A_2 + A_3)] - (A_1 \times A_2 \times A_3) \Rrightarrow B \quad [7] \qquad (2.1)$$

2.1 Representation of a parenthesized expression

It is clear that one must make a few statements about the nature of parenthesized expressions so that the machine can calculate the desired machine code. For this purpose one expresses the parenthesized expression in numerical form:

To illustrate, we examine the above-mentioned example (2.1); this expression contains as "elements" parentheses, operators, operands, and a result; the latter will henceforth likewise be classified as an operand. We denote these elements, without regard to the special nature of the sequence, by $E_1, E_2, \ldots, E_N$, and precede the entire formula with a blank element E_o . Specifically in our example:

$$E_1 = [\; , \; E_2 = A_1 \; , \; E_3 = : \; , \; \ldots , \; E_{19} = B \quad .$$

We must, of course, make several assumptions about the parenthesized expression: First, it must be mathematically meaningful, in that, for example, two operators may not immediately be adjacent to each other. In

[7] The sumbol $\Rrightarrow$ (so-called "replacement"-symbol) should signify, as in the proposal by K. Zuse, not that a conditional equation exists, but that the result calculated from the quantities on the left should be characterized by B . Accordingly, in $x \Rightarrow p$, the replacement symbol means that p should take on the value x .

addition, expressions like $ab+c$ always should be written as $(a \times b)+c$.[8] It is permissible however to combine more than 2 operands in one set of parentheses when the connecting operators are either entirely multiplication or then exclusively addition and subtraction. The machine naturally executes such complex phrases in single steps, e.g. $(a+b-c+d)$ as $([(a+b)-c]+d)$.

In converting ("arithmetizing") a parenthesized expression, each of its elements E_k has two numbers a_k and b_k associated with it. To start, the a_k's are defined as follows:

$$\left.\begin{array}{l} a_o = 0 \\ a_k = a_{k-1} + 1 \text{ , if } E_k \text{ is a left parenthesis or an operand} \\ a_k = a_{k-1} - 1 \text{ , if } E_k \text{ is a right parenthesis or an operator} \\ a_{N+1} \text{ is the Q-symbol and denotes the end of a parenthesized} \\ \quad\text{expression} \end{array}\right\} \quad (2.2)$$

The b_k's are determined as follows:

For $k = 0$ or for a left parenthesis, $b_k = 010000$, which is the machine code for the instruction to "Read from the cell 0 ."

For the replacement symbol or for a right parenthesis, $b_k = 200000$ which is the instruction S 0 .

If E_k is an operator, b_k is the instruction for the execution of this operation upon the number which is in cell 0 , e.g.: $b_k = 020000$ for addition, $= 030000$ for subtraction, $= 060000$ for multiplication, and $= 070000$ for division.

Finally, if E_k is an operand, b_k is its address.

[8] This prerequisite can be eliminated, as C. Bohm shows (Diss. ETH, still unpublished).

For the given example (2.1), the following diagram shows the sequence of the numbers a_k and b_k, with a graphic representation of a_k:

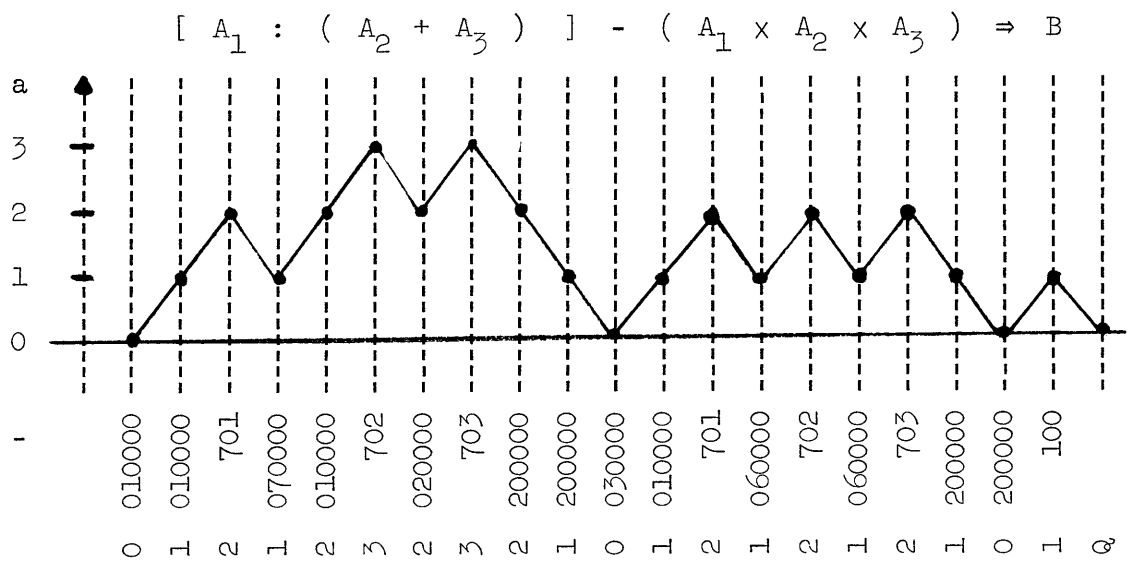


Figure 2

It is assumed that the numbers $A_i, \ldots, B$ are stored in cells $700+k, \ldots, 100$, respectively. From the graphic representation, one recognizes immediately that the peaks correspond to the operands of the expression, and the valleys to the operators, while the parentheses are on the "slopes".

But it also shows that it suffices to introduce only the series of b_k's into the machine, since from these the machine can calculate the a_k's itself, according to:

$$\left.\begin{aligned} a_o &= 0 \\ a_k &= a_{k-1} + \operatorname{sgn}(15000 - b_k) \qquad (k = 1, \ldots, N) \end{aligned}\right\} \qquad (2.3)$$

In the process of course $b_k < 15000$ for an operand (addresses are always less than 1000), as for a left parenthesis while for right parentheses and operators $b_k > 15000$.

With help from the two number sequences a_k and b_k , which completely characterize a parenthesized expression, the machine can automatically solve the following problems:

(a) Numerical interpretation of a formula, by which the numbers to be inserted as operands are determined by their addresses. To this end, the expression is transformed step by step from the inside out to a single number by execution of the respective operations commanded at any given time.

(b) The formula should not be numerically interpreted; instead the machine code for the numerical evluation itself should be calculated.

Although these two problems are fundamentally different, the solutions to them utilize nearly the same methods. But we want to direct our attention toward problem (b), for this has great implications for the planned machine. The necessary calculation for the interpretation of a parenthesized expression or the respective generation of its machine-code program is very extensive, in many cases much longer than the evaluation of the solution itself with help from the finished machine code. It is therefore unsuitable to interpret the same parenthetical expression with the help of the sequences a_k and b_k , for example 100 times, instead of first generating the machine code which would then be used 100 times. Only in very fast electronic calculating machines could the calculating expense for problem (a) be tolerable.

2.2 The principle of code generation [9]

For a given parenthesized expression K, one must first manually punch the b_k values in the correct sequence into a punched tape, for which a preparation device would be useful. For our purpose it would have a keyboard which, besides operators and addresses, also has parentheses and replacement operators. If several such expressions are present, one separates them with Q symbols. In the terminology of §1.4, every parenthesized expression represents one block on the tape.

The actual calculation of the machine code begins with the insertion of the b_k values successively in the machine, which can easily calculate the associated a_k values by (2.3) and, with the b_k's, store them in the following order:

$$a_o b_o a_1 b_1, \ldots, a_N b_N Q \quad .$$

These values are called the image of the expression. Then the altitude H of the expression, that is, the maximum of the a_k's, calculated by the following recursion formula:

$$\left.\begin{aligned} & h_o = 0 \\ & h_k = \mathrm{Max}[h_{k-1}, a_k] \quad (\text{for } k = 1, \ldots, N) \\ & \text{Then } H = h_N \quad . \end{aligned}\right\} \qquad (2.4)$$

[9] In further developments of this work, references are often made to single boxes of the structure diagrams 1-3 to be found at the end of the work, and one may find:

Box 101 - 122 in structure diagram 1
Box 201 - 214 in structure diagram 2
Box 301 - 315 in structure diagram 3 .

Next the expression, described by the values a_k, b_k, is finally decomposed by many reduction steps to be described shortly, while at the same time the desired instruction sequence is built and immediately punched. The generated punched tape is directly usable as an instruction tape.

We denote the original input expression by K_1, and its state after p-1 reduction steps by K_p. Accordingly, one denotes the elements of K_p by E_k^p, its image by a_k^p, b_k^p $(k = 0,1,\ldots,N_p)$ and its altitude by H_p.

2.3 Development of a reduction step

We start out with the parenthesized expression K_p, generated from the original K by p-1 reduction steps. Because the reduction must begin with the innermost parentheses, the p-th reduction step consists of seeking out these parentheses with the use of the a_k^p values and then manipulating them. Namely, one recognizes the operands most deeply nested in parentheses because for them a_k^p reaches the maximum value H_p. Because as a rule $a_k^p = H_p$ for several operands, one first searches left to right among them for the one with the smallest k :

In the preceding reduction step, i_{p-1} should be a specific index value with the condition that $a_k^p < H_p$ for $k < i_{p-1}$. Then beginning with $k = i_{p-1}$ one determines the smallest number k such that $a_k^p = H_p$ (see box 107 in structure diagram 1); this index we call i_p. Therefore:

$$a_k^p \begin{cases} < H_p & \text{for} \quad k < i_p \\ = H_p & \text{for} \quad k = i_p \end{cases} \tag{2.5}$$

It can occur that there exists no such index value i_p ; that is, for all k between i_{p-1} and N_p , $a_k^p < H_p$. In this case, one decrements H_p by 1 and searches further for i_p beginning with $k = 0$ (box 105, 106).

$E_{i_p}^p$ then is necessarily an operand, as a glance at formula (2.2) and Figure 2 immediately shows. Furthermore, as a rule $E_{i_{p+2}}^p$ is also and operand and $E_{i_{p+1}}^p$ the operator between. In general, m operands[10] $E_{i_{p+2\mu}}^p$ $(\mu = 0,\ldots,m-1)$ and $m-1$ operators $E_{i_{p+2\mu-1}}^p$ $(\mu = 1,2,\ldots,m-1)$ are in the same set of parentheses, and the task is to determine the sequence of these operations, including the associated storage of intermediate results denoted by R_p . The instruction sequence for a one-address machine of the form described in §1 reads:

	Operation	Operand
1st instruction	A	$E_{i_p}^p$
2nd instruction	$E_{i_{p+1}}^p$	$E_{i_{p+2}}^p$
⋮	⋮	⋮
m-th instruction	$E_{i_{p+2m-3}}^p$	$E_{i_{p+2m-2}}^p$
m+1st instruction	S	R_p .

According to the conventions of §2.1, the address of the operand for the k-th instruction $(k = 2,\ldots,m)$ is $b_{i_{p+2k-2}}^p$, and the operation

[10] m can easily be determined by the machine as the smallest positive number such that $a_{i_{p+2m}}^p < H_p$ (see box 110).

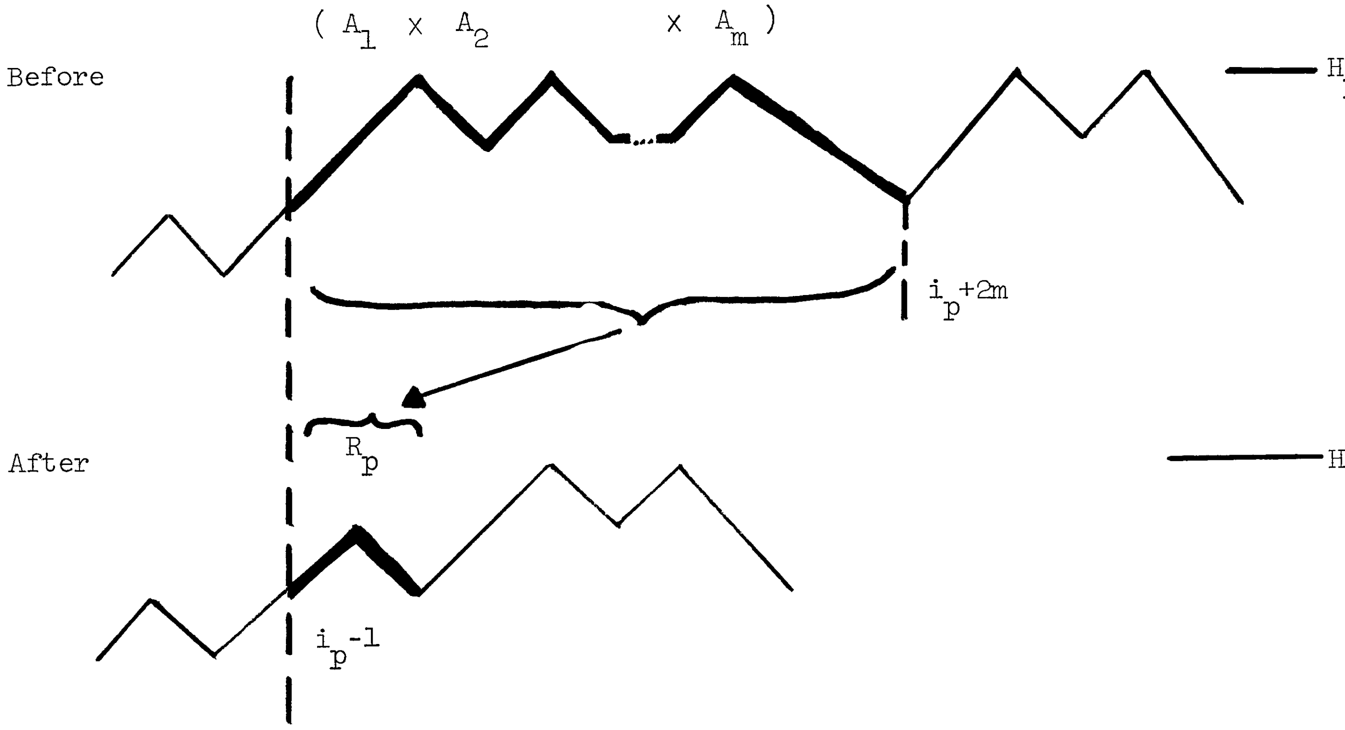
code (followed by 4 zeros) for the operation $E^p_{i_{p+2k-3}}$ is $b^p_{i_{p+2k-3}}$. Therefore one receives the numerical form of this instruction simply through the addition of the two numbers $b_{i_{p+2k-3}}$ and $b_{i_{p+2k-2}}$. This rule also holds for $k = 1$, since the element $E^p_{i_{p-1}}$ must be either a left parenthesis or the empty element E_o , and thus the number $b^p_{i_{p-1}}$ just represents the instruction to read from the cell 0 .

For the address of an intermediate result R_p which will appear again as an operand in further processing of the reduction, one can set up an index (we choose 1000-p), assuming that the cell in question is indeed free during the calculation. Accordingly, one obtains the desired instruction sequence for combining the m operands in the same set of parentheses as follows:

1st instruction:	$b^p_{i_{p-1}} + b^p_{i_p}$
2nd instruction:	$b^p_{i_{p+1}} + b^p_{i_{p+2}}$
⋮	
m-th instruction:	$b^p_{i_{p+2m-3}} + b^p_{i_{p+2m-2}}$
m+1st instruction:	201000 - p .

All that remains is to punch this number in the instruction tape (box 111), for which the instruction LB is used (see §1.4).

Furthermore, the p-th reduction step consists of the replacement of these m operands, the two parentheses, and the $m-1$ operators, altogether 2m+1 elements E^p_k (namely $k = i_p-1$ through $k = i_p+2m-1$), by the result R_p (see Figure 3). Moreover, 2m places become free in this manner; the succeeding elements are moved up 2m places into them



(box 116-119). Thus, the new parenthesized expression K_{p+1} is defined as follows:

$$E^{p+1}_k = \begin{cases} E^p_k & \text{for } k < i_p-1 \\ R_p & \text{for } k = i_p-1 \\ E^p_{k+2m} & \text{for } k = i_p, i_p+1, \ldots, N_p \end{cases} \quad .$$ [11]

$$N_{p+1} = N_p-2m$$

$$H_{p+1} = H_p$$

[11] The machine determines the index N not through numbering, but always as the smallest positive number such that $a_{N_{p+1}} = Q$. The instruction CQ serves this purpose.

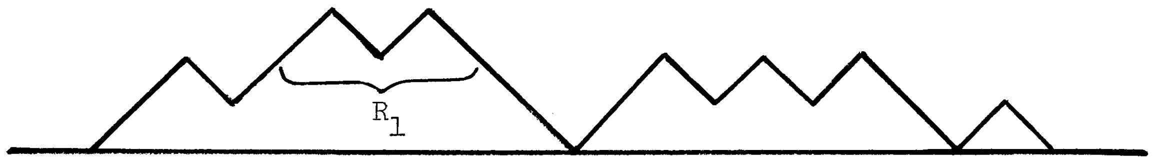
Appropriate changes yield the image a_k and b_k . Only for the new operand R_p , thus for $k = i_p-1$, must one define a_k^{p+1} and b_k^{p+1} : Because R_p is an operand while $E_{i_{p-1}}^p$ was a left parenthesis, $a_{i_{p-1}}$ (using 2.2) remains unchanged, as Figure 3 also shows. On the other hand, the new b-values for this position must be hhe same as the address of R_p , 1000-p . Finally, one sets $a_{N_{p+1}}^{p+1} = Q$, that is, the Q at the end must also be moved up 2m places.

Figure 4 illustrates the stepwise analysis of the parenthesized expression (2.1) given as an example, and to the right the simultaneous construction of the instruction sequence.

Decomposition of the parenthesized expression

Composition of the instruction sequence

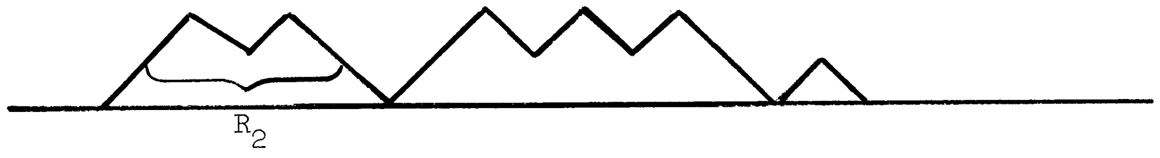
K_1: $[A_1 : (A_2 + A_3)] - (A_1 \times A_2 \times A_3) \Rrightarrow B$



1st reduction: $H_1 = 3$, $i_1 = 5$, $m = 2$; $R_1 = A_1+A_2$

A 702
+ 703
S 999

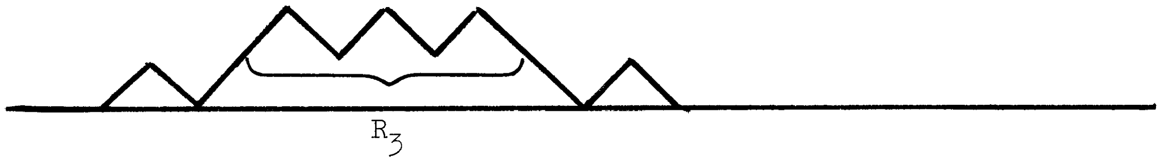
K_2: $[A_1 : R_1] - (A_1 \times A_2 \times A_3) \Rrightarrow B$



2nd reduction: $H_2 = 2$, $i_2 = 2$, $m = 2$; $R_2 = A_1:R_1$

A 701
: 999
S 998

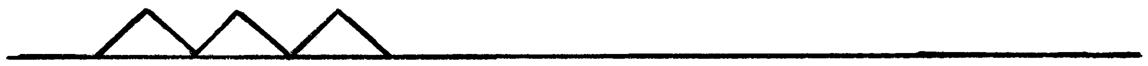
K_3: $R_2 - (A_1 \times A_2 \times A_3) \Rrightarrow B$



3rd reduction: $H_3 = 2$, $i_3 = 4$, $m = 3$; $R_3 = A_1 \times A_2 \times A_3$

A 701
x 702
x 703
S 997

K_4: $R_2 - R_3 \Rrightarrow B$



4th reduction: $H_4 = 1$, $i_4 = 1$, $m = 3$; $B = R_2-R_3$ (Halt)

A 998
- 997
S 100

End

2.4 Operations with only one operand

The calculation of the machine code with the help of the image essentially depends upon the fact that every arithmetic operation always joins its two surrounding operands. But when operations with only one operand appear, such as $|x|$ or $\sqrt{x}$, but also $\sin x$ or e^x, special provisions must be made. We make the following case distinctions:

(a) Operations with one operand which are executed by the machine through a single command (e.g. $|x|$) :

As seen by the machine, these operators follow after the operands, and a typical example reads:

$$\text{For} \quad |a+b-c| \Rightarrow d \quad \left\{\begin{array}{ll} A & a \\ + & b \\ - & c \\ |x| & \\ s & d \end{array}\right.$$

(b) Operations which are not built into the machine and therefore must be executed with the help of a subroutine::

The instruction sequence for a simple example, namely $\sin(a+bx) \Rightarrow y$, consists of the following:

```
A    x  }
x    b  }  Calculation of the argument
+    a  }

BZ   4     Storage of 1+program counter in IR_4
C    sin   Jump to the sin subroutine
--------  }
--------  }  Empty instructions
S    y     Storage of the result.
```

With this arrangement, care must be taken in the sin subroutine because the argument from the main routine still remains in Op (BZ does not change Op). In the same manner, the function value calculated in the subroutine must be placed in Op prior to returning to the main routine, because further calculations require this result. For this purpose, it is assumed that Op is not changed by the jump instruction.

Because the instruction $\langle BZ\rangle + 2$ should be executed after the subroutine and the number $\langle BZ\rangle+1$ is in IR_4 on account of the BZ-instruction, the correct return to the main routine is produced through the instruction C 4 001 placed at the end of the subroutine (both of the empty instructions are connected with the fact that only a left instruction in a memory cell can be a target instruction).

In both cases (a) and (b) we have, immediately before the store command (which corresponds to a right parenthesis or an equals sign), another operator which is succeeded by no other operands. This obviously upsets the arrangement of the number sequence a_k and b_k. To remedy the situation, one constructs a false operand with the address 0 immediately after this operation, as if the operation would be carried out on this operand. Moreover in case (b), when the operation is executed through a subroutine, one must still insert the BZ instruction and both empty instructions. The machine recognizes case (b) by the b_k value, which is ≥ 350000 for such an operator (see box 121). In summary, the following rules apply to the determination of the image for operations with only one operand:

The operand, that is, the expression to which the operation is to be applied, is enclosed with the operator in a set of parentheses; the operator and the empty operand come at the end.

For the operator, b_k is initialized as an image value with: the operation code with four zeros for case (a), or with the instruction calling the subroutine under consideration for case (b).

Through the application of these rules, the meaning of the a_k's and the calculation of the machine code according to §2.2 and § 2.3 is unchanged.

2.5 Application to differential equations

The machine code for numerical integration of ordinary differential equations falls in two parts, namely a permanent routine, which is determined by the integration method applied, and an individual routine which depends only upon the differential equation involved. One is accustomed to producing the latter from case to case and inserting it into the permanent routine.[12]

But now one can also consider a differential equation as a parenthesized expression and let the machine compile the machine code belonging to it and insert it into the permanent routine. As was shown in §2.4, this means that the occurrence of elementary transcendental functions in a differential equation also gives no difficulty. Even

[12] How one inserts such subroutines into a main routine is described in detail in [3] (Part II, Vol. III) and in [6].

though no great saving of time occurs with simply-constructed differential equations, one nevertheless eliminates a source of error through the exclusion of manual code generation. With respect to the considerable loss of time which errors in the machine code cause, automatic code generation has considerable significance.

3. Application to Iterative Problems

3.1 The extension of iterative programs

The methods in §2 apply themselves only to problems of a linear nature which play only a subordinate role in applied mathematics. The methods should therefore be modified so that they may also be applied to iterative problems.

Therein occur two different problem situations, in so far as for an iterative calculation one can set up either a cyclic program or through the explicit enumeration of all the operations a correspondingly longer linear program. (This transition to the linear machine code we call "the extension of an iteration program".) The linear (extended) program is of course much longer than the iterative and is fixed with regard to the running indices, but it solves the given problem significantly faster, because in the iterative routine extra operations are added to the essential arithmetic operations for the control of the loop termination. Of course, the disadvantages of the iterative program are of all the less importance the greater the calculating expenditure per value of the running index.

For complex iterative problems, one may expand at least the innermost loops with profit while leaving the iterative nature of the superposed loops unchanged. Using the machine described in §1, this speed-up technique would reduce the calculation time for a matrix multiplication by 40% from the original amount. In summary, we shall therefore stipulate:

When, owing to the flexibility of the machine, it is possible to produce such compressed programs as in the given example of matrix multiplication in [4], §4.7, this compression is always made at the cost of execution speed, and even a doubling of the execution time does not appear to be unusual. It is therefore recommended -- particularly with a machine which is not very fast -- not to carry the flexibility of the machine to extremes, but rather to take longer programs into the bargain and to simplify the structure. But in order not to have to manually punch the extended programs, which can become very long, one can, as should be clear in the following, let the machine calculate them.

3.2 Representation of running indices

When running indices occur with a single operand in a parenthesized expression, it means that the numerical evaluation is carried out repeatedly (namely once for each value combination of the indices) such that the addresses of the operands in question are to change in a definite pattern with every iteration.

If we want to automatically compile such a program, we must communicate to the machine in which way the address of an operand $u_{\iota_1 \iota_2 \cdots \iota_n}$ depends upon the indices $\iota_1, \iota_2, \ldots, \iota_n$. The following suggestion of course restricts itself to such cases where this dependency is linear:

$$\langle u_{\iota_1 \iota_2 \iota_3 \cdots \iota_n} \rangle = b + \sum_{\rho=1}^{n} \iota_\rho I_\rho \quad . \qquad (3.1)$$

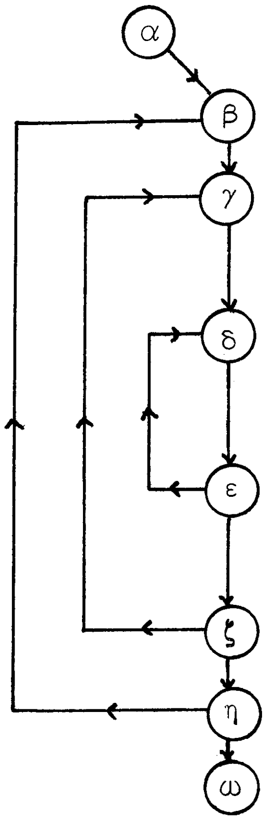
Then one not only attaches 2 image values a and b to the operand u, but also the so-called index image values $10^{-3}I_1, 10^{-6}I_2, \ldots, 10^{-3n}I_n$, which determine the dependency of the address upon the indices. When, for example, the four image values $3(=a)$, $59(=b)$, 2×10^{-3} and -50×10^{-9} belong to an element E, this means that the address of E depends upon 2 indices ι_1 and ι_3:

$$\langle E \rangle = 52 + 2\iota_1 - 50\iota_3 \quad .$$

The image values belonging to an element E are stored one after another in the sequence $a, b, 10^{-3}I_1, 10^{-6}I_2, \ldots$ etc., but for which one generally omits an unused index value to conserve memory.[13] The last index value is followed by the a-value of the next element in the expression.

Finally it must be specified through which values the indices must run. In a problem which is composed of several formulas, this appears somewhat as follows (dealing with matrix multiplication):

[13] One can do this, because the clear relation of the index image values to the different indices is already guaranteed through the factor $10^{-3\rho}$.



```
For  i = 1(1)n: [14]
For  k = 1(1)n:
  0 ⇛ $h_0$
For  j = 1(1)n:
$h_{j-1} + (a_{ij} \times b_{jk}) \Rrightarrow h_j$                (3.2)
End index j
  $h_n \Rrightarrow c_{ik}$
End index k
End index i
Halt
```

Besides the actual parenthesized expression, one must clearly also numerically encode the directions "For i = ..." and "End index i". But first we want to clarify the meaning of these directions on the basis of the structure of the problem[15]: With the statement "For i = 1(1)n" a new index i begins to run; consequently this statement corresponds to the point β in Figure 5, and the statement

---

[14] In the text in §3 and §4, as well as in structure diagram Figure 7, the convenient i,j,k replace the indices $\iota_1, \iota_2, \iota_3$ for simplification of notation. But one must avoid confusion with the index i employed for the numeration of the elements of an expression in the text §2, as well as in structure diagram 1, 2, 3, Figure 6 and Figure 8.

[15] See Figure 5; a detailed structure diagram follows in Figure 7.

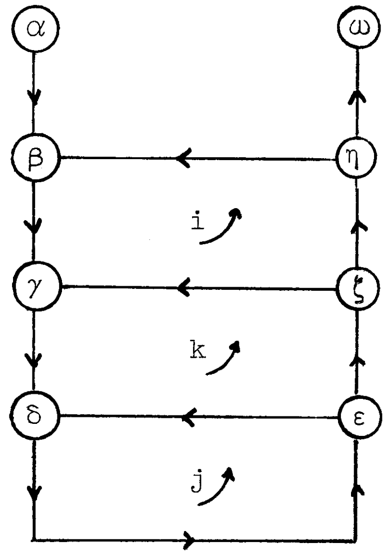


Figure 5

"For $k = 1(1)n$" to the point γ . Formulas which are to be evaluated only once -- there are none in our example -- would be adjoined in the interval $\alpha\beta$ or $\eta\omega$. But the formula $0 \Rightarrow h_o$, which to be sure contains no running indices, nevertheless is to be evaluated for each i and k and therefore belongs to segment $\gamma\delta$. We then have 3 running indices in $\delta\varepsilon$; at point ε it is tested whether the index j has reached its end value. If not, one jumps back to point δ with a simultaneous incrementation of j by 1 ; if yes, the index j expires and the calculation proceeds to the formula $h_n \Rightarrow c_{ik}$, which belongs to segment $\varepsilon\zeta$. We see, therefore, that the point ε corresponds to the statement "End index j" . Analogies are valid for the points ζ and η .

For the following we define an index level s . We assign to every point of the calculation (that is, every point of the structure diagram) a number s which determines the number of running indices at this point.

This same s we also assign to the index just initialized and call s its level. In our example, i has level 1 , j level 3 , and k level 2 .

The matrix multiplication is completely described by the formula 3.2, in that this formula uniquely determines the structure diagram, as indicated to the left of (3.2). It is enough, therefore, to numerically encode the description (3.2) of the problem, taking into consideration the following rules:

(1) After every parenthesized expression (the encoding itself has already been explained) is a Q-symbol.

(2) A statement "For $\iota_\rho = u(v)w$" is encoded with 5 numbers, after which follows a Q-symbol to separate the exression from succeeding expressions or other statements. "For" is represented by a very large number, for example, 10^{12} ; then the index ι_ρ by the number of the cell which is provided for the storage of the numerical value of the index; for this we reserve the cell $45 + 5\rho$. Then the numbers u,v,w follow.

(3) The statement "End index ρ" is represented independently of ρ through 2 successive Q's, exactly like the end of the entire problem.

To clarify, the image values are shown to the right for the section taken from the matrix multiplication,[16]

[16] The image value a is omitted (see footnote 18, on page).

"For $j = 1(1)10$" (α)

$h_{j-1}+(a_{ij} \times b_{jk}) \Rightarrow h_j$ (β)

"End index j" (γ)

for which the following memory cells are reserved:

99 for h_j (independent of j, for one no longer needs the old h-values)

89+10i+j (cells 100-199) for a_{ij}

189+10+j (cells 200-299) for b_{jk}

$$\begin{array}{rl}
 & \cdot \\
 & \cdot \\
\alpha & \left\{\begin{array}{c} Q \\ 10^{12} \\ 55 \\ 1 \\ 1 \\ 10 \\ Q \end{array}\right. \\
\beta & \left\{\begin{array}{c} 99 \\ + \\ (\\ 89 \\ 10\cdot 10^{-3} \\ 10^{-6} \\ \times \\ 189 \\ 10\cdot 10^{-6} \\ 10^{-9} \\ ) \\ \Rightarrow \\ 99 \end{array}\right. \\
\gamma & \left\{\begin{array}{c} Q \\ Q \end{array}\right. \\
 & \cdot \\
 & \cdot \\
 & \cdot
\end{array}$$

3.3 Calculation of machine code for an iterative problem[17/]

The series of image values which characterizes a problem according to §3.2 would first be punched into a tape[18/] and then inserted in an unchanged sequence together with the Q-symbols into the memory of the machine. The machine should then calculate the extended machine code from these image values.

This calculation proceeds, similar to in §2, in that the machine searches the series of image values thoroughly and builds the sequence of instructions; but in doing so the dependence of the addresses of the indices must be taken into account: If a non-integral number follows the image values a_k, b_k of an element E_k, this is not a_{k+1}, but an index image value for the element E_k which is to be multiplied by $10^{3\rho}$ and the index value and to be added to b_k, according to (3.1). This naturally necessitates several changes in structure diagram 1 between box 110 and box 121 (see Figure 6).

But as soon as one finds the number 10^{12} in the sequence of image values, this announces a new statement "For ..."; a new index ι_ρ begins to run and one finds in the following 4 cells the specification of the index, in any case its beginning value, step width, and end value. This instruction is now executed, in so far as the index level s established in §3.2 is incremented by 1 and at the same time the

17/ See footnote 9, page 16.

18/ One can also leave out the value a, because the machine can calculate this from b according to (2.3).

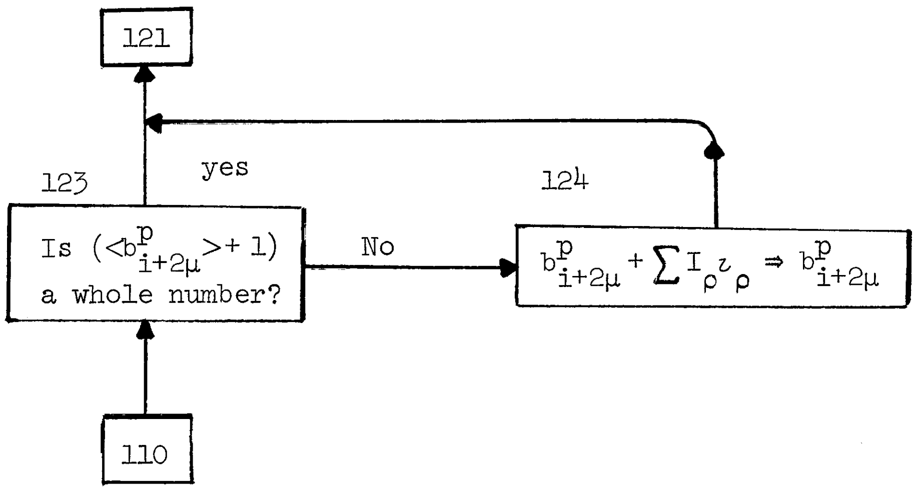

121
123
yes
124
Is $(<b^p_{i+2\mu}>+1)$ a whole number?
No
$b^p_{i+2\mu} + \sum I_\rho \iota_\rho \Rightarrow b^p_{i+2\mu}$
110

Figure 6

beginning value of the new index ι_ρ is transferred to the cell $45+5\rho$ provided for it, the step width in $46+5\rho$ and the end value in $47+5\rho$ respectively (box 204 in structure diagram 2). Moreover, the address of the new index ι_ρ must be stored somewhere, and to be sure in such a way that the addresses of indices of lower levels are st~~e~~ill remembered and immediately available when s is decremented. This can be done best by immediately storing the address of every newly initialized index in the cell s . We have, therefore:

(s) = the address of the running index $(= 45+5\rho)$

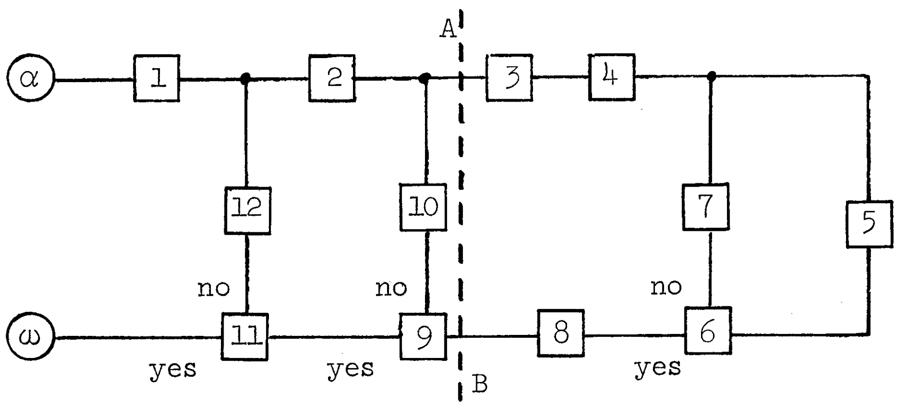
((s)) = the value of the running index $(= \iota_\rho)$.

Accordingly, the processing of the parenthesized expressions following the statement can be initiated (box 206) until a double Q-symbol indicates the end of the effect of the "for-"statement (box 207). Then the running index is to be incremented by the prescribed amount (box 212) and all parenthesized expressions from that point on where the index has reached its end value (box 209); it is discarded with the next double Q-symbol, and s is decremented by 1 (box 210). The calculation of the machine code is complete when s has again reached the value 0 and then another double Q appears.

3.4 Piecewise extension of iterative programs

In complex iterative problems it is usually wasteful to produce a completely extended program for the following reason: The extension of the innermost loops (which belong to the indices of the highest level) can result in a considerable shortening of calculation time, while the corresponding extension of the remaining loops produces only a nominal gain, but enormously lengthens the program. But because the process just described allows only the complete extension of all loops, it must be supplemented for piecewise extension. We want to illustrate this with the example of matrix multiplication $\sum_{1}^{n} a_{ij}b_{jk} \Rightarrow c_{ik}$ (see structure diagram Figure 7): Of the three loops, the j-loop is the innermost and therefore should be extended; this process conforms to the explicit description of the above-mentioned sum:
$a_{i1}b_{1k} + a_{i2}b_{2k} + \dots + a_{in}b_{nk} \Rightarrow c_{ik}$. The resulting program is then only twice iterative, namely over i and k .

Figure 7 shows the structure diagram for the matrix multiplication. It is convenient to partition the diagram along the dotted line because the index j which is to be eliminated occurs only in the right-hand side; therefore the extension of the program takes place exclusively in the right half. For the left half one produces the machine code separately; it must only be connected to the calculated machine code at points A and B .



Legend:

1 : $1 \hookrightarrow i$	7 : $j+1 \hookrightarrow j$
2 : $1 \hookrightarrow k$	8 : $z_n \Rrightarrow c_{ik}$
3 : $1 \hookrightarrow j$	9 : Is $k = n$?
4 : $0 \Rrightarrow z_o$	10 : $k+1 \hookrightarrow k$
5 : $z_{j-1} + (a_{ij} \times b_{jk}) \Rrightarrow z_j$	11 : Is $i = n$?
6: Is $j = n$?	12 : $i+1 \hookrightarrow i$

Figure 7

For the calculation of the sequence of instructions, one must analyze the dependency of the addresses upon indices with respect to j on the one hand and i,k on the other: If E is an operand, its address can be represented as follows: $\langle E \rangle = b + j \times J + f(i,k)$. Then one stores $f(i,k)$ in one of the index registers, for example IR_5 , and calls up the operand E with an instruction which contains 5 for its index digit and $b+j \times J$ for its address. Because, however, $f(i,k)$ is constant in the right part of the structure diagram, one can generate the code for this part according to previous methods (with a single running index j), in which one shifts the processing of the quantity $f(i,k)$ to the left-hand side.

In our example, only 3 elements with variable addresses occur, namely a_{ij}, b_{jk}, and c_{ik}. They should be stored:

a_{ij} in cell $\alpha + j + [ni]$

b_{jk} in cell $\beta + nj + [k]$

c_{ik} in cell $\gamma + [ni + k]$.

The bracketed terms correspond to the $f(i,k)$; we assign them to index registers 1, 2, and 3 respectively, but then must observe in the formation of the machine code for the left half that a change of the values stored in index registers 1, 2, 3 is connected with each change of the indices i,k (box 1, 2, 10, and 12 in Figure 7).

The problem contained in the right side of the structure diagram for which we have to complete the machine code can be formulated as follows:

$$\left.\begin{array}{l} 0 \Rightarrow z_o \\ \text{For } j = 1(1)n \\ z_{j-1} + (a_{ij} \times b_{jk}) \Rightarrow z_j \\ \text{End index } j \\ z_n \Rightarrow c_{ik} \\ \text{Halt.} \end{array}\right\} \qquad (3.4)$$

The formation of the image and the generation of the machine code follows from §3.2 and §3.3, but it must be borne in mind that the quantities a_{ij}, b_{jk} and c_{ik} are called up with the help of the index registers and therefore the index-digits should be added to their addresses in the machine code. The image value b and the index image values for these 3 quantities read:

$$\left.\begin{matrix} 1000+\alpha \\ 10^{-6} \end{matrix}\right\} \text{ for } a_{ij}, \quad \left.\begin{matrix} 2000+\beta \\ n\cdot 10^{-6} \end{matrix}\right\} \text{ for } b_{jk}, \quad 300+\gamma \text{ for } c_{ik}.$$

3.5 <u>An application of a special nature</u>

We shall calculate the extended machine code for the multiplication of a matrix with a vector, whereby certain elements of the matrix can be assumed to be zero. This situation, occurring quite frequently especially in work with systems of linear differential equations, permits a condensation of the machine code by complete extension, which can be very important. The problem is as follows:

```
For  i = 1(1)n
  0 ⇒ $y_i$
For  k = 1(1)n
$y_i + a_{ik} \times x_k) \Rightarrow y_i$
End Index k
End Index i
Halt
```
(3.5)

The calculation of the machine code essentially follows the process already explained. But because the formula $y_i + (a_{ik} \times x_k) \Rightarrow y_i$ should only be evaluated for such value pairs i,k for which $a_{ik} \neq 0$, the calculation of the corresponding instructions should be suppressed, which can be achieved through a small change in structure diagram 2.

3.6 Variable index limits

The methods presented in §3.3 for code generation suffer somewhat from the restriction that fixed numbers must be inserted for index limits, while in practice it is very often the case that an index runs between limits which are dependent upon other indices. A classic example in this direction is the solution of a system of linear equations $\sum a_{ik}x_k = b_i$ using the elimination method of Gauss-Banachiewicz (see [3.6]):

```
For $k = 1(1)n$
For $i = 1(1)k-1$
  $a_{ik} \Rrightarrow h_o$
For $j = 1(1)i-1$
$h_{j-1} + (t_{ij} \times t_{jk}) \Rrightarrow h_j$
End index j
$-(h_{i-1} : t_{ii}) \Rrightarrow t_{ik}$
End index i
For $i = k(1)n$
  $a_{ik} \Rrightarrow h_o$
For $j = 1(1)i-1$
$h_{j-1} + (t_{ij} \times t_{jk}) \Rrightarrow h_j$                                        (3.6)
End index j
$h_{i-1} \Rrightarrow t_{ik}$
End index i
End index k
For $i = n(-1)1$
  $b_i \Rrightarrow h_o$
For $j = i+1(1)n$
$h_{j-1} + (t_{ij} \times x_j) \Rrightarrow h_j$
End index j
  $h_n \Rrightarrow x_i$
End index i
Halt.
```

It is not hard to extend the theory once more to variable index limits. If one of the index limits in "For $\iota_\rho = u(v)w$" is linearly dependent upon the indices of lower level, for example, $w = w_o + \sum w_\rho \iota_\rho$, then one representes the w_ρ exactly as the I_ρ in §3.2, namely through $w_\rho \times 10^{-3\rho}$, and stores these amounts associated with w_o.

In this case, one should check every time during code generation whether the index limits are variable, which is expressed by the occurrence of non-integral numbers in the values succeeding "For". This causes several changes in structure diagram 2 (similar to the changes in structure diagram 1 given in Figure 6 for the index-dependency of addresses).

Furthermore, one must check in every statement "For $\iota_\rho = u(v)w$" put into operation, whether $\frac{w-u}{v} + 1$ (the number of values which ι_ρ should run through) is positive; otherwise the succeeding parenthesized expressions should remain inoperative until the next double Q of the same index level. In particular, this case occurs several times in (3.6); e.g. for $k = 1$ the statement "For $i = 1(1)k-1$" (second line in (3.6)), which then takes the form "For $i = 1(1)0$", is not executable, so that the succeeding lines up to the statement "End index i" (8 lines) are jumped over.

4. Compilation of an Iterative Program[19]

For a fast electronic computer, the reason asserted in §3.1 for the extension of programs may be of little significance. For this reason, foregoing a detailed explanation, several paragraphs are devoted to the generation of unextended machine code for an iterative problem. In conjunction with §3.4, we mention that one can carry out the code generation through a combination of the methods presented in §3 and §4 in such a way that by exercising an option, single loops of the problem can be extended, while, using the same plan, the remaining loops retain their iterative nature.

The compilation of an iterative program is harder with this condition, because the addresses of the operands occurring in the formulas can depend upon the indices in completely different ways, as the example of matrix multiplication dealt with in §3.4 straightforwardly shows. But one can simply proceed in such a way that one calculates every index-dependent address prior to the operation concerned, stores it in an index register, and then calls on the operands with the help of the index-digit.

For the actual implementation of the compilation of an iterative program, it should first be noticed that the encoding of a problem follows exactly as in §3. Furthermore, the code generation is the same as before in principle, that is, the machine analyzes the image and with it builds

[19] See footnote 9, page 16.

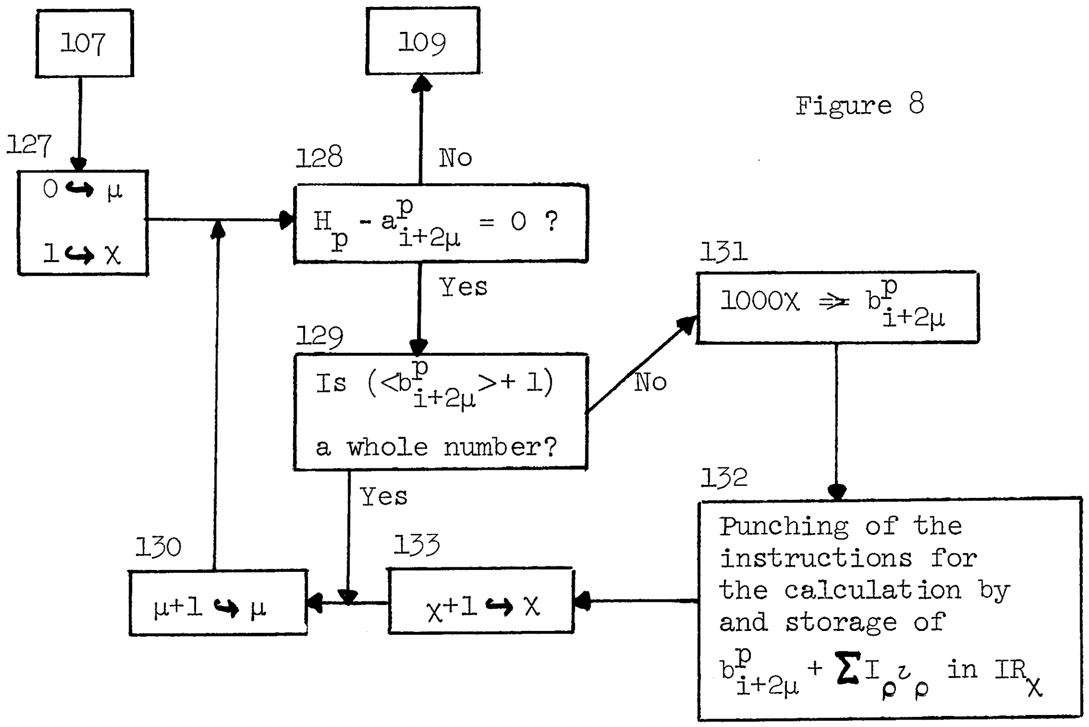
the sequence of instructions, as is described in §2.3. But in regards to §3, it undergoes substantial changes in reference to the running indices: The instruction sequence is set up only once for each expression; in return, the instructions of the succeeding numerical evaluation must be worked out once for every value combination of the indices. To this end naturally, the ncessary provisions should be built into the calculated machine code. This implies:

(a) For the operands: Before every set of parentheses the operands occurring within must be examined for variable addresses, and for each such operand the instruction sequence for the calculation of the address and its storage in an index register must be set up. Only after this may one calculate the instruction sequence for the actual numerical evaluation of the expression. For example, for the operand e_{ij} with $<e_{ij}> = 100 + 30i + 2j$ from the index image values 100, 30×10^{-3}, 2×10^{-6}, one sets up the following series of instructions:

```
 Z 0 100
 S 0 099
 Z 0 030
 x 0 050  (x i)
 + 0 099
 S 0 099
 Z 0 002
 x 0 055  (x j)
 + 0 099
SI 0 001
```

which causes $<e_{ij}>$ to be stored in IR_1 . To this end, the address 0 ,

Figure 8



107
109
127
$0 \hookrightarrow \mu$
$1 \hookrightarrow \chi$
128
No
$H_p - a^p_{i+2\mu} = 0$?
Yes
129
Is $(<b^p_{i+2\mu}> + 1)$ a whole number?
No
131
$1000\chi \Rightarrow b^p_{i+2\mu}$
132
Punching of the instructions for the calculation by and storage of $b^p_{i+2\mu} + \sum I_\rho \iota_\rho$ in IR_χ
Yes
130
$\mu+1 \hookrightarrow \mu$
133
$\chi+1 \hookrightarrow \chi$

```
   Z   (r+2)
   S   (s)
   Z   (r+3)
   S   (s)+1
   Z   (r+4)
   S   (s)+2
  BZ   0 009
   A   0 000
  SI   0 008
   Z   9 002
   S   8 080
empty instruction
```
(3.7)

The last 6 instructions in (3.7) have the objective of storing the address of the instruction following these 12 instructions in the cell $80 + s$. This instruction is the target instruction of the jump which leads back from the end of theloop to its beginning (corresponding to one of the connecting lines $\eta \rightarrow \beta$, $\zeta \rightarrow \gamma$, $\varepsilon \rightarrow \delta$ in Figure 5). Its address must be stored so that at the end of the loop one can correctly formulate the above-mentioned jump instruction. The empty instruction insures that this jump instruction has the desired effect whenever the targe tinstruction is sotred in the right half of a memory cell.

Then the processing of the succeeding parenth^{e}sized expression X continues (box 306) until the next double Q, where again several special instructions are inserted into the machine code, which, for the running index ι_ρ read as follows:

```
 Z 0 080      )
 + 0 000      |
SI 0 009      |
 A 9 000      |
SI 0 009      |
 A 0 045+5ρ   }        (3.8)
 - 0 047+5ρ   |
Co 9 000      |
 A 0 045+5ρ   |
 + 0 046+5ρ   |
 S 0 045+5ρ  .)
```

These instructions test whether the running index has reached its end value, and return to the beginning of the loop in question with the simultaneous increase of the index itself by the increment when this is not yet the case (box 309). The first 5 of these instructions prepare for the jump instruction 9 000, by fetching the address of the target instruction out of the cell $80+s$ and storing it in IR_9. The machine can easily insert the addresses $45+5\rho$, $46+5\rho$, $47+5\rho$ occurring in (3.8) into the calculated code itself because of the assumption that $45+5\rho$ is stored in cell s.

Furthermore, the index level s is decremented by 1 at the double Q (box 310); thereupon the search for image values can be continued. When the index s has sunk to 0 and then a double Q follows once more, the end of the problem has been reached; one inserts the halt instruction into the code (box 308 and 315).

The occurrence of variable index limits creates no new difficulties except those already mentioned in §3.6; one must insert into the sequence of instructions for the storage of the index limits (3.7) the calculation itself of the limits from the image values.

Appendix: Structure diagrams 1 - 3

Explanation: It is assumed that the image values of an interpreted parenthesized expression are already stored in the cells 500ff; the index r always means the address of an image value in the following.

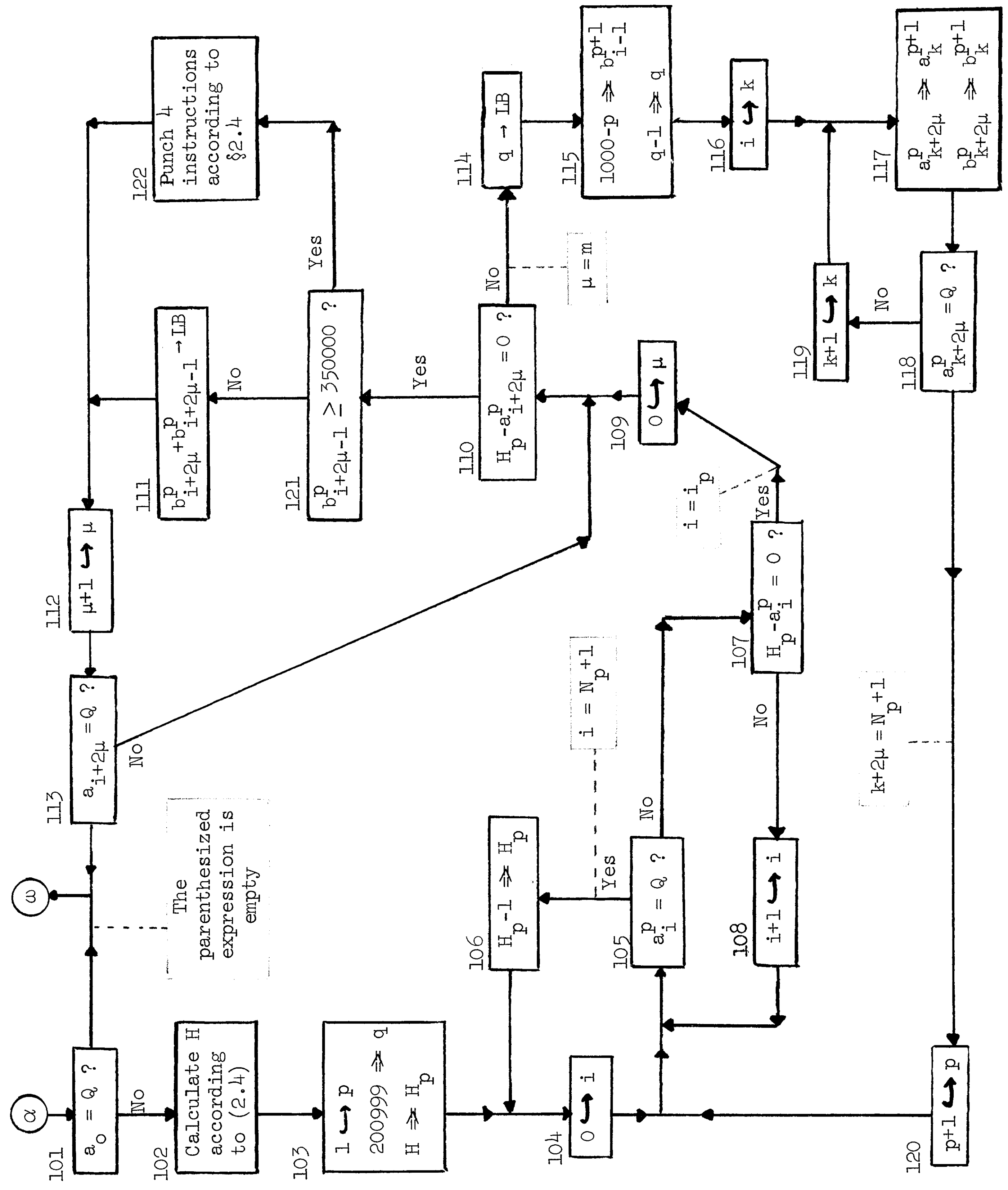
In other respects, the explanation of the symbols ( ), < >, $\rightarrow$ have been alluded to in §1.1. The meaning of the sumbol $\hookrightarrow$ should be explained:

$a \hookrightarrow \iota$, in which a number or an index already set up stands on the left, means: A new index ι with initial value a is to be introduced.

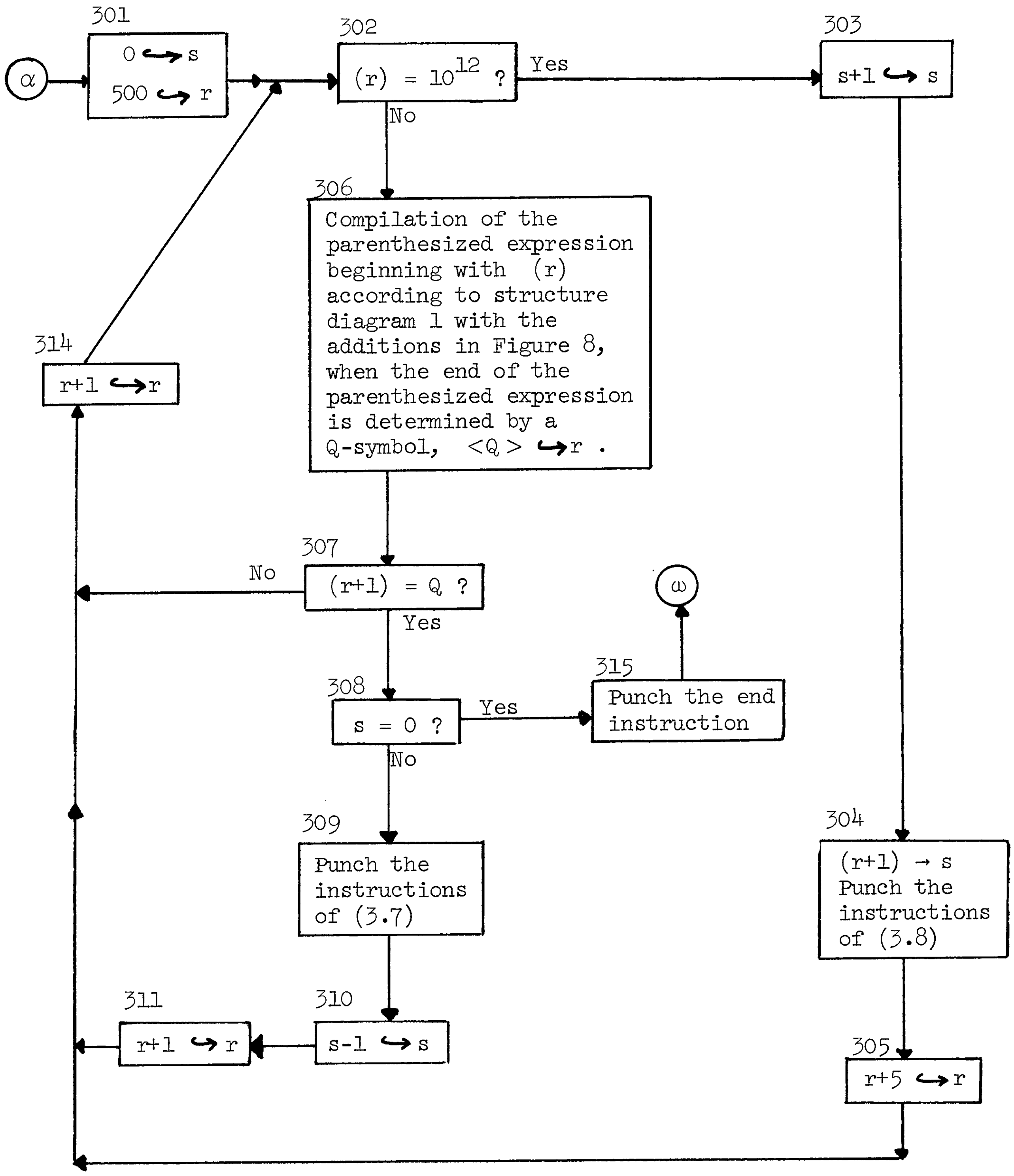
$i+1 \hookrightarrow \iota$ The index ι is to be incremented by 1 at this point; more precisely: the earlier value $\iota+1$ is henceforth to be associated with ι . This causes the automatic updating of all quantities which depend upon the index ι , for example, thereby associating x_ι to $x_{\iota-1}$.

but also the index digit 1 , should be placed in the instruction which should call the operand $<e_{ij}>$ out of the memory. Figure 8 shows the important changes up to now in structure diagram 1 which are to be placed between boxes 107 and 109.

(b) For the indices: When one comes across a new statement "For ..." in the search for image values, this indicates the beginning of a new subloop. As in §3, the index level is incremented by 1 (box 303), then the address of the new index (namely $45 + 5\rho$) is transferred to the cell s , and the sequence of instructions consisting of the first 6 instructions in (3.7) is to be calculated, which serves to store the beginning value, the increment, and the end value of the new index in the cells $45 + 5\rho$, $46 + 5\rho$, $47 + 5\rho$ allotted to them:

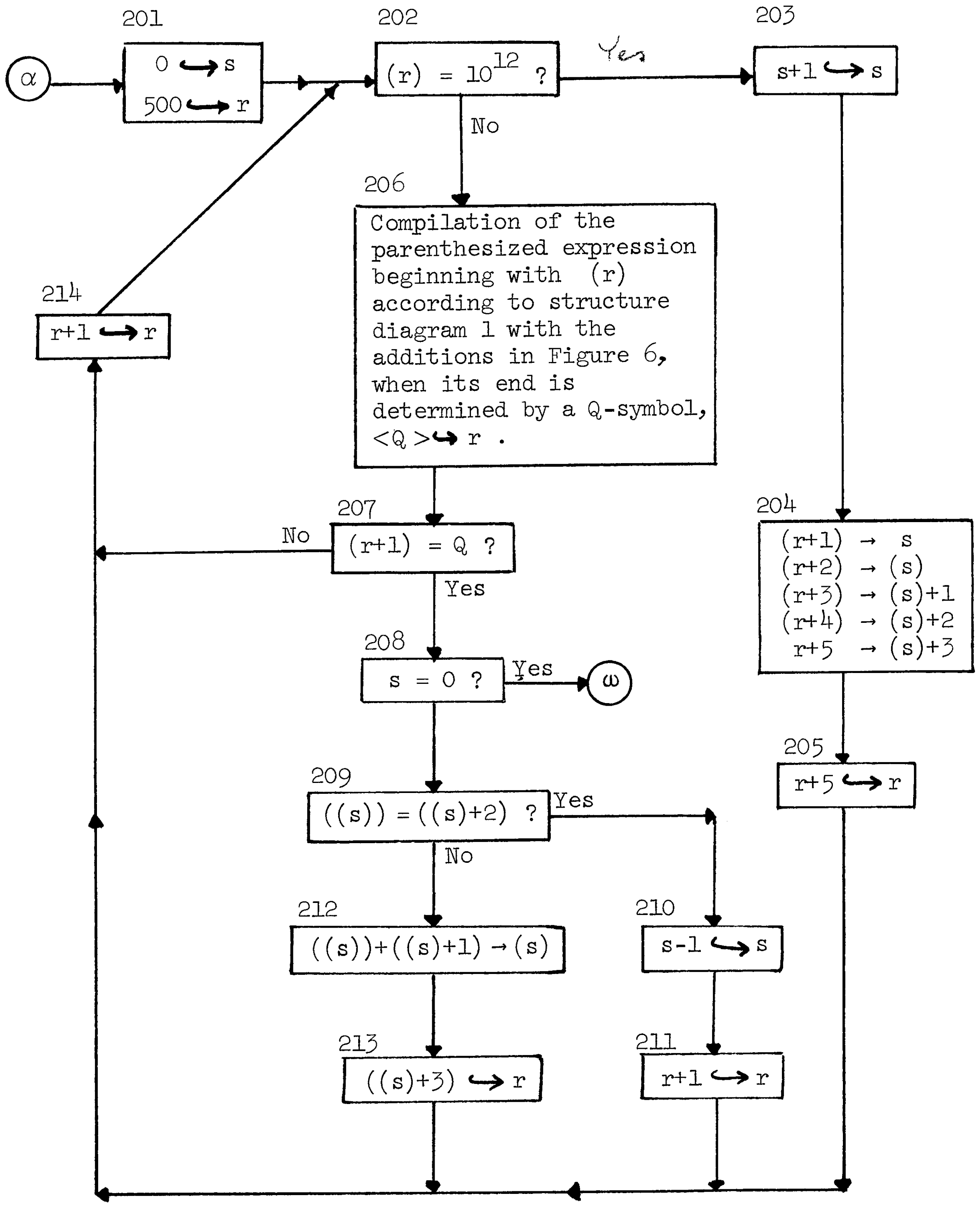

α
101 $a_o = Q$?
No
102 Calculate H according to (2.4)
103 $1 \curvearrowright p$ $20999 \Rrightarrow q$ $H \Rrightarrow H_p$
104 $0 \curvearrowright i$
105 $a^p_i = Q$?
Yes
No
106 $H_p - 1 \Rrightarrow H_p$
107 $H_p - a^p_i = 0$?
Yes
No
108 $i+1 \curvearrowright i$
$i = N_p + 1$
$i = i_p$
109 $0 \curvearrowright \mu$
110 $H_p - a^p_{i+2\mu} = 0$?
Yes
No
$\mu = m$
121 $b^p_{i+2\mu-1} \geq 350000$?
Yes
No
111 $b^p_{i+2\mu} + b^p_{i+2\mu-1} \rightarrow LB$
122 Punch 4 instructions according to §2.4
112 $\mu+1 \curvearrowright \mu$
113 $a_{i+2\mu} = Q$?
No
The parenthesized expression is empty
ω
114 $q \rightarrow LB$
115 $1000-p \Rrightarrow b^{p+1}_{i-1}$ $q-1 \Rrightarrow q$
116 $i \curvearrowright k$
117 $a^p_{k+2\mu} \Rrightarrow a^{p+1}_k$ $b^p_{k+2\mu} \Rrightarrow b^{p+1}_k$
118 $a^p_{k+2\mu} = Q$?
No
119 $k+1 \curvearrowright k$
$k+2\mu = N_p + 1$
120 $p+1 \curvearrowright p$

Structure diagram 1



301
$0 \hookrightarrow s$
$500 \hookrightarrow r$
302
$(r) = 10^{12}$?
Yes
No
303
$s+1 \hookrightarrow s$
306
Compilation of the parenthesized expression beginning with (r) according to structure diagram 1 with the additions in Figure 8, when the end of the parenthesized expression is determined by a Q-symbol, $\langle Q \rangle \hookrightarrow r$.
314
$r+1 \hookrightarrow r$
307
No
$(r+1) = Q$?
Yes
308
$s = 0$?
Yes
No
315
Punch the end instruction
309
Punch the instructions of (3.7)
304
$(r+1) \rightarrow s$
Punch the instructions of (3.8)
311
$r+1 \hookrightarrow r$
310
$s-1 \hookrightarrow s$
305
$r+5 \hookrightarrow r$

Structure diagram 3



Structure diagram 2

Bibliography

[1] Aiken, H. H. Description of a Magnetic Drum Calculator, Harvard University Press, Cambridge, Mass. 1952.

[2] Kilburn, T. "The University of Manchester Universal High Speed Digital Computing Machine," Nature, Vol. 164, pp. 184.

[3] Neumann, J. von and Goldstine, H. H. "Planning and Coding of Problems for an Electronic Computing Instrument," Institute for Advanced Study, Princeton, New Jersey, 1947.

[4] Rutishauser, H., Speiser A., and Stiefel, E. "Programmgesteuerte Rechenmaschinen," Mitteilungen R. 2 aus dem Institut für angewandte Mathematik der ETH. Zürich, 1951.

[5] Speiser, A. "Entwurf eines elektronischen Rechengerätes. Mitteilung Nr. 1 aus dem Institut für angewandte Mathematik der ETH. Zürich 1950.

[6] Wilkes, M. W., Wheeler, D. G., Gill, St. The Preparation of Programs for an Electronic Digital Computer, Addison Wesley Press, Cambridge, Mass. 1951.

[7] Zuse, K. "Ueber den allgemeinen Plankalkül als Mittel zur Formulierung schematisch kombinativer Aufgaben. Archiv der Mathematik, Vol. 1, (1948/49), pp. 441-449.

[8] Kjellberg, G., Neovius, G. "The BARK, a Swedish General Purpose Relay Computer," MTAC, Vol. 5, 1951, pp. 29-34.

[9] Rutishauser, H., "Automatische Rechenplanfertigung bei programmgesteuerten Rechenmaschinen. (Kurze Mitteilung)." ZAMP, Vol. 3, 1952, pp. 312-313.