Rice Institute Computer Project

Programming Memorandum #1

March 1, 1959

Applications to Automatic Coding - Part 1.

Array manipulation and the algebraic formula language.

J. K. Iliffe

Contents. Page

Introduction.

Some potentialities of the Rice Computer code are examined herein, and a formalism is described for the machine representation of algebraic formulae which has advantages in compilation, interpretation, and symbolic manipulation. The interpretive process is considered in detail and presented in Appendix I. A tentative formal language is specified in Appendix II.

Readers familiar with the latest developments in machine code should omit Section 1.

Characteristics of the Rice Computer code.

1.0 Full details of the Rice Institute Computer are given elsewhere. In this section we shall give just sufficient information for an appreciation of subsequent applications.

This computer is an asynchronous, parallel, binary, single address, electrostatic storage machine with an arithmetic unit consisting of eight 54-bit machine words (in "flip-flop" registers) and a control section consisting of eight 15-bit words, each equivalent in length to one full address in the main storage. The arithmetic is in one's complement form, both fixed and floating point, with a six-bit exponent (representing a power of 256) and 48-bit mantissa in the latter case. Fixed point arithmetic uses only the 48-bit fractional part of the word.

Each machine word contains, besides the 54 bits which represent a number, two "tag" bits which can be treated as a label (we distinguish four cases: no tag, tag 1,2, and 3) on a number or an operation.

Input and output is by means of 6-hole punched paper tape, fast line printer, and magnetic tape.

1.1 In the arithmetic unit there is a single length accumulator (56 bits), its extension (R), a subtrahend register (S), four temporary stores (T4...T7) and a null (zero) register (Z). Access time to one of these registers is of the order of one microsecond. Each is addressable as a normal memory cell. These are later referred to as A-registers. In comparison with the IBM 704, U is roughly equivalent to AC, R to MQ, and S to the storage register.

1.2 The control unit consists of eight B-lines, one of which is the control (or location) counter, and another is designated the "pathfinder" (PF) which automatically serves the purpose conventionally assigned to index register #4 on the IBM 704. As well as these special uses each B-register is available as an indexing box and is manipulated by a conventional set of orders.

1.3 There is a further set of six addressable 15-bit fast access registers, consisting of sense light, mode light, trapping light, indicator, and increment (X) registers, and the second pathfinder. The sense lights are used as manual or automatic switches in the usual way. Mode lights are also subject to manual and automatic control, but can not be interrogated in the same way as sense lights. A mode light affects the detailed functioning of a class of machine orders, and may be thought of as controlling the sequencing of a single order in a way analogous to sense light control of the main program. We thus have the option of operating in a rounding, significance, or trapping mode (or any combination of these) without formally changing the program in the machine. SPE CIA REG TER

Overflow and tag indicators are placed in the indicator register, and may be reset at any time. The X register may be used for incrementing B-registers or controlling jumps by a preset amount. The second pathfinder plays a role complementary to the "first" pathfinder and is designed to assist mainly in check-out programs.

1.4 Each order entering the instruction register I is decoded in a similar way. The format of the order distinguishes four fields and although there is some overlap in timing it is convenient (and correct) to regard the decoding as taking place in the following sequence: IN- STR TIO

1. Field 1 (6 bits) Some number from an A- or B-register is brought into U. If n is such a number, we have the options of bringing in n, -n, $|n|$, or $-|n|$.
2. Field 4 (27 bits) The address field is decoded and a number is brought into S. Details of this decoding are given in section 1.6 where it will be noted that the B-registers are additive, that infinite indirect addressing is possible, and that if m is the address which is finally formed in I we have the option

of bringing into S one of M, -M, $|M|$, $-|M|$ or (M), -(M), $|(M)|$ or $-|(M)|$.

3. Field 2 (15 bits) This is the operation code itself, which generally performs some operation on S, U, R, or possibly a B-register. The codes are built up on a modal principle, and a significant proportion of the 2^{15} possible orders is meaningful. Some of the more unusual orders are discussed in section 1.5.

4. Field 3 (6 bits) After the operation is completed several options are provided for incrementing B-registers or transferring quantities in the arithmetic or control units. These are as follows:

(I) Store U (or R) in some A (or B) register

(II) Modify some B register by $\pm$ 1.

(III) Modify some B register by +X.

(IV) Send the final (effective) address from the instruction register to some B register.

An assembly program for the machine allows full flexibility in the synthesis of the order, whilst gaining the convenience of symbolic codes. An example of a single order is

T4 ADD A +B1+ B2, B1+ 1

which has the effect of adding to T4 the contents of Location (A +(B1)+ (B2)) and leaving the result in U. Also B1 is incremented by 1 after completion of the order.

1.5 Apart from conventional orders for arithmetic and logic and B-line modification, the presence of tags calls for a set of orders to control and test these. The arithmetic unit contains both two tag bits, which are set and reset as each number is brought to S, and three tag indicators which are set by the tag bits (tag 1, 2, or 3) and remain set until interrogated by a control order. Thus they act as a basic "memory"

UN-USU. ORD

of the type of number which has come into the arithmetic unit, and can be used as such for a variety of purposes such as loop control, boundary point recognition, tracing, etc. A number may be stored with tags unchanged, or with a specified tag.

Control orders provide for testing whether any or all of up to three conditions is satisfied and taking one of four possible courses of action if the test is successful. Thus any of 2^{12} alternative control orders may be executed.

Twelve of the sixteen binary logical functions are provided.

A repeat mode is available to facilitate table searching procedures. After entering this mode (by an order or manually) the next command is repeated until either a tagged number is brought to S or a test is satisfied.

1.6 Possibly the most important new logical feature in the machine is the use of tags for arbitrary trap transfers when the machine is in the trapping mode (if the machine is not in the trapping mode, tags have no effect except in setting the arithmetic tag indicators). The principle of trapping is that for certain purposes it is advantageous to interrupt the normal sequencing of orders at specified points in order to perform some other task such as input-output control, giving checking information, interpreting the data in S or the order in the instruction register which is about to be obeyed, or taking action on some unusual arithmetic condition such as an overflow.

TRA
PIN

The trapping register is used to monitor these conditions, and each bit of the register which is set equal to 1 is used for interrupting control when both the machine is in the trapping mode and the condition to which this bit corresponds is satisfied. In addition, the option is provided with tag traps of trapping before executing the order (field 2) or after executing field 3. When a transfer takes

place, the next order is taken from one of seven fixed addresses, depending on the condition causing the transfer.

The following conditions may arise and cause a trap transfer:

a) Mantissa overflow: trap after order has been obeyed.

b) Exponent overflow: trap after order has been obeyed.

c) Sign of U mantissa = 1: trap after order has been obeyed.

d) Sign of U mantissa = 0: trap after order has been obeyed.

e) Tag bit 1, 2 or 3 set in control unit: trap before or after field 2.

f) Tag bit 1, 2 or 3 set in arithmetic unit: trap before field 2.

Some bits of the trapping register are spare for additional tests. The detail of field 4 is so arranged that trapping may occur during an indirect addressing chain. That is to say, we have the sequence in decoding field 4 of:

1. If control trap condition is met, trap and turn off indicator.
2. Modify address portion of field 4, M, by adding in the contents of specified B-registers.
$$M + \sum_i B_i \rightarrow M$$
3. Test indirect address bit. If 1, bring bits 31 - 54 and tags from location M and return to 1. If 0, proceed to 4.
4. Examine bit 28 of I. If 1, transfer M to S. If 0, transfer (M) to S.
5. Adjust sign of S according to bits 29, 30 of I (i.e., $\pm S, \pm |S|$).
6. End of field 4. Proceed to test arithmetic tag bits.

It will be noted that some ambiguity in the cause of trapping may arise since separate addresses are not provided for each possible trap. It is up to the coder to resolve such problems arising from multiple trapping lights.

2 A scheme for array manipulation.

In this and succeeding sections practical applications of the Rice Institute Computer are considered. These are illustrated by elementary programs in symbolic form, and more detailed programs are given in Appendix I.

2.1 Indirect array referencing. Consider an array of n numbers $a_1 \ldots a_n$ stored in consecutive locations A ... A+n - 1. There are several equivalent ways of referring to the ith number a_i in this set (with an address independent of i) and each assumes that (B1) (i.e. the contents of B-register #1) is i.

a) We could write CLA A+B1-1.

b) We could write CLA *AC (an asterisk denotes indirect addressing)

where (AC)=ZER A+B1-1.

Obviously this process could continue indefinitely through the indirect addressing feature of the machine. AC is called a codeword for the array A.

Now consider an array of mn numbers $a_{11}, a_{12}, \ldots a_{mn}$ stored in consecutive locations such that a_{ij} is found in A+(i-1)n+j-1. Again, a "direct" indexing instruction would depend on having (B1) = (i-1)n and (B2) = j-1, say, and then we would write down

CLA A+B1+B2

to call out a_{ij}. The fact that B1 and B2 do not contain the "true" indices (i,j) leads us to enquire whether a scheme can be devised with this property, with the following result:

Firstly, we set up an array of m codewords of the form:

(AJ) : ZER A+B2-1
(AJ+1) : ZER A+ n +B2-1
⋮
(AJ+ m-1): ZER A+ (m-1)n + B2-1 .

Secondly, a single codeword AIJ is stored:

(AIJ) : ZER *AJ+B1-1 (2.1.1)

Then supposing (B1)= i and (B2) = j the order

CLA *AIJ

calls out the required element a_{ij}.

It will be noted that in each codeword, only the address field and its B-modifiers (field 4) are used, and the sign modifiers (bits 28 - 30) are retained from the original order.

Thus CLA - |AIJ| brings - $|a_{ij}|$ to U.

The above method is quite general, and can be applied to arrays of any order up to 6. Further, any array can have any number of sets of codewords for the convenience of the coder. An illustration of this case is the matrix transposition routine given in Appendix I(i). It should be remarked, however, that the circumstances in which transposition is necessary are effectively reduced by the availability of such codewords, since we can say the matrix is virtually transposed by writing down a new set, say:

```
(AI)      :  ZER   A+B1-1
(AI+1)    :  ZER   A+n+B1-1
               :
(AI+m-1):  ZER   A+(m-1)n+B1-1
 AJI      :  ZER  *AI+B2-1                    ...  (2.1.2)
```

Then if (B1) = i and (B2) = j the reference

```
CLA   *AJI
```

calls out the a_{ji} element of the array.

The actual storage allocation of the array is unimportant, except that it should be in an ordered fashion with respect to the last index. We conventionally assume it to be stored with the last index varying most frequently. For an $I_1 \times I_2 \ldots I_k$ array we require c_1^k codewords where

$$c_1^k = 1 + I_1 (1 + I_2(1 + \ldots(1 + I_{k-1}) \ldots)) \qquad (2.1.3)$$

and in this case it can be shown it is most economical to choose

$$I_k \geqslant I_{k-1} \cdots \geqslant I_1 \qquad (2.1.4)$$

for then c_1^k is minimised.

There is still some loss in time and space in using codewords to set against the gain in coding simplicity. This can be reduced by generating codewords only for as long as they are

required, and then overwriting, and also by transferring the single codeword which is the "key" to the array to a fast access register during an inner loop (in the above example, we may send (AIJ) to T4, say).

Relative addressing within arrays and between corresponding elements of different arrays is accomplished by using only one set of codewords. Thus, if we have three matrices A, B, C, of equal size with elements a_{ij}, b_{ij}, c_{ij} and a set of codewords for A starting in AIJ, the order

CLA *AIJ, I→B3

will bring a_{ij} to U if (B1) = i and (B2) = j, and also store the address in which a_{ij} is stored in B3. Subsequently, references to b_{ij} and c_{ij} may be made relative to this B register.

2.2 Use of tags for control purposes. In cases where the array is stored in full in a regular manner many efficient ways of scanning it systematically exist, and the formalism of sec. 2.1 is particularly useful when it is necessary to use a scanning "pattern" other than by "row" or by "column", for example a diagonal scan in which i + j = a constant. The size of the array determines the range of indices, and it is a straightforward matter to set and test these using conventional orders.

It is also possible, however, to use the "tag" labels to indicate when the end of a row, say, has been reached, and to rely on these for indexing purposes. With two tag bits, a two-dimensional array is handled conveniently, tag 1 indicating the last word in a row and tag 2 the last word in a column. In this way quite general subroutines may be written to operate on any array of a given order, independent of its size (in a later memorandum we shall consider subroutines which operate on arrays of any order).

In cases where two tag bits are not available, or the formalism of sec.2.1 is being used, a simple alternative is possible using only one tag bit which may also be used for higher order arrays. An example of this scheme in which codewords are tagged

is given in Appendix I(ii), and it will be noted that it is based on the idea of tagging the last word in any linear array, either of numbers or codewords. In the following sections where this idea is used, we shall always reserve tag 1 for this purpose:

The last word of an array is always denoted by tag 1 (2.2.

2.3 Use of tags in data interpretation. For many purposes, as remarked above, the size of an array is immaterial. However, information of this kind must be readily available both to the coder and to the routines which will be required to manipulate arrays and codewords in storage. It is also desirable to be able to store arrays in "condensed" form, when many of the elements are zero and substantial economies in space can thereby be made. A typical example of the latter situation is found in a linear programming problem where there exists a large matrix (say 100 × 200 elements) in which perhaps only 20% of the elements are non-zero.

We therefore suggest extending the functions of the codeword to give the maximum and minimum values attained by the index of the array to which it refers, and also the maximum allowed value of the index. Denoting these quantities by i_{kmax}, i_{kmin}, and I_k respectively, we allocate nine bits to each (which allows a range of index up to 512) and store them in the left half of the codeword.

Type 1 codeword

9	9	9	3	1	8	15
i_{kmax}	i_{kmin}	I_k	IM	IA	BM	M

... (2.3.1)

Normally, we have $I_k = i_{kmax}$ and $i_{kmin} = 1$. However, in cases where the array is condensed we may have $1 \leq i_{kmin} \leq i_{kmax} \leq I_k$. Note that the three bits in the codeword denoted by IM are still unused, and also the general rule that if the sub-array is another set of codewords, IA = 1 (indirect address bit) but if the sub-array is a set of numbers, IA = 0. The BM field indicates which B-registers are used to modify M.

The methods of the previous section are unaltered in a "normal" situation where the array is stored in full. However, in a condensed array we have to ensure, for each i, that $i_{kmax} \geq i$, & $i \geq i_{kmin}$, before extracting the corresponding element. If $i > i_{kmax}$ or $i < i_{kmin}$, the element has value zero. This can be arranged in a number of ways, either by extracting i_{kmax}, i_{kmin}, and I_k before entering a loop, and modifying the control orders accordingly (or by a priori knowledge of the values of i_{kmax}, i_{kmin} as in the case, for example, of a band matrix); or by the intervention of an interpretive scheme which recognises the codeword and automatically extracts the correct element of the array. Such a scheme will be detailed later, but we note that tag 2 will be used, by convention, to denote a codeword which is to be interpreted, and this will be used to trap out of an indirect addressing sequence.

Type 1 codewords only handle an array in which the terminal elements may be zero. In the most general case, non-zero elements are randomly distributed in the array, and we want to store only these. Moreover, we shall have occasion to add and delete elements from the array in an arbitrary fashion, and therefore a "chaining" system is proposed similar to that used by Newell and Shaw in the IPL language.(1) There is one codeword corresponding to each element of the sub-array giving the value of the index for that element, the location of the element in storage, the location of the codeword of the next element in the array, and the "order" of the current array, counting the first codeword in the sequence as order 1. We denote these quantities by i_k, M, $L(i'_k)$ and R.

Type 2 codeword

9	3	15	3	1	8	15
i_k	R	$L(i'_k)$	IM	IA	BM	M

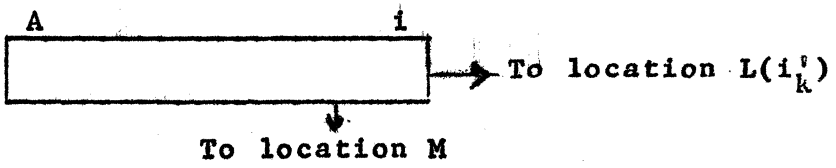
... (2.3.2)

Note that the right hand half of the type 2 codeword is similar in form to that of type 1.

These codewords are used extensively in what follows, and it is convenient to have a schematic representation of some

(1) See, for example, A.Newell and J.C.Shaw, "Programming the Logic Theory Machine". Proceedings of the 1957 Western Joint Computer Conference, p.230.

expressions, in which the codeword is a building block of the form



where A is the address of the codeword and i its tag (if any). Other information may be written inside the box.

For an example, consider the case of the M×N L.P. matrix given at the beginning of this section. A general element is a_{ij}, and the matrix is stored "by column", that is, with index i varying most rapidly. To call out element a_{ij} we set (B1) = i, (B2) = j and give the order

CLA *AIJ.

The codewords are as follows:

(AIJ): type 1 with i_{kmax} = N; i_{kmin} = 1; I_k = N:
IA = 1; Address = AI + B2 - 1; No tags.

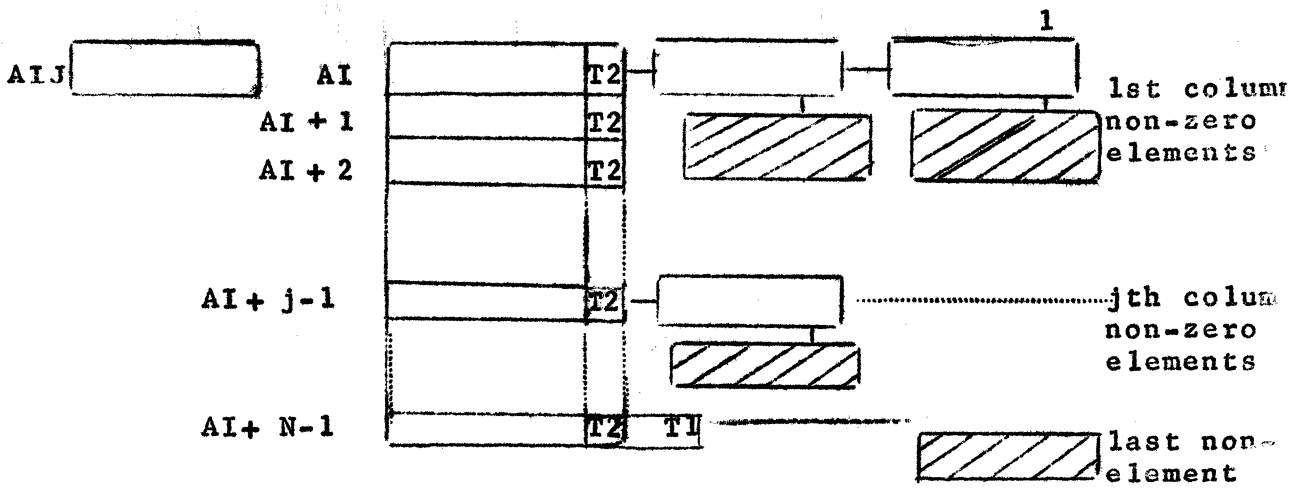
(AI): type 1 with i_{kmax} = M; i_{kmin} = 1; I_k = M
IA = 0; Address = FWA of codeword for 1st non-zero element of first column; Tag 2.

(AI 1): type 1 with i_{kmax} = M; i_{kmin} = 1; I_k = M
IA = 0; Address = FWA of codeword for 1st non-zero element of second column, Tag 2.

... etc. ...

(AI +N-1): type 1 with i_{kmax} = M; i_{kmin} = 1; I_k = M
IA = 0; Addres = FWA of codeword for 1st non-zero element of Nth column; Tags 1 and 2.

Finally, the elements are stored as a chain of type 2 codewords for each column of the matrix. Schematically, the following diagram shows the structure of the array (Shaded cells represent elements).



The only block of cells necessarily consecutive is AI to AI+ N-1, although it may be convenient to store elements in consecutive locations particularly when the matrix is being systematically scanned and rewritten. The indirect addressing sequence is trapped by tags on the second order codewords, and the interpretive program recognises the structure of the sub-arrays.

The importance of the above scheme in array manipulation is considered to be the simplicity of coding achieved by allowing the B-registers to contain true index values, and the fact that several different types of storage arrangement can be made formally equivalent (from the coder's point of view) by means of interpretive-type subroutines called in by the trapping feature of the computer. The normal disadvantages of interpretation are not found here, since "wasted" interpreting time is reduced to a small fraction of execution time.

3. Generalisation of the array concept.

3.0 In this section we shall extend the idea of an array described by codewords to the description of mathematical formulae within the machine. This is based in the first place on describing the formulae as arrays, and then using the conventions of the previous section to handle the array within the computer. It is characteristic of this method that the codewords provide both a natural description of the array and the means of handling it in computational and manipulative processes. We shall also see that we are left with the option of processing the array by means of a compiler-type program before starting a calculation or of starting the calculation directly and interrupting the program where it is necessary to interpret an operation or operand. The trapping feature brings the execution time of interpreter-type programs to within a small multiple of that for direct machine language codes.

3.1 A simple algebraic system. To start with, consider the following algebraic system.

We have two fundamental quantities handled by the machine, namely variables and constants. These are identical in form (floating point numbers) and differ only in their use in the calculation. We denote these by v and c respectively, or, if it is not necessary to distinguish between them, by x. Different quantities are distinguished by subscripts, as in v_7, but we do not suggest at this stage that they form part of an array in the sense of section 2.

There are two operators in the system, namely + (add) and × (multiply) and well-formed formulae (w.f.f.) containing these are defined as follows:

1. Given a formula (string of symbols) w, if w = some v or c then it is a w.f.f.
2. (a). If w is formed from a finite succession of w.f.f. connected by + signs, then w is well-formed, i.e.,

$w = w_1 + w_2 + \ldots + w_k$ is a w.f.f. For $1 \leq i \leq k$, w_i is a sub-formula of w.

(b). If w is formed from a finite succession of w.f.f. of the type (w_i) connected by $\times$ signs, then w is well-formed; i.e., $w = (w_1) \times (w_2) \times \ldots \times (w_k)$ is a w.f.f. For $1 \leq i \leq k$, w_i is a sub-formula of w.

3. If w is a w.f.f. then so is (w).
4. If w is a w.f.f. then so is (-w). For brevity, we may write $w_1 + (-w_2)$ as $w_1 - w_2$.

These are the only w.f.f. of the system. Formulae of type 2(a) are called Σ-formulae, or Σ-terms, and formulae of type 2(b) are Π-formulae or Π-terms. A particular Σ term is written as σ or σ_i, and a particular Π term as π or π_i.

Then we can say that any w.f.f. is given by some σ or π or x, and if it is σ or π, then each sub-formula is of the form σ or π or x, ... and so on until the formula has been decomposed into x-terms.

We define the value of a formula w as follows:

1. If w is x-type, the value of w is the number in the location associated with x.
2. If w is σ or π-type, the value of w is the result obtained by applying the machine operations of floating point addition (multiplication) to the sub-formulae of σ (π).
3. The value of (w) is the same as the value of w.
4. The value of (-w) is the negative of the value of w.

Consider first the representation of w as an array. In general the x-type formulae will be stored in a random manner in the computer, so it is not possible to use a type 1 codeword to describe a Σ or Π term. A modified type 2 codeword is chosen instead. As usual, tag 1 denotes the end of the array, tag 2 indicates that sub-arrays are to be interpreted. The three basic schemes are:

1. x-type. A single codeword gives the location of x and sign modifications (if any). There is no tag 2; there may be tag 1.

2. σ -type. A single codeword C gives the location of the first codeword of the chain, which carries a tag 2. C may include sign modification in the IM field and there may also be a tag 1.

3. π -type. A tag 2 codeword is used, as in (2), to indicate the beginning of a new w.f.f.

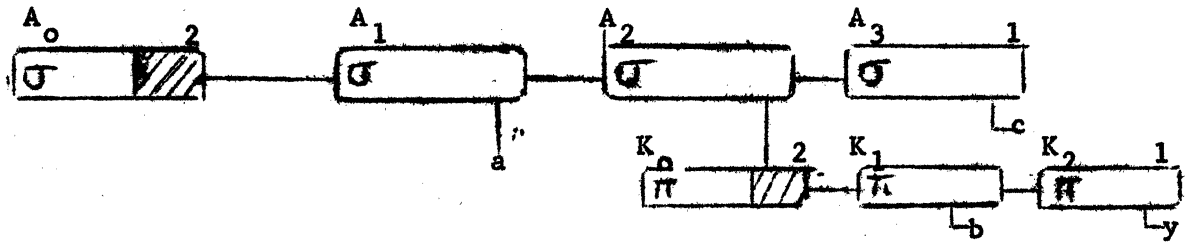
The modified codewords which we use give $L(i'_k)$ and M as before, but the remainder of the word is used to give sign inflexions, a description of the sub-array, and a code for the sub-array (or formula)

Modified type 2 codeword

6	6	15	3	1	8	15
Type	Code	L(next)	IM	IA	BM	M

As before, the IA bit is used to indicate that the sub-array is an array of codewords. If the sub-array is x type, it is effectively named by the (BM, M) portion of the word. If it is σ or π type, the name and type codes will be used as a "first order sieve" in comparing two arrays.

Thus the expression a +by +c is represented as the array:



In a sense, the expression is represented as a "tree" in which the terminal points are x-type formulae, and the structure of the tree corresponds to the structure of the expression. Note that two addresses (shaded) are not used here. Strictly speaking, a saving of two locations can be made in the above scheme, but a use for the additional codeword is suggested in Section 5.1.

We now illustrate two methods of evaluating a given expression. Firstly, consider the Σ -sequence of x-type formulae whose first data codeword is in the address given by (B1). Assume (T7)= 0 initially. Then we may write:

```
SIGMA          CLA   Z +B1,  U→R        Find. codeword
               LUR   d27,    U→B1       Find  address of ne
          T7   FAD   *R,     U→T7       Form  sum
               IF(NT1)TRA    SIGMA                  (3.1.1)
```

The $\prod$-sequence evaluator is similar, with FAD replaced by FMP.

Next suppose that the machine is controlled to execute a trap transfer whenever tag 2 enters the arithmetic unit before the instruction is obeyed, and that SIGMA is operating on the $\sum$-sequence a +by +c given above. Then as soon as (K_0) is brought to S (in the third order of the loop), a trap transfer takes place to a routine which organises a transfer of control to the PI program sequence before continuing with SIGMA. (The next section deals with the mechanics of this interpretation).

An alternative to (3.1.1) is possible, which produces a machine code rather than evaluating a $\sum$-term directly. Consider

```
SIGMA          CLA   Z +B1,  U→R        Find  codeword
               LUR   d27,    U→B1       Find  address of ne
               CLA   *R                 Trap transfer point
          R    ORU   SMASK              Form order
               STO   Z +B2,  B2 +1      Store in program
          IF(NT1)TRA  SIGMA-1, CC+ 1    Test
          SMASK  T7   FAD   Z, U→T7                (3.1.2)
```

In the above, (B2) gives the location count in the compiled program; (SMASK) gives the basic order in the compiled program, which is simply modified for a $\prod$-sequence.

[It is interesting to note an alternative code to 3.1.2: Suppose we make the type codes correspond exactly with the computer codes for floating point addition and multiplication, both in bit pattern and position in the word. Then suppose, when a basic cycle is started, that the codeword is placed in a temporary store. Then a <u>general</u> basic cycle can be written which extracts the function and address from the codeword into a mask to form the next order to be executed.]

The primary problem of translating a paper tape code into the machine representation of an expression is resolved only for an elementary algebraic expression, but the following considerations indicate it is not likely to prove much more involved in a more practical case. We have relied on compound w.f.f. being enclosed in parentheses in order that they may be distinguished readily, and this implies that where parentheses and multiplication signs are omitted in normal notation, they must be replaced by the machine by compounding such transformations as

ab becomes $a \times b$

with $\pm ab$ becomes $\pm(a \times b$

$ba\pm$ becomes $b \times a)\pm$

and so on.

When these replacements have been made, we note the important fact that the occurrence of a left parenthesis is associated with a tag 2 codeword, and a right parenthesis with tag 1. In this way, an array becomes almost a direct transcription, symbol by symbol, of an expression, provided reference addresses are placed correctly and operation codes inserted.

In fact, the process of scanning an expression to form an array can be defined neatly in a recursive fashion, and is illustrated in the Appendix I(iii).

Before proceeding to an elaboration of this scheme, we may summarise its apparent advantages as follows:

(1) The machine representation forms a "natural" link between computer code and formal algebraic expressions, which lends itself to direct manipulation by the machine.

(2) The option is provided of evaluating expressions by direct interpretation or by compiling a machine language program.

(3) The scheme is readily extended to more complicated operations and functions.

(4) The possibility exists of providing for data traps, where it is required to interpret and manipulate data in different forms such as double precision or complex numbers, matrices, etc., during execution of the object program.

(5) The scheme is consistent with the array manipulative system of section 2, and the above simple conventions provide for the direct evaluation of expressions such as

$$y_i = a - 3|x - x_i|,$$

by using the full address portion of the descriptive codeword.

3.2 Extensions of the representation. We shall not be concerned here with detailed extensions to the scheme outlined in sec.3.1, but will indicate some possibilities which have been explored and appear promising. In Appendix II the language we have constructed for writing mathematical formulae is presented in summary form.

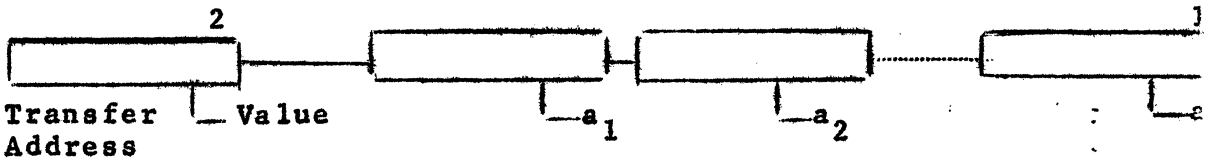
3.2.1 Division. This seems best handled by including it in a Π-sequence as the last term, so that the most general Π-type formula is

$$\Pi_i = w_1 w_2 \ldots w_{n-1} / w_n$$

which does not limit us in any way, since w_n may be any w.f.f. Thus the last codeword of a Π-sequence, which has a tag 1, also indicates whether this is a divisor or multiplier.

3.2.2 Series. The compatibility of sec.3.1 with sec.2 makes the summation of series of indexed terms a natural generalisation of the Σ evaluation, and correspondingly for a product of terms. Thus we include such expressions as $\Sigma_{i=0,\, n} a_i$ and $\Pi_{j=1,10} b_j$ as allowed terms. (The Flexowriter code includes Σ and Π and the facility for sub- and super-scripting)

3.2.3 Function Subroutines. Exponentiation (a binary function), trigonometric (unary) functions, and, more generally, n-ary functions may be represented by the array type



Function subroutines are written in such a way that they are compatible with the interpretive or compiling master program (this imposes only minor restrictions) and allow other functions as arguments.

3.2.4 Operand Interpretation. Most of the previous work is concerned with the interpretation of orders by means of basic loops such as 3.1.1. The potential use of trapping also includes the interpretation of numerical data at the time of program execution without any formal change in the program. This applies particularly in cases where the system is used as a compiler to produce a working program, which the coder may wish to apply, for experimental purposes, to single or double length fixed or floating point real or complex numbers or arrays, without recompiling. These options can be provided under sense and mode light control provided arithmetic orders in the basic loops are themselves trapped. In the same way, some details of array manipulation may be left to an interpretive program used by the object program where a variety of types of array is to be handled.

Thus we can envisage the use of data interpretive routines and array manipulative routines at both the compiling and execution stages of problem solving.

3.2.5 More basic loops. The ability to handle more basic loops depends primarily on the complexity of the recognition process. It would be desirable, for instance, to include the polynomial evaluation $\sum a_i x^i$ as a basic loop, and it seems feasible to recognise this. We should also consider including half term Fourier Series $\sum a_n \cos(nt)$, $\sum b_n \sin(nt)$ in a working system, but at this stage we reach a point where a fine distinction has to be made between the use of a Fourier series as an operand (the value of $\sum a_n \cos(nt)$) and as an operator or function ($\lambda(a_n, t) \sum a_n \cos(nt)$) and in doing so we draw a line between simple automatic coding and more refined algebraic processes which can be performed by the machine, but are better discussed in a later memorandum.

4.0 Interpretation of arrays.

In this section we return to an examination of the type of interpretive routine called for by the generalised arrays of section 3. It has been noted that any expression can be evaluated either by direct interpretation and computation or by interpretation leading to the compilation of a sequence of orders for computing the value of the expression at a later stage. We shall concern ourselves mainly with the former method, on the assumption that a closely parallel system is adequate for the latter.

4.1 The interpretive procedure. This can be stated quite simply. It is aimed at linking together a set of manipulative or computing sequences (CS) such as 3.1.1 to perform particular functions with particular operands, on the assumption that the operands are of a particular nature. In the event that the nature of the operand changes, the sequence is interrupted and further interpretation is necessary.

It is characteristic of the system that one CS may, through trapping, use itself in an iterative or recursive fashion, and it is known that provided (a) the subroutine does not modify itself and (b) some control over working storage and data allocation is exercised, this is allowable (these conditions are sufficient, but not necessary). Condition (a) is accounted for when the routine is written, and (b) is handled by the "working storage counter" which is controlled by the interpretive routine. (This is simply a B-register, and all references to working storage are modified by its contents, as in STO W+B4.) The ease with which the order code of this machine can be adapted to such processes should be noted.

Figures 4.1.1 and 4.1.2 give, in flow chart form, the way in which the interpreter acts. The concept of a subroutine "level" at any point (i.e., the number of steps "down" from the main routine) is useful, although not essential to the code. The full interpretive routine for the system of sec. 3.1 is

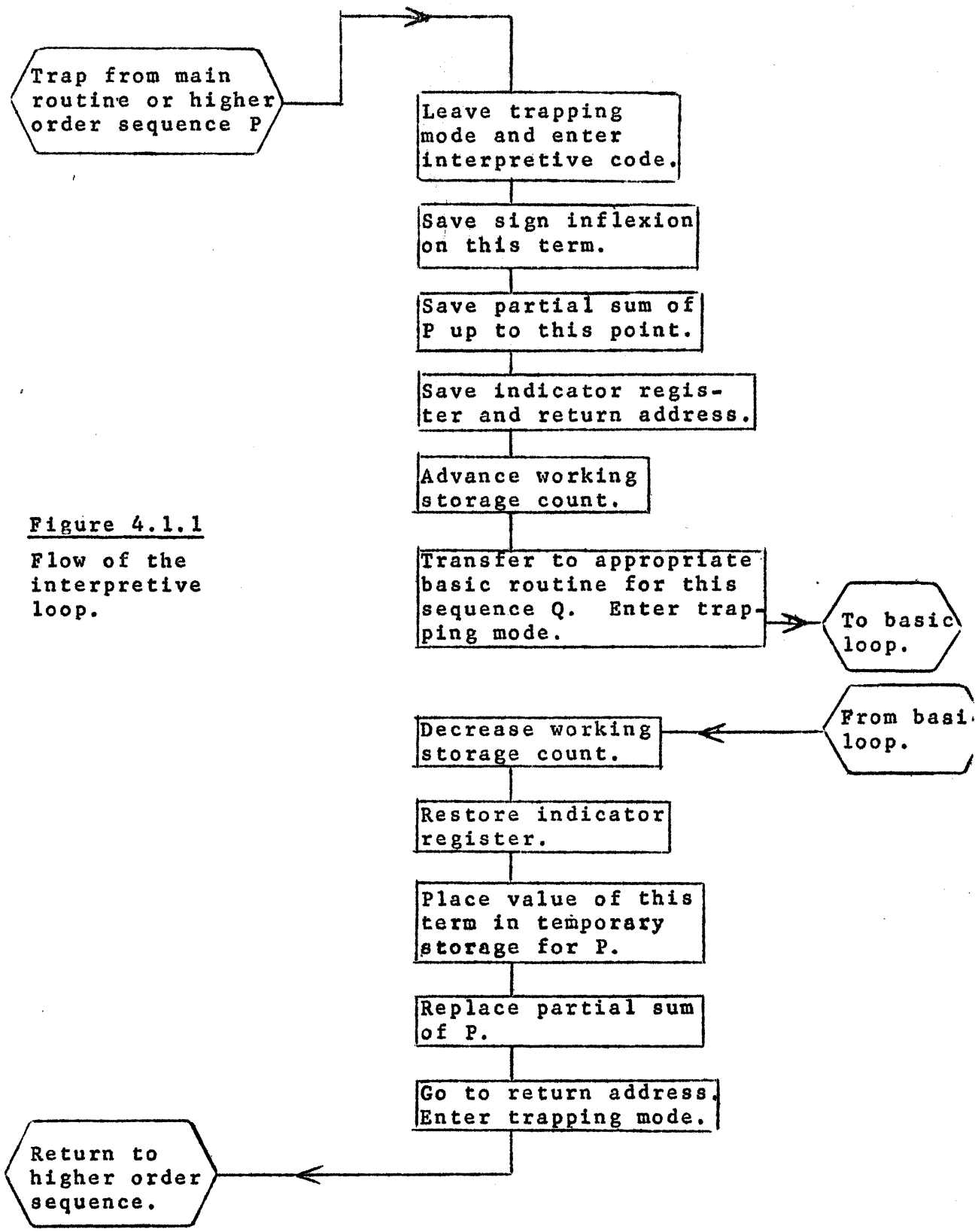


Figure 4.1.1

Flow of the interpretive loop.

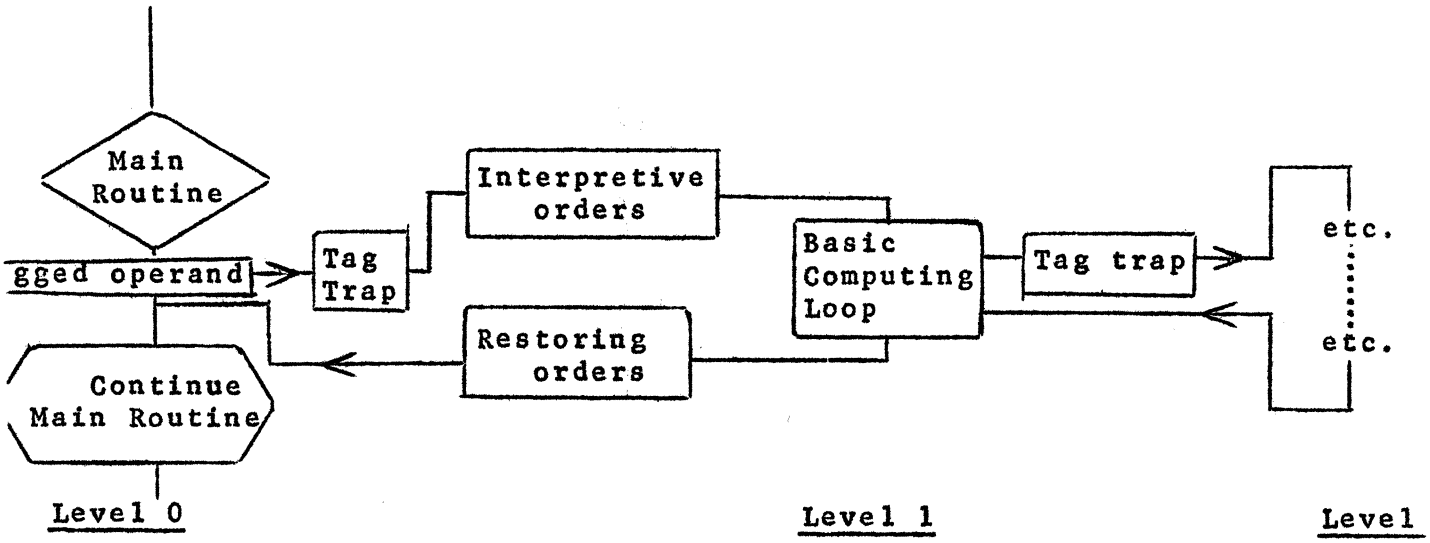


Figure 4.1.2. Relation of the Interpretive routine to the main routine and basic computing cycles.

given in Appendix I(iv). Its extension to more elaborate array structures is analogous. Whilst indexing within the expression is controlled by $\sum$ and $\prod$ operators, an expression may also be indexed from <u>without</u>, and is subject to control operators which are discussed in a later memorandum.

4.2 It should be noted in passing that use of recursive algorithms can lead to extreme inefficiency, as can be seen by applying the above methods to evaluating

$$J_n(x) = (2\ (n-1)J_{n-1}(x))/x - J_{n-2}(x)$$

4.3 A final remark should be made in that this scheme, like any other interpretive or quasi-interpretive system, is well adapted to giving check-out information at all stages of problem solving by means of trap transfers under manual control.

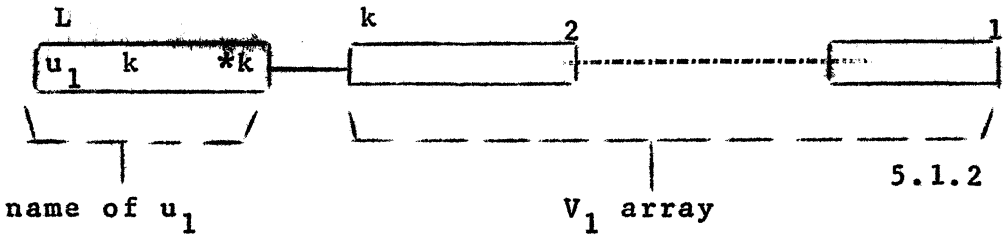
5. Manipulation of expressions and construction of program sequences.

We briefly indicate in this section some uses of the array representation in manipulation of a source program.

5.1 Array complexes. We have so far avoided saying what is understood by an arithmetic formula. It stands for some numerical value which is inserted in relational expressions such as $(u < v)$ or in arithmetic definitions of the form

$$u_1 = V_1 \qquad 5.1.1$$

where V_1 is a formula and u_1 is a variable symbol not defined elsewhere. In the machine, this is encoded as the array



5.1.2

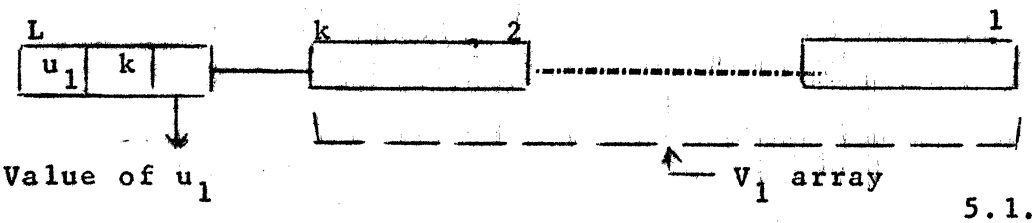
Note that both addresses in the codeword in L refer to k, the first codeword of the V-array. L is called the name of the expression for u_1.

Now we may have a set of definitions:

$$\begin{aligned} u_1 &= V_1(u_2 \ldots u_k x_1 \ldots x_n) \\ u_2 &= V_2(u_1 \ldots u_k x_1 \ldots x_n) \\ u_k &= V_k(u_1 \ldots u_{k-1} x_1 \ldots x_n) \end{aligned} \qquad 5.1.3$$

which express $(u_1 \ldots u_k)$ in terms of some other variables $(x_1 \ldots x_n)$ by means, possibly, of some "cross-connections". In circumstances (which we do not consider here) where the $u_1 \ldots u_k$ are properly defined, it is possible to form the program sequence (PS) evaluating $u_1 \ldots u_k$ given $(x_1 \ldots x_k)$ starting from any u_i and working through the list until they are all found.

Firstly, note that 5.1.3 is represented by a set of interconnected arrays, since any reference to u_1 in the definitions $u_2 \ldots u_k$ will be made via a symbol table to location L in the first array, and so on. Such a set is called an array complex, and it plays an important role in program construction. Secondly, we arrange to replace the second address in each array name by a reference to a location containing its value as soon as this is found, so it is not re-computed later, e.g., the u_1 array becomes



5.1.4

Ths basic CS loops are unchanged, and compiling (or interpreting) stops when all u_i are found. The order of calculating the u_i is unimportant, since the array complex "unwinds itself".

5.2 The independent variable table I.V.T. In the above array complex, if some x_q $(1 \leqslant q \leqslant n)$ is altered then the values of some (not necessarily all) u_i must be recomputed. Such u_i are conveniently found from the independent variable table, which is derived in turn from 5.1.3. It consists of a matrix T_{ij} of binary elements such that

$$T_{ij} = 1 \text{ if } u_j \text{ is dependent on } x_i$$

$$= 0 \text{ otherwise .} \qquad 5.2.1$$

In cases when x_i does change, then all u_j-arrays for which $T_{ij} = 1$ must be changed from form 5.1.4 to 5.1.2.

5.3 Equivalence. Two expressions are equivalent if they represent the same function of the same variables, and we would expect the equivalence of their arrays to be recognized. The present solution to the problem closely parallels the IPL approach, with a crude "sieve" contained in the expression name and a recursive

type matching process. In this case, since operators may have many arguments whose sequence is unimportant, some "standard" or "normal form" convention has to be adopted to give equivalent arrays a chance of being matched, and a simple numerical ordering of variable names has been evolved.

5.4 Re-ordering the expression. It is well known that the order in which an expression is evaluated affects the number of temporary stores used by the compiling algorithm. (Compare the calculation of

$$x = a + b\ (c + d\ (e + f))$$

with

$$x = ((e + f)d + c)b + a$$

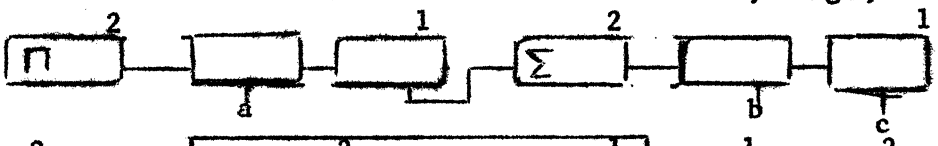
by left-to-right scan.)

The re-ordering necessary to minimise the use of working stores can be expressed in terms of array manipulation, and in fact made compatible with the "normalising" process of sec.5.3.

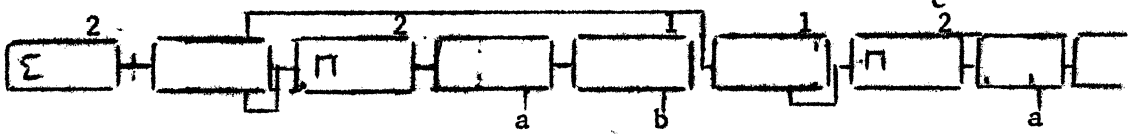
Other types of re-ordering may be considered (e.g., round-off minimisation) but they are less easily defined numerically. In any case, this is an optional feature in the compiler in case the coder already knows the "best" order of calculation in the expression.

5.5 Factorisation. The machine may fail to recognise two expressions as equivalent if one is factored and the other not, e.g.,

a(b + c):



ab + ac:



One solution is to "multiply out" all expressions before attempting to match them, and this does seem to be the only unique way of defining an expression in normal form since its algebraic factors are not unique. This process is likely to be interminably long and clumsy, and it seems more hopeful that some heuristic processes may be applied in attempting factorisation.

5.6 "Unpiling". We also wish to consider the converse problem, of whether a given program sequence is computationally equivalent to a given formula, which involves proceeding from code to array and from formula to array and attempting to match the two.

Firstly, note that subject to minor restrictions it is possible to proceed from a machine language to symbolic program of the API (assembly) type provided the final symbol table is available. Hence the problem is first to derive an array complex from a given symbolic code. A program of practical use would have to deal with control orders and data manipulations as well as as "straightforward" formula evaluations; by considering the latter first we may gain experience in this direction. Therefore consider a program sequence (PS) of arithmetic and store orders which may or may not evaluate a formula in the scheme of section 3.1.

The basic orders for evaluating a $\sum$ sequence $A_1 \pm A_2 \pm \ldots \pm A_a$ are

$$\begin{aligned} &\text{CLA} \quad A_1 \\ &\text{FAD} \quad \pm A_2 \\ &\text{FAD} \quad \pm A_3 \\ &\quad\vdots \\ &\text{FAD} \quad \pm A_a \\ &\text{STO} \quad W_a \end{aligned}$$

A $\prod$ -sequence is similar. Such a sequence can be recognised by the machine and coded as the $\sum$ formula $W_a = A_1 \pm A_2 \pm \ldots \pm A_a$.

Since in the simple algebra we proposed, only such sequences as these are possible we end up with a set of formulae of the type

$$\begin{aligned} W_a &= A_1 \Theta A_2 \ldots \Theta A_a \\ W_b &= B_1 \Theta B_2 \ldots \Theta B_b \\ &\vdots \\ W_n &= N_1 \Theta N_2 \ldots \Theta N_n \end{aligned}$$

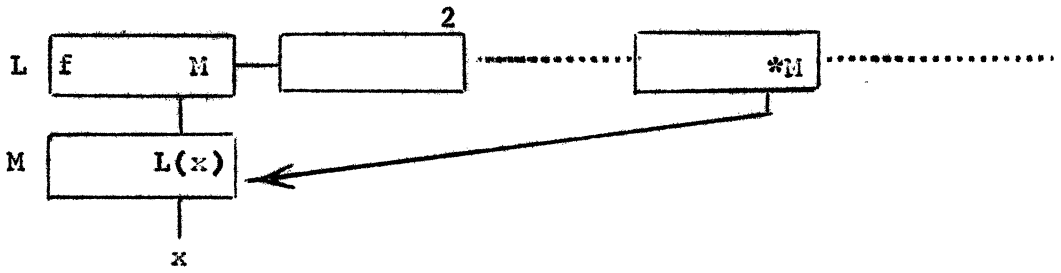
where A_1 ... N_n may include any of W_a ... W_n, and ⊛ stands for one of the allowed machine operations. But this is what has already been described (section 5.1) as an array complex (AC) and we may treat it as a single array for manipulative purposes.

It is hoped that a general approach along these lines will prove fruitful.

5.7 Subroutines. In cases where frequent applications of I.V.T. cause recalculation of some u_I, it is conveniently written as a function subroutine $u_I(u_1 \ldots u_k, x, \ldots x_n)$ to ensure that a compiler codes it in a closed form. The organisation of the subroutine hierarchy is a problem of control rather than formula language. However, expressions such as 5.1.1 may be generalised to include functional expressions of one or more arguments by an array which, instead of having a value reference in the name codeword, has a reference to a list of arguments, the analogue of a calling sequence in machine code. References to the argument are made, indirectly, via the argument list.

f(x) = ... x .....

becomes



Appendix I. Examples of codes using the conventions of this memorandum.

(i) Matrix transposition. (A) is an N× N matrix for which two sets of codewords have been set up, as described in section 2.1, with "key" entry words AIJ, AJI, which call out the a_{ij} and a_{ji} elements respectively, where (B1) = i, (B2) = j. Then we have:

```
TRANSP      Z   SB1   1, U→B2          (B1) = i = 1
                CLA   aN, B2 +1        N = order of matri
            -U  STO   X, B2 + 1        (B2) = j = 2
 AA         Z   CLA  *AIJ, I→B3        (a_ij)→U
                FST  *AJI              U→L(a_ji),(a_ji)→S
            S   STO   Z + B3           S→L(a_ij)
           -B2  IF(ZER)SKP X           is j = N?
                SCC  AA, B2 + 1        No
            B1  ADD  a2, U→B2          Yes
                IF(POS)SKP  -X, B1+ 1  Is i = N-1?
                SCC  AA                No
                TRA  PF                Yes. Return.
```

(ii) Control of Loops by tags. The following routine sums the squares of the elements of a vector a of any length, storing the result in Y. It is assumed that the a_i are stored as floating point numbers in successive locations starting in A+ 1 and finishing with an element bearing tag 1.

```
SQSUM       Z   STO  T4, U→B1
 NEXT           CLA  A + 1 + B1
                FMP  U, B1 + 1
                FAD  T4, U→T4
                IF(NTI)TRA NEXT
               T4   STO  Y
```

(iii) The basic computing sequences of section 3.1:

```
(a)   SIGMA      CLA    Z + B1, U→R
                 LUR    d27, U→B1
                 CLA    *R, U→T7
      SLOOP      CLA    Z + B1, U→R          Evaluation of
                 LUR    d27, U→B1            the Σ-sequence
           T7    FAD    *R, U→T7
           IF(NT1)TRA SLOOP

(b)   PI         CLA    Z + B1, U→R
                 LUR    d27, U→B1
                 CLA    *R, U→T7             Evaluation of
      PLOOP      CLA    Z +B1, U→R           the ⌈⌉-sequence
                 LUR    d27, U→B1
           T7    FMP    *R, U→T7
           IF(NT1)TRA PLOOP

(c)   INTERP   S RPL    R                    Interpretive loop.
             PF IF(ODD)TRA NOTSAV
                 LUL    d15
                 ORU    INDREG               Save indicators,
                 LUL    d15                  pathfinder, and
                 ORU    B1, B6 + 1           (B1).
                 STO    2 +B5
              B5 RPM    R, B5+ 1             Save inflexion on
               R STO    Z + B5               this sub-array
           T7    STO    B5-1
             Z   STO    INDREG, B5 + 1
                 LLS    d9, U→PF             Transfer code→PF
             Z   LLS    d15, U→B1
             Z   ETM    ,U→ R                Enter trap. mode.
                 SCC    (Transfer location) + PF, B5+ 1
                 Execute basic computing sequence
      RETURN     CLA    B5-1, U→B1           Return from CS
                 LUR    d15, B5-1            with value of
                 STO    INDREG, B5-1         result in T7.
```

```
        LUR   d15, U→PF          Restore registers.
PF      IF(ODD)TRA  NOTRES
T7      FST   B5-1
S       ETM   ,U→T7              Set reference
        CLA   B5+ 1, U→R         word in R
        SCC   PF-1, B6-1         Exit
```

Notes (i) A distinction is made between the cases of saving the partial result of the higher order array or not, and only routines for the former case are given here.

(ii) (B5) = working storage count; (B6) = level count

(d) Conversion of data input to arrays.

We consider the problem in quite general terms. The main difficulty comes from the concept of a hierarchy of binary operations permitting parentheses to be omitted. The subsidiary complication of allowing one operation sign (e.g. multiplication) to be omitted is easily dealt with.

When parentheses are inserted in a formula they are dealt with by tag traps as indicated below. We therefore consider a program Π designed to convert a formula F from punched paper tape codes to the array convention in storage, where F does not contain any internal parentheses.

Let $0_1\ 0_2 \ldots 0_n$ be a set of binary operations with associated values $r_1 \leqslant r_2 \ldots \leqslant r_n$, which rank them in the hierarchy. At any point the following quantities are in the machine:

(i) A^j, the array of completed codewords and formula names with its associated index j giving the next available memory space.

(ii) L^s, the list of partially completed sub-expressions S of F. Each term in L gives (a) j_1, the index of the first codeword of S in A; (b) j_2 the index of the last complete codeword of S in A; (c) the rank r_s of S; (d) the sign inflexion (if any) on S.

(iii) M, the codeword for the last complete sub-formula read, but not yet stored.

(iv) N, the codeword for the next sub-formula (in the case of F, this is always a variable symbol V with sign 0_n, rank r_n).

Then Π uses four primitive sub-processes on F:

π_1: Add M to A^j at level r_s

$\pi_2(r_i)$: Terminate current sub-expression with its codeword in M, and "step up" to the level r_i.

$\pi_3(r_i)$: Start new sub-expression in L^{s-1} with rank r_i and first numbers M and N.

π_4: Place the codeword for the next sub-expression in N. Then Π can be described by a symbolic control language (to be more fully explained in a later memorandum) as follows (for " " read "and then execute" and for Cp(q,r) read "if p execute q, if not p, execute r").

$$\Pi. = \pi_4 \rightarrow (M = N) \rightarrow \pi_4 \rightarrow \pi_3(r_n) \rightarrow \Pi(M, N, L^s).$$

$$\Pi(M, N, L^s). = C(r_s = r_n)((\pi_1 \rightarrow (M = N) \rightarrow \pi_4), (C(r_s < r_n) ((\pi_3(r_n) \rightarrow \pi_4), (\pi_2(\min(r_{s-1}, r_n))))) \rightarrow \Pi(M, N, L^s).$$

Use of Tags. Characters read from paper tape are edited and checked by the input routine for illegal characters and character pairs before combining signs with variable symbols to present to Π. The edited symbols are placed in an input buffer region D^k before being transferred by π_4 to N. Left and right parentheses or their equivalent ( ) ↓ ↑ etc) cause a tagged codeword to be placed in D^k, with the following effect in π_4:

Tag 1 (Right parenthesis): Indicates the end of F. π_2 is executed to "step up" through the remaining incomplete expressions to the final codeword for F, which is stored in N before continuing.

Tag 2: Interrupts the current F and transfers to a new Π.

In addition, tag 3 is used to indicate the end of data stored in D^k, initiating a further read-in from tape using the editing program. In this way it is hoped to find an optimal value for the amount of algebraic code to process at a time.

Appendix II: The Formula Language

This is a tentative description of a language for presenting mathematical formulae to the Rice Institute Computer, based on the compiling and interpretive features of the programs described in the rest of this memorandum. It is intended to be sufficiently general, however, to be applied to many other formal systems. This is obviously a long-term project, but many useful sub- (and super-) systems can be devised, such as that of sec.3.1.

(i) Variables. Any single symbol a, b ...z, A, B, ...Z, α, β, γ, σ, λ, π (58 in all) stands for a variable or operand in this language, which, unless otherwise indicated, is assumed to be a single precision floating point number. The list of distinct symbols is enlarged by allowing any of the above which do not appear in an array declaration to be subscripted with up to four Flexowriter characters.
Thus x, K_{Mass}, $\lambda_{t \leq 7}$ are allowed variable symbols. The subscripts have no significance other than as distinguishing marks.

(ii) Constants. The symbols .0123456789 express constants as decimal numbers in conventional form, which are converted to floating point form unless they appear in subscripts or index expressions.

(iii) Arrays. Any symbol may be declared to stand for an array of either one dimension or two. By convention, elements of an array are indicated by subscripting the array symbol. It is not necessary to specify the size of the array, since this can be controlled by the data input program, but by doing so some execution time is saved. The dimension may be symbolic.
Thus
Matrix Q (5, 10)
Vector Π
Matrix K (m,n)

are valid array declarations. An element of an array may itself be an array, as in

Vector $Q_{i,j}(10)$

in which the kth element is denoted by $(Q_{i,j})_k$.

(iv) Subscripts. The general subscript form is $i \pm n$ where i is a variable symbol and n an integer constant. By appearing as a subscript i is defined as an index variable and is always evaluated using integer arithmetic modulo 512.

(v) Operations. The symbols +, -, /, × have their normal arithmetic meanings when appearing between elementary operands. Where possible, they are also interpreted correctly between arrays. At times, however, it may be desirable to leave the interpretation of some operations to program execution time, in which case the superscript * is used, as in $+^*$, $-^*$, $/^*$, $\times^*$. These cause interpretive trap transfers when the order reaches the control unit. The normal × may be omitted if so desired.

(vi) Operand types. As with operations, the operands may be labelled with * to cause an interpretive trap transfer when the operand reaches the arithmetic unit during program execution.

(vii) Formulae. A formula is a string of operand and operation symbols, whose meaningfulness is defined in a recursive fashion. Some examples illustrate the type of formula which is allowed:

$a + 2bx + cx^2$

$(a_i + |\lambda - .0019|)^{\frac{1}{2}}$

$(A^* + B^*)/n^*$

$\sin|x + y|$ where "sin","cos" appear in

$\cos\ (x +^* y)$ the subroutine list.

$\gamma(1 + r - r^2)$ where "γ" appears as a functional expression

$\sum_{i=1,\ 10} a_i x^i$

$$\prod_{i=1,5} \cos((.1+i)t)$$

(viii) Expressions are of the form

$$u = V \qquad \ldots\ldots (A)$$

where V is a formula independent of u and u is a variable not defined elsewhere.

Example: $y = ax^2 + bx + c$ is a normal expression

We distinguish two special cases:

I. If u is defined as an index variable, then (A) is an <u>index expression</u> and V is evaluated using integer arithmetic modulo 512.

II. If u is of the form $u(x_1 x_2 \ldots x_r)$ then (A) is a <u>functional expression</u> expressing u as a single valued function of the r independent variables $x_1 \ldots x_r$.

The formula language consists of a set of array declarations (iii) and expressions (viii). To form a meaningful computer program, these must be linked by a control language, which organises program sequences and input-output functions together with storage allocation. However, simple sequences may be specified entirely in this language.

Example. Matrix A (10,10)

$$y = (x/90)^{\frac{1}{2}}$$

$$x = \sum_{i=1,10} \sum_{j=1+i,10} A_{ij}^2$$

$$\sum_{i=1,10} \sum_{j=1,i-1} A_{ij}^2$$

This defines y as the root mean square of the off-diagonal elements of A.