AN ADDRESSLESS CODING SCHEME BASED ON MATHEMATICAL NOTATION

by

C.L. Hamblin

N.S.W. University of Technology

1. INTRODUCTION

I suppose everyone would agree that programming a modern digital computer is a laborious and sometimes extremely complicated task. But not everyone would agree what, if anything, should be done about it. Some programmers seem to be just plain defeatist, to accept the situation as a consequence of the laws of nature. Others place a great deal of reliance on so-called "interpretive" schemes and use secondary instruction codes which are simpler, or do more per instruction, than the primary codes of current machines; but unless such schemes are designed ad hoc for particular problems or classes of problems they are liable to be extremely wasteful of machine time, and even in some cases to waste the programmer's time as well. I do not personally think that there is any general solution of the immediate practical problem: a computing organisation must simply build up its library of programs - which may of course include special-purpose interpretive schemes - and look to its efficiency in adapting them.

On the other hand there is a good deal to be said for research in what I should like to call the "pure" theory of coding; not only because some of us will, God willing, in the course of time be designing new computers but also because there are things in it which may be of use to us in the meantime. And there is also a third, if less tangible, reason: it can give us a better grasp of what our programming problems are and of what machines do and are capable of doing.

There is of course no one ideal computer code. But if we limit consideration to computers for general mathematical and mathematico-logical applications, i.e. to computers in the mainstream of the development process, a certain amount can be said which is at least fairly widely applicable. The code which is obviously the most convenient for the programmer of mathematical problems is the one he ordinarily uses in formulating them, namely the symbolism of ordinary mathematics. Once a problem has been formulated mathematically - formulated, I mean, as a clear sequence of mathematical operations - it is in principle in a form which could be accepted by a machine. The remainder of the task of the present-day programmer is essentially a task of translation from one already adequate language into a radically different one, which is at the same time vastly more redundant in that he must make many decisions regarding storage addresses of both numbers and instructions.

Ordinary mathematical symbolism has two things to commend it. The first is simply that it is already familiar to the programmer, and in fact common to all programmers and to their customers. The second is that in the course of time it has evolved by natural selection into an extremely efficient language. A good symbolism of this sort is not to be lightly set aside: one is reminded of the numerous cases in the history of mathematics in which the invention of a notation has represented the crucial step in the development of some branch of the subject.

2. SYMBOL ORDER

If some features of mathematical notation are unsuitable for our purposes we should not of course hesitate to modify them, provided we do so with a full consciousness of the issues involved. In the last hundred years or so a good deal of attention has been devoted by formal logicians to the study and invention of symbol-systems for logical purposes. Not very much of this is immediately relevant here, and work on machine-codes uncovers its own problems and possibilities; but one logical notation in particular I should like to mention.

A minor complication of algebraic symbolism concerns its conventions regarding "bracketing". For many years now, however, logicians have used a system of writing logical formulae which eliminates brackets entirely - the so-called "Polish" notation*. The trick is to be single-minded in always writing operator-symbols in front of the symbols for the entities which are operated on. In the formula "a + b", for example, the operation of addition is applied to the numbers a and b: in Polish notation it would be written "+ab". The result is that if a second operation is applied to the first, as in "(a + b) x c", no brackets are needed: the Polish notation for this would be "x+abc", whereas in the case of the formula "a + (b x c)" it would be "+a x bc".

For machine use this system has the disadvantage that the order in which the operator-symbols occur in the formula is the reverse of the order in which the operations are supposed to be performed. It is perfectly feasible, however, to use a "reverse Polish" notation in which the operators follow the operands: i.e. in place of "a + b" we can write "ab+", and in place of "(a + b) x c" we can write "ab+cx". It is now not very difficult to demonstrate that each symbol of a formula can be regarded as a machine instruction. Let us imagine we have a number of storage locations arranged in a linear order and reserved as working-space in connection with the arithmetic unit. I shall refer to these locations collectively as "the accumulator" and to individual locations as "cells". When a number is required for an arithmetic operation it is placed in the first vacant cell, i.e. in cell 1 if this does not already contain a number, otherwise in cell 2 or etc.; and when a diadic operation such as addition or multiplication is carried out it is always on the numbers most recently transferred in, i.e. the numbers in the last two occupied cells. The operation of addition, more specifically, will be carried out by adding from the last occupied cell to the next last, the last cell being "emptied". Now the result of any formula in "reverse Polish" notation, interpreted as a sequence of instructions, will be to calculate the number represented by the formula and leave it in cell 1. For example the formula "a b + c x" operates in detail as follows:-

(i) "a": number a transferred to cell 1.

(ii) "b": number b transferred to cell 2.

(iii) "+": contents of cell 2 added to those of cell 1, leaving a + b in cell 1 and cell 2 empty.

(iv) "c": number c transferred to cell 2.

(v) "x": contents of cell 2 multiplied by those of cell 1, result (a + b) x c in cell 1 and cell 2 empty.

---

*See e.g. Chwistek, L., The limits of science, Eng. tr. London, Kegan Paul, 1948.

In this example only two cells are used. But if the formula had been written in the equivalent form "c a b + x", three cells would first have been filled with numbers before any operations were carried out. This system of having what might be called a "running accumulator" has the advantage, implicit in the mathematical symbolism, of permitting intermediate results to be "held" pending the calculation of additional terms.

A monadic operator, i.e. an operator on a single number, will be written after its operand in the same way and will represent an operation on the number in the last occupied cell. Triadic and higher-order operators are rare, but can be accommodated by the same means, as also can operators giving more than one result - including the special case of a double-length result. As coding possibilities all these are important since they would be needed as subroutine operations if not as elementary ones.

3. VARIABLES AND SUFFIXES

Now let us look at some of the other properties of mathematical notation. Numbers can be written in a mathematical formula directly as a sequence of the numerals 0-9 with or without a decimal point; and it would be very convenient, if not strictly necessary, to be able to write them in this way in a program. Algebraic representation is more important, however, since provided some means is available of identifying numbers with letters any constant number required can be represented by a letter anyway,

The most obvious interpretation of algebraic variables is that they should represent storage locations. This, however, is only a start, because the entire resources of the Latin and Greek alphabets are insufficient to name even the fast-access storage locations in a present-day computer, and many more numbers are required in some problems. How does mathematics manage to formulate these problems? I think the answer lies in suffix-notation.

Suffixes have at least three functions. In the first place they indefinitely enlarge the alphabet, since after one has run through a, b, c... etc. one can start on a_1, b_1, c_1 ... etc. and then a_2, b_2, c_2 ... and so on. But one can also have algebraic suffixes, and this leads us to their second function: they enable us to classify numbers into groups, and to generalise about members of a group or to indicate the performance of some operation involving the members of the group as a whole. Thus if an operation is to be carried out on a group of numbers x_1, x_2, ... x_8, say, we indicate it as carried out on "x_j", and make a note in the margin about the range of values of j; and this simple notational trick is a sort of analogue of a loop in a present-day flow diagram.

The third facility that suffixes provide is a rudimentary "function" notation. A set of values of a function can be represented by a variable with a range of suffix-values; and when an algebraic suffix is used it may be considered as representing, albeit usually on a different scale, the argument of the function. Thus x_j is a function of j, and the symbol "x" can here be considered as a sort of operator-symbol. It follows that to conform with "reverse Polish" algebraic notation we should write it after the symbol "j"; and since j is an ordinary variable we should write it as such and not (as in ordinary notation) provide a special range of suffix-symbols. Actually it is the suffixed variable and not the suffix which should have a special notation. For simplicity, however, we might prefer to have one special symbol rather than an alphabet of them, and I suggest a symbol "/", written between the two. For example for "x_j" we can write "j/x".

Since double suffixes are of frequent use there is something to be said for making provision for them, and in conformity with what has been said we

could use one additional symbol "//" and e.g. in place of "a_{ij}" write "i j // a".

The logic of these notations, together with that of the algebra system outlined already, automatically guarantees the provision of a very large slice of the facilities normally associated with suffixes. For example, when a symbol-sequence such as "j / x" comes along the first thing that happens is that the number j is placed in the next vacant cell of the accumulator, and arithmetic operations may be carried out on it as they ordinarily are on suffixes: for example, for "a_{j+n}" we can write simply "j n + / a". Another straightforward possibility is the use of suffixed suffixes, e.g. as in "a_{j_k}", since without any further symbols or elementary operations we can write this in the form "k / j / a". The limits on the uses of this and allied notations are not so much logical as technical ones, i.e. store size and access time: for each doubly-suffixed variable represents a "matrix" of storage locations, and even with a moderate limit on the permissible numerical magnitudes of the suffixes --- say 32 --- one such matrix would take up an appreciable fraction of the total storage of present machines.

4. REPETITIVE OPERATIONS

I have said that ordinary mathematics indicates repetition of an operation for a range of suffix-values by means of a note in the margin, e.g. "(j = 1, 2, ... n)". This is however only part of the truth: there are other cases for which special notations are provided. An example of this is repeated summation, using the symbol "Σ". This kind of notation is logically easy enough to provide: all that is required is a special symbol, plus an indication of the variable of summation and of its limiting values. The limiting values of the summation are most conveniently written before the special symbol, and the variable-symbol after it.* There would follow the formula representing the summand, and finally some kind of "bracket" symbol to mark the summand off from the subsequent program.

One comment I should like to make here is that a summand is rather similar to a subroutine, "called into control" by the summation-symbol a number of times determined by the parameters. Furthermore, the possibility of nested subroutines is paralleled by the possibility of nested summations (or other repetitive operations), and by the possibility of repetitive operations occurring within subroutines or vice versa. It follows that if some kind of provision is made for automatic subroutining it goes some way towards meeting the case of repetitive operations.

The most important provision to be made towards automatic subroutines is what I should call a "nesting register": this is a sequence of storage locations for instruction addresses operated on something the principle of the running accumulator. It may even regularly store, in its last occupied cell, the address of the current instruction for use by control. When a subroutine is called in, the address of its first instruction is placed in the next vacant cell and becomes the current instruction address, the point of interruption of the main program being indicated by the address in the last cell but one. Then when the subroutine is completed the last occupied cell is cancelled and control reverts to the point of interruption.

---

*There is however an advantage in using a "name" symbol (see Appendix I).

If the same system is adopted for repetitive operations, some indication is needed in the nesting register to distinguish the entries from those of subroutines. It happens that it would be very convenient also to be able to store a "count" number. A quite general repetitive operation, representing simply an instruction to repeat an arbitrary specified sequence of operations for each of a range of parameter values, could then be provided quite simply. This is the analogue of the notation "(j = 1, 2, ... n)".

5. CONDITIONAL INSTRUCTIONS

Such a notation appears to be capable of coping with most of the applications requiring repetition of instructions, and hence explicit "looping" of programs is hardly needed. The need for "discrimination" is also reduced. There appear to be still cases, however, requiring these latter facilities; and it is necessary to consider how best they can be introduced into a scheme of this kind. I shall consider discrimination first.

Mathematics has no very satisfactory notation for discrimination: the nearest approach seems to be an arrangement such as:-

$$x = \begin{cases} a & (y \leqslant 0) \\ b & (y > 0). \end{cases}$$

On the other hand, formal logic has something to say on the subject, at least in principle: it can be pointed out that the basic requirement is for a logic of conditional instructions, of the form "If p then A", where p represents a proposition and A an instruction. If we assume that we shall have some symbol or symbols to represent the "if ... then ..." operation, what we are lacking is primarily a means of representing propositions. Now there is one important way in which propositions are represented in mathematics, and that is by means of equations and inequations. It is accordingly appropriate to introduce symbols such as "=", "$\leqslant$", "$\geqslant$" and so on (or a selection of them) and treat them as diadic operators which operate on pairs of numbers to give propositions. Numerically a proposition can be represented by its "truth-value", i.e. a digit (say the sign-digit) to indicate whether it is true or false. If we use "1" for true and "0" for false we accord with the usual computer convention for Boolean operations; and these may incidentally be introduced and will have their usual meanings when interpreted as operators on propositions. For example if "&" represents the Boolean "and" operation the proposition "s < x < b" can be represented as the conjunction of the propositions "a < x" and "x < b", taking the form "a x < x b < &"*.

We can represent the conditional instruction "If p then A" in some such form as "p →, A]": for example, the instruction "If x = y, add n to the number in the accumulator" becomes "x y = →, n +]". Methods can be devised of handling nesting of conditional instructions using the nesting register.

6. CONTROL TRANSFER

"Control transfer" instructions represent the biggest problem in this kind of notation; and none of the suggestions I can offer is completely satisfactory. Since the whole point of the notation is to save the

---

*A minor modification, the use of "all ones" for true and "not all ones" for false, permits an interesting application to the logical calculus of propositions. False mathematical propositions are in this case represented by "all zeros".

programmer from having to worry about addresses inside the machine it is undesirable at this stage to demand that he specify the instruction address to which control is to be transferred. An alternative is to insert recognisable "markers" in the program and provide means of instructing that control be transferred to such-and-such a marker; but unless the machine can be persuaded, say, to make a list of all marker addresses while reading in a program, this involves a search for the appropriate marker and is presumably ruled out on the grounds of speed.

The most important case of control transfer is that of subroutine entry; and in this case it seems less objectionable to demand listing (automatic or not) of instruction addresses. For the rest there are some hopes that control transfer may be unnecessary in other cases if a sufficiently flexible system of conditional instructions can be found. This possibility however requires more study and I shall not enlarge on it here.

7. REALISATION AS SECONDARY CODE

To help crystallise the above suggestions into something approaching a practical code, at least from the point of view of the programmer, I have prepared an interpretive program for Utecom and tried it out on a number of problems. It is of course rather wasteful of machine time, and I would not seriously put it forward in this form for general use. It completely vindicates my hopes, however, as regards programming convenience. In fact it has been a real joy to be able to write programs in a matter of minutes, and particularly to be able to write faultless programs. The latter point should be particularly emphasised since it has an economic importance: hardly any machine time has been wasted in program-checking, and problems have usually gone through in one uninterrupted machine session.

The code actually used differs from that suggested above only as regards the method of discrimination: the facility provided is a conditional transfer of control, the actual transfer being initiated by one of two alternative kinds of "skip" instruction which operate in conjunction with program place-markers and a search-routine. The "skip" instructions can also be used unconditionally. Various facilities have been added, e.g. for input and output of numbers from or to the usual punched cards, and for a number of longer operations such as exponentiation, radication and trigonometric functions which would not of course be elementary machine operations. A table of code-symbols, with some additional notes on their operation, is given in Appendix I.

Symbols are coded as groups of eight binary digits and are fed four-per-word (48 per card) into the machine by a routine which eliminates any spaces left between them (code 00000000 is not used): this facilitates the patching-together of a program without wasting storage, since component cards may simply be collected into a pack. Subroutines are simply attached to the program, and a special symbol is used to terminate program read-in and start the calculation at the first symbol. The scheme is unambitious in its system of numeration: all operations are carried out as if the numbers were expressed to 16 binary places (in the Utecom word of 32 digits) and always give single-length answers. Overflow checks are provided. The number store is divided like Gaul into three parts: (i) the "scalar" store (32 unsuffixed variables), (ii) the "vector" store (32 singly-suffixed variables) and (iii) the "matrix" store (four doubly-suffixed variables --- only a, b, c and d can carry double suffixes). Since suffixes can take values from 0 to 31, up to 5000 numbers can be stored. There is room for 512 instruction-symbols. The "running accumulator" has 31 cells (though no program so far has ever used more than about ten) and the nesting register has six. Some of these figures I would alter if I were to redesign

the scheme, but in general they have proved adequate. The exception is the number-store, which is unduly inflexible: it needs reorganisation to permit, for example, the use of much larger suffix-values under certain circumstances and could perhaps be treated sequentially such that "a_{32}" represents the same number as "b_0", and so on.

Some examples to illustrate programming techniques are given in Appendix II. To those who find these merely bewildering I can only point out that the notation is extremely compressed compared with, say, that of the customary coding-table; but is in consequence all the more convenient in the long run since when learnt it can be read (and written) with a facility that a coding-table never could.*

8. CONCLUSION

How feasible is it to build a machine embodying this kind of code? I do not know, having made no detailed attempt to investigate. But I should like to record the impression --- though it is admittedly scarcely more than that --- that the majority of the requirements could be met without much real difficulty by a judicious combination of elementary operation and built-in subroutine. The built-in subroutine facility is perhaps the crucial one: given this, the logical design process becomes one of deciding exactly what elementary operations are the best ones to use as building-bricks. A good deal must depend on the broad technical approach (e.g. on the kind of storage used) and hence can only be profitably discussed after some initial technical decisions are made. On the other hand there are also many possibilities of theoretical developments along the lines I have mentioned, and it should be fairly clear that seemingly technical problems can often be met and solved in this field simply by some basic re-thinking.

*And if "reverse Polish" symbol-order proves for some reason comprehensible only to formal logicians I have a final trump-card in the shape of a translation program which will accept a mathematical notation that is almost completely orthodox!

APPENDIX I

TABLE OF CODE-SYMBOLS

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	not used	0	/	//	a	q	(a)	(q)	π		x		sin		tms	R_b
1	,	1	disp	+	b	r	(b)	(r)	e	$\vec{1}$	rcp		cos		rep	R
2	;	2	neg	-	c	s	(c)	(s)	τ	$\vec{2}$	÷		tan			$R_{b/}$
3	*	3	mod		d	t	(d)	(t)	σ	$\vec{3}$	pow		cot			$R_{/}$
4	x	4	sig		ε	u	(ε)	(u)		$\vec{4}$	$\surd$		csc		$\int_8$	$R_{b//}$
5	]	5	int		f	v	(f)	(v)		$\vec{5}$	exp		sec		$\int_{16}$	$R_{//}$
6	end	6	frac	max	g	w	(g)	(w)		$\vec{6}$	log				$\int_{32}$	
7	buzz	7		min	h	x	(h)	(x)		$\vec{7}$	lg				$\int_{64}$	
8	unbz	8	dup	rev	i	y	(i)	(y)					$\sin^{-1}$		Σ	P_b
9	wait	9			j	z	(j)	(z)		1			$\cos^{-1}$		Π	P
10	=	10			k	α	(k)	(α)		2			$\csc^{-1}$		Σ_1	$P_{b/}$
11		11			l	β	(l)	(β)	ϕ_1	3			$\sec^{-1}$		Π_1	$P_{/}$
12		12		&	m	γ	(m)	(γ)	ϕ_2	4			$\tan^{-1}$			$P_{b//}$
13	<	13		$\equiv$	n	λ	(n)	(λ)	ϕ_3	5			$\cot^{-1}$			$P_{//}$
14	>	14	$\sim$	v	θ	μ	(θ)	(μ)	ϕ_4	6						
15		.	?		p	ν	(p)	(ν)	ϕ_5	7			ang			#

Notes

Column 0

"," used to separate numeral sequences.

";" cancels last occupied cell.

"*" and " " program place-markers: the latter is to be followed by a reference number.

"wait" stops program until manual signal given.

Column 2

"disp" displays contents of last occupied cell on output lights.

"neg" monadic operation taking negative of a number (cf. "−" diadic (operation denoting subtraction).

"int" and "frac" monadic operations giving integral and fractional parts respectively.

"dup" duplicates number from last occupied cell in next cell.

"~" Boolean negation.

"?" discrimination instruction, means "if number in last occupied cell is all ones ignore the next skip (any symbol of column 9)": last occupied cell is cancelled.

Column 3

"rev" interchanges contents of last two occupied cells.

"&", "≡", "v", "⊃" Boolean conjunction, equivalence, disjunction and implication respectively.

Columns 6 and 7 "Names"

These correspond with the variables of columns 4 and 5. A name transfers the contents of the last occupied cell (without cancelling) into the number-store.
Names can be suffixed in the same way as variables.

Column 8 Constants

"τ" truth (all ones).

"σ" sign-digit.

"ϕ_1" to "ϕ_5" digit-patterns for use as truth-tables of elementary propositions.

Column 9 "Skips"

"→" (preceded by number) skip to subroutine of indicated reference number.

"$\vec{1}$" etc. skip right to first (etc.) asterisk

"←" (preceded by number) skip to indicated reference number.

"1" etc. skip left to first (etc.) asterisk.

Column 10

"pow" diadic operation such that "a b pow" represents a^b.

"lg" diadic operation, logarithm to given base.

Column 12

"ang" diadic operation, angle of complex number.

Column 14 Repetitive operations

"tms" simple repetition a number of times given by preceding number, e.g. "n tms .....]".

"rep" repetition for increasing values of parameter,
e.g. "a b rep (j) .....]".

"$\int_8$" etc. definite integration by Simpson's rule, for different numbers of intervals, e.g.

"a b $\int_8$ (x) .....]".

"Σ" etc. repeated sum, product, logical sum and logical product respectively, e.g. "a b Σ (j) .....]".

Column 15

Read and punch, with or without conversion, to accumulator or vector store or matrix store.

"#" ends program read-in.

APPENDIX II

PROGRAMMING EXAMPLES

Of these examples the first six are simple illustrations of methods of using various symbols; the seventh is a "subroutine"; and the eighth is the main part of a working program.

1. $\sqrt{a^2 + 2ab\cos\theta + b^2}$ (result in accumulator).

```
a 2 pow a b 2 θ cos x x x + b 2 pow + √
```

2. Put $x = \frac{1}{2}(a + b)$, $y = \frac{1}{2}(a - b)$.

```
a b + 2 ÷ (x) ; a b - 2 ÷ (y) ;
```

3. Put $t_j = a_j^2$ $(j = 1, 2, \ldots, n)$.

```
1 n rep (j) j / a 2 pow j / (t) ; ]
```

4. $\sum_{i,j=1}^{n} a_{ij} x^i x^j$ (result in accumulator).

```
1 n Σ (i) 1 n Σ (j) i j // a x j pow x ]
                           x i pow x ]
```

5. $\frac{1}{s\sqrt{2\pi}} \int_a^b e^{-x^2/(2s^2)}\,dx$ (result in accumulator)

```
a b ∫ (x) x s ÷ 2 pow 2 ÷ neg exp ] s π 2 x √ x ÷
```

6. If x is odd, add one to y.

```
x 1 & 1 = ? 1 y 1 + (y) ; *
```

7. Iterate from initial value unity using the formula $x_{(n+1)} = \frac{1}{2}(x_{(n)} + y/x_{(n)})$ until $|y - x^2| \leqslant \varepsilon$ leaving result in accumulator.

```
1 * (x) y x ÷ + 2 ÷ dup 2 pow y - mod ε⩽ ? 1
```

8. Solve the linear equations

$$\sum_{j=1}^{n} a_{ij} x_j = a_{i,n+1} \quad (i = 1, 2, \ldots, n)$$

Crout's method is used.

```
1, 1 // a (p); 2 n 1 + rep (j) 1 j // a p ÷ 1 j // (a); ]
2 n rep (q) q n rep (i) i q // a 1 q 1 - Σ (k) i k //
a k q // a x ] - i q // (a); ] q 1 + n 1 + rep (j)
q j // a 1 q 1 - Σ (k) q k // a k j // a x ] - q q // a
÷ q j // (a); ]]
n n 1 + // a n / (x); 1 n 1 - rep (q) n q - (i);
i n 1 + // a i 1 + n Σ (k) i k // a k / x x ] -
i / (x); ]
```

Mr. R. Davis English Electric Co.

Mr. Hamblin said that schemes such as his would only apply to certain ranges of work. Would he indicate the range for which he thinks his scheme is suitable?

Mr. C. L. Hamblin (in reply)

It is not severely restricted - it has all the elementary operations of normal computers. The main difficulty appears to be in organisation of storage.

Professor T.M. Cherry University of Melbourne

Have you a symbol to show when a variable is finished with, so that the location may be used for other purposes?

Mr. C.L. Hamblin (in reply)

There is no provision for this. However, the same letter may be used for different variables at different stages in a calculation.

Dr. T. Pearcy Commonwealth Scientific and Industrial Research Organization.

I feel that Mr. Hamblin is asking us to learn a new language whereas automatic programming should use the language of mathematics.

Mr. C.L. Hamblin (in reply)

You cannot quite do this. Ordinary mathematics has things which cannot simply be coded into a programme. One has to learn an exact language.

Dr. S. Gill Ferrati Limited

I am not worried by the author's notation. I think it could be learnt more easily than some of the other schemes which have been suggested. But has the author considered using forward Polish notation and making the machine read it backwards?

Mr. C.L. Hamblin (in reply)

Yes, I have, but this system has certain advantages.