Annual Review in Automatic Programming

2

Edited by

RICHARD GOODMAN

Automatic Programming Information Centre, Brighton College of Technology

The Use of the Genie System in Numerical Calculation

J. K. ILIFFE

The Rice University Computer Project, Houston, Texas

SYMPOSIUM PUBLICATIONS DIVISION

PERGAMON PRESS

NEW YORK · OXFORD · LONDON · PARIS

1961

The Use of the Genie System in Numerical Calculation

J. K. ILIFFE
The Rice University Computer Project, Houston, Texas

1. GENERAL INTRODUCTION

THE set of codes designed for interpreting formal expressions on the Rice University computer is termed the 'Genie system', or simply 'Genie'. Although most of the concepts employed therein may be applied on any machine of the type used for a decade, it is probably true that intensive exploitation of a given set of orders and logical properties influences the code designer to a greater extent than he is aware, and it is appropriate to make some initial remarks on the properties of the Rice computer, and the circumstances in which the Genie system is used.

Design specifications were made in 1958 at a time when no machine was available with a word length suitable for extended numerical calculations; the choice of an instruction code for the machine using a 56-bit word has led to some features which, if not novel, at least do not appear to exist in such a combination on another computer.

The machine is binary, and has main memory of 32,767 words, principally in electrostatic storage which is parallel in operation. Access to the memory is shared by four magnetic tape units, a fast line printer and the central processing unit. Primary input is by punched paper tape prepared on an 88-character Flexowriter with 1/2-line super- and subscript shifts. The principal features which are relevant to the present discussion may be summarized as follows:

1. A *location number* is 15 bits in length. It represents, in one's complement notation, a numeral in the range (−16,383, +16,383).

2. The machine contains 8 conventional *B*-registers each of length 15 bits, one of which is the 'control counter' used in sequencing programs. Thus simple routines may be written in absolute code in completely relocatable form.

3. An *address* consists of 24 bits, divided into groups A_1, A_2 and A_3 of 1, 8 and 15 bits respectively. A location number is formed from an address in the following manner:

(*a*) The contents of up to 8 *B*-registers, selected by A_2, are added to the number represented by A_3.

(*b*) If A_1 is zero, the 15-bit number resulting from (*a*) is the required location number. If A_1 is one, the result of (*a*) is used to select a word in memory from which a 24-bit address is taken, and the process (*a*) is repeated, and (*b*) reapplied.

Thus indirect addressing to any depth with *B*-modification is permissible (Fig. 1).

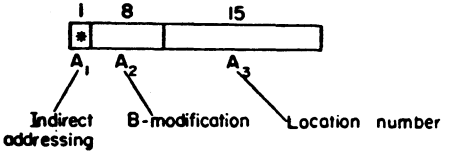

FIG. 1. An address.

4. An *instruction* is a 54-bit number containing up to three *operands* and two *orders*. An operand is specified either by an address or a *truncated* location number of 4 bits, which selects one of 16 special fast access (1 μsec) registers. Two of the operands may undergo sign modifications selected by the instruction. The location number determined by the address may itself be used as an operand, by using an immediate addressing option. An *order* is either a 15-bit *order code* or a part of an *auxiliary operation* which involves certain of the fast registers. In the order code are about 1000 non-trivial orders with perhaps 200 of these in regular use. The auxiliary operations are 64 in number, and include such frequently used orders as *B*-register incrementing and decrementing (thus including 'jump' orders) and transfers between fast access registers. It is our experience that the present organization combines many advantages of single- and three-address machines, achieving more compact programs than would be expected, even on the basis of the longer word length (Fig. 2).

5. Associated with each word in memory are two *tag bits* or labels which do not take part in its function as a number or instruction. They are detected separately in the arithmetic and control sections of the machine, and may be interrogated and set to 0 or 1 by conventional orders. Alternatively, they may be interrogated automatically when the machine is in the 'trapping mode' described in the next paragraph.

6. Under *normal* sequencing conditions, instructions are taken from consecutively numbered memory cells to the control unit for execution, unless a conventional 'transfer of control' takes place as a result of an order. In executing an instruction, two operands are brought to special registers in the arithmetic unit, the designated order is then obeyed, and

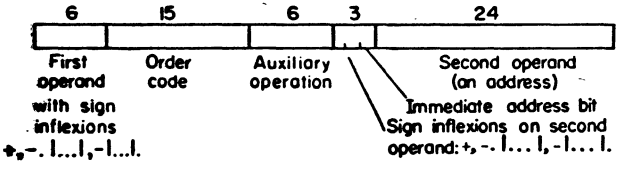

FIG. 2. An instruction.

finally the auxiliary operation is performed. When the machine is in the *trapping mode*, normal sequencing may be interrupted if certain conditions arise in the control or arithmetic units. The point at which interruption or 'trapping' occurs depends upon the intentions of the coder, but there seem to be three natural points at which the option should be provided:

(*a*) Immediately an instruction enters the control unit. This is useful if any part of the instruction requires an interpretation not provided in the machine hardware.

(*b*) Immediately an operand enters the arithmetic unit. This is useful again if the operand is of some 'non-normal' type, requiring interpretation.

(*c*) Immediately after an order has been executed, in case some particular condition, such as an overflow, has arisen.

All three possibilities are provided on the Rice machine. Cases (*a*) and (*b*) are normally controlled by tagged instructions and data. Apart from the obvious ability to employ these features in selective interpretive and tracing routines, they are of importance in the control of the Genie codes. In the latter respect a simulated interruption system would begin to lose practical value.

Some remarks on experiences in hand coding the machine are appropriate at this point, since they illustrate both the direction and purpose of our efforts in automatic coding. The instructions admit of a compact and meaningful symbolic representation which is used by almost all coders. Using the symbolic assembly program, correct codes can be written reasonably concisely and quickly if no more than a casual regard for optimization is taken. By 'optimization' we mean reaching an acceptable compromise between time and space requirements in a given code, such that a reduction in one cannot be reached without an unacceptable increase in the other. It involves principally the choice of the index registers and fast registers which will be used, taking into account the ability to use truncated operands and orders, and the requirements of subordinated or concatenated pieces of coding in these respects. The detailed optimization of code is an interesting and normally rewarding problem. At the same time, it frequently amounts to a combinatorial problem of such magnitude, even for short codes of 20 or 30 instructions, that the coder is willing to accept a solution he believes to be near-optimal rather than devote time to investigating all possibilities. The point to make here is that we consider all coding, hand or automatic, to take part in four stages:

(*a*) A description of an algorithm.

(*b*) An initial equivalence transformation of the algorithm dictated by the properties of the machine and the translation process.

(*c*) A translation from the algorithm description to a sequential machine code.

(*d*) A final equivalence transformation of the machine code based upon certain optimizing criteria.

In practice, it is rare to consider these stages separately, but since it is our purpose to assign the last three to a machine, and since each involves an amount of experimentation, it is advantageous to separate them. The present paper is concerned with stages (*a*) and (*c*), as far as they affect the user of Genie codes. In fact, some of the descriptive forms permitted in Genie are precisely those convenient for describing experiments on stages (*b*) and (*d*), which are the subject of continued investigation. Whilst refined optimizing processes are of considerable technical interest, and probably of future importance, it remains to be seen whether, even on a machine as versatile as the one we are concerned with, they will gain favour over crude but fast methods. However, it does seem possible that a good solution to the combinatorial problem can be attained at stage (*d*).

In the writer's view, the task of designing an automatic coding 'language' for a particular class of problems is for the user of the language, and not the designer of automatic coding systems. In the sense that he, too, has a class of problems that can be described formally, he is a user with prejudices of his own, but in the main he cannot anticipate the tastes of those using other formal languages beyond attempting to provide for as wide a choice of form as possible. Only in this way does it seem possible that an acceptable 'universal' language will develop in any particular branch of mathematics. Although the system may be anchored at one or two corners in safe theoretical grounds, it is supported in the main by its use and abuse: in the present case, the users are chemists, physicists and mathematicians, with a small number of large problems of an experimental numerical nature. If it appears that Genie is designed for coders with some fluency in the symbolic representation of algorithms, then the circumstances of its use may provide an explanation.

2. PARTICULAR ASPECTS OF GENIE ORGANIZATION

Genie is concerned in general with the definition of *objects* belonging to certain computable domains, and the execution of particular *operations* between or upon these objects. Both the objects and the operations are under the control of the coder, but if he can find an acceptable machine realization for them both, then he can call upon the mechanism of Genie to assist in a calculation. Applications of Genie have been made to domains of objects which include 'numbers', 'integers', 'names', 'symbols', 'equations', 'formulae', 'truth-values' and 'instructions';* generally, operations on such objects fall into closed groups so that we can talk of a 'language' in which operations between objects in a certain domain, say 'integers' or 'truth-values', take place. It is practical to give a description of such 'languages' in parametric terms, and although in the present paper particular reference is made to the conventional language forms of algebra, it should be understood that most of the constructions illustrated are operand independent in the sense that a given formal representation may be variously interpreted in a number of domains.

Formal calculation proceeds by *naming* objects of interest, and assigning *values* to such objects either by means of equations (which use names and operations), or, in the case of linguistic objects, by exhibiting specific examples of the objects. The fact that early applications of Genie have been made to linguistic objects should not obscure the more general conception of the system.

* Each defined in some specialized sense corresponding closely to general usage.

Two features are characteristic of most coding systems in active use at the present time and may be selected as ones which have been modified in Genie. They are as follows:

(*A*) The pattern of sequential coding imposed by machine conventions in the past decade still dominates algebraic coding languages. The 'instruction' has become a 'statement', but the only advance towards non-sequential description has been to permit the use of formulae and substitution-type equations.

(*B*) The 'translation' process is conceived in terms separate from the 'execution' of a program.

In their primitive forms, the deficiencies of both these features have long been recognized, and a number of attempts have been made to use the sequencing implicit in recursive definitions in order to derive more compact descriptions of algorithms. Any text on recursive functions supplies schemata which may form the basis of such descriptions, but an immediate attempt to allow descriptions of a general type falls into difficulties of another sort. Firstly, our machines are finite in both capacity and speed, and the type of program resulting from general recursive definitions may overstrain both these quantities. Secondly, from the point of view of practical description of algorithms, recursive definition is only useful up to a point where rapid visual comprehension is possible: beyond that, it is self-defeating, and a survey of any text on numerical procedures would bear this point out. Most expressions belong to a relatively small class of primitive recursive functions, and complicated algorithms are described by reverting to a sequential presentation of these expressions, although the proportion of these which consist of simple substitutional equalities is probably quite small.

One of the objectives of Genie, therefore, is to extend the basic forms of definition which can readily be understood, and simply and efficiently encoded. The precise form of these extensions is not of vital importance, but some examples of those in current use are given in the next Section.

An improvement on the second feature (*B*) is less easy to achieve, although its necessity can be demonstrated by various expedients currently in use to circumvent it. These include 'load-and-go' techniques, the 'subroutine library', 'subroutine package', the 'executive system' and various 'monitoring' systems. To recognize the similarities of all computing processes involves a unification of these ideas, and the removal of one of the main obstacles to efficient automatic coding, in its widest sense. In Genie the separation of processes of various types has been removed by

generalizing the concept of *evaluation* to include objects with values in any domain of interest, and by precise control of names to make evaluation an automatic process in cases where no ambiguity can arise. At its simplest level, this generalization can be illustrated by a numerical example. Suppose a and b have values 2·6 and −1·8 respectively. Then if a formula '$a + b$' is encountered in the course of constructing code, it will automatically be replaced by the simpler formula '0·8'. Similarly, if π is a given constant number, and sin a given constant function, the formula 'sin $(\pi/6)$' would automatically be replaced by the simpler formula '0·5'. Other consequences are:

(i) That a program is executed automatically if it has been defined and all parameters have been given values.

(ii) That an executive system of some power is included in the formalism of Genie; machine operation is a continuous process from program to program.

(iii) That storage control for vectors and matrices is continually exercised, space for arrays of variable size being taken only for the period of time in which they are used.

One of the most difficult problems of automatic coding, yet one to which a sophisticated solution is required when continuous evaluation techniques are used, is that of the identity of names in different parts of a description. A coder frequently divides his problem into segments, each logically independent, but referring to the same operands: in each segment of code, references to 'common' operands must be distinguished from purely internal names. He may use constant numbers and routines of his own definition, he may use routines from a 'library'; he may also use one or more of the languages of Genie; and finally his code may involve analytical manipulations which retain symbolic names. All these requirements put constraints on the way names can be handled internally. A solution is proposed in Genie, which is summarized in Section 4, which links symbol control to a hierarchy of definitions given to the machine. This is probably by no means a final solution, but it is feasible, and corresponds in an approximate way with conventional usage where the latter has any describable meaning.

3. STATIC PROPERTIES OF ALGORITHM DESCRIPTIONS

We shall now define, and illustrate by means of examples, the extensions which have been made in methods of making value assignments in Genie.

The examples will make use of three fundamental forms of expression which will be used with little further explanation:

(i) *Algebraic formulae*

Numbers and operation signs appear with their usual significance. Names stand for numbers, or sets of numbers, or for functions which assume numerical values.

*Example E*1. The five expressions which follow in quotation marks are algebraic formulae:

$$\text{`}-a + |b - kt|\text{'}, \quad \text{`}A_{i,i+m} + 3 \sin 2B_i\text{'}, \quad \text{`}K_{i_{j,s}}\text{'},$$
$$\text{`}\sum_{i=n}^{m} A_i \sin B_i t\text{'}, \quad \text{`}(a, b, c) - 2{\cdot}535G\text{'}.$$

(ii) *Predicates*

Relations and Boolean connectives appear with their usual significance. Names stand for propositions which assume the values 'true' or 'false'.

*Example E*2. The four expressions which follow in quotation marks are predicates:

$$\text{`}a < b^2 \text{ and } b \neq m - 1\text{'}, \quad \text{`}x < 0 \text{ or } x \geq 1\text{'},$$
$$\text{`}(x < 0 \text{ and } y < 0) \quad \text{or} \quad x \geq 0\text{'}, \quad \text{`}-a \leq x \leq a\text{'}.$$

Note that in the absence of parentheses the operation *and* takes precedence over *or*, so that in the third example the parentheses may equally be omitted.

(iii) *Program Schemes*

These are defined basically in the same sense as those of Yu I. Yanov (Ref. 1).* Names stand either for elementary operators or for predicates.

* *Editorial Note:*

If $A_{i_1}, A_{i_2}, \ldots, A_{i_s}$ are operators and p is a predicate, the line

$$A_{i_1} p \underset{m}{\lfloor\!\_} A_{i_2} \ldots \underset{m}{\_\rfloor} A_{i_3} \ldots$$

is read as follows:

Execute A_{i_1}; test p; if true ($p = 1$), execute A_{i_2}, etc.
if false ($p = 0$), execute A_{i_3}, the operator to the immediate right of the right stroke $\underset{m}{\_\rfloor}$ corresponding to the left stroke $\underset{m}{\lfloor\!\_}$ immediately following p.

Yanov uses O to denote the identically false predicate.

An alternative notation, adopted by Lyapunov, uses $\overset{m}{\uparrow}$ and $\underset{m}{\downarrow}$ in place of $\underset{m}{\lfloor\!\_}$ and $\underset{m}{\_\rfloor}$ respectively and w for the identically false predicate.

See Yu I. Yanov, 'Logical Schemes for Algorithms' (*Problems of Cybernetics* I, pp. 82 *et seq.*)—R.G.

Two primitive operators are used, the 'right stroke' $\underset{i}{\rfloor}$ and 'left stroke' $\underset{i}{\lfloor}$: other special signs may be defined in terms of these. The 'conditional operator' consists of a predicate name followed by a left stroke.

*Example E*3. The following three expressions in quotation marks are program schemes:

$$\text{`}\underset{2}{\rfloor} A\alpha \underset{1}{\lfloor} B0 \underset{2}{\lfloor} \underset{1}{\rfloor}\text{'}, \quad \text{`}\alpha \underset{1}{\lfloor} \beta \underset{2}{\lfloor} A \underset{1}{\rfloor} \underset{2}{\rfloor}\text{'}, \quad \text{`}I \underset{2}{\rfloor} \alpha \underset{1}{\lfloor} AD0 \underset{2}{\lfloor} \underset{1}{\rfloor}\text{'}$$

(The reader may verify that these program schemes are identical with those given in the 'block diagram' form of Fig. 3.)

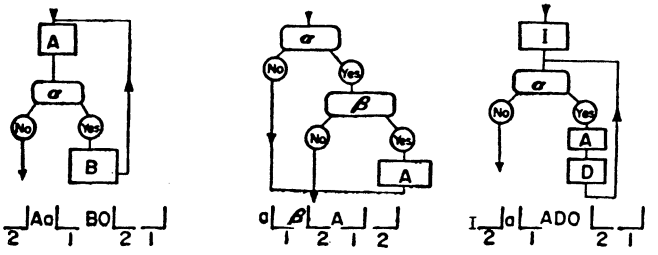

FIG. 3. Block diagrams and program schemes.

Effective calculation proceeds by assigning values in a particular domain to named objects. The elementary means of doing this is provided by an *equation.*

*Definition D*1. An equation is an expression of the form:

$$\alpha \approx \mathfrak{F} \tag{1}$$

where α stands for a (possibly subscripted) name, and $\mathfrak{F}$ stands for a formula.* The sign '$\approx$' is one of general equality, and in any particular instance of an equation it is replaced by a particular equality sign which indicates the type (i.e. domain) of object being defined.

*Example E*4. The expression on the next line is an equation:

$$H = 4kt^2 + gt/2$$

In this, the sign '=' is the particular equality sign associated with algebraic formulae, of which an instance appears to the right. It is inferred from the equation that H has, *in some context*, the numerical value assigned to the formula on the right-hand side. The words in italics are important

* For concrete examples, the words 'algebraic formula' (as illustrated above) should be understood here.

and will be given a more precise meaning later (see *D*9). Another example of an equation is on the next line:

$$M \quad \text{if} \quad a < 0 \quad \text{or} \quad b \geq 10$$

In this case, 'if' is a particular equality sign associated with the predicate on the right-hand side. It is inferred that M assumes the truth value of the predicate in a certain context.

*Definition D*2. A conditional equation is an expression of the form:

$$\alpha \approx \mathfrak{F}_1 \text{ if } \mathfrak{P}_1, \mathfrak{F}_2 \text{ if } \mathfrak{P}_2, \ldots, \mathfrak{F}_n \text{ if } \mathfrak{P}_n, \mathfrak{F}_0 \tag{2}$$

where α is a (possibly subscripted) name, and $\mathfrak{F}_i$, $i = 0, 1, \ldots, n$ stands for a formula, $\mathfrak{P}_i$, $i = 1, 2, \ldots, n$ stands for a predicate, n being a fixed integer in any particular instance of a conditional equation. The construction '. . .' is used here as an extra-linguistic device in a sequence, implying the presence of all terms up to the nth.

*Example E*5. The next line contains a conditional equation:

$$L = 1{\cdot}0 \text{ if } x < 0, \quad 1{\cdot}0 - x \text{ if } 0 \leq x < 1{\cdot}0, \quad 0$$

The interpretation of a conditional equation is that it is scanned from left to right until a true predicate is found. If this is $\mathfrak{P}_j$, then α is assigned the value of $\mathfrak{F}_j$. If no true predicate is found, the α is given the value of $\mathfrak{F}_0$. In case $\overset{n}{\underset{i=1}{V}} \mathfrak{P}_i$ is true, $\mathfrak{F}_0$ may be omitted; it may also be omitted in case α has previously been evaluated, and the value is to be unchanged if $\overset{n}{\underset{i=1}{V}} \mathfrak{P}_i$ is false.

*Definition D*3. A *preceding values recurrence scheme of order r* consists of a set of not more than $r + 1$ equations (or conditional equations) which define values of a *simply subscripted* name which we shall denote by α_i. One of the equations is a *recurrence relation* defining α_i in terms of α_{i-1}, $\alpha_{i-2}, \ldots, \alpha_{i-r}$; the remaining equations define the *initial values* α_0, $\alpha_1, \ldots, \alpha_{r-1}$. If a particular initial value is not defined it is assumed to be zero. By the term 'simple subscript' is meant a single name, standing for an integer.

*Example E*6. The following is an example of a preceding values recurrence scheme of order 2:

$$u_i = xu_{i-1} + u_{i-2}, \quad u_1 = 2 - 4xy$$

It should be noted that the quantity which is defined in a recurrence scheme is the general term 'α_I' for an integer value of I. Each use of the

general term causes an iteration to be made on the recurrence relation until the required term is found. The index 'i' is a dummy variable and may be replaced throughout by any other name provided it does not occur in the recurrence scheme.

The use of subscripts

It is appropriate to point out some properties of subscripts as they are used in Genie. In general, a name may stand for a single object, or a one or two dimensional array of objects from a particular domain. Depending on the organization of the array it may be treated as a *set*,* a *vector*, or a *matrix*. A subscript is any algebraic formula which assumes positive (non-zero) integer values. No name may have more than two subscripts but they may, where appropriate, contain subscripted names. Under certain circumstances an element of an array may itself be an array.

There are, however, two circumstances in which it seems profitable to employ a restricted subscript form. The first of these has already been described in the recurrence scheme. The second is connected with the minimalization operator used in the theory of recursive functions,† and has particularly strong applications in conjunction with the recurrence scheme. In it, if α is the name of a one dimensional array, we shall permit the form '$\alpha_{\mathfrak{P}}$' where $\mathfrak{P}$ stands for a predicate, with the meaning 'the first α in the sequence $\alpha_1, \alpha_2, \ldots, \alpha_i, \ldots$' such that $\mathfrak{P}$ is true. Normally, $\mathfrak{P}$ will involve elements of α, but we permit only those indexed by the dummy subscript 'i' and by constant differences '$i - 1$', '$i - 2$', . . ., '$i - r$': then i is identified with the index used in generating successive terms of the array.

Example E7.1. Let A be a vector. Then $A_{A_i<0}$ is the first negative element of A, and $A_{A_i\neq 0}$ is the first non-zero element.

E7.2. Let P be the predicate: $|u_{i-1} - u_i| \leq e$. Let u be determined by the recurrence scheme: $u_i = (u_{i-1} + H/u_{i-1})/2$, $u_0 = 1{\cdot}0$. Then the term u_P determines to an accuracy e a square root of the number H.

We are now in a position to complete the description of formal expressions in Genie.

Definition D4. A *primary definition* is an equation, conditional equation, or recurrence scheme.

* A *set* is an object of irregular structure, which is used, *inter alia*, for the representation of formulae. Details of its use and properties are outside the scope of the present paper.

† For example: $\mu[y; \mathfrak{P}(y)]$ with the meaning 'the least value of y such that $\mathfrak{P}(y)$ is true, y assuming the values 1, 2, ...'

*Definition D*5. A *dependent variable* is the name occurring to the left of the equality sign in a primary definition, disregarding its subscript. An *independent variable* is one of the names occurring in the formula(e) to the right of equality signs, or in the subscript of a dependent variable (disregarding dummy subscripts).

*Example E*8. The examples *E*4, *E*5 and *E*6 all give primary definitions. The dependent variables in these are H, M, L and u respectively. The independent variables are the sets (k, t, g), (a, b), (x) and (x, y) respectively.

In general, we shall represent a primary definition by an expression of the form:

$$\alpha \approx \mathscr{D}(\beta) \tag{3}$$

where α stands for the dependent variable, (β) for the set of independent variables, and $\mathscr{D}$ for the *definition schema* which is being used.

*Definition D*6. Consider the set of N primary definitions:

$$\alpha^{(i)} \approx \mathscr{D}^{(i)}(\beta^{(i)}) \qquad i = 1, 2, \ldots, N \tag{4}$$

The set is said to be *consistent* if all the $\alpha^{(i)}$, $i = 1, 2, \ldots, N$ are distinct (taking subsripts into account). The set is *cyclic* if the following condition holds:

$$\bigwedge_{i=1}^{N-1} (\alpha^{(i+1)} \in \beta^{(i)}) \quad \text{and} \quad (\alpha^{(1)} \in \beta^{(N)}) \tag{5}$$

*Example E*9. The following set of three equations is cyclic:

$$x = 2y + t, \quad t = 3m - 1, \quad m = \cos x$$

*Definition D*7. A *definition* in Genie is a set of M primary definitions such that (i) it is consistent; (ii) no cyclic subset exists; (iii) a unique dependent variable exists which does not occur as an independent variable.

*Example E*10. The following set of four equations is a definition:

$$x = 2y^3 + 3y - b, \quad y = a - b, \quad a = 4{\cdot}52, \quad b = 2{\cdot}91$$

In a definition, the unique dependent variable is the *principal variable*, (x in *E*10); the primary definition containing the principal variable is the *principal equation*. By convention, the principal equation is always written down first in the definition; using this fact it is possible to relax condition (iii) to allow the principal variable to occur elsewhere in the definition as an independent variable, so that in a suitable context a definition may be regarded as 'redefining' its principal variable in terms of a previously assigned value. The remaining primary definitions are

termed *auxiliary equations*, and their dependent variables (y, a and b in $E10$) are *auxiliary variables*. Let γ denote the union of the sets of independent variables occurring in a definition, excluding the names of auxiliary variables; then we shall write:

$$\alpha \approx \mathscr{D}(\gamma) \tag{6}$$

where α is the name of the principal variable, to indicate its dependence on the values assigned to names in γ. (In $E10$, γ is null.) For many purposes, a definition can be regarded as a generalized form of primary definition, and in fact it can be replaced by an equivalent primary definition if none of the auxiliary equations are recurrence schemes or function definitions ($D8$). Conversely, a primary definition can be regarded as a special case of definition, consisting of one element. The identity of form between (6) and (3) is therefore appropriate.

In the set γ in (6) (or β in (3)), values are assigned in the context in which a definition (or primary definition) is used. In the usual way, a subset π of γ may be chosen to represent *parameters* of a definition:

$$\alpha(\pi) \approx \mathscr{D}(\gamma) \tag{7}$$

In this case α may be used as a *function name* in formulae, with appropriate specification of parameter values.

Definition D8. A *function name* is the principal variable of a definition with which has been associated a set π of parameter names. A *function* is the operator mapping the set of objects designated by π into the single object which is the value of the function name. If, in (7), the sets π and γ are identical, the function is said to be in *library form*, a term closely corresponding with conventional usage.

Example E11. Let:

$$f(a, b, c, x) = ax^2 + bx + c$$

Then f is a function name with four parameters. Let A be a vector which can be written $A = (a, b, c)$. Then the function F, where

$$F(A, x) = A_1x^2 + A_2x + A_3$$

has two parameters, but assumes the same values as f. More generally, let B be a vector of length n; then consider the function P defined by:

$$P(B, x) = u_n, \quad u_i = xu_{i-1} + B_i$$

Clearly, $P(B, x)$ assumes the values of the polynomial $\sum_{i=1}^{n} B_i x^{n-i}$. Both f and F are library functions, but P is not, since it depends on the value of n, not listed as a parameter. One way of remedying this is to make use of

the special function 'dim' which, applied to a vector, assumes the integer value of the dimension of the vector:

$$P(B, x) = u_n, \quad u_i = xu_{i-1} + B, \quad n = \dim B$$

Strictly, P is still defined relative to the meaning of the name 'dim', but it is possible, as we shall see, to write a definition within a context in which a set of standard function names is 'understood' without explicit definition.

*Definition D*9. A *definition set* (*ds*) is a set of which each element is either a definition or a definition set. If there are M elements, $\mathscr{E}_1, \mathscr{E}_2 \ldots, \mathscr{E}_M$ in the set it will be written* as $[\mathscr{E}_1; \mathscr{E}_2; \ldots; \mathscr{E}_M]$. A definition set may be named for future reference, thus:

$$S[\mathscr{E}_1; \mathscr{E}_2; \ldots; \mathscr{E}_M] \tag{8}$$

denotes the definition set S.

All definitions in Genie occur within some definition set, although at the simplest level of coding the programmer may be unconscious of it. The significance of this is that a *context* is automatically provided in which a particular definition can be said to 'hold': *each principal variable which appears in a definition set has constant meaning throughout that set and in all definition sets which it contains; it has no meaning outside the given ds.*

*Example E*12. Consider the expression:

$$\begin{aligned} &S[\pi = 3{\cdot}14159;\ g = -32{\cdot}2;\ V = 1000{\cdot}0 \\ &E[Q(t) = vt + gt^2/2;\ v = V \sin(\pi/6) \\ &H = Q(2{\cdot}5)]] \end{aligned}$$

This consists of a definition set S containing elements π, g, V and E,† the first three being defined as numbers, and the last as a definition set with elements Q, v and H. Here, Q, v and H are defined within E but not within S; π, g and V are defined within S and within E. It has been assumed that S occurs within a 'higher order' definition set in which 'sin' is defined.

Some non-trivial calculations can be carried out with the help of definition sets, but in the main recourse must be made to description of algorithms by sequential definitions which are obeyed one after another, according to some sequencing rule. These are described by the following two constructions:

* In all written texts, the semicolon may be dispensed with by using a *line convention*, viz. each element is distinguished from the preceding one by placing it on a new line. In cases where a single definition extends over several lines, the text on the second and succeeding lines is indented by about 30 character positions.

† It is convenient to identify elements by their principal variables or names.

*Definition D*10. A *command sequence* (*cs*) is an ordered set of one or more definitions. If there are M terms $\mathscr{E}_1, \mathscr{E}_2, \ldots, \mathscr{E}_M$ in the sequence, it will be written* $\{\mathscr{E}_1; \mathscr{E}_2; \ldots; \mathscr{E}_M\}$. For reference purposes, a *cs* may be named, thus:

$$L\{\mathscr{E}_1; \mathscr{E}_2; \ldots; \mathscr{E}_M\} \tag{9}$$

defines the *cs* L.

*Definition D*11. A *program* is an explicit representation of a program scheme, using command sequences to describe the elementary operators, and appropriate syntactic constructions to replace left strokes, right strokes, and derived operators.

*Example E*13. The choice of representation of a program is very much a matter of taste, and a large class of apparently differing representations can be shown to be equivalent by simple transformations. The names of command sequences usually play a double role in such representations: they identify the elementary operators, and they serve as 'program points' controlling transfers of control. For this reason, each occurrence of a given command sequence is unique, and must be identified by a different name. Given a sequence of right strokes $\underset{i_1}{\rfloor}\ \underset{i_2}{\rfloor} \ldots \underset{i_\tau}{\rfloor} A$ in a *ps*, an obvious first step in obtaining a representation is to identify the index of each right stroke with the name A of the *cs* which follows these. Then each conditional operator can be replaced by an appropriate conditional or unconditional transfer, i.e.

'$\mathfrak{P}_j \underset{i_j}{\rfloor}$' is represented by: 'go to # A if not $\mathfrak{P}_j$', for $j = 1, 2, \ldots, \tau$

'$O \underset{i_j}{\rfloor}$' is represented by: 'go to # A'

The iterative operators formed by compounding left and right strokes are represented by a 'for . . . repeat' or 'for . . . repeat until $\mathfrak{P}$' constructions, $\mathfrak{P}$ standing for a predicate which determines the end of the iteration. With the addition of a 'stop' operator, this completes the list of elementary control representations. The use of the special sign '#' has several important applications: it is employed in '# A' to denote the *machine address* of the quantity A rather than the value of A. It is then possible to define a vector such as (# A, # B, # C) = K and select a branch of control by an index i as in 'go to K_i'. Another multiple branch is made with the help of a conditional equation:

$$\text{go to } \#A \text{ if } P, \ \#B \text{ if } Q, \ \#C$$

* As in definition sets, the semicolon can be dispensed with by using the line convention. It is customary to place the name of the *cs* in the left-hand margin of the text, alongside the first definition.

The analysis of programs follows conventional lines. The set of all principal variables occurring in command sequences is the analogue of the principal variable of a definition; a subset of these, termed the *output parameters*, is named to denote a particular group of computed quantities of interest. Similarly, of all the independent variables occurring in a program, a particular group of interest, named the *input parameters*, may be selected, and these play a role analogous to the parameters of a function definition. By selecting a single output parameter, and storing it in a fixed machine location, a program name may be used in the same way as a function name, to stand for its value in formulae. More generally, a program is conceived as an operator transforming a given set of input parameters, π_i, to the values of a set of output parameters, π_0. It may be used, with appropriately assigned parameters, in place of a *cs* in a program.

*Example E*14. The following is an example of a program.

$K(F, a, b, c, V, \#H) . = SEQ$
S function F
$I = V(b - a)$; $n = 1$; $h = (b - a)/2$; $J = h(F(a) + F(b))$
A $M = J + 4h \sum_{i=1}^{n} F(a + (2i - 1)h)$
go to $\#B$ if $|M - I| \leq c$
$I = M$; $J = (M + J)/4$; $n = 2n$; $h = h/2$
go to $\#A$
B $H = M/3$
END

Here K is a program with input parameters F, a, b, c and V, and a single output parameter H. One of the parameters is a function (F) and is declared to be so. The complete representation of the program scheme is parenthesized by the words SEQ and END. The symbol '. =' is the special equality sign associated with program schemes. It can easily be shown that this is a representation of a program scheme which can be written:

$$K(F, a, b, c, V, \mathrm{H}) . = S \underset{2}{\rfloor} Ap \underset{1}{\lfloor} \_ UO \underset{2}{\lfloor} \_ \underset{1}{\rfloor} B$$

where p stands for the predicate 'not $|M - I| \leq c$' and U for the unnamed *cs* on the fifth line of the program. The use of the summation operation 'Σ' in the *cs A* should be noted: since it is encoded in a very direct fashion it would normally lead to inefficient coding, but in this particular case no more efficient algorithm can be deduced in the absence of information about F. K is a library program in the sense defined above.

It could equally be put into the form of a function, since it has a single output parameter.

The range of definition of names occurring in a program raises some complicated problems concerning the habits of programmers, and it has been settled in Genie by reference to the context in which a program appears. This is essentially a dynamic problem, discussed more fully in the next Section, but in general terms the assumption is made that all names occurring in a program are internal to it, i.e. they have no meaning outside the limits of the program unless they have occurred *earlier in the context in which the program is defined.* Accordingly, *D*9 is modified as follows:

*Definition D*9.* A definition set (*ds*) is a set of which each element is either a definition or a definition set or a program.

In this way a set of programs can be written, sharing some common data (in a general sense, meaning any definable quantity, object or function or program) but otherwise being independent. Whether long parameter lists are used in communicating information between programs, or whether use is made of the 'common' region as defined by the context of a program is a matter which the programmer can decide. It should be noted that by naming the arithmetic and control registers as part of the context some optimization may be attempted by the coder, and compatibility with the symbolic assembly program is achieved.

4. DYNAMIC PROPERTIES OF ALGORITHM DESCRIPTIONS

At the highest organizational level, Genie is concerned with the interaction of two definition sets: one initially in the machine and one written by the coder. The result of the interaction is a new definition set in the machine and (possibly) some printed output information of interest to the coder. Logically, output control has a subordinate position in the system: it can be effected in a variety of ways of differing elaboration, subject to manual control or not, but the occurrence of an output command or subroutine has no effect on the process of evaluation, other than to delay it. On the other hand, the system can be regarded from one point of view simply as an input routine whose function is to read definitions from paper tape, realize them internally in preassigned binary form, and keep a record of the nominal interdependence of named objects in each context. New information is read from the input tape in 'units' of one definition, and each of these is analysed before proceeding to the next. The complete behaviour of the machine as an information processor is determined if its interaction with a definition in a given context, and its

interaction with the boundaries of programs and definition sets, is given, and we shall first summarize this behaviour.

A definition set may be regarded as a 'tree' (Fig. 4), the 'branch points' of which (*A*, *B*, *E*, *G*, *L*) are named definition sets and programs, the 'terminal' points (*C*, *D*, *F*, *H*, *K*, *W*, *X*, *Y* and *Z*) corresponding to definitions or command sequences. Given any named quantity (say *H*) in a tree, it can refer by name to any quantity on the same branch, or connected to it by a downward or sideways movement on the tree; (*H* may refer to *K*, *L*, *F*, *G*, *B*, *C*, *D*, *E* and *A*). The set of points so named is the *context* of the given quantity, in the sense defined in the last Section. (Clearly, the contexts of *H*, *K* and *L* are identical.) It is necessary for the

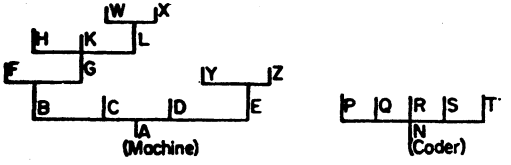

FIG. 4. Definition sets.

coder to describe the context in which his definition set (*N*) is to be interpreted. This is done by a special expression which precedes the *ds* when it is introduced to the machine. Suppose it is required to interpret *N* in the same context as *H*; then the correct expression would be:

$$\text{Context } A/B/G \tag{10}$$

This statement can be regarded as 'activating' a set of names in the machine *ds*, all other names being disregarded. The set thus activated constitutes the *symbol table*, *St*, which contains not only representations of symbolic names but also all pertinent information concerning the symbol. The following is a partial list of information retained in each entry of the *St*:

(i) A symbolic name.

(ii) Syntactic properties of the symbol, i.e. whether it is a parameter, fixed name (with a fixed location number), or name of general type.

(iii) The type of object for which the name stands: integer, program scheme, floating point number, truth-value, character, command sequence, etc.

(iv) Whether the name represents a scalar, vector, matrix or set.

(v) Whether the name stands for a function or not.

(vi) Whether the name is defined in a command sequence or definition set.

(vii) If defined in a *cs*, the address assigned to the object is retained. If defined in a definition set, either its formal definition or its actual value is retained.

In processing a definition set or program, *St* expands as new symbols are used: it is always assumed that a new symbol will be defined in the context in which it first appears; occasionally this leads to slightly artificial constructions when it is desired to place a name in a 'higher level' definition set than the one in which it first occurs.

*Example E*15. A common example of the need for giving meaning to a name outside the context in which it first appears is in writing a program to operate on some unspecified data, whose value will be given immediately prior to or during the execution of the program, where for some reason it is required not to place the name of the data on the input parameter list. In the absence of a detailed flow analysis, it is assumed in constructing a program that all newly named quantities have meaning inside the program only. Thus, suppose we redefine the program of *E*14 as a function with parameters *a*, *b* and *c*, taking the values of *V* and *F* from the definition set in which *K* is defined. A correct way of writing this is:

{Function *F*
Number *V*
$K(a, b, c) . = S\underset{2}{\_}\rfloor Ap\lfloor\underset{1}{\_} UO\lfloor\underset{2}{\_}\ \underset{1}{\_}\rfloor B\}$

—where all symbols have the same meaning as before, with the exception of the *cs B* which causes the value of the function *K* to be stored in a fixed machine location instead of at *H*. At a later time, appropriate definitions of *F* and *V* may be given, and then *K* can be used in a formula in the usual way. An interesting application of trap transfers on data arises in situations of this sort, for we may use one of the tag conditions to indicate that a named object has not been given a value; if an attempt is made to use it in a calculation, automatic trapping will occur to a routine in which corrective action is taken.

Figure 5 shows schematically the machine definition set after the interaction of the *ds*s of Fig. 4, in the context (10). Were this all the action of the machine, nothing more than a binary representation of the formal definitions would have been achieved, but it is in fact accompanied by continuous application of the evaluation principle illustrated in Section 2. We shall now indicate briefly what is meant by this.

The process of assigning a value to a name, which is the basis of all calculation, falls into two parts: firstly, determining in which domain

it takes its values; secondly, determining a representation of its value in that domain. One of the restrictions of the present codes is that there is a unique solution to each of these problems. One of the simplifications imposed is that the first problem is solved for each name by a process of *type evaluation* which is applied at the first occurrence of the name in a given context. Type can be determined in one of three ways:

(i) *By declaration.* This simply asserts that a name stands for an object in a particular domain. A declaration, if used, must always precede the first occurrence of a name in a definition or program.

(ii) *By assumption.* If the first occurrence of a name is in a formula, then in the absence of further information it is assumed to be of a type associated with that particular class of formulae.

(iii) *By implication.* If the first occurrence is as a dependent variable, its type can be inferred from the nature of the formula constituting the right-hand side of the equation.

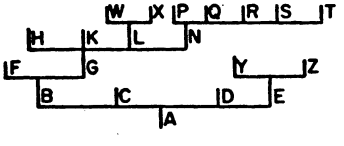

FIG. 5.

*Example E*16.1. Two examples of declarations follow:

Matrix P
Function f

*E*16.2. Referring to the elementary expressions whose existence was assumed at the beginning of Section 3: the value of a name in an algebraic formula is assumed to be a single precision floating point number; in a predicate it is assumed to be a truth value; in a program scheme it is assumed to be a command sequence, unless it immediately precedes a left stroke, in which case it is a truth-value. The value of a name occurring in a subscript in any formula is assumed to be an integer.

*E*16.3. Given an algebraic formula, it is possible to deduce the type of object it defines in a straightforward way from a knowledge of the type of each name in it and the properties of its unary and binary connectives. The latter are given in the so-called *Type Reduction Table*, an example of which, for the binary connective '+', is given in Fig. 6. Type derivations for other classes of formulae are similarly described. Note that

dimensionality is ignored in matrix and vector operations (any inconsistencies are detected during 'execution'), and also that in place of the 'invalid' combinations (Λ) it is a simple matter to substitute a program which gives diagnostic information.

The second stage of value assignment is analogous to type evaluation, although possibly more complex in application. Consider the following expression, which represents a definition of an object α by a schema $\mathscr{D}$, dependent on the values of a set of objects whose names are represented by γ:

$$\alpha \approx \mathscr{D}(\gamma) \tag{11}$$

We shall say that (11) constitutes a formal definition of α. By the methods of *E*16 the type of each member of γ can be determined. In order to

		Type of b			
		Integer	Number	Vector	Matrix
Type of a	Integer	Integer	Number	Λ	Λ
	Number	Number	Number	Λ	Λ
	Vector	Λ	Λ	Vector	Λ
	Matrix	Λ	Λ	Λ	Matrix

FIG. 6. Type reduction table for the connective '+' in the algebraic formula '$a + b$'.

determine the value of α it is necessary that values be assigned to each member of γ, that algorithms be given for reducing $\mathscr{D}$ to a sequence of steps each involving only formula evaluation, and that the properties of each unary and binary connective in a formula be given. The algorithms operate in an obvious manner, arranging the equations of a definition in the correct order for evaluation, and encoding conditional equations and recurrence schemes. The evaluation of a formula is controlled by a *Value Reduction Table* which gives, for each unary and binary connective, and each possible operand or operands, a Rule by which the result of application of the operation can be determined. Frequently, this Rule will be given in the form of a function defined by other languages, but ultimately, by continued application of the evaluation process, it will be expressed in the form of code executable by the machine.

*Example E*17. Consider a restricted class of algebraic formulae consisting simply of names and the two binary operations '+' and '×'. It is easy to derive, for any formula, and using conventional rules of precedence, an expression involving simply two functions $P(a, b)$ and $M(a, b)$ which, applied to any two objects of a suitable type, determine the values of the expressions '$a + b$' and '$a \times b$' respectively. These are

the value reduction tables. Thus, the formula $a + b \times c \times d + e$ reduces to $P(a, P(M(b, M(c,d)), e))$, i.e. the formula is reduced to a prefix notation. We shall evaluate the formula in a domain $\mathscr{B}$ whose elements are sequences of single address machine instructions expressed in symbolic form, and a domain $\mathscr{A}$ whose elements are machine addresses. In $\mathscr{B}$, instructions are written $\langle$OPN, *addr*$\rangle$ where OPN is one of the four mnemonics CLA, ADD, MPY and STO, and *addr* is an address. A code sequence is written as, e.g. $\{\langle$CLA, $a\rangle$; $\langle$ADD, $c\rangle$; $\langle$ADD, $d\rangle\}$. Let $t_1, t_2, \ldots$ denote an 'inexhaustible' sequence of working stores. We shall construct elements of $\mathscr{B}$ which use working stores in sequence during a calculation, always 'vacating' them at the end of the code. The address T will be used for the 'next available t_i' in the instruction $\langle$STO, $T\rangle$, and for the 'last used t_i' in $\langle$ADD, $T\rangle$ or $\langle$MPY, $T\rangle$. The predicates Code (X) and Name (X) are respectively true if X is in $\mathscr{B}$, $\mathscr{A}$. To each name α in a formula corresponds a unique α', its value in $\mathscr{A}$. Let μ, ν be elements either of $\mathscr{A}$ or $\mathscr{B}$. Then P is defined by:

$$\begin{aligned} P(\mu, \nu) = {} & \{\mu; \langle\text{STO}, T\rangle; \nu; \langle\text{ADD}, T\rangle\} \text{ if Code } (\mu) \text{ and Code } (\nu), \\ & \{\mu; \langle\text{ADD}, \nu\rangle\} \text{ if Code } (\mu), \\ & \{\langle\nu; \langle\text{ADD}, \mu\rangle\} \text{ if Code } (\nu), \\ & \{\langle\text{CLA}, \mu\rangle; \langle\text{ADD}, \nu\rangle\} \end{aligned}$$

$M(\mu, \nu)$ is similar, with ADD replaced by MPY. Then we have, after replacing each name in the formula by its value in $\mathscr{A}$:

$$\begin{aligned} & P(a', P(M(b', M(c', d')), e')) \\ & \quad = P(a', P(M(b', \{\langle\text{CLA}, c'\rangle; \langle\text{MPY}, d'\rangle\}), e')) \\ & \quad = P(a', P(\{\langle\text{CLA}, c'\rangle; \langle\text{MPY}, d'\rangle; \langle\text{MPY}, b'\rangle\}, e')) \\ & \quad = P(a', \{\langle\text{CLA}, c'\rangle; \langle\text{MPY}, d'\rangle; \langle\text{MPY}, b'\rangle; \langle\text{ADD}, e'\rangle\}) \\ & \quad = \{\langle\text{CLA}, c'\rangle; \langle\text{MPY}, d'\rangle; \langle\text{MPY}, b'\rangle; \langle\text{ADD}, e'\rangle; \langle\text{ADD}, a'\rangle\} \end{aligned}$$

The result is a code for evaluating the formula $a + b \times c \times d + e$.

The preceding example illustrates the method which is applied in all code constructing and evaluating procedures in Genie. Formalizing code construction as an evaluation process makes the contemplation of processes in which code construction and numerical evaluation are intermingled a relatively simple thing, and this is the usual mode of operation. In the above example, if a, b, c and d were numerically defined quantities, the construction of code would be followed immediately by its execution. The same principle can be formulated in another way which more accurately describes the realization of command sequences: *if an operation can be performed, perform it; otherwise construct code which will perform it at a later date.* A succession of 'scans' of a piece of code can be envisaged,

ultimately reducing it to a single order, but at the present time only two are attempted: one during code construction and the second one when it is known that all independent variables have been defined.

*Example E*18. The realization of the expression in *E*12 is equivalent to the following definition set:

$$S[\pi = 3{\cdot}14159;\ g = -32{\cdot}2;\ V = 1000{\cdot}0$$
$$E[Q(t) = vt + gt^2/2;\ v = 500{\cdot}0;\ H = 1149{\cdot}375]]$$

The obvious example of numerically defined quantities has a parallel in the case of programs, which resolves into the problem of whether to include a particular function in another program in the form of an open or closed subroutine. The problem is one of optimization and a serious attempt at solving it can only be made when all the information about a given program or programs is known. This is not generally the case, but in the special class of very short programs, a simpler criterion can be used: whether or not the program itself is shorter than the sequence of instructions necessary to set up its parameters, save index registers, and so on. If this is the case, it is included as an open subroutine.*

5. SOME ASPECTS OF THE EVALUATION PROCESS

In most respects, the evaluation programs associated with code construction act in a way which is a direct solution to the problem on hand, without recourse to detailed optimizing procedures. This is a temporary phase in constructing an automatic coding system of this type, resulting from the intention of using symbolic techniques to examine a variety of optimizing procedures and the practical necessity of getting the system, which is experimental in some ways, to work with minimum delay. In other respects, some of the traditional problems of coding have been avoided by the formalism of Genie, or may be avoided by asking a small degree of cooperation from the programmer. We shall first discuss these, and then describe the methods of array representation which are used.

Optimizing processes

(i) *Recognition of equivalent sub-expressions.* This matter is delegated without hesitation to the coder, who is better equipped for it than the machine. The use of a definition in place of a single equation is sufficient to collect equivalent terms together, and often has the distinct advantage of shortening the amount of code to be written.

* Or *macro-order*, to use the current term.

*Example E*19. By recognizing equivalent sub-expressions in the following equation:

$$y = \exp(a + b) - 3(a - c + b) + 2x(a + b)^2$$

the equivalent definition may be formed:

$$y = \exp(t) - 3(t - c) + 2xt^2, \quad t = a + b$$

(ii) *Optimal use of fast access registers.* As mentioned in Section 1, this problem can be formulated in precise terms, and it is probable that given any complete set of codes, a solution could be found in a realistic amount of time. One of the difficulties of continuous evaluation is that, without being specially told, it is impossible for the machine to decide when a set of codes (a definition set, for example) is complete, and the addition of a further program may invalidate the optimization. For the time being, therefore, the matter is again left to the coder, who may use the names of fast registers to stand for numbers in any formula. The result of using a fast register name for '*t*' in *E*19, for example, is to save four orders and about 100 μsec of calculation time.

(iii) *Optimal use of index registers.* A serious problem in code construction is that whereas a *B*-register may be used to contain the value of a named index controlling an iterative loop, it is possible that a transfer of control outside the loop may leave the index undefined in memory. Unless a flow analysis is made as in Fortran, one is left with the precautionary alternative of keeping the index 'updated' in memory as well as in the *B*-register. In the present formalism there are three important occasions in which an iteration is performed without the possibility of a transfer taking place: the use of the 'Σ' operator in formulae; the use of recurrence schemes; and the use of a postfix 'for . . .' construction which is permitted to follow a single definition with subscripted principal variable, e.g.

$$y_i = A_i - xb_i, \quad \text{for} \quad i = 1, 2, \ldots, n$$

These three instances form a significant proportion of the commonly constructed iterative loops, and in encoding them more efficient methods can be used than the single 'for . . .; . . .; repeat' construction. (It can easily be seen that they correspond to the cases where a dummy index is in use.)

(iv) *Optimization of working storage.* Given any definition or program the named auxiliary variables are distinguished from the 'unnamed' quantities developed in the course of formula evaluation, and they receive different address allocations. The space required for auxiliary quantities is taken immediately prior to program execution from a 'free storage region' controlled by an independent routine, and returned after the end of the

program. Thus if a program is not in use it does not take up memory space for its data. The cells required for temporary storage during formula evaluation are similarly placed external to all programs, and are shared as successive programs are evaluated. This treatment fulfils, incidentally, the requirements to be met on the few occasions when it is useful to execute programs recursively.

The actual minimization of the number of cells used is considered to be unimportant in this system, unless a number of large arrays are being handled, since the total effect over a number of formulae and programs is not cumulative.

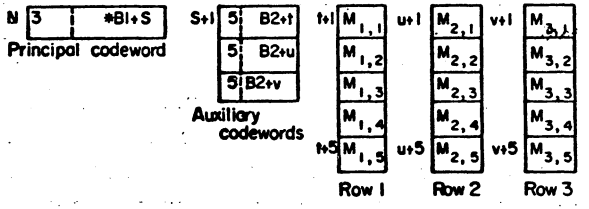

FIG. 7. Codewords for a 3 × 5 matrix.

The control of arrays

It is undesirable that the construction of code for handling arrays should depend on their size. This is avoided in Genie, as in other systems (Ref. 2), by the device of constructing one or more additional words to control selection of elements in the array. These so-called *codewords* are automatically generated when an array is defined. For a vector of length n, which is stored in n consecutive memory cells, a single codeword is required. For a matrix of m rows and n columns, $(m + 1)$ codewords are required, the first referring to a vector of m codewords, each of these referring to a vector of n elements which is a row of the matrix. The vector codeword, or the first matrix codeword, is termed the *principal codeword* of the array; its address plays a part in code construction analogous to that of the address of a simple numerical object.

*Example E*20. The main property of the codeword representation is illustrated with the help of Fig. 7, in which the codewords for a 3 × 5 matrix M are drawn. In order to address any particular element $M_{i,j}$ it is necessary to place i in B-register $B1$, and j in B-register $B2$. Then if N is the location number of the principal codeword, $M_{i,j}$ is obtained from the address $*N$, the asterisk denoting indirect addressing.

The codeword representation has several important consequences:

(i) All 'address calculation' at the time of execution is avoided: it is done once for all when the codewords are generated.

(ii) The space reserved for any array is precisely that which it occupies at a given time, and not a 'maximum size' specified by a dimension statement.

(iii) Elements of a vector may be moved to any part of storage, provided the codeword address is changed. As a particular case, programs in binary relativized form are instances of such vectors.

(iv) It is not required that matrices be stored in single blocks, only that their rows be in vector form.

(v) It permits certain 'virtual' operations, e.g. row interchange and transposition, without altering the position of any element in storage.

(vi) It does not require that the matrix rows be of equal length, and it can be adapted to the realization of symmetric and triangular matrices.

(vii) Only the address portion (24 bits) of each codeword is used in the indirect addressing sequence, and the remaining 30 bits and two tags can be used for storing other information, notably: (*a*) the size of the vector which is being referred to, and (*b*) the type of object it represents, such as Euclidean vector, polynomial, multilength number, etc.

*Example E*21. The special function 'dim' may be used in two ways with a vector argument:

$n = \text{dim } V$, asserts that n assumes the integer value of the number of elements in the (previously defined) vector V.

$\text{dim } V = n$, asserts that V is a vector of n elements, and as a result appropriate space reservation is made.

The main drawbacks to the codeword representation are the space occupied by the codewords, which is usually small, and the additional time taken by indirect addressing sequences. The latter can be reduced by placing codewords in the fast registers, and this is done in the manually constructed programs for basic operations on vectors and matrices; the problem of automatically doing this awaits investigation.

Implicit arrays

In any definition the principal variable may be simply subscripted, and the definition schema made to depend on the value of the subscript. Thus:

$$f_i(x) = x \quad \text{if} \quad \text{Even (i)}, \quad -x$$

defines a vector function, f, of unspecified size:

$$a_{i,j} = A_{j,i} \quad \text{if} \quad i \leq j, \quad -A_{j,i}$$

defines a matrix a in terms of a second matrix A. At present, such array

definitions are confined to those occurring in definition sets,* and they are logically equivalent to functions. However, this is a powerful syntactic device, closely resembling mathematical notation, and it seems convenient for that reason to retain the distinct form. In code construction, no distinction is drawn between implicitly and explicitly defined arrays. In program execution, the indirect addressing sequence is interrupted by a tag trap on the principal codeword of an implicit array, which sends control to the appropriate routine for calculating the value of the element. Thus algorithms operating on explicit arrays may be applied without change to implicit arrays.

6. CONCLUSION

It is difficult to compare one system of automatic coding with another: in most instances they differ sufficiently in purpose and economic circumstance to invalidate any possible criteria of comparison. At the same time, the power of symbol manipulative languages (instances of which appear in the present system) supports the view that most apparent differences of form in an algebraic language can almost trivially be rectified and those that remain are accounted for by the tastes of the code designer.

Differences of evaluation technique are not of first importance to the user of a system.† Indirectly, however, they will affect him to the extent that he requires flexibility and speed in problem solution. In this respect, the use of a hierarchy of definition sets in Genie is its key feature, leading to the application of the continuous evaluation principle, to the definition of *context*, and the formalization of automatic operating, debugging and symbolic correction schemes.

One of the main objectives in writing Genie is to permit the description of both numerical and analytical processes in the *same* formal system. It will be recalled that from a formal definition (11) the principal variable can only be evaluated if value assignments have been made to each independent variable. If, by some syntactic device, an evaluation is called for by the coder, the occurrence of a name with unassigned value will result in a failure of the evaluation process at that point (by a 'trap transfer'). This is not necessary, however, if provision is made for the

* The implication of our remarks is that *for all values of the subscripts*, the subscripted definition holds true; were this the case for definitions appearing in command sequences, an ambiguity of meaning would result, since the more natural meaning is that for the subscript values *currently defined*, the array element has a particular value. Thus implicit arrays may be used, but not defined, in programs.

† Although, on a machine of 12 digit accuracy, it becomes irksome to have to replace constants such as '3·9 sin (π − 2·5)' by their numerical values, and it is convenient to have them evaluated automatically during code construction.

formal manipulation of names within formulae, and there is no theoretical limit to the complexity of such processes which can be accommodated. A more difficult problem is that of placing formal manipulations within an unambiguous context, and our present investigations, although incomplete, indicate that again definition sets may be more suitable tools than programs.

7. ACKNOWLEDGEMENT

The work described in this paper was supported in part by funds supplied under National Science Foundation Grant G–7648.

8. REFERENCES

1. YANOV, YU. I., 'On Equivalence and Transformation of Program Schemes', *Dokl. Akad. Nauk, SSSR*, **113,** No. 1 (1957).
2. WELLS, M. *et al.*, MADCAP (A programming system for MANIAC II). See p. 115, this volume.