Applications to Automatic Coding - Part 2

The Elements of the Genie System

J.K. Iliffe

1. Introduction

The aim of the Genie system is to provide a set of routines to be used in constructing programs for The Rice Institute Computer in both numerical and symbol manipulative fields. Precise forms of coding language are not defined here since it is our intention to allow the coder a fairly wide choice of the forms he will use; however, as starting points in this inquiry, certain languages adequate for the description of numerical and analytical processes in a natural fashion were considered in detail, and by putting these in parametric form, it was hoped to achieve greater generality. The formula language F1l and the symbolic assembly program AP2 are examples of forms permitted in Genie,* and these will be used below for purposes of illustration.

* See Programming Memorandum #5.

It is clear that a large part of this type work is independent of any particular machine, and that it is to our advantage to keep it this way as far as possible. Consequently, most of the characterization of 'source languages' which follows is machine independent up to the point where a realization is chosen for a particular computer: the choice of representation and provision of basic sequences of machine code are matters for intervention by a coding specialist, al-

though the problem of minimizing the amount of work required in this respect is receiving attention. The point here is that the realization is no concern of the average customer.

Construction of symbol manipulating routines is in its infancy: much work is being done and some is documented, but none has received such general acceptance that we can refer to this in order to shorten the task of description. Unfortunately, therefore, much of the present memorandum consists of statements of the obvious which are necessary to illustrate an approach to the subject rather than to advance new facts. By its very nature a description (using symbols) of symbol manipulative routines gives rise to speculations, of which some are interesting and others lead to endless tail-chasing: most of these are latent in the structure of Genie, awaiting further investigation. Our primary objective, which is the production of an efficient translating system of some sophistication, does not permit diversion to these at the present stage.

It may be worthwhile, however, to try to make one aspect of our approach explicit; and this is concerned with the distinction between syntactical and formal systems which is found in logic, although the following presentation is far from the ultimate logical precision which one would hope to achieve. Whereas both systems ascribe predicates to certain sets of objects, they differ in the way in which the objects are understood. In the first (syntactical) case the objects are <u>words</u> which are strings of letters in an <u>alphabet</u>; in the second case, the objects are generated from primitive <u>atoms</u> by <u>operations</u>. If, following H. B. Curry[1], we call the objects of a formal system

the obs, then they consist of an inductive class whose basic elements are the atomic obs, such that the application of an operation of degree K to an ordered sequence of K obs is an ob. Obs constructed in different ways are in general distinct. A representation of the formal system is obtained by assigning a unique concrete object to each ob in such a way that distinct objects are assigned to distinct obs. Now it is possible to describe the properties of a syntactical or a formal system in a metalanguage which contains names for the objects of each system, verbs for the predicates, and so on. The unavoidable fact is, of course, that to communicate anything at all about the formal system, we must have a symbolic representation for it, and at first sight it then becomes difficult to distinguish it from a syntactical system. The distinction becomes clearer when one observes that a formal system is invariant with respect to changes in representation, so that, for example, no distinction is made between the propositional calculus in prefix or infix notation, or between an arithmetic formula as specified by a FORTRAN statement or as stored in a list structure in a machine.

As a matter of taste, the idea of a formal system is preferred here, and it affects the way Genie is constructed and described. As Curry[1] points out, syntactical systems can be reduced to formal ones by formalizing the operation of concatenation, so nothing is thereby lost and the structure of Genie is particularly designed for investigating more sophisticated languages than are currently allowed. It seems to the writer that from the practical point of view the present scheme has advantages over other systems in that the next step towards

sophistication is made at less cost.

Our objectives have thus been to produce a machine system whose external behavior is such that is recognizes certain descriptions of 'source languages' in a given canonical form and proceeds to read sequences of symbols which represent descriptions of computational procedures and, where possible, to 'execute' such procedures in a wide sense. Internally, the system is as general and homogeneous as possible, with an organic structure which permits continual growth and modification of the set of languages which is in use. It will become apparent that Genie differs from comparable systems in a number of ways, but these can mostly be traced down to a single significant change, viz, the elimination of the separate ideas of 'assembly', 'compiling', 'execution' (in the old sense), 'interpretation' and so on, by means of a general principle of evaluation which includes all these processes and allows them to be controlled automatically by the machine. This gives to Genie a more dynamic character and leads to some changes in the attitude of the coder to the machine which may loosely be described as putting the two in a 'conversational' frame of mind. To be sure, the old concepts are recoverable, but it is felt that this study may lead to advances in the use of parallel machines.

In the next section, some remarks are made on explicit and implicit sequencing of processes, and the behavior of machines in this respect is characterized. Such an analysis is in fact independent of the descriptive form chosen, and the definition of a class of such forms is delayed until Section 3. It is then necessary to determine the classes of objects which can be represented by the Genie languages, and the types of opera-

tions which can take place between them. These are chosen largely with our current interests in mind although in principle any other collection of objects and operations could be chosen. It is then possible (Section 4) to describe the evaluation process, which is common to all languages. In Section 5, the class of machines which we are interested in is introduced with particular reference to the Rice Institute Computer and the realization chosen for it.

2. Sequencing and Procedures

It is an accepted hypothesis that all effective procedures can be represented on one hand by recursive functions, or on the other by descriptions of Turing machines; while we wish to preserve a link with theoretical concepts, it is clear that neither of these extreme representations is suitable for practical or 'natural' descriptions of a procedure. The choice of description is a subjective matter, and the first observation concerning our current aims is that <u>within</u> Genie the coder can choose from a spectrum of forms running (almost) from one extreme to the other.

As the element of a procedure description, we shall take a <u>definition</u> and write it for the time being* as:

---

* In this Section, all symbols are part of the descriptive metalanguage, other than those appearing in expressions given as 'examples'. In this, Greek and script Roman letters are employed, together with primes and subscripts which extend the class of distinct symbols, and certain special signs are introduced. The numerals have their usual interpretation as integers.

$$\alpha = \mathcal{D} \qquad (2.1)$$

where α stands for the object defined by the <u>definitional schema</u> $\mathcal{D}$. We shall assume it is intuitively understood that (2.1) describes the way in which α is to be constructed from other objects and operations between them. If the other objects are $\beta, \beta', \beta'', \ldots, \beta^{(n)}$, constituting a set $\mathcal{B}$, we can demonstrate this by writing:

$$\alpha = \mathcal{D}(\mathcal{B}) \qquad (2.2)$$

For this definition to be effective, it is necessary that the β's be known; in other words, they also must be given definitions which are <u>auxiliary</u> to (2.2). Clearly a notion of <u>sequencing</u> is introduced at this point; the β's must be defined <u>before</u> α can be defined, and it is customary to exhibit this sequencing in two ways: (1) by a linear* (spatial or temporal) sequencing of definitions which directs the order in which they are to be obeyed; (2) by the implicit sequencing of recursive function definition, which we shall discuss first.

*Disregarding, for the time being, such two-dimensional presentations as flow charts and 'displayed' formulae on the grounds that these have to be linearized in the first place to get them into the machine.

Let us assume for the moment that the technique of formula evaluation to be described in Section 4 is understood. Then by an <u>equation</u> we understand a definition in which $\mathcal{D}$ (see 2.2) is a formula in the set of variables $\mathcal{B}$ (say). For practical purposes it is not possible to go to the lengths of allowing defini-

tions which are general recursive, but nothing is lost by this since all the functions of any use that we know of are primitive recursive, and in any case general recursive functions can be described by explicit sequencing methods. We shall therefore give as standard means of expression four definition schemata which are primitive recursive or directly reducible to primitive recursive forms(2).

Schema I. [Definition by equation]

" $\alpha = \mathcal{J}(\mathcal{B})$ "

where α stands for an object, $\mathcal{B}$ a set of objects, and $\mathcal{J}$ for a formula in one of the admissible languages* of Genie.

---

* In the sense to be described in Section 3.

Schema II. [Definition by composition]

"$\alpha_1 = \mathcal{J}_1(\mathcal{B}_1),\ \alpha_2 = \mathcal{J}_2(\mathcal{B}_2),\ldots,\alpha_r = \mathcal{J}_r(\mathcal{B}_r)$"

where for $i = 1,2,\ldots,r$, α_i stands for an object, $\mathcal{B}_i$ a set of objects, $\mathcal{J}_i$ a formula in one of the admissible languages of Genie and where no sequence of positive integers $K_1,K_2,\ldots,K_p$, $p \leq r$ exists such that:

(1) $K_i \neq K_j$ for $i \neq j$

(2) $\alpha_{K_{i+1}} \in \mathcal{B}_{K_i}$ for $i = 1,2,\ldots,p-1$

(3) $K_p = K_1$

(This condition is imposed to prevent circularity of definition). Schema II, the first equation is the principal equation, α_1 is the principal variable, the remaining equations are auxiliary equations defining auxiliary variables.

For the next two schemata, the conventional notation of sub-

scripting is introduced with the implication that the subscripted object belongs to a finite ordered set of objects, the ordering being controlled by applying the arithmetic successor relation to the subscript.

Schema III. [Preceding values recursion on one index]

$$\alpha = \mathcal{G}(\lambda_p), \lambda_i = \mathcal{J}(\mathcal{B}, \lambda_{i-1}, \lambda_{i-2}, \ldots, \lambda_{i-r}), \lambda_o = \mathcal{J}_o(\mathcal{B}_o),$$
$$\lambda_1 = \mathcal{J}_1(\mathcal{B}_1), \ldots, \lambda_{r-1} = \mathcal{J}_{r-1}(\mathcal{B}_{r-1})$$

where λ_o is the initial member of the set of objects λ and $\mathcal{G}$, $\mathcal{J}, \mathcal{J}_o, \ldots, \mathcal{J}_{r-1}$ are admissible formulae. Here, i and λ are auxiliary variables of the definition, r is a fixed integer which in any instance of Schema III gives the order of the recurrence relation, and p stands for a positive integer which may be determined either by an auxiliary equation or by a previous definition.

Schema IV. [Conditional definition]

$$\text{"}\alpha = \mathcal{J}_1 \text{ if } \mathcal{C}_1, \mathcal{J}_2 \text{ if } \mathcal{C}_2, \ldots, \mathcal{J}_\mu \text{ if } \mathcal{C}_\mu, \mathcal{J}_o\text{"}$$

where $\mathcal{J}_1, \ldots, \mathcal{J}_o$ are admissible formulae, $\mathcal{C}_1, \ldots, \mathcal{C}_\mu$ are predicates, and "if" is a special operation.

This completes the set of implicitly sequenced definitions. It is clear that the precise form is unimportant, and in fact the initial Genie schemata are slightly more complicated than these. It is our experience that a lot of fluency is to be gained by extending the forms of definition beyond simple equations in this way, and that this leads to more efficient machine codes than would be obtained by the same amount of effort expended on a less concise system. What is important in a system of this sort is that the ability of the machine to sort out implicitly sequenced definitions should be roughly comparable with that

of the machine user, and it is at this point that the practical and formal approaches to the description of calculations begin to diverge. There would be no formal difficulty, for example, in allowing any equation appearing in one of the above schemata to be replaced by instances of another (or the same) schema: the practical difficulties arise when consideration is given to the problem of identifier control which is discussed in the next paragraph and in trying to explain these to a potential coder. It must always be assumed that the coder is aiming at things other than linquistic elegance, and the less obtrusive the coding rules are the more useful they are likely to be.

A more difficult concept to formalize is that of the identification of objects whose names appear in different definitions. A simple solution, corresponding to current practice in most compiling systems, is to establish a unique correspondence between names and the objects which they identify, so that a given name can stand only for one object. This is objectionable on various grounds: firstly, in large problems, a genuine shortage of names may develop; secondly, it does not distinguish easily between the 'essential' and 'inessential' objects in a definition; thirdly, it does not provide for the discrimination in meaning which is dependent on the context in which a name appears; fourthly, it leads to crude generalizations in describing the interaction of a machine with a set of definitions. The rule (R1) for Symbol Table control which follows is an attempt to improve on this situation.

Rule 1A (Symbol Table control for implicitly sequenced definitions)

In a given Definition Schema of one of Types I - IV, the identi-

fiers of the auxiliary variables have meaning only within the definitions in which they occur, and may not be referenced from outside that definition. If, in (2.2), $\mathcal{A}$ denotes the set of auxiliary variables of the definition, then the set $\{\mathcal{B}-\mathcal{A}\}$ is termed, when non-empty, the set of external variables of the definition. Given a group $\mathcal{H}$ of definitions, its Symbol Table is obtained as the union of the sets of external variables, and the set of principal variables. It is important to note that, given any identifier appearing in the Symbol Table, it is either defined or undefined in $\mathcal{H}$. In the event that the set of undefined identifiers is empty, $\mathcal{H}$ is said to be complete.

It is also evident from R1A that the addition of another definition to a set such as $\mathcal{H}$ will in general change the categories into which its symbols fall: in order to prevent this happening, $\mathcal{H}$ may be 'closed', and we shall denote this condition by writing it in brackets, e.g. "$[\mathcal{H}]$". In this case, we have:

Rule 1B (Symbol Table control for sets of definitions)
Given a closed set of definitions $[\mathcal{H}]$, the identifiers of the principal variables have no meaning outside $[\mathcal{H}]$. Only the undefined identifiers of $\mathcal{H}$ constitute the external variables of $[\mathcal{H}]$.

A certain amount of non-trivial calculation may be controlled by taking sets of definitions from Schemata I - IV, but when all else fails it is natural that recourse should be made to explicitly sequenced definitions, particularly where complicated iterative procedures are involved. It is also natural that the formalization of this idea should bend towards the established

practice of writing programs for sequential machines. Consequently we have:

Schema V. [Sequential definition]

$$\alpha = \{\alpha_1 = \mathcal{D}_1(\beta_1);\ \alpha_2 = \mathcal{D}_2(\beta_2);\ldots;\alpha_n = \mathcal{D}_n(\beta_n)\}$$

where the parentheses "$\{\ldots\}$" enclose an ordered set of n definition schemata taken from Types I - IV, and α stands for the set of external variables determined by R1C below. Associated with a sequential definition is a Sequencing Rule (R2) which is given after the discussion of functional forms which follows.

Rule 1C (Symbol Table control for sequential definitions)

The external variables of a sequential definition (sd) are defined by reference to a list $\mathcal{L}$ of identifiers which is assumed to be given immediately prior to examining the sd. Then only those identifiers appearing in $\mathcal{L}$ and in the sd constitute the external variables. Within the sd, auxiliary variables may be identified with members of $\mathcal{L}$ or with principal variables which appear earlier (to the left of them) in the given sequence. No 'definition' appearing in the sd is regarded as such in the static sense that we have assumed with regard to Schemata I - IV; for this reason, sub-definitions which appear inside sequential definitions will be termed commands.

Functional definitions

From the preceding rules and remarks associated with them, it is possible to deduce the identifiers which are external to any given definition. Let (2.2) be such a definition, and let $\{\beta - a\}$ denote its external variable set. It will be seen in Section 4 that (2.2) cannot be evaluated until values are assigned to each external variable. Then in the sense that the

definiend is itself external to the definition schema which defines it, we can regard incomplete schemata as functions from a class determined by one subset of the external variables (the arguments) into a second class determined by a second subset (the results). In Schemata I - IV we shall adopt the usual convention of listing the arguments in parentheses following the definiend or result; in Schema V, both arguments and results will be listed, following a new identifier which is used to stand for the sd itself.

Example: $f(x) = a + bx,\ a = \sin x,\ b = \cos x$

This constitutes an instance of Schema II in functional form with argument x and result f. It is still incomplete, since it depends also on the identifiers "sin" and "cos".

Example: $P(a,b,c,\#s,\#t) = \{x = b^2 - ac;\ x = x^{1/2};\ s = (-b + x)/a;\ t = (-b-x)/a\}$

This constitutes an instance of Schema V in functional form with arguments a,b and c, and results s and t. It is complete apart from the identifiers "2" and "1/2". Note the use of the sign "#" to identify the results. "P" is the identifier for the sd.

The functional use of Schemata I - IV identifies the result as a <u>function letter</u> which may be used in formulae as described in the next section. The functional use of Schema V identifies the name of the sd as a <u>procedure name</u> which may be used in the final basic schema:

<u>Schema VI</u>. [Definition by procedure]

$$\Gamma(\alpha,\#\beta)$$

where Γ is a procedure name, and α stands for the set of arguments and $\#\beta$ for the set of values.

Any functional definition which contains no external names other than the given parameters is said to be in library form. If this is not the case, but the external identifiers all belong to a certain set of symbols $\mathcal{A}$, then the definition is in library form with respect to $\mathcal{A}$. This corresponds closely with current usage. We are now in a position to give R2, which takes the point of view that all functional definitions are special cases of sequential definitions, consisting of a (sequence of one) single command.

Rule 2. (The General Sequencing Rule)

In order to control the use of function and procedure names, the sequencing process is described in an inductive manner.

Basis: An ordered set of pairs <M, J> is to be constructed, initially consisting of the single pair <0, 0>.

Induction step: When it is necessary to evaluate a definition Γ, the number of pairs is increased by one by adding the pair $\langle \Gamma, 1 \rangle$ to the set. Then the first definition of Γ is executed. After executing each definition, the second member of the last pair is advanced by 1, and the corresponding new definition is executed. This procedure is terminated by the special "return" definition which causes the last pair to be removed from P before the next definition is selected. The whole process terminates when the 'zero' definition is encountered.

In current terminology, P is a slightly generalized 'path-finder' list, and the last J at any point is the 'command counter'. It is essential in programming practice, of course, that the 'command counter' be available to be used as a variable in commands of the sd, and we shall assume this is the case. It is

not usual for all of P to be arbitrarily accessible, only that the 'last' entry be known at any time. In giving examples, and in actual coding, we shall identify the command counter's value symbolically at any point to which it is necessary to refer to it.

To summarize the position, it will be recalled that in order to manipulate certain unspecified classes of objects, a set of six descriptive forms or definitional schemata has been specified, dependent upon further analysis of formulae which will be given in the next section. Among the schemata are both primitive recursive forms, and sequential definitions controlled by an inductive sequencing rule which permits the use of function and procedure names. Rules have also been given for the identification or discrimination between objects appearing in separate definitions by means of the list of external variables (the Symbol Table).

It is now necessary to examine the permitted structure of formulae.

3. Formula Syntax

The remarks of the foregoing section have been made as far as possible without presupposing the existence of any symbolic representation, but of course this is essential for practical purposes. It is computer-oriented in some degree, since syntactic structures which require many 'scans' for effective recognition have not been permitted, and what has been aimed at is a maximum degree of flexibility with a single (left-to-right) formula scan.

Consider a fixed and finite set of characters or marks $\zeta_1, \zeta_2, \ldots, \zeta_K$ forming an alphabet $\mathcal{A}$. In examples, we shall use upper and lower case Roman letters, together with Arabic numerals and certain arithmetic signs and parentheses as specific illustrations of members of $\mathcal{A}$. Unless otherwise specified, the lower case Greek letters will be used as variables taking values in the domain $\mathcal{A}$. 'Script' Roman and upper case Greek letters will be used in various ways which will be defined as they are introduced. Finally, subscripts may be used on any symbol to distinguish that one from others taking values in the same domain, or in the manner already illustrated in recurrence equations, Schema III.

By a <u>string</u> we mean a finite sequence of marks in $\mathcal{A}$. We shall take it that the ideas of 'first' and 'last' members, of predecessor and successor relationships, and of 'occurrence' of one string within another are intuitively understood. We shall use "Λ" to denote the null (empty) string, with no members. To use strings in the representation of formulae, we shall associate with each ζ in $\mathcal{A}$ three parameters: the <u>type</u>, $\mathcal{J}(\zeta)$, the <u>subtype</u>, $\mathcal{U}(\zeta)$ and the rank $\mathcal{V}(\zeta)$. Where no ambiguity can arise, we will use the abbreviation $\mathcal{J}_i$ for $\mathcal{J}(\zeta_i)$. The significance of the parameters is implicit in the following rules.

<u>Symbolic characters</u>

If $\mathcal{J} = 1$, then ζ is symbolic. The symbolic characters are used in the construction of names according to:

<u>Definition 1</u>

The occurrence of the string $\eta = \alpha_1\alpha_2\ldots\alpha_p$ in the string $\beta\eta\gamma$ is a <u>name</u> if and only if the following conditions hold:

(i) $\mathcal{I}(\alpha_1) = 1$

(ii) If $p \geq 2$, $\mathcal{I}(\alpha_j) = 1$ or 2 for $j = 2,3,\ldots,p$

(iii) If $p \geq 2$, $\mathcal{U}(\alpha_1) = 0$

(iv) $\mathcal{V}(\alpha_j) \leq \mathcal{V}(\alpha_{j+1})$ for all $1 \leq j < p$

(v) Either $\beta \equiv \Lambda$ or $\mathcal{I}(\beta) = 1$ and $\mathcal{V}(\beta) < \mathcal{V}(\alpha_1)$
or $\mathcal{I}(\beta) = 1$ and $\mathcal{U}(\beta) = 1$
or $\mathcal{I}(\beta) \neq 1$

(vi) Either $\gamma \equiv \Lambda$ or $\mathcal{I}(\gamma) = 1$ or 2 and $\mathcal{V}(\gamma) > \mathcal{V}(\alpha_p)$
or $\mathcal{I}(\gamma) \neq 1$ or 2

To aid in understanding the above, we remark that name construction depends on whether a character, standing alone, is to be taken as a name (sub-type 1), or whether it can be joined by succeeding characters on the right. In the latter case, a method of ranking determines the length of the name, i.e., characters belonging to the same name must be of non-increasing rank, reading from left to right across the string.

Numeric characters

If $\mathcal{I} = 2$, then $\mathcal{S}$ is numeric. The numeric characters are used in the construction of numerals according to:

Definition 2

The occurrence of the string $\mathcal{R} \equiv \alpha_1\alpha_2\ldots\alpha_p$ in the string $\beta\mathcal{R}\gamma$ is a numeral if, and only if, the following conditions hold:

(i) $\mathcal{I}(\alpha_i) = 2$ for $i = 1,2,\ldots,p$

(ii) Either $\beta \equiv \Lambda$ or $\mathcal{I}(\beta) = 1$ and $\mathcal{U}(\beta) = 1$
or $\mathcal{I}(\beta) = 1$ and $\mathcal{V}(\beta) < \mathcal{V}(\alpha_1)$
or $\mathcal{I}(\beta) \neq 1$

(iii) $\mathcal{I}(\beta) \neq 2$ and $\mathcal{I}(\gamma) \neq 2$

In the Genie system, numerals are representations either of natural integers or of the limited range of fractional quantities handled by a computer. Unlike the names, numerals are unique representatives, whereas the names, by means of definitions, can be made to stand for any one of a specified set of quantities. If the specified set has just one member, we can talk of the 'constant name' or just 'constant'; otherwise, a 'variable name' or 'variable'.

Definitions 1 and 2 permit a given string to be scanned once to recognize in a unique way the names and numerals occurring in it. A simple procedure for doing this can be given. Let us assume, for practical purposes, that there exist preassigned upper limits on the lengths of strings which can be distinguished from one another, i.e., if two numerals are identical in their first ν characters, then they are regarded as standing for the same number no matter how many additional characters each may contain, and similarly for names with a constant length μ or more. It follows that the totality of distinct numerals is finite, though possibly large, and similarly for names. Let $\mathcal{N}$ denote the set of names. Let $\mathcal{R}$ denote the set of numbers.

Now construct the alphabet $\mathcal{A}^*$ from $\mathcal{A}$ by removing all characters of type 1 or 2 and replacing them by a set of symbolic characters of sub-type 1 in (1-1) correspondence with members of $\mathcal{N}$, and by a set of numeric characters in (1-1) correspondence with members of $\mathcal{R}$. In what follows we shall consider strings formed from $\mathcal{A}^*$.

Operations

If $\mathcal{J} = 3$, then $\mathcal{Z}$ is an operation. The operation characters

are further classified by sub-type:

(i) If $\mathcal{U} = 0$, then $\mathcal{S}$ is a <u>punctuation</u> operation.

(ii) If $\mathcal{U} = 1$, then $\mathcal{S}$ is a <u>unary</u> operation.

(iii) If $\mathcal{U} = 2$, then $\mathcal{S}$ is a <u>binary</u> operation.

(iv) If $\mathcal{U} = 3$, and if the preceding character is type 1 or 2 or right parenthetic (see below), then $\mathcal{S}$ is a binary operation. Otherwise it is unary.

<u>Parenthetic characters</u>

If $\mathcal{I} = 4$, $\mathcal{S}$ is a parenthetic character. At each occurrence, it is either <u>left</u> or <u>right</u>, but this may depend on the context in which it appears. It is therfore necessary to describe parentheses in terms of the complete processing of a given string, say $\mathcal{A} = \sigma_1\sigma_2\ldots\sigma_s\Lambda$. To do this, we shall construct an auxiliary string $\mathcal{G}$ of 'unmatched' left parentheses, by the following rules:

(1) Let $\mathcal{G} = \Lambda$ initially

(2) If $\mathcal{A} = \Lambda$, proceed to step (5). Otherwise, let σ be the first character of $\mathcal{A}$. If $\mathcal{I}(\sigma) \neq 4$, go to step (4).

(3) If $\mathcal{U}(\sigma) = 0$, σ is l.p. (left parenthetic) and it is added to $\mathcal{G}$. If $\mathcal{U}(\sigma) = 1$, σ is r.p. (right parenthetic); let g be the last character added to $\mathcal{G}$. Then if $\mathcal{V}(g) = \mathcal{V}(\sigma)$, σ is 'accepted', and g is removed from $\mathcal{G}$; otherwise an 'alarm' condition is set up and σ is 'rejected'. If $\mathcal{U}(\sigma) = 2$, σ is said to be <u>conditionally symmetric</u>, i.e., if $\mathcal{V}(g) = \mathcal{V}(\sigma)$ then σ is r.p. and action is taken as for $\mathcal{U}(\sigma) = 1$; otherwise, σ is taken to be l.p. as for $(\sigma) = 0$. Finally, if $\mathcal{U}(\sigma) = 3$, σ is <u>conditionally asymmetric</u>. In this case <u>two</u> ranks $\mathcal{V}$ and $\mathcal{V}'$ are associated with σ. Then if $\mathcal{V}'(\sigma) = \mathcal{V}(g)$, σ is r.p., and otherwise, it is l.p.

(4) σ is removed from the beginning of $\mathcal{S}$, and step (2) is carried out.

(5) At this point, the process terminates. $\mathcal{S}$ is said to be well-formed with respect to parentheses if both (a) no character in $\mathcal{S}$ has been 'rejected' in step (3) and (b) $\mathcal{Y} = \Lambda$ finally, i.e., no unmatched parentheses remain. We define the parenthetic depth of a character σ in $\mathcal{S}$ to be the number of elements in $\mathcal{Y}$ at the time when σ is examined in step (2).

As an illustration of the different types of parentheses which the above definitions are aimed to allow, we can give, from the formula language:

'Normal' type : $\mathcal{S} \equiv a + B(1-K)(1+K)$

Symmetric : $\mathcal{S} \equiv a + |B - (|x - y|)|$

Asymmetric : $\mathcal{S} \equiv a \downarrow j \uparrow + b \downarrow i, K \uparrow \uparrow 2 \downarrow$

Formulae

There is a duality in the use of operations and parentheses in formal expressions; the device of 'ranking' binary operations in order to avoid an excessive use of parentheses is well known, and we have paralleled this by admitting 'implied' operations to be associated with parentheses. The unifying figure is that of the rank of operation or parenthesis, (which is distinct from the rank of symbolic characters used in constructing the names of $\mathcal{A}$). Thus, a character " $\downarrow$" for which $\mathcal{J}(\downarrow) = 4$, $\mathcal{U}(\downarrow) = 3$ and $\mathcal{V}(\downarrow) = 5$ may be replaced, when it is left parenthetic, by the pair of characters "$_\tau$(" where τ is an operation of rank 5. This is precisely the treatment given to sub- and superscripting parentheses illustrated for the formula language above. In general, a left parenthetic character

may imply the existence of any operation of specified rank, either binary or unary, which is to precede it.

Consider the following three procedures:

P1: the application of Definitions 1 and 2 to a given string in $\mathcal{A}$ to obtain a string in $\mathcal{A}^*$.

P2: the application of the parenthetic scan to determine the sub-type of each parenthetic character, and the substitution of "τ(" for each left parenthetic character σ for which $\mathcal{V}(\sigma) = \mathcal{V}(\tau)$ and τ is an operation, and the substitution of ")" for each right parenthetic character.

P3: the process of inserting a specified 'implied' binary operation between pairs of characters which occur as "$\mathcal{E}\,\mathcal{E}'$", "$\mathcal{E}($", "$)\mathcal{E}$" or "$)($", where $\mathcal{E}$ and $\mathcal{E}'$ are names or numerals.

Let $\mathcal{S}^*$ be the string formed from $\mathcal{S}$ by the application of P1, P2 and P3. Then we have:

Definition 3

$\mathcal{S}^*$ is a well-formed-formula (w.f.f.) if it is well formed with respect to parentheses and can be constructed by the following inductive process:

(i) If $\mathcal{S}^*$ is a name or numeral, it is a w.f.f.

(ii) If $\mathcal{S}^*$ has one of the forms "$(\mathcal{S}_1)$", "$\alpha_1\mathcal{S}_1$", "$\mathcal{S}_1\alpha_2\mathcal{S}_2$" where $\mathcal{S}_1$ and $\mathcal{S}_2$ are w.f.f. and $\mathcal{F}(\alpha_i) = 3$ and $\mathcal{U}(\alpha_i) = i$, then $\mathcal{S}^*$ is a w.f.f.

The above definitions and rules determine many language forms of interest. We remark that, in the absence of parentheses, a w.f.f. consists of a string of names, each possibly 'inflected' by unary operations, connected together by binary operations. Replacing any name in the string by a w.f.f. gives a new w.f.f.

It will be convenient to use a special notation for a string constituting a w.f.f. by the above definition. Let $\mathcal{E}$ stand for either a name or a number or a w.f.f. enclosed by parentheses, possibly preceded by a number of unary operations. Evidently $\mathcal{S}*$ can then be written in the form:

$$\mathcal{S}* \equiv \mathcal{E}_1\omega_1\mathcal{E}_2\omega_2\cdots\omega_{n-1}\mathcal{E}_n$$

where ω_i stands for a binary operation, i = 1,2,...,n-1. Now if $\mathcal{V}(\omega_i) = r_i$ (the rank of the operation), there is evidently at least one operation $\omega*$ of the lowest rank r* and for reasons which will be apparent in the next section, we will write:

$$\mathcal{S}* \equiv \mathcal{F}_1\omega*\mathcal{F}_2\cdots\omega*\mathcal{F}_p$$

where $\mathcal{F}_i$, i = 1,2,...,p, stands for a w.f.f. A w.f.f. such as this is said to be of <u>rank r*</u>. It will also be abbreviated to:

$$\mathcal{S}* \equiv \psi_r^p\mathcal{F}_i$$

<u>Further devices</u>

A practical problem which is worth mentioning here is that of stretching a limited machine alphabet to cover frequently occurring situations. The analysis of this section has so far been concerned with the slightly idealized alphabet $\mathcal{A}$ and its abstraction $\mathcal{A}*$. In point of fact we have to produce $\mathcal{A}$-characters from some mechanical set = $(h_1, h_2, \ldots, h_t)$; for example, we want to include in $\mathcal{A}$ a mark such as "≠" and will achieve it by the marks in $\mathcal{M}$ "=", 'backspace', "/", assuming these exist. Also, it is undersirable to complicate $\mathcal{A}$ with case control marks and the like and these can be disposed of at a stage prior to the main reduction. The following two devices may be used by the coder for simple string manipulation.

(1) Character Expansion, P4

Let h be a character in $\mathcal{H}$ but not in $\mathcal{A}$. Then it is possible to 'expand' h into an occurrence of a given string "$\alpha\beta\gamma$" say, the characters of which may or may not all be in $\mathcal{A}$. If not, the process continues until a string in $\mathcal{A}$ is found.

(2) String Contraction, P5

Let A be a string, say $h_1h_2h_3$. Then it is possible to contract all occurrences of A into occurrences of a single character δ which may be in $\mathcal{H}$ or in $\mathcal{A}$. If in $\mathcal{H}$ but not in $\mathcal{A}$, δ may be expanded under P4 to a new string.

* * * * * * * * * * *

To summarize this section, we have described a symbolic representation of formulae based on an alphabet $\mathcal{H}$, in which strings are reduced first by two elementary string transformations (P4 and P5) to strings in an alphabet $\mathcal{A}$, from which they are further reduced to formulae in the alphabet $\mathcal{A}^*$ by the procedures P1, P2 and P3 which in turn make reference to the parametric values $\mathcal{I}, \mathcal{U}, \mathcal{V}, \mathcal{Y}$ assigned to elements of $\mathcal{A}$. This whole process is determined by a set of values contained in a Character Table which is referred to as each formula is processed. This gives the following information:

(1) The alphabets $\mathcal{A}$ and $\mathcal{H}$

(2) Expansion rules for elements of $\mathcal{H}$ not in $\mathcal{A}$

(3) String contractions in $\mathcal{H}$

(4) Parameter values for all characters in $\mathcal{A}$

(5) The 'limiting lengths' for names (μ) and numerals (ν)

(6) The radix of number representation and rule for representing numerals and names.

(7) The designated 'implied' binary operation

The main point of this technique is to allow rapid switches from one symbolism to another in reading sets of formulae, and to allow experimentation by the coder in obtaining the presentation he wants. In the Genie system, formula scans are made by a routine called TSCAN, which has various facilities for detecting syntactic errors. It is evident that a by-product of TSCAN is the list of identifiers required by the processors of the previous section. Beyond this, further analysis is dependent on the meaning of names which are used, i.e., the classes of objects for which they stand.

4. Semantics

In a formal system, names and formulas must be made to 'stand for' something, for only then can they be used as the means of displaying relationships between objects of interest, and revealing new relationships by means of formal manipulation. The classical way of providing an interpretation is to give a model or representation for the logical system on hand. By this device, the consistency of the system can be demonstrated, or at least made to depend on the consistency of another system, and an interpretation is provided. It is the second achievement which interests us here, and in this section we shall demonstrate the methods by which a representation is used in evaluation.

Viewing the immediate applications we have in mind for Genie, it can be stated that the classes of objects falling under investigation will include: characters, names, numbers,

formulae, and definitions. Moreover, for various reasons, we shall want to manipulate these not only one at a time, but also in linear (vector) and rectangular (matrix) arrays. They may be represented by names or numbers, as appropriate. However, a name may only represent one object other than itself. Thus, in the equation:

$$G = 4t + 1$$

the names "G" and "t" stand (presumably) for numbers, but in listing the quantities on which G is dependent, "t" would stand for itself.

The problem of deciding what a name does stand for is more complicated in practice than theory, and the three methods which are available are as follows:

(1) By declaration: as in "K is a vector" or "K is a definition";

(2) By assumption: as in the equation above, where we assume G and t are numerical objects;

(3) By deduction: for example, we deduce from an equation such as:

$$y = (\sin g, \cos g, o) + (x_1, x_2, x_3)$$

that y is itself a vector.

Now method (1) is infallible, but somewhat clumsy and we should like to avoid it where either of the other methods is sufficient. Method (2) depends upon an underlying assumption which is made whenever a formula is read, and we shall associate with each Character Table a particular class of objects which names will be assumed to represent <u>unless otherwise determined</u> by declaration or deduction. Various methods of selecting the

appropriate Character Table are possible, mostly dependent on a 'key' character which immediately procedes the formula. Thus, in the case of the algebraic formula language, the key character is "=" and all names are assumed to stand for numbers. In the case of algebraic program descriptions, the key character is ".=" and all names are assumed to stand for definitions. Finally, in order to use method (3) it is necessary to know which binary and unary operations are permitted on which classes of objects, and what classes the resultant objects belong to. This information is given in the Reduction Table, given below.

First, we will summarize the evaluation process applied to formulae. It is evident from the inductive definition of a w.f.f. that in order to have an evaluation process for a formula, it is sufficient that evaluation processes be given just for the binary operations applied to two objects, or for the unary operations applied to one object, provided unambiguous rules of precedence are given.

Let a source language $\mathcal{L}$ be determined by the group $\{\mathcal{N}, \mathcal{R}, \mathcal{U}, \mathcal{B}\}$, respectively the sets of names, numerals, unary and binary operations in $\mathcal{L}$. Then we have:

Definition 4

An evaluation model $\mathcal{M}$ of $\mathcal{L}$ consists of a set of objects $\mathcal{S}$, and two reduction rules $\mathcal{K}^{(U)}$ and $\mathcal{K}^{(B)}$, with the following properties:

(i) To each name α in $\mathcal{N}$ corresponds a subset $\mathcal{S}_\alpha$ in $\mathcal{S}$; and to each numeral ν in $\mathcal{R}$ corresponds* a unique ν' in $\mathcal{S}_\nu \in \mathcal{S}$.

*

A prime will be used to denote elements in $\mathcal{M}$ corresponding to particular elements of $\mathcal{L}$. This will also be written, e.g.,

"$\nu' \rightarrow \nu$", and it is termed a 'value assignment'.

(The subset of numbers.)

(ii) To each unary or binary operation ω in $\mathcal{L}$ corresponds a unique unary or binary operation ω' in $\mathcal{M}$ of the same rank. Parentheses in $\mathcal{L}$ correspond to parentheses in $\mathcal{M}$.

(iii) A w.f.f. in $\mathcal{L}$ reduces to w.f.f. in $\mathcal{M}$ if each name in the w.f.f. is replaced by some $\alpha' \varsigma \mathcal{S}_\alpha$, each numeral ν is replaced by its corresponding ν', and operations and parentheses in $\mathcal{L}$ are carried over to the corresponding operations and parentheses in $\mathcal{M}$.

(iv) If α' and β' are objects of $\mathcal{S}_\alpha$ and $\mathcal{S}_\beta$ respectively, then for each binary operation ω' there exists a rule $\mathcal{K}^{(B)}(r,\alpha',\beta',\gamma')$, r being the rank of ω', by which the formula "$\alpha'\omega'\beta'$" may be reduced to an element γ' in some subset $\mathcal{S}_\gamma$ of $\mathcal{S}$.

(v) If α' is an element of $\mathcal{S}_\alpha$ and $\mathcal{U}'$ corresponds to a unary operation in $\mathcal{M}$ of rank t, then there exists a rule $\mathcal{K}^{(U)}(t,\alpha',\beta')$ which reduces the expression "$\mathcal{U}'\alpha'$" to an element β' of $\mathcal{S}_\beta$.

Potentially, there may be infinitely many rules $\mathcal{K}^{(B)}$ and $\mathcal{K}^{(U)}$ since there may be infinitely many elements in $\mathcal{S}_\alpha$ and $\mathcal{S}_\beta$ (say). Even a large number would be awkward to handle, and it is avoided in practice by analysing the elements of $\mathcal{S}$ and expressing them as compound elements from another model $\mathcal{M}'$ of $\mathcal{M}$, with only a small number of rules. In this way, the general rules of decimal arithmetic addition, for example, can be built up from a simple rule of 100 elements, expressed in a 10 X 10 addition table. It is often convenient to write the

result of applying $\mathcal{K}^{(B)}$ as:

$$\gamma' = \mathcal{K}_r^{(B)}(\alpha', \beta') \tag{4.1}$$

and similarly, the result of applying $\mathcal{K}^{(U)}$ is:

$$\beta' = \mathcal{K}_t^{(U)}(\alpha') \tag{4.2}$$

In this case, $\mathcal{K}_r^{(B)}$ and $\mathcal{K}_t^{(U)}$ are termed valuation operators. Let the disjoint subclasses of different types of object in be $\mathcal{S}_\alpha$, $\mathcal{S}_\beta$, and $\mathcal{S}_\gamma$. Then to specify the properties of valuation operators completely, they can be written in the form of tables, one for each operation of each rank.

		Right Operand Sub-Class		
	r	$\mathcal{S}_\alpha$	$\mathcal{S}_\beta$	$\mathcal{S}_\gamma$
Left Operand Sub-Class	$\mathcal{S}_\alpha$	$\mathcal{K}_r^{(B)}(\alpha',\alpha'):\lambda_{\alpha\alpha}$	$\mathcal{K}_r^{(B)}(\alpha',\beta'):\lambda_{\alpha\beta}$	$\mathcal{K}_r^{(B)}(\alpha',\gamma'):\lambda_{\alpha\gamma}$
	$\mathcal{S}_\beta$	$\mathcal{K}_r^{(B)}(\beta',\alpha'):\lambda_{\beta\alpha}$	$\mathcal{K}_r^{(B)}(\beta',\beta'):\lambda_{\beta\beta}$	$\mathcal{K}_r^{(B)}(\beta',\gamma'):\lambda_{\beta\gamma}$
	$\mathcal{S}_\gamma$	$\mathcal{K}_r^{(B)}(\gamma',\alpha'):\lambda_{\gamma\alpha}$	$\mathcal{K}_r^{(B)}(\gamma',\beta'):\lambda_{\gamma\beta}$	$\mathcal{K}_r^{(B)}(\gamma',\gamma'):\lambda_{\gamma\gamma}$

TABLE 1. Reduction table for a binary operation of rank r

In Table 1, the table entry in row $\mathcal{S}_\beta$, column $\mathcal{S}_\gamma$, for example, is $\mathcal{K}_r^{(B)}(\beta',\gamma')$, together with the class to which the reduced formula belongs, $\mathcal{S}_{\lambda_{\beta\gamma}}$. Table 2 is a reduction table

for a unary operation of rank t. With the aid of these rules and the substitution rule which will now be given, a value in $\mathcal{M}$ of any w.f.f. in $\mathcal{L}$ may be obtained.

t	Valuation
$\mathcal{S}_\alpha$	$\mathcal{K}^{(U)}_t(\alpha'):\mu_\alpha$
$\mathcal{S}_\beta$	$\mathcal{K}^{(U)}_t(\beta'):\mu_\beta$
$\mathcal{S}_\gamma$	$\mathcal{K}^{(U)}_t(\gamma'):\mu_\gamma$

TABLE 2. Reduction table for
a unary operation of rank t

Rule 3 (The Substitution Rule and Ranking conditions in)

(a) Let $\bar{\mathcal{F}}$ be a w.f.f. in $\mathcal{M}$ consisting of one or more unary operations followed by the element σ' of $\mathcal{L}$. Let U'_t be the highest ranking operation in $\bar{\mathcal{F}}$, with rank t. Let $\Phi(\bar{\mathcal{F}})$ be a w.f.f. in which $\bar{\mathcal{F}}$ occurs with no immediately preceding unary operations. Then the first substitution can be written:

R3A:
$$\frac{\Phi(\bar{\mathcal{F}}),\ \rho' = \mathcal{K}^{(u)}_t(\sigma')}{\Phi(\bar{\mathcal{G}})}$$

where $\bar{\mathcal{G}}$ is obtained from $\bar{\mathcal{F}}$ by replacing σ' by ρ' and deleting <u>all</u> occurrences of U'_t from the string preceding σ'.

(b) Let $\bar{\mathcal{F}}$ be a w.f.f. in $\mathcal{M}$ consisting of an element σ' of $\mathcal{L}$ in parentheses. Then the second substitution rule can be written:

R3B: $\dfrac{(\sigma')}{\sigma'}$

(c) Let $\Phi(\mathcal{F})$ be a w.f.f. in $\mathcal{M}$ in which the highest ranking binary operation at zero parenthetic level is ω, of rank r, and suppose that this occurs in a w.f.f. $\mathcal{F}$ of rank r and maximum length $p \geq 2$, i.e. $\mathcal{F} \equiv \sigma', \omega\sigma'_2\omega\ldots\omega\sigma'_p$, where $\sigma'_i, i = 1,2,\ldots,p$ are elements of $\mathcal{S}$. Then the third substitution rule can be written in either of the forms:

R3Ca: $$\frac{\Phi(\mathcal{F}), \rho'_{1,2} = \mathcal{K}^{(B)}_r(\sigma'_1, \sigma'_2)}{\Phi(\mathcal{G})}$$

where $\mathcal{G}$ is given by:

$$\mathcal{G} \equiv \rho'_{1,2}\omega\sigma'_3\ldots\omega\sigma'_p$$

Alternatively:

R3Cb: $$\frac{\Phi(\mathcal{F}), \rho'_{p-1,p} = \mathcal{K}^{(B)}_r(\sigma'_{p-1}\omega\sigma'_p)}{\Phi(\mathcal{J})}$$

where $\mathcal{J}$ is given by:

$$\mathcal{J} \equiv \sigma'_1\omega\sigma'_2\ldots\omega\sigma'_{p-2}\rho'_{p-1,p}.$$

R3C merely states the usual arithmetic rule of precedence which is to be applied when parentheses are omitted, and the R3A extends it to cover unary operations. In addition, a choice is offered in the binary case, which may or may not be significant, of evaluating formulae in a 'left-to-right' (R3Ca) or 'right-to-left' (R3Cb) order. In Genie, the code may choose either of these, or he may leave it to form. It should also be noted that we shall assume that R3C also caters for instances where function names appear in formulae, and that the reduction table has provision for appropriate action in such cases. At

this level, function evaluation in arithmetic formulae, for example, can be regarded just as an elaborate form of multiplication.

An example of evaluation

To illustrate these points we shall take a simple arithmetic language $\mathcal{L}_o$ containing the names "a", "b", "c" and "d", and the binary operations "+" and "X" of rank 1 and 2 respectively. As a model, $\mathcal{M}$, we take a single class of objects (numbers) $\mathcal{S} \equiv \{0', 1', 2'\}$, with just 3 elements. Then the reduction tables for $\mathcal{M}$ and its two operations are final (since no separate sub-class of $\mathcal{S}$ exists), and the valuation operators $\mathcal{K}_1^{(B)}$ and $\mathcal{K}_2^{(B)}$ are given in Table 3a, 3b. Let $\mathcal{F}$ be the $\mathcal{L}_o$ formula "a + b X c + d", and let it be required to determine the value of $\mathcal{F}$ for the value assignments '1'→a', '2'→b', '1'→c', and '0'→d' in $\mathcal{M}$. Thus,

$\mathcal{K}_1^{(B)}$	0'	1'	2'
0'	0'	1'	2'
1'	1'	2'	0'
2'	2'	0'	1'

TABLE 3a

$\mathcal{K}_2^{(B)}$	0'	1'	2'
0'	0'	0'	0'
1'	0'	1'	2'
2'	0'	2'	1'

TABLE 3b

we have:

$\mathcal{F} \equiv$ a + b X c + d in $\mathcal{L}_o$

whence: $\mathcal{F} \equiv$ 1' +' 2' X' 1' +' 0' in $\mathcal{M}_o$

hence: 1' +' 2' +' 0' by $\mathcal{S}$2a and $\mathcal{K}_2^{(B)}$(2',1')

and 0' +' 0' by $\mathcal{S}$2a and $\mathcal{K}_1^{(B)}$(1',2')

and $\mathcal{J}' \equiv 0'$ by 2a and $\mathcal{K}_1^{(B)}(0',0')$ which completes the evaluation of $\mathcal{J}$ in $\mathcal{M}$. Readers familiar with formal calculation will recall more elegant developments of this sort.

It can be deduced quite simply from the above definitions that if $\mathcal{J}$ is a w.f.f. of rank r in $\mathcal{L}$ with elementary sub-terms, then its corresponding w.f.f. in $\mathcal{M}$ can be reduced to a recursive expression involving the evaluation operator $\mathcal{K}_r^{(B)}$. Thus:

$$\mathcal{J} \equiv \alpha_1 \omega_r \alpha_2 \ldots \omega_r \alpha_p$$

whence: $$\mathcal{J}' \equiv \alpha_1' \omega_1' \alpha_2' \ldots \omega_r' \alpha_p'$$

whence: $$\mathcal{J}' = \mathcal{K}_r^{(B)}(\mathcal{K}_r^{(B)}(\ldots(\mathcal{K}_r^{(B)}(\alpha_1',\alpha_2'),\ldots),\alpha_{p-1}'),\alpha_p'$$

A slightly different form follows by applying R3Cb rather than R3Ca. This form is important, and may be conceived as the result of applying a <u>general</u> evaluation operator $\mathcal{K}$ to the form $\mathcal{J}$:

$$\mathcal{K}(\mathcal{J}) = \mathcal{K}(\psi_r^p \alpha'_i) = \mathcal{K}_r^{(B)}(\mathcal{K}_r^{(B)}(\ldots\mathcal{K}_r^{(B)}(\alpha_1',\alpha_2'),\ldots)\alpha_{p-1}')\alpha_p')$$

With this concept, it is possible to express in operational form the result of applying the evaluation operator to any w.f.f. in $\mathcal{M}$ of rank r:

$$\mathcal{K}(\mathcal{G}) = \mathcal{K}(\psi_r^p \mathcal{E}_i) = \mathcal{K}_r^{(B)}(\mathcal{K}_r^{(B)}\ldots\mathcal{K}_r^{(B)}(\mathcal{K}(\mathcal{E}_1),\mathcal{K}(\mathcal{E}_2),\ldots),$$
$$\mathcal{K}(\mathcal{E}_{p-1})),\mathcal{K}(\mathcal{E}_p)).$$

where $\mathcal{E}_i$ stands for a sub-w.f.f. of $\mathcal{G}$.

<u>A second evaluation example</u>

To illustrate the generality of the evaluation technique we shall take the simple language $\mathcal{L}_o$ and evaluate it on a new model $\mathcal{N}$, in which $\mathcal{S}$ consists of two sub-classes, $\mathcal{S}_a$, the class of 'simple' objects which are values of objects in $\mathcal{L}_o$, and $\mathcal{S}_b$ which is the class consisting of all ordered linear sequences of pairs of objects <K,L>, where L ε $\mathcal{S}_a$ and K is a

member of the set {CLA, ADD, MPY, STO} which we will term orders. The reduction rules for "X'" are given in Table 4, and the rules for "+'" are similar.

Rank 2	$\mathcal{S}_a$	$\mathcal{S}_b$
$\mathcal{S}_a$	$\mathcal{K}_2^{(B)}(a,a);b$	$\mathcal{K}_2^{(B)}(a,b);b$
$\mathcal{S}_b$	$\mathcal{K}_2^{(B)}(b,a);b$	$\mathcal{K}_2^{(B)}(b,b);b$

TABLE 4. Reduction table for X' in N

For all i, let $a'_i \in \mathcal{S}_a$ and $b'_i \in \mathcal{S}_b$. Then the evaluation operators can be defined by the following procedures:

$$\mathcal{K}_2^{(B)}(a'_1,a'_2) = (\langle CLA,a'_1\rangle, \langle MPY,a'_2\rangle)$$

$$\mathcal{K}_2^{(B)}(a'_1,b'_1) = (b'_1,\langle MPY,a'_1\rangle)$$

$$\mathcal{K}_2^{(B)}(b'_1,a'_1) = (b'_1,\langle MPY,a'_1\rangle)$$

$$\mathcal{K}_2^{(B)}(b'_1,b'_2) = (b'_1,\langle STO,t'_d\rangle,b'_2,\langle MPY,t'_d\rangle)$$

where t'_d is a member of $\mathcal{S}_a$ not corresponding to any element in $\mathcal{M}$ and d is the parenthetic depth of the operation sign in "b'_1 X' b'_2 ".

The rules for "+'" are similar, with "MPY" replaced by "ADD" throughout. Now consider the formula $\mathcal{G} \equiv a + b \times c + d$ in $\mathcal{L}_o$. Its reduction under the value-assignment "a'→a", "b'→b", "c'→c" and "d'→d" is as follows:

$$\mathcal{G} \equiv a + b \times c + d$$

whence: $\mathcal{F} \equiv a' +' b' \times' c' +' d'$

hence: $a' +' (\langle CLA,b'\rangle,\langle MPY,c'\rangle) +' d'$

and $(\langle CLA,b'\rangle,\langle MPY,c'\rangle,\langle ADD,a'\rangle) +' d'$

and finally: $\mathcal{F}' \equiv (\langle CLA,b'\rangle,\langle MPY,c'\rangle,\langle ADD,a'\rangle,\langle ADD,d'\rangle)$

which completes the evaluation of $\mathcal{F}$ in $\mathcal{N}$. As another example, take:

$\mathcal{F} \equiv a \times b + c \times d$

whence: $\overline{\mathcal{F}} \equiv a' \times' b' +' c' \times' d'$

hence: $(\langle CLA,a'\rangle,\langle MPY,b'\rangle) +' (\langle CLA,c'\rangle,\langle MPY,d'\rangle)$

by a double application of $\mathcal{L}_{2a}$ and $\mathcal{K}_2^{(B)}(a_1', a_2')$.

and $\mathcal{F}' = (\langle CLA,a'\rangle,\langle MPY,b'\rangle,\langle STO,t_1'\rangle,$

$\langle CLA,c'\rangle,\langle MPY,d'\rangle,\langle ADD,t_1'\rangle)$

which completes the evaluation of $\mathcal{F}$ in $\mathcal{N}$.

From the foregoing two simple examples, it is hoped the reader will infer the general structure of evaluation processes in Genie.

Another example of the type of formula which can be evaluated is provided by the predicates introduced in Section 2 in the definition of Schema IV. By this we understand a formula constructed from elementary terms with values 0 (false) and 1 (true), and the binary connectives "or" and "and" and the unary operation "not" of the propositional calculus. It is usual to assume that the rank of "and" exceeds that of "or". In this case an evaluation procedure is exactly equivalent to familiar logical device of evaluation by truth table.

Evidently many of the so-called 'algebraic coding languages' in present use involve several of the 'languages' which we have been categorizing. Almost all of them permit the use of some

form of algebraic expression, and a 'Boolean' expression. In addition, it can easily be seen that the 'meta-language' used in Section 2 to describe definitional schemata and their properties can itself be put into a form which satisfies the requirements of Sections 3 and 4. This leads to a 'control language' which is extremely useful in the manipulation of other expressions. For example, a sequential definition may be written in the form:

Example: P .= A → B → C → Return

where "P" is the name of the sequence, and "A", "B", "C" stand for commands. The special equality sign ".=" is used to distinguish the formula which follows as belonging to the control language rather than the formula language; "→" is a 'sequencing' operation.

Hence, in practice, we have to contend with a mixture of language conventions, and to have the ability to switch rapidly from one set to another. This is a familiar enough situation in reading a mathematical text, where the mingling of English phrases and formal expressions generally causes no confusion: if there is a chance of this happening, the writer would use constructions such as 'the formula .....' and 'the equation....'. Much the same thing as this is done in Genie, i.e. the language in which a formula is to be given is indicated by a special sign which precedes it.* Often, as indicated above, a particular

---

* These special signs have universal significance, i.e. are common to all the admissible Genie languages.

---

form of 'equals' sign is sufficient; at other times, as in

using predicates, the special sign "if" serves the same purpose. In terms of computing, a change from one language to another can be made by the trivial device of changing one word which controls an indirect addressing sequence.

One of the immediate applications of Genie is to systems whose objects are to be 'symbols', 'formulae' and 'equations', i.e. linguistic elements. In such cases a familiar situation arises in that formulae may contain not the names of objects but concrete examples of the objects themselves, which must be displayed between quotation marks, as in the following example from a hypothetical symbol manipulating language:

Example: T := A then "+" then either "1" or "X"

Theoretically, the numbers appearing in an arithmetic formula could be treated in the same way, and this is the case internally to Genie, but their importance merits a special external treatment.

The necessity for the devices of 'embedding' one language within another and 'displaying' symbolic objects is purely practical, and may be removed by the use of additional names.

To summarize the results of this section, we have required that a Reduction Table be associated with each formal system which effectively determines a model in which its formulae may be evaluated. Each binary operation may be declared to be associative or not, and if not it may be specified to have left-to-right or right-to-left precedence. As a control device, each formula must be preceded by an operator which effectively selects the Character Table and Reduction Table for the ensuing formula, which is terminated by some punctuation opera-

tion at parenthetic level zero. It is quite possible that elements in the chosen model may be the representation of some formal system which in turn may be subject to evaluation, so that we conceive of evaluation as a cascading process which continues until no further reduction can be made. The process of evaluation is initiated by an application of the Execution Rule R4 which is described in the next section.

5. Machine Realization

The class of machines we are concerned with is difficult to describe in an abstract fashion. To be sure, they are all finite automata in a strict sense, but they are also organized in a way dictated by history, economy and usage, and it would be an elaborate process to describe such organization in general terms. Our purpose in using the idea of a formal system in giving definitions was to extend the classes of objects which could be defined without unduly extending the mechanism of value assignment. When our interest turns to enlarging the class of machines on which valuations can be made, then the task of generalization will be faced. For the sake of present simplicity, however, we shall consider just that narrow class of machines into which presentday sequential computers fall.

In short, a machine contains a binary, addressable storage unit S containing W words of b bits each, together with a further n bits of unaddressable storage T. Also, there exists a finite set of m command operators $K_1, K_2, \ldots, K_m$ such that, for any assignment of bits to S, T, which we can write $\langle S', T' \rangle$,

there exists a rule:

$$<S",T"< = K_t(<S',T'>),\ t = 1,2,...,m$$

which determines uniquely a new assignment <S",T">.

We shall assume the reader is familiar with the characteristics of stored program machines, with the techniques of micro-coding, interpretive coding, and with specific examples of single and multiple address order codes. In particular, it should be noted that if a machine contains a program P in p of its storage cells, then it becomes equivalent to a new machine with $W_1 = W-p$ storage cells, and a set of command operators $K_1, K_2, ...K_m, P$. For, given a machine configuration <S',T'>, it is possible to determine uniquely the configuration <S",T"> resulting from the application of p.

Implicit in the design of the machine is a sequencing rule, which determines, for any state <S',T'>, the command operator K_t which is next to be applied. It does so by observing a particular group of cells in T (the control counter) which designate that word in S which contains the code "t" of K_t. The order code is such that only a small subset of S, not more than 4 or 5 words at most, need be considered in the application of a given K_t. In this subset, one or more words may be always contained, by implication (the 'accumulator', 'quotient' registers, etc.), and the others explicitly stated by the command code t. For this reason it is simpler to regard the set $\{K_t\}$ as the product of the sets of order codes $\{O_i\}$ and addresses $\{A_j\}$, the latter referring to elements of S, perhaps, as in instances of B-modification, after 'inflection' by elements in T.

The purpose of this section is to show how such a machine may be used for general evaluation. This requires a representation in the machine of the command operators and operands of the model. There is evidently two ways of achieving a representation of the evaluation operators: either the machine has in the set $\{O_i\}$ an order which is precisely equivalent to one of the given operators, or it does not. In the first case a 'direct' representation of the evaluator is possible; in the second case, the evaluation must be represented by a sequence of orders, which is equivalent to modifying the machine in the way indicated above, and the 'indirect' representation is achieved through the use of an open or closed subroutine. We shall always assume that it is possible to name a single operand by means of one address, so that a unary operation acting on a given operand can be represneted by just one command operator in the direct case. In considering binary operations we shall adopt the convention, since we have in mind primarily a single address machine, that the first operand is always contained in a fixed storage register 'U', and the second is to be specified by an address. In this case, too, a direct representation of a binary operation amounts to a single machine command operator. Given this restriction on the first operand, and given that the result of any operation, unary or binary, resides in U after execution of the operation, it is possible with the help of the Substitution Rule R3 to derive a sequence of command operators for evaluating given w.f.f. on a particular machine. The precise details are easy enough to work out. The only point to make here is that by describing the evaluation operators in terms of command operators we have sufficient in-

formation to effect an evaluation process on the machine. Any other information that is required is for improving the efficiency of the process in one way or another. By thus separating the essential from the inessential, we can achieve a great degree of flexibility in the types of evaluation procedure which are applied, and it becomes easier to decide how much effort should be expended in minimizing time or space requirements in the translated code.

Clearly, sequences of command operators in the machine are instances of Schema V, and references to special sequences of commands used as closed subroutines are instances of Schema VI. We shall assume, therefore, that the machine obeys a Sequencing Rule similar to R2 and that it maintains a 'pathfinder' list of references to subroutines which are in use. In any given machine, of course, this rule is more or less contained in the hardware, although none has been built with completely automatic control of the pathfinder list.

Before considering in detail the realization of operands it is necessary to discuss the ways in which they are to be addressed by a command operator. From the remarks in Section 2 and the present section, we can see that operands may be classified into five groups:

(1) Named auxiliary variables

(2) Named external variables

(3) Named parameters of a definition

(4) Implicit variables - 'U', the quotient-register, B-registers, control counter, etc.

(5) Unnamed auxiliary variables, generated as in the reduction from implicit to explicit sequencing form, or by 'dis-

played' operands and numbers.

There are various deductions to be made from this classification. First, note that in some cases it is relative to the structure of the definitions: external variables of one definition may be the auxiliary variables of another, and in fact the distinction parallels exactly the Symbol Table control rule R2 given in Section 2. In Figure 1, for example, we have represented the definitions D1 and D2 inside the set of definition D3 which is in library form. We then know that the external variables of D1 and D2 are auxiliary to D3, that the auxiliary variables of D1 have no significance outside D1, and similarly for D2. P is a pointer which moves steadily through each definition as it is processed, and in Figure 2 is represented the form of the Symbol Table at various stages of processing.

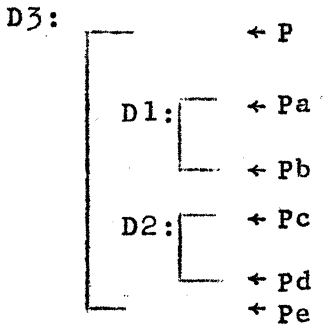


FIGURE 1

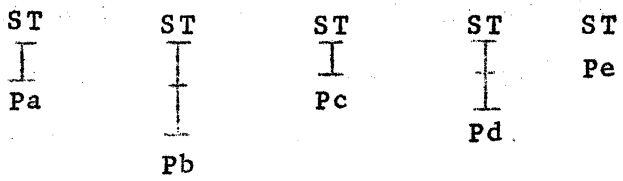


FIGURE 2

In position Pa it is at the beginning of D1, at Pb at the end of D1, at Pc at the beginning of D2, and so on. The conclusion to be drawn from this is that providing the evaluation of D1 does not call for the evaluation of D2, or vice versa, then the maximum number of named variables in use at any one time during the execution of D3 is the same as the maximum length

attained by the Symbol Table. This fact is used in assigning addresses to names appearing in D3. For any library program a similar argument holds, and a region of storage is reserved for named variables in groups (2) and (3). It is not necessary for such space to be permanently reserved, and it is taken immediately prior to the execution of a library routine and relinquished afterwards.

The restriction imposed on the use of D2 by D1 in the last paragraph can be removed in the case of library definitions: they may use one another, and themselves, to any degree of complexity, and are thus the most convenient computing units to handle.

Each definition in functional form refers to its parameters in a region of storage whose address, by convention, is contained in a certain fixed cell in the machine. The fact that an address may in fact refer to another address for its data leads to great flexibility here. With regard to the implicitly used operands - little can be said of general significance, since the use of these is closely related to machine design. It is sufficient to recognize their existence and consider the problem of what to do when cells in this group are used in different ways by two interacting subroutines. For if D1(I) and D2(I) denote the sets of names used in definitions D1 and D2 respectively, then clearly just that set $D1(I) \cap D2(I)$ must be 'saved' when D1 calls for the use of D2, and 'unsaved' afterwards. This is a simple algorithm to code, and it is achieved with the help of the working storage region described in the next paragraph.

Concerning the last group of unnamed objects (5), it is clear from the nature of things that one of these cannot possibly be referred to in another formula, or in the same part of the same formula, and a simpler addressing scheme can be used. Moreover, from the inductive nature of the evaluation procedure, it follows it is only necessary to make the 'intermediate results' available in the inverse order to that of their creation, so that a linear 'pushdown' list is maintained for results such as these, and it is shared by all routines which use one another, simply extending and contracting the list as they need. Exactly the same treatment can be given to variables which are required to be saved from the implicit variable list.

To summarize the present position, it has been shown that to use a machine for evaluation purposes it is necessary to express the evaluation operators of a particular formal system in terms of the command operators of the machine. This, and the reduction of formulae to an explicitly sequenced form leads to the formation of 'programs' in the machine which are all ultimately dependent on the automatic execution of the command operators by the machine hardware, although it is not always convenient to conceive of programs in these terms. Each program may refer to its operands through one of the five lists containing parameters, named variables (internal), named variables (external), unnamed variables, or implicit operands.

We now pass to the description of the realization of operands in the machine. It has already been said that the objects of study in any Genie language are very general in nature, and the fact that for immediate purposes we allow them to be just

'numbers' or 'names' should not detract from this. At a future date we may equally chose 'lines', 'points', or 'sets'. The realization is best given with reference to a particular machine, in this instance the Rice Institute Computer, which is a single address machine with a word length of 56 bits, two of which are 'control' tags, and do not enter the arithmetic unit. A given address may at any time denote either the location of an operand, or a representation of the operand itself or another address. Each address may be modified by up to eight B-registers at one time, one of these being the control counter. These features are not essential, but do lead to a great deal of efficiency in this type of work. The chosen realizations are as follows:

(i) Numbers

Numbers are given in one of two binary forms. If a number is a positive or negative integer with value less than 2^{14}, it has a direct integer representation in the address portion of a word. In other cases a number has 'floating point' representation of a conventional type in a full machine word, which also includes the integer representation.

(ii) Names

These too have both a direct and conventional form. A name consisting of a single character may be given directly in the address portion of a word. In other cases a name of up to nine characters in length is stored to the left hand side of a full machine word. If it is of eight characters or less, it is terminated by a special character code.

(iii) Formulae

The realization of formulae is based on the idea of formula 'rank' which was derived from the inductive definition of w.f.f.

in Section 3. First, the names appearing in formulae are realized by their (Symbol Table) addresses. Second, unary operations which apply to names appear in coded form in the same word as their addresses (they may be up to seven in number). Last, a formula of rank r with p terms is represented by an associative list of p+1 words. The first is a heading word which contains certain information about the list. The address of the list (i.e. of the formula) is the address of the heading word. The remaining p words give the p addresses and unary operations on sub-formulae. In Figure 4a, each box represents a term in the list, and the arrow out to the right connects it to its 'next' element. Inside the box is written the name of the element to which it refers, together with any unary operations which apply to it. If the term refers to another formula, a downward pointing arrow is drawn from the box to the 'data'. Figure 4c shows a complete associative list. The last word in the list completes a 'loop' back to the heading word, which is marked with a control tag "2", so that the detection of "2" in a 'data' word is sufficient to indicate that the data consists of a new list, and detection of "2" in the 'next' word indicates that there are no more terms in the current list. The heading word itself contains a reference to a higher order list, so it is always possible to trace a path through a formula to arrive at any given level relative to a particular point.

FIGURE 4a: The list of rank r: $\Psi_r^p a_j$

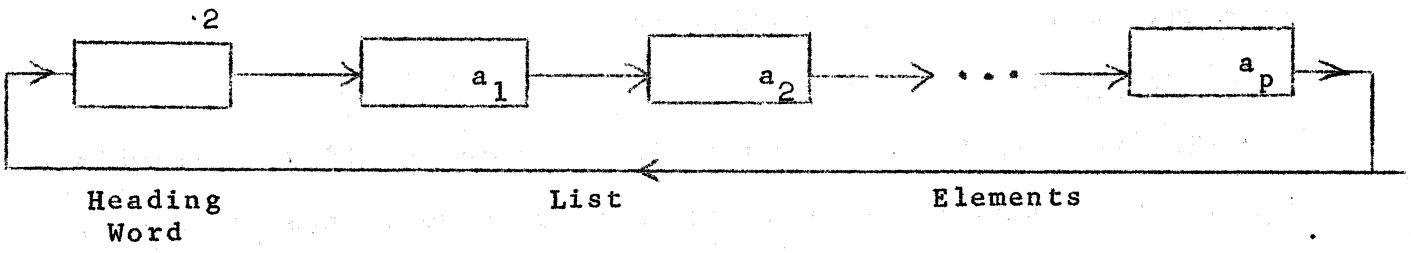


FIGURE 4b: List words on the Rice Institute Computer

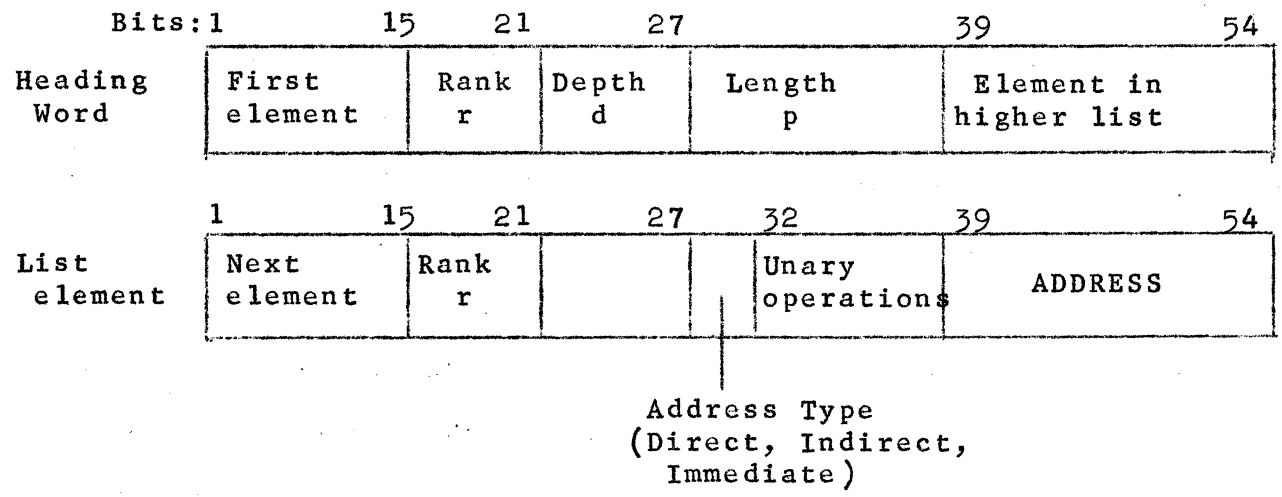
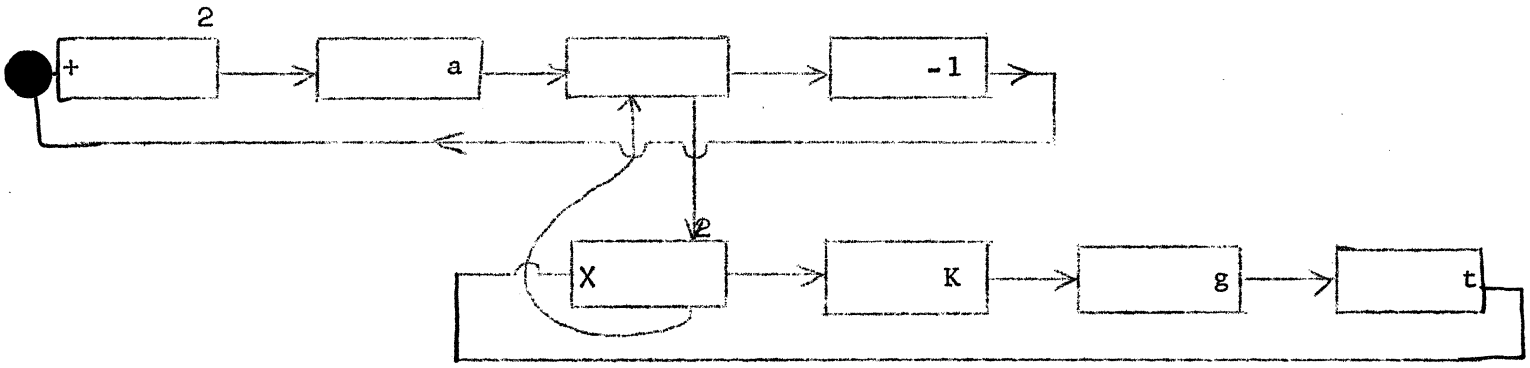


FIGURE 4c: Realization of the formula "a + Kgt - 1"



(iv) Definitions in Schemata I - IV

When a non-functional definition is given by means of an equation, it is represented by placing the heading word for the defining formula in a second table, the Value Table (VT) whose elements are in (1-1) correspondence with the elements of the Symbol Table (ST). If the formula consists simply of a name or number then the representation of that element is placed in the VT. Associated with each formula is a list of external variable names, and this is appended to the appropriate ST entry.

Figure 5 shows the realization of two formulae in this way. When a definition is given, as in Schemata II and III, in terms of several formulae each of these is at first represented in the way described above, (Figure 6a). It is a simple matter, however, to process this representation in order to remove the auxiliary variables from ST, leaving a 'formula' representation of slightly more complex nature (Figure 6b), at the same time obtaining,as before, a list of external variables which is appended to the appropriate ST entry. Other information which is contained in the ST word indicates the 'language' form of the formula, the nature of the definiend, of the schema by which it is defined, and whether or not it is in functional form. If it is given in functional form, then the representation is given in sub-section (v) below.

FIGURE 5: Realization of the equations: "y = 2 + 5x" and "x = a + 3b"

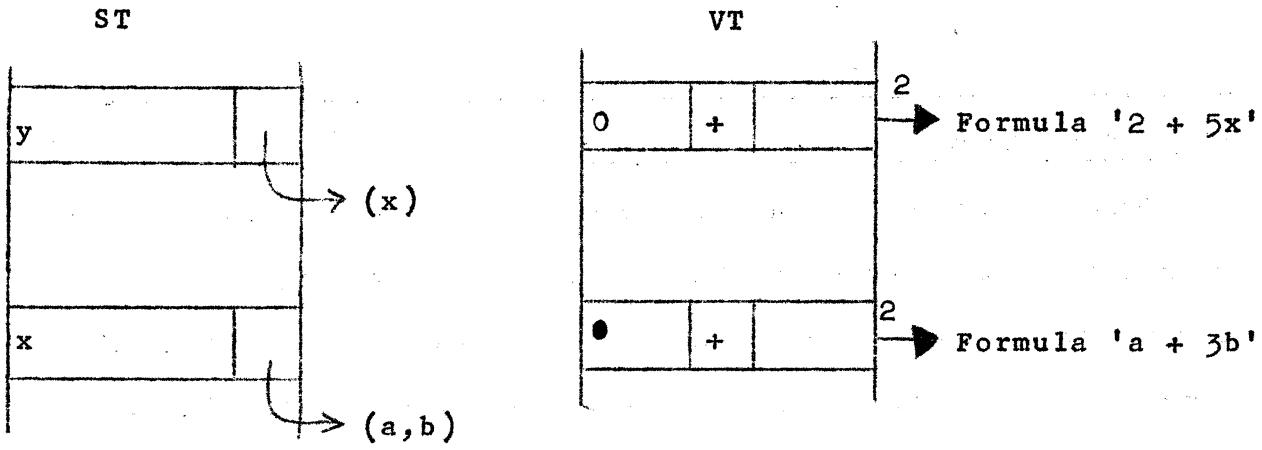
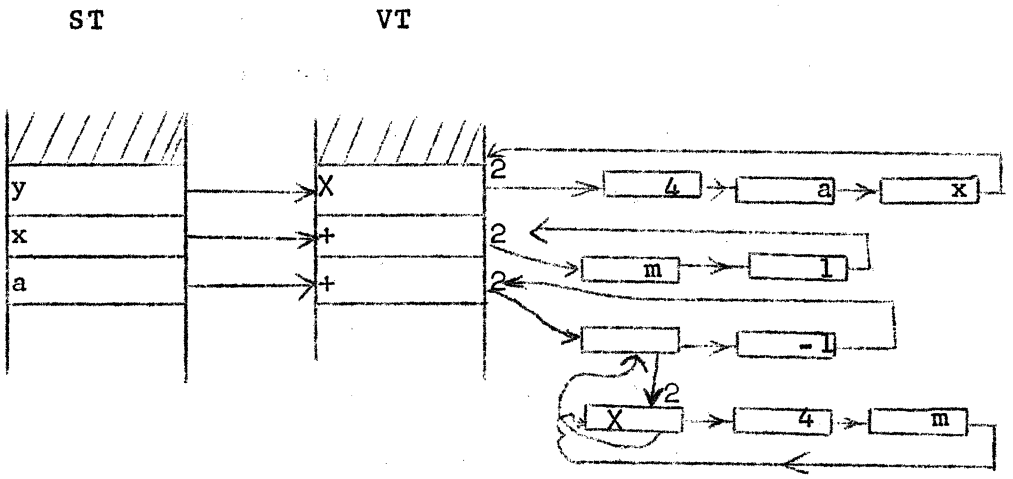


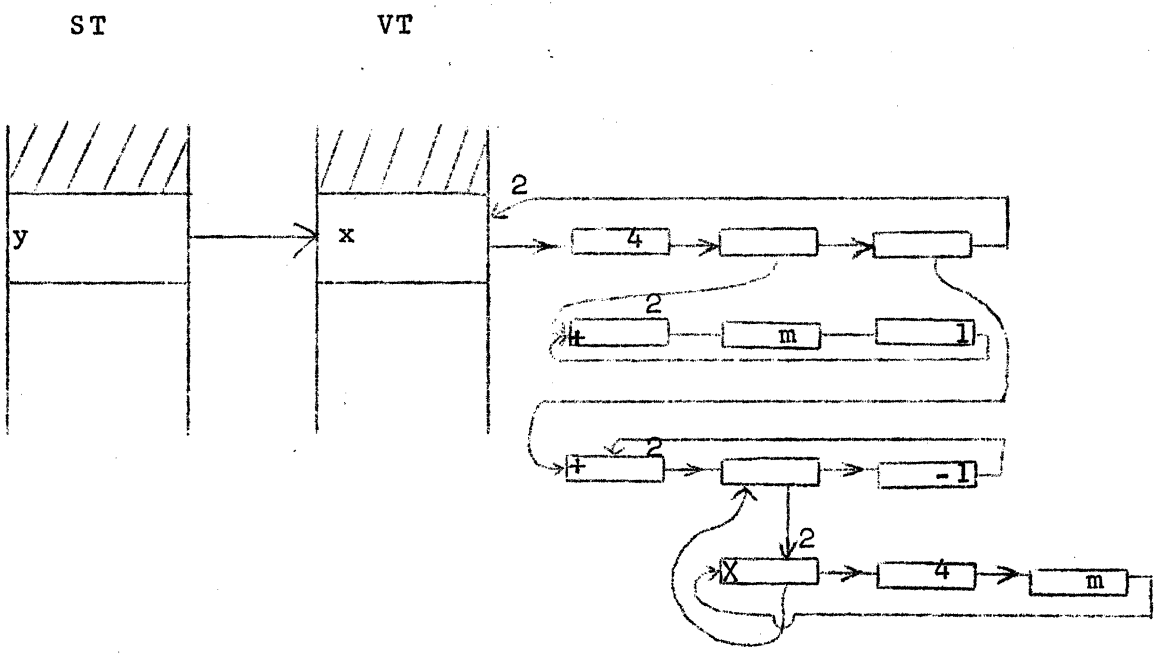
FIGURE 6: Realization of the definition:

"y = 4ax,x = m + 1, a = 4m - 1"

(a) The initial list structures:



(b) The final list structures:



(v) Functional Definitions, and Schemata V - VI

For practical reasons, sequential definitions and functions are realized by first effecting the translation to 'program' form which was indicated at the beginning of this section, on the basis of the Reduction Tables for the various definitions which are given. This realization resembles machine code closely, except that it retains additional information for ease of modification. It is very quickly converted into genuine machine code, and in certain important instances no alteration is necessary. For historical reasons, we refer to this as '8-code'. By retaining 8-code and the portion of ST associated with it a completely symbolic representation of code can be regained for reference purposes, and in practice these are automatically retained on tape (paper or magnetic) until the coder decides that the program is running satisfactorily.

This concludes the summary of operand realizations that are at present provided in Genie. They will be reported in greater detail elsewhere. In addition to these, however, provision is made for the arrangement of operands into regular arrays of one or two dimensions: the vector and matrix forms. Where such forms are desirable, appropriate evaluation operators may be provided in the model and the operand names in formulae can then refer to such arrays either as a whole, or element by element. All array elements must be of the same type, and they may themselves be arrays; they may also be defined by reference to their position in the arrays, as in the case of triangular or band matrices. The argument for restricting Genie to arrays of not more than two dimensions in that in this way we are in-

cluding by far the majority of instances which are useful, and for which efficient manipulative algorithms are known. In any case, higher dimensional arrays can usually be reduced to arrays of arrays at little cost. We have adopted the universal convention that elements of arrays shall be indicated by subscripts which take positive integer values.

We are now in a position to give the final rule which controls all processes of evaluation.

Rule 4. (The Execution Rule)

A machine is controlled by a (supposedly infinite) linear sequence of characters. In this it recognizes names, numbers, operations, formulae, equations and definitions according to the rules given in the preceding sections. For each name it determines what type of operand it stands for, either by implicit assumption, by deduction, or by explicit declaration. It also determines which other named objects a definition depends upon, and whether or not they have themselves been defined. If they have, then the dependent variable is evaluated according to the model which its definition implies. By a continuous scanning process, this continues until no more evaluations can be made on any entry in the Symbol Table. A further definition is then read in, and the process repeated. An example will illustrate some of the consequences of this rule.

Example: We shall assume that the machine contains the Character Table and Reduction Table for an albegraic formula language. Consider the following set of commands and definitions.

$[\pi = 3.14159$

$\sin(x) .= \{...\text{commands evaluating } \sin(x)...\}$

A is an integer

$Q[K = \sin(\pi/4) - 2\sin(3\pi/4)$

```
Q(t,#u)  .= {S = 3t^2 + st - K
            g = 2K + 1
            m = A sin 3K + A^2 g
            u = m + S/m 3 ]
            ...etc....]
```

The opening bracket delimits this set of definitions; when it is closed, all the information provided inside it will be erased from memory and the machine will be back in its original state. This is not always desirable, so a set of definitions may be named; in this case when it is closed all the information contained inside it is removed from the ST but saved in temporary storage. The name of the definition set is retained on ST and may be used to recall the information to ST at any time. In this way any user or group of users may build up private subroutine libraries, languages, etc. and call them into the machine at the start of an evaluation run, as well as using the main facilities of the machine.

The first two definitions in the example give "π" as a number and "sin" as a function. Following this, the number "A" is mentioned in order to place it on ST as an external variable of the set of definitions which follows. This is named "Q", and any subsequent call for it will bring into ST all the definitions contained in the following brackets. Here "K" is defined in terms of "sin" and "π", which are recognized as known quantities and an evaluation of "K" follows. The name"Q" is also given to the sequential definition, which is the main definition of this set in the sense that all others (K) are auxiliary to it. Q has an input parameter "t" and output "u".

In the commands which follow, "g" is recognized as depending on the external quantity "K" only and this has previously been defined, so that a simpler command can be written here. No further simplifications can be made and Q is retained in storage in its δ-code form. After the closing bracket only "π", "A", "sin" and "Q" remain in ST. After the final closing bracket, nothing remains in ST and the machine proceeds to the next definition set.

The Execution Rule implies a continuous control of the finite number of storage elements in a given machine. In the present Genie system this is achieved by placing all 'free' storage cells in a list of special structure which allows the evaluation operators to 'give' and 'take' as many cells as they need for application. This works well until storage space begins to be exhausted, and then various recovery operations can be called into play, aimed first to rearrange active regions of memory to bring all the available cells together in a single block, and finally to put all definitions not currently in use into temporary storage. In the latter connection, it will be noted that the Pathfinder list gives a complete description of routines in use at a given time. While such techniques as these have been shown to be feasible, further experience will be necessary before reliable conclusions can be drawn with regard to automatic storage control algorithms. It may well be that the algorithms will vary with the type of application for which the machine is in use.

Many of the automatic features of Genie are also available under programmer control, transfers to and from magnetic tape

storage being a case in point, and control of 'execution' of programs being another. It is interesting to note that in a machine with adequate trapping features some protection can be afforded against instances where an evaluation procedure is started before all the requisite terms have been defined, or when some of them are in auxiliary storage, and appropriate recovery steps taken. Such techniques seem very promising and suggest a new application of interpretive-type programs, the usefulness of which has been in doubt since the advent of reliable compilers.

It should also be noted that the Execution Rule can be treated simply as a loading routine, and as an input routine for use during execution of a program. In addition to this, individual routines may be used for the input of numbers during program execution, outside the control of the Execution Rule. Printed output does not affect the results of an evaluation process at all, and it may be obtained either in the form of definitions or formulae in certain standard formats. Otherwise, suitable output subroutines may be compiled by selecting a descriptive language and defining it to Genie. A useful technique involving output routines is to recognize certain 'print' operators "Π" (say) which may appear at any point in a formula and result in the printing of its operand when that is evaluated, e.g. in the definition:

$$y = (2\Pi (\Sigma_{i=1}^{10} A_i) + \Pi \Sigma_{i=1}^{10} (A_i^2))/N$$

the presence of the operators "Π" would result in the printing of the results of the two summations during the evaluation of y without otherwise affecting the result. Normally, such operators

are controlled by sense switches.*

---

* These and other techniques are discussed in detail in a separate memorandum.

---

6. Conclusions

In terms of the presently active generation of computers, the Genie structure provides at a low cost in programming the basis upon which conventional assembly, compiling, loading and interpretive routines can be achieved. On the Rice Institute Computer, Genie itself takes less than 2,000 orders, and on these both an assembly program and an algebraic compiler of some sophistication can be built with the addition of about 500 orders apiece. Even on a machine such as the IBM 709, it is probable that the basic effect could be achieved in less than 15,000 orders. More important, however, is the fact that the addition of more complicated languages becomes increasingly easy since their evaluation operators may be described in terms of those defined previously.* There is an absolutely minimal

---

* The simplest language in Genie is, of course, binary machine code.

---

increase in storage requirements, represented by the size of the combined Character and Reduction Tables and their appendages, but this may be as little as 100 cells on a given machine. Often the Tables for two languages will differ only slightly and can be combined into one with only slight cost in changing

from one language to another. That this can be achieved, at the same time bringing a unified approach to computing machine applications seems to weigh in favor of the type of analysis represented by Genie.

There are two cases in which Genie can be applied to more recently designed types of machine, almost without alteration. The first is in computers with parallel computing units and shared memories. In this case the Symbol Table and all definitions stemming from it will be in the memory as before. Now, however, there will be two or more units applying the Execution Rule to the Symbol Table, each subject to the same conditions as before except that it will be possible to detect, of any undefined symbol, whether it is in process of definition by some part of the machine. It may then be desirable to delay the evaluation of quantities which are dependent on this, or in some circumstances a second evaluation may be started, safeguarded by a trapping interlock device, in the hope that the first evaluation will be completed before its argument is required by the second process. A similar situation arises even on single sequence machines when a program is dependent on data supplied by a parallel operating input device.

An allied problem is concerned with human interaction with the definitions in the machine. Such interaction is ineffective without communication at the symbolic level which is provided by Genie, and continuous control of problem execution is mandatory. The Symbol Control Rule was devised with this application in mind, where several coders simultaneously share the machine and the 'subroutine library' without getting their

own mnemonics and subroutine names confused. Preliminary studies indicate that the present rules are adequate for time-sharing between a number of mathematician-operators and a long production routine, even on a single sequence machine, because of the additional information carried in ST and the pathfinder list, which enables evaluation to be interrupted at almost any point in a program.

It may be argued that the simplicity of Genie is achieved by oversimplifying the translation process, but we feel that this is untrue. An actual process of translation from a formula to sequential code consists of three parts:

(i) An initial equivalence transformation of the formula.

(ii) The translation into sequential code.

(iii) A final equivalence transformation of the sequential code.

Now in fact we have described only (ii), but Genie does include routines of types (i) and (iii) which are applied with varying success in order to improve the efficiency of the evaluation. The problem is easy enough to state,* but not easy to

---

* Equivalence transformation for arithmetic, for example, are implicit in the axioms of Peano; equivalence transformation for code have been given by Iu.A.Ianov, 'On equivalence and transformation of Program Schemes', Dok.Akad.Nauk. S.S.S.R., 113- No.1, 1957.

---

solve, and since we are interested in solving the problem by machine, the first step is to analyze it into these three stages. At a later date we may hope to achieve the synthesis arrived at by a human coder. Another relevent point here is that (iii),

for example, is applicable to the results of many different translators, and it is best applied after, rather than during code construction.

The formal description of computing processes from a theoretical standpoint has been given by A.P.Ershov[3], and this approach also appears to have a wider application at the practical level than can be achieved by syntactic means. One of the most beneficial results we can hope for is that with the flexibility offered by routines of the Genie type a powerful language will come into universal uses in communication with machines.

7. Acknowledgements

This investigation has benefitted greatly from testing and criticism by Miss Jane Griffin, Mrs. Ann Heard, Mrs. Jo Kathryn Mann and Mr. Carruth McGehee, who wrote the first version of Genie for the Rice Institute Computer. This work was supported in part by the National Science Foundation Grant, number G-7648.

8. References

(1) H.B. Curry, 'On definitions in formal systems', Logique et Analyse, 3-4, August 1958.

(2) S.C. Kleene, 'Introduction to Metamathematics', Van Nostrand and Co., Princeton, 1950.

(3) A.P. Ershov, 'On operator algorithms', Dok.Akad.Nauk.S.S.S.R. 122 No.6, p967, 1958.