Standards

H. BRIGHT, Editor

Specification Languages for Mechanical Languages and Their Processors*— A Baker's Dozen

A Set of Examples Presented to ASA X3.4 Subcommittee

BY SAUL GORN

Editor's Prescript:

This paper is being submitted simultaneously to the membership of the ACM and to the ASA X3.4 Subcommittee on Programming Language Standardization. Dr. Gorn heads Working Group X3.4.1, which is charged with the responsibility of establishing standards for the description of programming languages.

It is this Department's impression that some people are searching for an ideal programming language which will serve all users, for all purposes, on all machines. We feel that such a goal is not likely to be reached in the foreseeable future. We also feel, however, that the computing art has now progressed to a point where computer users can and will recognize the economic justifiability of attaining general understanding of precisely what certain particular programming languages consist of. In short, the computing community has now matured to a point where we should recognize the need for discipline and clarity in communication among humans, relative to communication between humans and machines. In contributing to the technique of language specification, this paper aids language development.—H.S.B.

Introduction

Many varieties of Mechanical Languages and the languages which specify their syntax exist. By many techniques and devices these syntactical languages specify concepts and processes. In these languages it is possible to specify the same object in different ways to obtain clarity from different points of view. The choice of the language depends upon its convenience in specifying or communicating a concept.

The purpose of this paper is to show to what extent the languages illustrated are capable of specifying such mechanical languages or their processors, and to what extent these specifying languages are equivalent in their ease of mechanical translation among themselves.

To dispel some of the confusion as to the power of applicability of these languages, one trivial example is worked over in a dozen different ways. The simplicity of this example provides the link permitting the comparison of the methods of specification. The processor (used by McNaughton as an illustration of logical nets) being specified we will call the *triple sequence alarm*. The corresponding input language of strings of zeros and ones we may call the *triple one sequenced strings*. The processor itself as shown in Fig. A may be considered a *data generator* or a *triple one sequenced string recognizer*.

The dozen specifying languages can be given the titles and classifications of Table 1. The groups not separated by double lines are mechanically translatable into one another. Translation between these groups calls for heuristic methods.

The examples will illustrate some of the terminology presented in the author's paper "Some Basic Terminology Connected with Mechanical Languages and Their Processors" (*Comm. ACM 4* (Aug. 1961), 336). Some of these languages are essentially one-dimensional (linear) and

* The material in this paper comes from the University of Pennsylvania's Office of Computer Research and Education and is an outcome of the work jointly supported by the National Science Foundation Grant G-14096 and the Air Force Office of Scientific Research AF-49(638)951.

some are essentially two-dimensional (graphic and tabular); some are purely sequential, and some permit simultaneous action; some are more suitable for behavioral (recognitional) specification, and some are better for structural (generative) specification; one is a command language where the others are descriptive; and one is a sublanguage of natural language where the others are mechanical. Finally, some are more suited to specifying languages and some to specifying processors.

For example, one might have a language which is suited to specifying languages structurally but which can be used to specify the *processors* of those languages behaviorally.

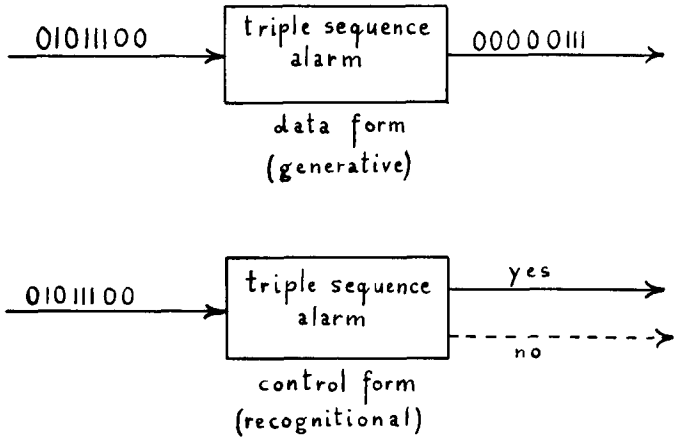

FIG. A

1. Natural Language

The "triple sequence alarm" is a device with one input and one output. The input will accept sequences of signals of equal duration chosen from two standard signals which we will designate by the symbols 0 and 1. The output will emit during each of the signal duration intervals one of the two signals, 0 or 1, acceptable by the input. The device is such that a signal 0 will be produced at each signal duration interval until the first occurrence of a signal 1 which had signals 1 as its two immediate predecessors; at that duration interval and for each successive interval until the end of the input string the output signal will be 1.

What we have just given is a descriptive, behavioral specification of the triple sequence alarm processor in a linear sequential sublanguage of a natural language.

TABLE 1. *Classification of Specifying Languages*

Title	*Dimension*	*Timing*	*Mode*	*Structural or Behavioral*	*Processor or Language*
1 Natural Language	1?	Sequential or Simultaneous	Descriptive or Command	Structural or Behavioral	Processor or Language
2 Regular Expression	1	Sequential	Descriptive	Structural..... Behavioral.....	Language Processor
3 Backus Normal Form	1	Sequential	Descriptive	Structural	Language
4 Trees	2	Sequential	Descriptive	Structural	Language
5 Prefix	1	Sequential	Descriptive	Structural	Language
6 State Diagram	2	Simultaneous	Descriptive	Behavioral or Structural	Processor
7 Symbol-State Diagram	2	Simultaneous	Descriptive	Behavioral or Structural	Processor
8 Incidence-Matrix	2	Simultaneous	Descriptive	Behavioral or Structural	Processor
9 Logical Net	2	Simultaneous	Descriptive	Structural	Processor
10 Propositional Logic With Time Variable	1	Simultaneous	Descriptive	Structural	Processor
11 Logic Equivalents	1	Simultaneous	Mixed	Structural	Processor
12 Turing Machine	1 or 2	Simultaneous	Descriptive	Behavioral or Structural	Processor
13 Flow Chart	2	Sequential or Simultaneous	Mixed or Command	Behavioral or Structural	Processor

2. Regular Expressions

The language-naming language we now present will have, as basic processors, generators and recognizers for a certain class of inifinite classes of finite strings of zeros and ones. One such infinite class, for example, is the class of all finite strings of zeros and ones containing somewhere three successive ones; that is, the class of all input strings for which the triple sequence alarm ends with a signal of 1.

The language is that of Kleene's [15] "regular expressions" as modified by McNaughton [16]. We now specify the "regular expression language-naming language" in natural language (our next example will present the specification in a mechanical language), but in a generative manner known as a "production system" (see Gorn [7, 8, 9]).

The regular expression language is a linear sequential language of strings of characters chosen from the alphabet {0, 1, (,), *, ∨}. Each string of characters from this alphabet which is a regular expression will be a partial specification of a linear sequential language of strings of characters chosen from the alphabet {0, 1}; it will be a structural specification when a processor is specified which will generate all these strings. The purely syntactic generative specification of the regular expression language is the following *production system:*

SY_a The one-character string, 0, is a regular expression.

SY_b The one-character string, 1, is a regular expression.

SY_c If a string designated by α is a regular expression, then the string composed in left to right order of "(", the characters of α, ")", and "*" is also a regular expression (briefly, if α is a regular expression, then so is "(α)*").

SY_d If the strings designated by α and β are regular expressions, then so is the string constructed by taking all the characters from left to right of α and following them immediately on the right by all of the characters from left to right of β (briefly, if α and β are regular expressions, then so is $\alpha\beta$). This basic procedure for all linear sequential languages is called "concatenation".

SY_e If the strings designated by α and β are regular expressions, then so is the string beginning with "(" concatenated on the right by the characters of α, followed on the right by "∨", then concatenated on the right by the characters of β, and ending with the character ")"; or more briefly stated, if α and β are regular expressions, then so is "$(\alpha \vee \beta)$".

Any particular regular expression is generated by "programming" a particular procedure of sequenced steps from the "order code manual" a, b, c, d, e. For example, applying productions a, b, and d, several times each, produces the strings $\alpha_1 = 1$, $\alpha_2 = 0$, $\alpha_3 = 10$, $\alpha_4 = 110$, and $\alpha_5 = 111$; applying e several times produces the strings $\beta_1 = (0 \vee 1)$, $\beta_2 = ((0 \vee 10) \vee 110)$; applying production c several times produces $\delta_1 = ((0 \vee 1))^*$ and $\delta_2 = (((0 \vee 10) \vee 110))^*$; finally, applying production d several times produces the regular expression

$$\alpha = (((0 \vee 10) \vee 110))^*111((0 \vee 1))^*$$

The specification of the "semantic content" of regular expressions can parallel the syntactic generative specification as follows:

SE_a The regular expression 0 designates the class of "words" or strings over the alphabet {0, 1}, which class contains the single element 0; i.e. 0 designates the unit class of "binary strings", {0}.

SE_b The regular expression 1 designates the unit class of "binary strings", {1}.

SE_c The regular expression designated by $(\alpha)^*$ designates the class obtained from the class designated by α as follows: if α_1, α_2, α_3, $\cdots$, etc. designate any binary strings belonging to α, then any strings of the form "null", α_1, $\alpha_1\alpha_2$, $\alpha_1\alpha_2\alpha_3$, $\cdots$ belong to $(\alpha)^*$.

SE_d The regular expression designated by $\alpha\beta$ designates the class obtained from the two classes designated by α and β by concatenating any "word" belonging to α on the left of any word belonging to β:

$$\alpha\beta = \{\alpha_i\beta_j : \alpha_i \in \alpha, \quad \beta_j \in \beta\},$$

to use the mathematical notation.

SE_e The class of words designated by a regular expression $(\alpha \vee \beta)$ is obtained by taking the union of the classes designated by regular expressions α and β, respectively, i.e. $\delta \in (\alpha \vee \beta)$ if either $\delta \in \alpha$ or $\delta \in \beta$. Thus, it is easy to see that the regular expression

$$\alpha = (((0 \vee 10) \vee 110))^*111((0 \vee 1))^*$$

constructed above denotes the class of strings (the "extent" of the language) comprising the language of strings of zeros and ones mentioned above, i.e. the class of all finite strings of zeros and ones containing somewhere three successive ones. The stated regular expression is even more specific in that it marks the first occurrence, reading from left to right, of three successive ones.

It is obvious that the regular expression type of language need not be restricted to languages with binary alphabet, {0, 1}.

3. Backus Normal Form

We now present a mechanical language designed for generative syntactic specification of digital linear sequential mechanical languages. Its use is called "specification by Backus normal form", because John Backus of IBM introduced it into the report of the international committee on ALGOL 60 (see *Comm. ACM* [21]). It was first presented in [25]. We will first specify the language using natural language as the syntax language, and will then illustrate its use by specifying in it both the language of regular expressions and the language of strings for which the triple sequence alarm was a recognizer.

A Backus syntactic specification language is a linear sequential language over an alphabet (infinite) composed of the following symbols:

a. "bracket" symbols "⟨" and "⟩"

b. "production" symbol "::="

c. "choice" symbol "|"

d. two alphabets called "names" and "symbols": Both of these are often selections of words from a prior alphabet, and can be infinite in number (not in length). If a control processor is provided which will distinguish in any given occurrence the characters within these words from the characters in

a, b, and c, then such characters as ⟨, ⟩, :, ::, =, :=, ::=, and | may appear within them. Usually this is not the case and the "names" and "symbols" chosen are restricted in format with respect to the occurrence of these characters.

The individual *names* are intended to denote *auxiliary* languages called *syntactic types* of the language being specified; a string of symbols consisting of "⟨" followed on the right by a name and then followed by "⟩" is intended to designate the extent of the name, i.e. the set of strings in the specified language of which it is the name—or, put another way, surrounding a name by these brackets is an operation transforming the intent designated by the name to the set of strings forming the corresponding extent. The intent of a language is the set of processors which will operate upon it.

A *symbol* is intended to name itself; in other words, the individual *symbols* are designated strings of characters from the alphabet of the language being specified, and often are designations *by the objects themselves* of auxiliary languages with only one word, namely that string. The symbol "|" is used to chain together alternative forms, each of which yields composite syntactic types which are auxiliary languages formed by concatenation. Thus a symbol concatenated with a symbol designates another symbol, a bracketed name concatenated with a symbol designates the set of words obtained by concatenating that symbol with each word of the set designated by the name, a bracketed name concatenated with a bracketed name designates the *product set* of all strings designated by one concatenated on the indicated side by all strings designated by the other.

By using a string of the form: *bracketed name* concatenated on the right by "::=" concatenated on the right by *a composite name* constructed with "|", bracketed names, symbols, and concatenation we obtain in a very compact form a set of production rules for the sublanguage designated by the bracketed name at the extreme left of the string.

For example, the language specified by the regular expression

$$\alpha = (((0 \vee 10) \vee 110))^*111((0 \vee 1))^*$$

cited in example 2, and given the name *triple one sequenced strings*, could be specified in Backus normal form as follows:

T_1 ⟨null string⟩ ::=
T_2 ⟨single character string⟩ ::= 0 | 1
T_3 ⟨elementary excluded string⟩ ::= 0 | 10 | 110
T_4 ⟨general excluded string⟩ ::= ⟨null string⟩ | ⟨general excluded string⟩ ⟨elementary excluded string⟩
T_5 ⟨general string⟩ ::= ⟨null string⟩ | ⟨general string⟩ ⟨single character string⟩
T_6 ⟨triple one sequenced string⟩ ::= ⟨general excluded string⟩ 111 ⟨general string⟩

Note the tacit assumption in T_4 and T_5 that the null string concatenated on either side by a string yields that string. Note also the similarity in form between T_4 and T_5; it will clearly appear whenever a regular expression ending with an asterisk is being specified.

A Backus syntactic specification is a compact and formal specification, in an appropriate language, of a production system. The specification restrictions implicit in Backus normal form place the languages so specified in the class of "Phrase Structure Languages" (see Chomsky [4, 5]).

As another example, a mechanical specification of the regular expression language which we specified in natural language in example 2 is the following:

R_1 ⟨regular expression⟩ ::= ⟨elementary expression⟩ | ⟨or expression⟩ | ⟨star expression⟩
R_2 ⟨star expression⟩ ::= (⟨regular expression⟩)*
R_3 ⟨or expression⟩ ::= (⟨regular expression⟩ ∨ ⟨regular expression⟩)
R_4 ⟨elementary expression⟩ ::= ⟨null string⟩ | 0 | 1 | ⟨regular expression⟩ ⟨regular expression⟩

Finally, let us give as much specification as we can of ⟨Backus language⟩ in Backus language:

Let the two alphabets (decidable languages), ⟨name⟩, and ⟨symbol⟩, be already specified; let, further, the symbols "←" and "I" and the operation of juxtaposition have the meanings we have described above for "::=", "|", and concatenation; finally, let "::=" and "|" be two special characters of the symbol alphabet. Then

B_1 ⟨subject⟩ ← ⟨name⟩
B_2 ⟨predicate⟩ ← ⟨name⟩ I ⟨symbol⟩ I ⟨predicate⟩ | ⟨predicate⟩ I ⟨predicate⟩ ⟨predicate⟩
B_3 ⟨Backus language⟩ ← ⟨subject⟩ ::= ⟨predicate⟩ I ⟨subject⟩ ::=

Clearly, there is much more than just the Backus specification in this specification.

4. Trees

Let us now consider a two-dimensional language for syntactic structural specification. Let us call it a language of *rooted trees*, or, when no confusion is possible, simply *tree language*. We now change our method of specification to suit the purposes of our main example, the specification of the triple sequence alarm. Instead of specifying the tree language directly, we will specify the relevant sublanguage of the tree language as the end product of the following generative rules of translation from the generating program for a regular expression.

Corresponding to each of the generating productions SY_a-SY_e of regular expressions, as given in Section 2, there is a translating rule to obtain recursively the ramified structure (i.e the tree) in our two-dimensional language. The rules may be applied "from the *end-points* up" or "from the *root* down".

A tree is a two-dimensional structure of the type known in combinatorial topology as an *acyclic graph*. We will scan it from top to bottom, placing the *node* called *root* at the top. It is composed of *nodes* and connecting lines called *branches*. Into each node except the root, one and only one connecting line will *descend*. The nodes from which no connecting lines descend will be called *end-points*. From each

node which is not an end-point, one or more connecting lines may descend depending upon the *stratification number* or *order* of the node. The nodes will be considered to be storage cells each capable of containing one character from a specified alphabet; each such character is assumed to have a corresponding natural number (including zero, for end-points) called its stratification number. The storage cell at a node of order n may contain only characters of stratification number n. It will be convenient, while a tree is being *constructed*, to permit characters at an end-point which do not belong to the stratified alphabet but to an auxiliary alphabet of *variables* whose values can be trees; this will permit us to generate a tree by hanging trees at end-points of already generated trees.

The trees corresponding to regular expressions are then specified as follows: The alphabet is composed of:

a. connecting lines (lengths unspecified).
b. the characters of the alphabet of the language specified by the regular expressions—for example, for the regular expression of which the triple sequence alarm is the recognizer, the alphabet is {0, 1}. Each of these characters will have stratification number zero.
c. the character "*", of stratification number one. The translation process will automatically distinguish its use in the tree language from its use in regular expressions; i.e. it will automatically select its "interpretation". Similarly for the character "$\vee$", except that its stratification number is two.
d. the new character C of stratification number two, to which there corresponds the operation of "concatenation" in SY_d, but no character in the regular expression alphabet.
e. the alphabet of auxiliary variables, α, β, γ, $\cdots$ having no characters in common with the alphabet specified in b.

The translation rules for transforming regular expressions into trees are then the following:

$T_{a,b}$ The characters of the alphabet of the language specified by the regular expression (e.g. 0 and 1) will be end-points.
T_c $(\alpha)^*$ will be replaced by a of Figure 1.
T_d $\alpha\beta$ will be replaced by b of Figure 1.
T_e $(\alpha \vee \beta)$ will be replaced by c of Figure 1.

Although nothing is said about the order in which we recognize the applicability of or apply these translation rules, just as nothing was said about the order of application of SY_{a-e} in the generation of a regular expression, sequencing programs for them can nevertheless be designed to drive all variables out of the trees in process of generation, with an end result having only characters mentioned in b as end-points.

For example, our regular expression,

$$\alpha = (((0 \vee 10) \vee 110))^*111((0 \vee 1))^*$$

translates into the tree shown in Figure 2.

Because of the fact that we deliberately ignored the "associativity" of the $\vee$-operator (at the expense of extra parentheses, this was done to make the productions SY_{a-e} simple), there is never any doubt about the ordering of the branches following an $\vee$-*node*. Since, however, concatenation is essentially associative, choice of the possible concatenation schemes is arbitrary. For example, the tree a of

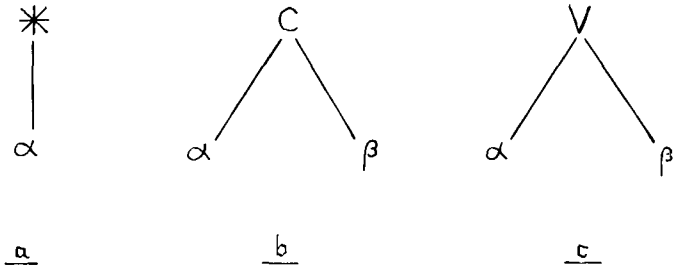

FIG. 1

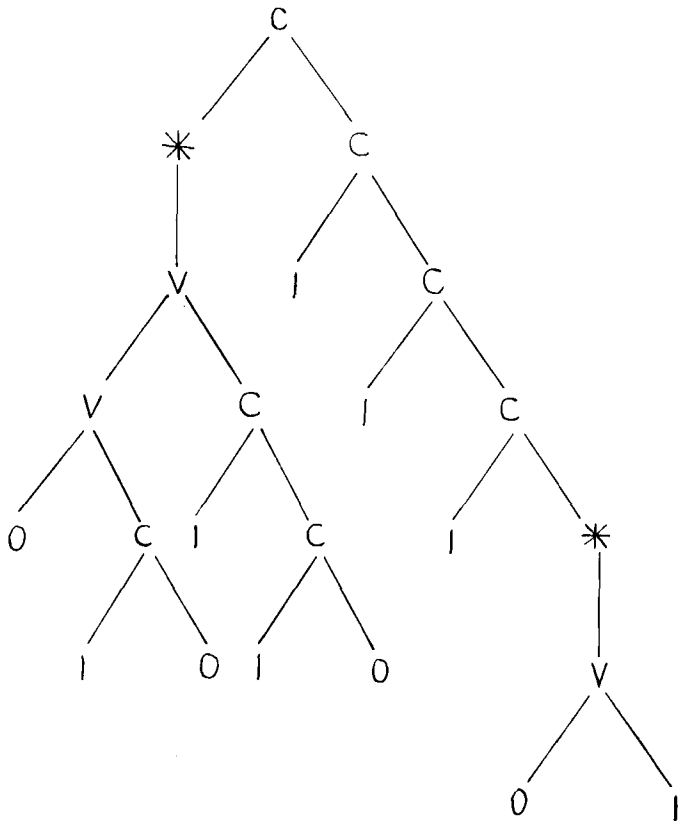

FIG. 2

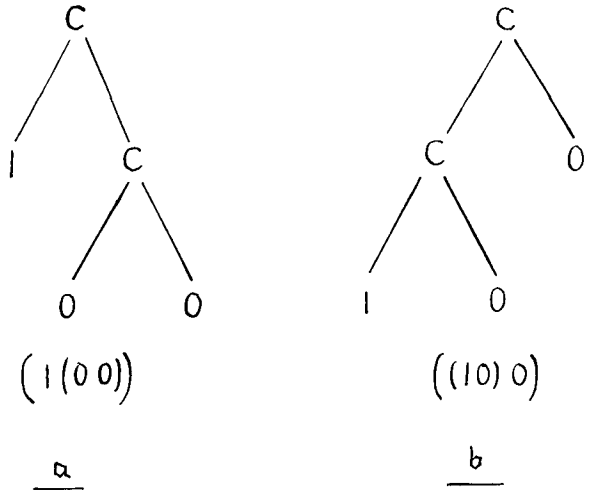

FIG. 3

Figure 3 is *equivalent* to the tree b of Figure 3. A regular expression has a translation into a number of trees, but they are all *equivalent.*

5. Prefix Language

Whereas our first example specified a processor, the next three each specified a language of which that processor is a recognizer. It was clear, however, that in each case, if we had used other and larger alphabets, the specifications themselves would have undergone only unessential changes. In short, we have specified families of languages, i.e. language functions in which the alphabet is a free variable. Let us now present a linear sequential language (function) for the designation of trees which is more directly concerned with tree processing than, say, the original regular expressions. The type is called a *prefix language* (see Gorn [7, 8, 11]).

Let us specify it by means of the translation process from the two-dimensional languages of example 4. The alphabet in each case is that of the nodes in the tree languages.

Given a tree, it may be specified in prefix language by arranging the node characters in a string as follows: Begin by placing the root character at the head of the string; after each character there are three mutually exclusive possibilities for the next character:

a. The character is not at an end-node: in this case the next character will be that immediate successor in the tree which is on the left-most branch emanating from the node.
b. The character is an end-node, but is not the rightmost end; in this case retrace to the first predecessor not all of whose immediate successors have been placed in the string yet and find the leftmost branch which has not yet been traced; the next character to be placed on the right end of the generated string is the immediate successor on this branch.
c. The character is the rightmost end node: in this case the translation is complete.

For example, the triple-sequence specifying tree of our last example has as designation in prefix language the string:

C * ∨ ∨ 0 C 1 0 C 1 C 1 0 C 1 C 1 C 1 * ∨ 0 1

In short, the order from left to right in which the node characters appear in the prefix language is the order in which a rat seeking the last end would first meet the nodes of the tree as a maze, if the rat could remember that it had already been at a node and preferred to keep to its right and to continue down.

The manner of specification of the translation process is an example of a *priority control* with an overriding priority to case a (see Gorn [8]).

For *logical* alphabets this language will be recognized as the *Polish parenthesis-free notation*, which has also been frequently applied to "algebraic" alphabets. All the *operators* are prefixed and none is used as an infix as in the traditional notation for binary operators (e.g. $x+y$).

Prefix languages systematically replace *control characters* (parentheses) by a standard scanning process, i.e. certain characters may be replaced by interpreters if the *command characters* are given an appropriate scanning priority.

Such syntactic concepts as *scope of a character, depth of a character, tree address of a character,* etc. are readily defined for prefix languages, and give rise to such standard processors as *scope analyzers, depth analyzers, tree address generators and selectors,* etc. in addition to the usual generators and recognizers of words. There is also a simple standard production system for the specification of all prefix languages (see Gorn [8]).

Oettinger [22] has also designed standard translation processors between prefix languages and corresponding languages with infix notation.

6. State Diagrams

Let us now return from examples of language specification to examples of languages for processor specification. Consider next a two-dimensional language for processor specification (mainly behavioral) called *state diagrams* (see Mealy [17], and Moore [19]). We begin by developing the state diagram of the triple sequence alarm.

We imagine that the triple sequence alarm begins in an *initial state,* S_1, when it is presented with a string of zeros and ones at its input. If the input string is read from left to right, the processor will always return to state S_1 if a zero is scanned before three successive ones have appeared, i.e. S_1 might be called the *disappointed state.* If the processor is in state S_1 and a one appears, there is a *transition* to *the first waiting state* S_2. If the processor is in state S_2 and a one appears, there is a transition to the *second waiting state* S_3. If the processor is in state S_3 and a one appears, there is a transition to the final state S_4. If the processor is in state S_4, all transitions are back to state S_4 whether a zero or a one appears. The state diagram is a directed graph whose nodes are marked with state names, and whose branches between nodes refer to transitions and are marked with arrows and with the output signal resulting from an input signal. Thus a state diagram for the triple sequence alarm is shown in Figure 4.

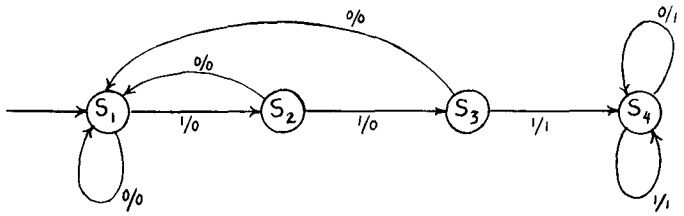

FIG. 4

The two-dimensional language of state diagrams has the following alphabet:

a. the input signal alphabet
b. the character "/"
c. the output signal alphabet
d. the alphabet of *state names*
e. the circles of variable size
f. *connecting arrows* of variable size

Intermediate alphabets include:

a. *state nodes* composed of circles containing state names; one marked with an initial arrow is called the *initial state*;
b. *input-output designators* composed of strings of three characters, reading from left to right: an input alphabet character, a "/", and an output alphabet character;
c. *transition designators* composed of a directed arrow and an input-output designator in its neighborhood and "closer" to it than to any other directed arrow.

A state diagram is composed of distinctly named nodes (no two with the same name) connected by transition designators and such that there is an exiting transition from each state for each input character. Thus the number of transitions is equal to the number of input characters times the number of states.

The procedure for translation into state diagram language of a natural language specification of a processor is at least as ill-defined as the latter.

7. Symbol-State Diagrams

The next two-dimensional language for processor specification is simpler than the state diagram in that it does not require the complicated transition designators, but is more complicated in that it calls for two directed graphs per processor. If we introduce a third alphabet for transition names, then the second directed graph needed will have its nodes labelled by input-output characters instead of by state names. The state diagram for the triple sequence alarm then translates into the 'Symbol-State diagram' shown in Figure 5.

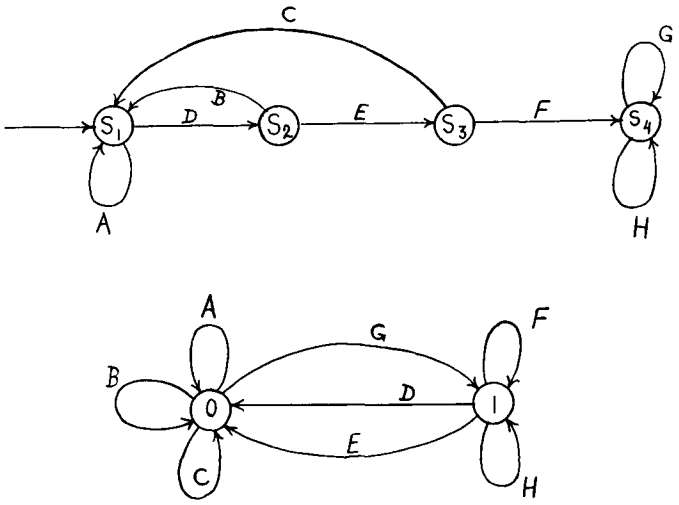

FIG. 5

8. Incidence Matrix

Another two-dimensional language for specification of a processor is an example of a *tabular format language* such as the type used to exhibit matrices in mathematics, and double entry tables in general. It is most simply specified by a translation procedure from the symbol-state diagram language. Let us call it the language of *symbol-transition-state matrices.*

The symbol-transition matrix has its rows labelled by symbols and its columns labelled by transition names. The transition-state matrix has its rows labelled by transition names and its columns labelled by state names. The alphabet of *matrix elements* is composed of four characters: 0, +1, −1, and ±1. An entry of 0 is placed at the intersection of a row and column in either matrix if the designated transition neither enters nor exits from the designated state in one case, and neither enters nor exits from the designated symbol in the other. An entry of ±1 is placed at the appropriate cell if the transition arrow both enters and exits from the state-node or symbol-node respectively. An entry of +1 or −1 is placed in the appropriate cell of the appropriate matrix if the transition enters or exits, respectively, from the corresponding node in the corresponding graph.

Thus the symbol-transition-state specification of the triple sequence alarm is the following pair of tables:

transition / character	A	B	C	D	E	F	G	H
0	±1	±1	±1	+1	+1	0	−1	0
1	0	0	0	−1	−1	±1	+1	±1

state / transition	S_1	S_2	S_3	S_4
A	±1	0	0	0
B	+1	−1	0	0
C	+1	0	−1	0
D	−1	+1	0	0
E	0	−1	+1	0
F	0	0	−1	+1
G	0	0	0	±1
H	0	0	0	±1

The language for either matrix could have been simplified to use only a binary entry alphabet {0, 1}. For the symbol-transition matrix, this would require distinguishing input and output characters, and having a separate row for each. For the transition-state matrix, it would call for a distinction between *states-on-entry* and *states-on-exit* and would require a separate column for each.

There are a number of well-known algorithms for processing these *incidence-matrices* and also for the derivation of *path-matrices* and *connection matrices.* These algorithms yield results useful in switching theory (Hohn [13, 14]), combinatorial topology (Veblen [23]), and graph theory (Berge [1]).

9. Logical Net

The next language type we consider is again two-dimensional, can specify simultaneous action, and is

most useful for structural specifications of processors. Languages of this type are therefore much used in logical design (Burks-Copi [3], Burks-Wright [2], Wilkes [24], etc.).

As a simplified example, let us specify one called *logical net language.*

The logical net language is a two-dimensional language with the following alphabet:

a. an alphabet of *gate-type designators*—we will use the characters $\vee$, $\wedge$, and $\sim$, calling them OR, AND, and NOT, respectively; each character of this alphabet has a "stratification number", 1 for $\sim$, and 2 for $\wedge$ and $\vee$;
b. an alphabet of *delay designators*—in our example we will use $\{j, k\}$;
c. an alphabet of *wire designators*—in our example we will use $\{i, j, k, p, q, r, s, u, v, w\}$;
d. a rectangular box of fixed size, called a *delay*, and designating a unit time interval between *entrance* and *exit*;
e. a circle of fixed size, called a *gate*;
f. connecting lines of varying size with an arrow head at one end, called *wires*;
g. a *dot* called an AND-*node*, and designating simultaneous continuations of a wire in a number of directions;
h. a " " symbol in a variety of orientations to designate the nonconnection of crossing wires.

Among the intermediate alphabets we might mention:

a. *gate-designators* composed of gates containing a gate-type character—thus $\textcircled{\sim}$ is a NOT-*gate*, $\textcircled{\vee}$ is an OR-*gate*, $\textcircled{\wedge}$ is an AND-*gate*;
b. *delay symbols* composed of a delay rectangle containing a delay designator;
c. *delay elements* composed of a delay symbol with one input wire and one output wire;
d. *gate elements* are gates with one *output wire* and a number of *input wires* equal to the stratification number of the gate type designator in the gate designator.

A logical net is, then, a two-dimensional connected array of these elements in which wires intersect only at AND-nodes and not at U-symbols. At each time unit it is assumed that each wire is "transmitting" one of two signals, designated by 0 and 1, respectively. For example, Figure 6 illustrates a logical net for the triple sequence alarm.

Here i marks the unique *input wire* and u the unique *output wire.* A logical net containing no delay elements is called a *combinational net.* For example, the net contained in the dotted box marked C is a combinational net with three input wires (i, j, k), and three output wires (u, v, w). A logical net which is not a combinational net is called a *sequential net.*

There is a heuristic translation procedure permitting one to translate from the behavioral specification of a processor which is provided by a state diagram into a structural specification of that processor in the logical net language. Let us illustrate this translation procedure for the triple sequence alarm.

First of all, there is the theorem to the effect that a logical net containing n unit delays corresponds to a processor having 2^n states. Second, there is a *normal-form* theorem to the effect that every net with I inputs, U outputs and D delay units is equivalent to a net constructed from a combinational net C with I+D inputs and U+D outputs where D inputs and D outputs are paired to feed back from an output of C through a delay and back into the corresponding input of C. The net in the diagram of Figure 6 is an example.

Thus the triple sequence alarm, having four states, requires a minimum of two delays, j and k, with outputs similarly marked. Let their inputs be v and w respectively, and let state S_1 correspond to outputs $j = 0, k = 0$; S_2 to $j = 0, k = 1$; S_3 to $j = 1, k = 0$; and S_4 to $j = 1, k = 1$. The triple sequence alarm calls for an output 1 only for the three transitions F, G, and H, so that $u = 1$ if and only if either $j = k = 1$ (for G and H from state S_4) or $i = 1$ and $j = 1, k = 0$ (for F from state S_3); in other words, the required combinational net C must have $u = jk + ij\bar{k} = j(i + k)$ in the notation of boolean algebra. This condition on u was therefore obtained by examining the entrances into state S_4. Now consider the entrances into all states for which $j = 1$, i.e. the states S_3 and S_4 via the transitions E, F, G, and H; the transition E occurs when the input is 1 $(i = 1)$ and the preceding state was S_3 $(j = 0, k = 1)$; hence $v = 1$ if and only if either $i = 1$, $j = 0$, and $k = 1$, or $j(i + k) = 1$ as before; this yields the condition $v = i\bar{j}k + j(i + k)$ which "simplifies" to $v = ik + j(i + k)$. Finally, the entrances into all states for which $k = 1$ is provided by the transitions D, F, G, H, yielding $w = i\bar{j}\bar{k} + j(i + k)$ because D is marked for an input 1 at state S_1 where $j = k = 0$; manipulation in boolean algebra permits us to *simplify* this condition into $w = i\bar{k} + j(i + k)$.

Rewriting these three conditions with our net operators then yields the following behavioral specification of the required combinational net C:

$$u = [j \wedge (i \vee k)],$$
$$v = [j \wedge (i \vee k)] \vee (i \wedge k),$$
$$w = [j \wedge (i \vee k)] \vee (i \wedge \sim k).$$

These behavioral specifications can be met economically by introducing intermediate gates and lines as fol-

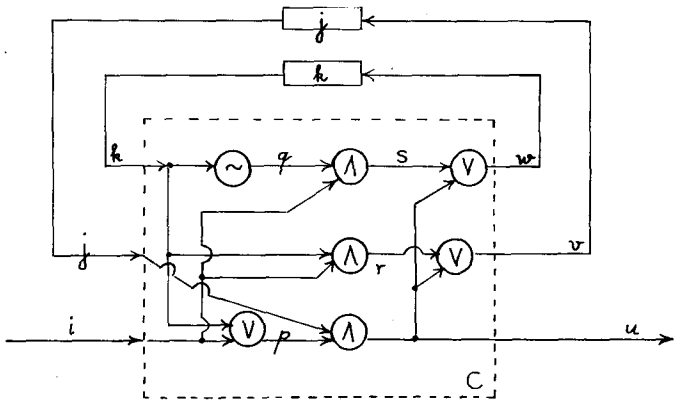

FIG. 6

lows:

$$p = i \vee k$$
$$q = \sim k$$
$$r = i \wedge k$$
$$s = i \wedge q$$

to yield

$$u = j \wedge p$$
$$v = u \vee r$$
$$w = u \vee s.$$

Thus a NOT-gate is needed from k, and an AND-node is needed from k to lead to an input to the AND-gate for r and to the OR-gate for p. Similarly, an AND-node is needed to lead to another input of the AND-gate for r and to an input of the AND-gate for s, whose other input is q. Finally, AND-nodes are needed from u to the OR-gates yielding v and w from the other inputs r and s respectively.

These detailed requirements yield the complete logical net.

10. Propositional Logic with a Time Variable

We can now specify a descriptive linear, simultaneous action language within which we can provide structural specifications of sequential processors. This is the language of *propositional logic with time as a variable.*

The language of propositional logic with time as a variable has the following alphabet:

a. an alphabet of ordinal numbers to designate time units
b. the characters − and + to designate time unit differences in the past or in the future
c. the character t to designate a variable instant of time
d. the control characters "(" and ")" for the construction of *signal designators* and the bracketing of *logical expressions*
e. a *logical operator* alphabet, e.g. $\sim$, $\wedge$, and $\vee$ to designate NOT, AND, and OR, respectively
f. the character "=" to designate the declarative verb *is identical as a signal with*
g. the control character ";" to serve as a statement separator
h. an alphabet of *wire names* (e.g. p, q, $\cdots$), but not to include t

Intermediate languages (syntactic types) will include:

a. ⟨signal designator⟩ ::= ⟨wire name⟩(⟨time designator⟩)
b. ⟨time designator⟩ ::= $t \mid t - \langle\text{ordinal}\rangle \mid t + \langle\text{ordinal}\rangle$
c. ⟨logical expression⟩ to be similar to those of logic, but using the alphabet of signal designators as variables
d. ⟨statement⟩ ::= ⟨signal designator⟩ = ⟨logical expression⟩

A mechanical procedure can be specified to translate from the logical net language into the language of propositional logic with time as a variable. When applied to the triple sequence alarm it yields the following specification:

$u(t) = p(t) \wedge j(t)$; $p(t) = i(t) \vee k(t)$; $v(t) = u(t) \vee r(t)$;
$r(t) = i(t) \wedge k(t)$; $w(t) = s(t) \vee u(t)$; $s(t) = q(t) \wedge i(t)$;
$q(t) = \sim k(t)$; $j(t) = v(t-1)$; $k(t) = w(t-1)$;

This specification can be manipulated in various ways to eliminate various wires which are not inputs or outputs of C. It can also be manipulated (algebraically) to yield the output as a function of the input and supply a check against an initial behavioral specification of a processor.

11. Logical Equations

The language of propositional logic with time as a variable may be mechanically translated into a mixed (descriptive and command), linear, simultaneous action language for structural specification of processors. This language has been called *logical equations* (see Gorn-Inger-man-Crozier [12]).

The alphabet of logical equations adds to that of propositional logic with time as a variable the following characters:

a. 0 and 1 to designate the *base alphabet* of the processor, i.e. its two signal levels
b. "→" to designate the imperative verb (command) *transmit to*
c. "≡>" to designate the causal relationship between having such and such a state and having such and such a transmission occur—this is a *control verb*

Important sublanguages (*clauses*) are, then:

a. ⟨signal⟩ ::= 0 | 1
b. ⟨action clause⟩ ::= ⟨signal⟩ → ⟨signal designator⟩
c. ⟨state clause⟩ ::= ⟨logical expression⟩ = ⟨signal designator⟩
d. ⟨control sentence⟩ ::= ⟨state clause⟩ ≡> ⟨action clause⟩

Although other systems of logical equations might be more *economical* in specifying a processor, one can always be obtained by immediate translation from a specification in the language of propositional logic with time as a variable by transforming each statement: ⟨signal designator⟩ = ⟨logical expression⟩ into the control sentence: ⟨logical expression⟩ = 1 ≡> 1 → ⟨signal designator⟩. Thus the specification of the triple sequence alarm would begin as follows:

$$p(t) \wedge j(t) = 1 \equiv> 1 \rightarrow u(t);$$
$$i(t) \vee k(t) = 1 \equiv> 1 \rightarrow p(t);$$
$$\text{etc.}$$

This language is clearly equivalent to a two-column tabular format language in which states in logical equation form are entered in the first column and the corresponding actions are entered in the second column.

12. Turing Machines

The next language class has already been of great theoretical utility in the foundations of mathematics and logic. It is that group of languages in which *Turing machines* are specified (see Davis [6], Moore [20], and Minsky [18]). In the idealized forms needed for the theory it is assumed that there is available a machine which can be in a number of states S_1, S_2, . . . and which scans one or more *tapes*, each a sequential arrangement of *cells* capable of containing one character each of an alphabet called *symbols*. Actions result from a state and an input symbol, and in the idealized forms these actions are composed of transitions to other states, motions of the tape or tapes one cell to the right or left, erasures in cells, and placing new symbols in cells. A specification language for Turing

machines must therefore begin with the following alphabets:

a. state names
b. symbols
c. special action names, e.g. R and L for move tape right or left

The language itself can then be specified in a manner similar to that in example 11 above, or in the two-column tabular format mentioned there.

The triple sequence alarm may be specified in this manner if we assume:

a. an *input tape* which always moves one cell to the right at each action, and whose current cell is the storage position i, and
b. an *output tape* which always moves one cell to the right at each action, and whose current cell is the storage position u;
c. either the state name aphabet of the state diagram above, S_1, S_2, S_3, S_4, or the state name alphabet $(jk) = (00), (01), (10), (11)$ of the corresponding outputs of the pair of delay elements in the logical net above; let us call the two delays d_1 and d_2, and consider them one-bit storage cells whose contents last one time unit each.

The specification of the triple sequence alarm can therefore simplify into either of the following tables:

State d_1d_2:	Symbol i	New State d_1d_2:	Output Symbol u
0 0	0	0 0	0
0 0	1	0 1	0
0 1	0	0 0	0
0 1	1	1 0	0
1 0	0	0 0	0
1 0	1	1 1	1
1 1	0	1 1	1
1 1	1	1 1	1

	S_1	S_2	S_3	S_4
0	S_10	S_10	S_10	S_41
1	S_20	S_30	S_41	S_41

13. Flow chart

The last language class we would like to list for processor specification (structural) is the two-dimensional sequential command language class called *flow charts* which is much used by programmers. Such languages may be simultaneously acting, as in *micro-flow charts*, or purely sequential. One such language is specified in Gorn [10] together with a translation procedure to the linear sequential command languages known as *machine codes*. A translation procedure from *logical equations* into micro-flow-charts is described in Gorn-Ingerman-Crozier [12].

The flow-chart languages are good to exhibit at a glance the main *control flow* of a program; the *data flow*, however, though specified completely, is certainly not obvious. Control transformation of data into commands sometimes are obvious (as in programmed switches) and sometimes are not (as in variable instructions which are *computed*).

Processors may be *simulated* (i.e. specified by a program which only partially translates their full specification) at many levels of detail, from the bit logic at micro-level, through the behavior at register level (as in an *interpreter* or an *automonitor*), into a simulation of the behavioral specifications only (i.e. of input-output behavior only). The flow chart in Figure 7 shows an input-output simulator of the triple sequence alarm: the *current input bit* is stored in p, the preceding one in p_1, and the one before that in p_2. The current output signal is stored in L.

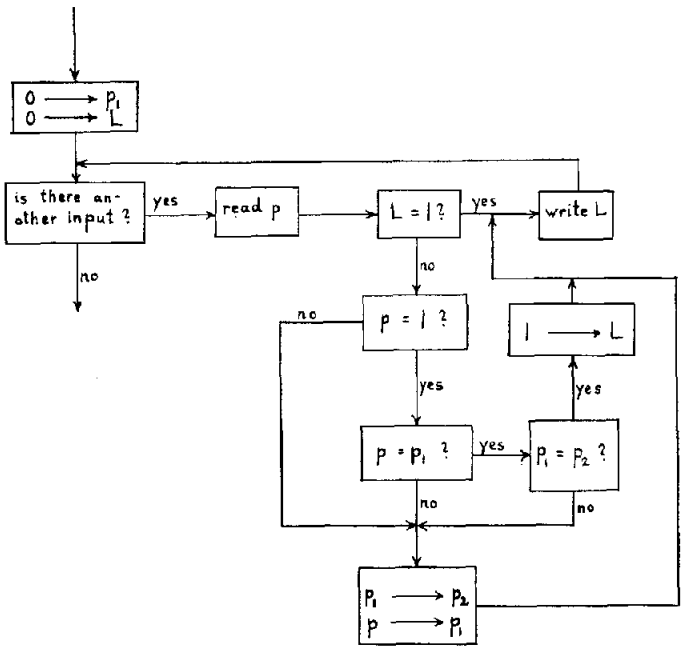

FIG. 7

A specification in flow-chart language of a translation processor from the linear language of algebraic expressions to a linear language of *machine code* may be found in Wegstein [26].

Acknowledgment

The author wishes to thank Frank A. Williams, Jr. for his help with the Introduction.

REFERENCES

1. BERGE, C. *Théorie des Graphes et ses Applications*. Paris, Dunod, 1959.
2. BURKS, A. W., AND WRIGHT, J. B. Theory of logical nets. *Proc. IRE, 41* (1953), 1357–1365.
3. BURKS, A. W., AND COPI, I. M. The logical design of an idealized general-purpose computer. *J. Franklin Inst. 261* (1956), 299–314; 421–436.
4. CHOMSKY, N. On certain formal properties of grammars. *Informat. Contr. 2* (June 1959), 137–167.
5. CHOMSKY, N. Three models for the description of language. *IRE Trans. Informat. Theory IT-2*, 3 (Sept. 1956), 113–124.
6 DAVIS, M. *Computability and Unsolvability*. New York, McGraw-Hill, 1958.
7. GORN, S. Common programming language task, PT. I, Sec. 5. Final Report AD59UR1, U.S. Army Signal Corps, Contract No. DA-36-039-sc-75047. The Moore School of Electrical Engineering, University of Pennsylvania (July, 31, 1959).
8. GORN, S. Common programming language task, Pt. I. Final Report, AD60UR1, U.S. Army Signal Corps, Contract No.

DA-36-039-sc-75047. The Moore School of Electrical Engineering, University of Pennsylvania (June 30, 1960).
9. Gorn, S. Some basic terminology connected with mechanical languages and their processors. *Com. ACM 4* (Aug. 1961).
10. Gorn, S. Standardized programming methods and universal coding. *J. ACM 4* (July 1957), 254–273.
11. Gorn, S. The treatment of ambiguity and paradox in mechanical languages. AFOSR TN-603-61, U.S. Air Force Contract No. AF-49(638)-951; The Office of Computer Research and Education, University of Pennsylvania (Apr. 1961).
12. Gorn, S., Ingerman, P. Z., and Crozier, J. B. On the construction of micro-flowcharts. *Commu. ACM 2* (Oct. 1960), 27–31.
13. Hohn, F. E. Boolean matrices and the design of combinational relay switching circuit. *Bell System Tech. J. 34* (Jan. 1955), 177–202.
14. Hohn, F. E., Seshu, S., and Aufenkamp, D. D. The theory of nets. *IRE Trans. Electr. Comput. EC-6*, 3 (1957), 154–161.
15. Kleene, S. C. Representation of events in nerve nets and finite automata. In *Automata Studies*, C. E. Shannon and J. McCarthy, eds., pp. 3–41. Princeton, Princeton University Press, 1956.
16. McNaughton, R., and Yamada, H. Regular expressions and state graphs for automata. WADC TN 59-192, U.S. Air Force Contract AF 33 (616)-5886; The Moore School of Electrical Engineering, University of Pennsylvania, Report No. 60-06 (July 1, 1959).
17. Mealy, G. H. Method of synthesizing sequential circuits. *Bell System Tech. J. 34* (Sept. 1955), 1045–1079.
18. Minsky, M. A 6-symbol 7-state universal Turing machine. Report No. 54G-0027 Contract AF 19(604)5200, Aug. 17, 1960, Lincoln Lab., MIT
19. Moore, E. F. Gedanken-experiments on sequential machines. In *Automata Studies*, C. E. Shannon and J. McCarthy, eds. pp. 129–153. Princeton, Princeton University Press, 1956.
20. Moore, E. F. A simplified universal Turing machine. Presented to the Association for Computing Machinery, 1952.
21. Naur, P. (ed.) Report on the algorithmic language ALGOL 60. *Comm. ACM 3* (May, 1960), 299–314.
22. Oettinger, A. G. Automatic syntactic analysis and the pushdown store. *Proc. Symposia in Applied Mathematics*, American Mathematical Society, 1961.
23. Veblen, O. *Analysis Situs*. New York, American Mathematical Society, 1931.
24. Wilkes, M. V. *Automatic Digital Computers*. London, Methuen, 1957.
25. Backus, J. W. The syntax and semantics of the proposed International Algebraic Language of the Zurich ACM-GAMM Conference. Presented at the First International Conference on Information Processing (ICIP) June 13-23, 1959 in Paris, France. International Business Machines Corp., New York, USA.
26. Wegstein, J. A general purpose pseudocode. Annual ACM Conf., Ann Arbor, Mich., June 1954.

A more extensive bibliography and further discussion of some of these languages may be found in:

27. McNaughton, R. The theory of automata—a survey. In *Advances in Computers*, v. 2, Franz L. Alt (ed.) New York, Academic Press, 1961.

Editor's Postscript:

Many of the concepts discussed in Reference 9, published here in the August 1961 issue of Communications, are of considerable importance and will be more meaningful following a reading of this paper. If you have not read the earlier paper, we urge you to do so.—H.S.B.

What is Proprietary In Mathematical Programming?— Impressions of a Panel Discussion

By L. Wheaton Smith
Richfield Oil Company, Los Angeles, California

A panel discussion on "What is Proprietary in Mathematical Programming?" was sponsored by the Special Interest Committee on Mathematical Programming of the ACM during a Hall of Discussion/on September 7th at the 16TH National ACM meeting in Los Angeles. This note consists solely of the impressions garnered by the moderator of the panel and does not necessarily represent the position of any of the panelists or other participants in the discussion.

The panelists were: Willis H. Ware, Associate Head, Computer Sciences Department, The RAND Corporation, Santa Monica, California; William Orchard-Hays, Vice President and Director of the Information Processing Division, CEIR, Inc., Arlington, Virginia; and Dale O. Cooper, Consultant, Bonner and Moore Engineering Associates, Houston, Texas.

Each panelist presented a brief statement of his organization's policy with regard to the matter under discussion. In addition, a brief, written statement by H. W. Crandall, Manager, Electronic Computer Center, Standard Oil Company of California, was read by the moderator.

There is no doubt that many of the applications of mathematical programming are highly proprietary in regard to content and even their existence. The discussion considered basic mathematics, algorithms and computer codes. With regard to all three categories, the attitudes expressed during the floor discussion ranged from complete and open distribution, through limited distribution, to complete secrecy. It was pointed out that the question of publication of mathematics and algorithms is exactly analogous to that of publication of scientific and technical information in any other field.

It was the sense of the discussion that any organization has the right to hold as proprietary any information that they wish to so classify. (The question of national interest and national defense did not arise.) The only area of question was, to what extent restricting the exchange of information and computer codes was consistent with the enlightened self-interest of an organization. Again, it should be emphasized that it was the sense of the discussion that each organization must make its own determination of policy in these matters.

Representatives of several organizations who are not in the business of supplying methods and codes indicated that they usually gave away general-purpose computer routines through computer manufacturers or users' groups. This was justified on the basis that they had recovered their own investment in these routines and felt that exchange of routines should be encouraged, since they them-

(*Please turn to Page 594*)