ARPA ORDER NO.: 189-1

R-876-ARPA

March 1972

A New Approach to Programming Man-Machine Interfaces

R. H. Anderson and W. L. Sibley

A Report prepared for

ADVANCED RESEARCH PROJECTS AGENCY

Rand
SANTA MONICA, CA. 90406

This research is supported by the Advanced Research Projects Agency under Contract No. DAHC15 67 C 0141. Views or conclusions contained in this study should not be interpreted as representing the official opinion or policy of Rand or of ARPA.

ARPA ORDER NO.: 189-1

R-876-ARPA

March 1972

A New Approach to Programming Man-Machine Interfaces

R. H. Anderson and W. L. Sibley

A Report prepared for

ADVANCED RESEARCH PROJECTS AGENCY

Rand
SANTA MONICA, CA 90406

Bibliographies of Selected Rand Publications

Rand maintains a number of special subject bibliographies containing abstracts of Rand publications in fields of wide current interest. The following bibliographies are available upon request:

Aerodynamics • Arms Control • Civil Defense
Communication Satellites • Communication Systems
Communist China • Computer Simulation • Computing Technology
Decisionmaking • Game Theory • Maintenance
Middle East • Policy Sciences • Program Budgeting
SIMSCRIPT and Its Applications • Southeast Asia
Space Technology and Planning • Statistics • Systems Analysis
USSR/East Europe • Weapon Systems Acquisition
Weather Forecasting and Control

To obtain copies of these bibliographies, and to receive information on how to obtain copies of individual publications, write to: Communications Department, Rand, 1700 Main Street, Santa Monica, California 90406.

Published by The Rand Corporation

PREFACE

If one can picture an engineer being given close and immediate control over the computer solution of a design problem, it should not seem too surprising that industry has realized up to eightfold increases in productivity through close coupling of man and computer. The intimacy of the engineer-computer interaction, however, requires a man-oriented scheme for interfacing the engineer with complex data structures and computer systems.

Man-machine interfaces for complex computer systems are traditionally difficult to build and modify, and they are becoming more complicated and expensive because, as their use increases, more is demanded of them. To complete the programming of a single interface, with traditional languages and techniques, for the highly interactive systems required today by government and industry can cost a quarter of a million dollars and take as long as a year.

And yet this interface, which is difficult to modify, ought to be flexible because computer systems that interact with humans must be flexible. The technical specialist will find in this report a new approach toward achieving such flexibility through the use of pattern-replacement rules capable of operating on complex data structures. These rules completely govern interface behavior; they are so organized as to be easily augmented, modified, or deleted, thus altering interface behavior. These pattern-replacement rules are, in effect, a succinct high-level computer programming language.

This work was done for ARPA's Office of Information Processing Techniques and is an integral part of an overall program to apply current computer technology to defense-related requirements. This report, however, should appeal only to a limited audience of specialists interested in higher-order computer programming techniques.

SUMMARY

This report discusses the application of recent developments in web languages and machine learning of heuristics to problems in programming man-machine interfaces. It suggests that these problems arise from the requirements for interface flexibility, the complexity of data interrelationships, and the use of multiple communication channels.

The basic theses of the report are that:

- o Labelled directed graphs, or webs, are an appropriate data-base organization for man-machine interaction;
- o Web grammars, in the form of pattern-replacement rules, can be used to manipulate that data base;
- o Pattern-replacement rules can be viewed as heuristics suitable for machine learning.

The first point draws upon experience gathered over the last ten years by workers in the field of interactive systems. The relationships of an object or action to other objects (actions) are as important as the existence of the object (action). The relationships are generally represented in various forms of directed graphs, e.g., lists or ring structures.

A natural extension of normal programming languages (based on strings of alphanumeric characters) are languages involving labelled, directed graphs. These languages are defined by grammars that consist of graph patterns to be matched against the data base, and graph replacements to be made if the match is successful. One can then consider a collection of such pattern-replacement rules to be a schema for dynamically controlling the data base.

These rules have two important properties:

- o Their form is homogeneous with that of the data base;
- o They may be individually manipulated.

If each rule is considered as a heuristic describing the behavior of the interface, then that behavior may be modified by proper adjustment of the rules. The adjustment may require the insertion of a new

ACKNOWLEDGEMENTS

We would like to thank T. O. Ellis and J. F. Heafner for their participation in early discussions about an adaptive communicator, and Professor E. Feigenbaum for helpful discussions concerning artificial intelligence.

CONTENTS

FIGURES

I. INTRODUCTION

The man-machine interface for a complex interactive system is traditionally difficult to build and to change once built. However, computer systems that interact with humans must be flexible and adaptive to meet the changing requirements of various users and problems.

We believe man-machine interactive systems are inherently multidimensional. By this we mean that the basic problem is recognizing patterns of events that occur over multiple input/output (I/O) channels in both space and time. Moreover, these events do not occur in isolation, but must be interpreted within a rich and varied context.

The work described in this report was motivated by the desire to build a flexible interface mechanism between a user and an application program. Figure 1 shows such an adaptive communicator interposed between an application program and a set of I/O devices. We assume that the communicator is coupled with the I/O devices by device-dependent programs that transform raw signals from these devices into such low-level logical objects as "STROKE" and "POINT," and that perform the inverse transformation for output devices. We envision such an adaptive communicator as a series of context-analysis systems, each having a separate context and set of operating rules, and communicating with each other by transmitting logical objects from the context of one system to the context of another. Each of these context-analysis systems could deal with logical objects at a particular level of generalization and, as a result of data-base transformations, could create logical objects of greater or lesser generality and transmit them to the appropriate companion system for consideration within the system's context. For example, one system could deal with strokes drawn on a tablet and, in the appropriate context, could transform them into logical entities, representing such components as a resistor or a capacitor. These components, once discovered, would be transmitted to another system, whose context consists of individual components. When patterns of components (e.g., representing filters) are discovered by the operation of that second system, they are then treated as higher-level logical objects and passed to a third system, whose context deals primarily with

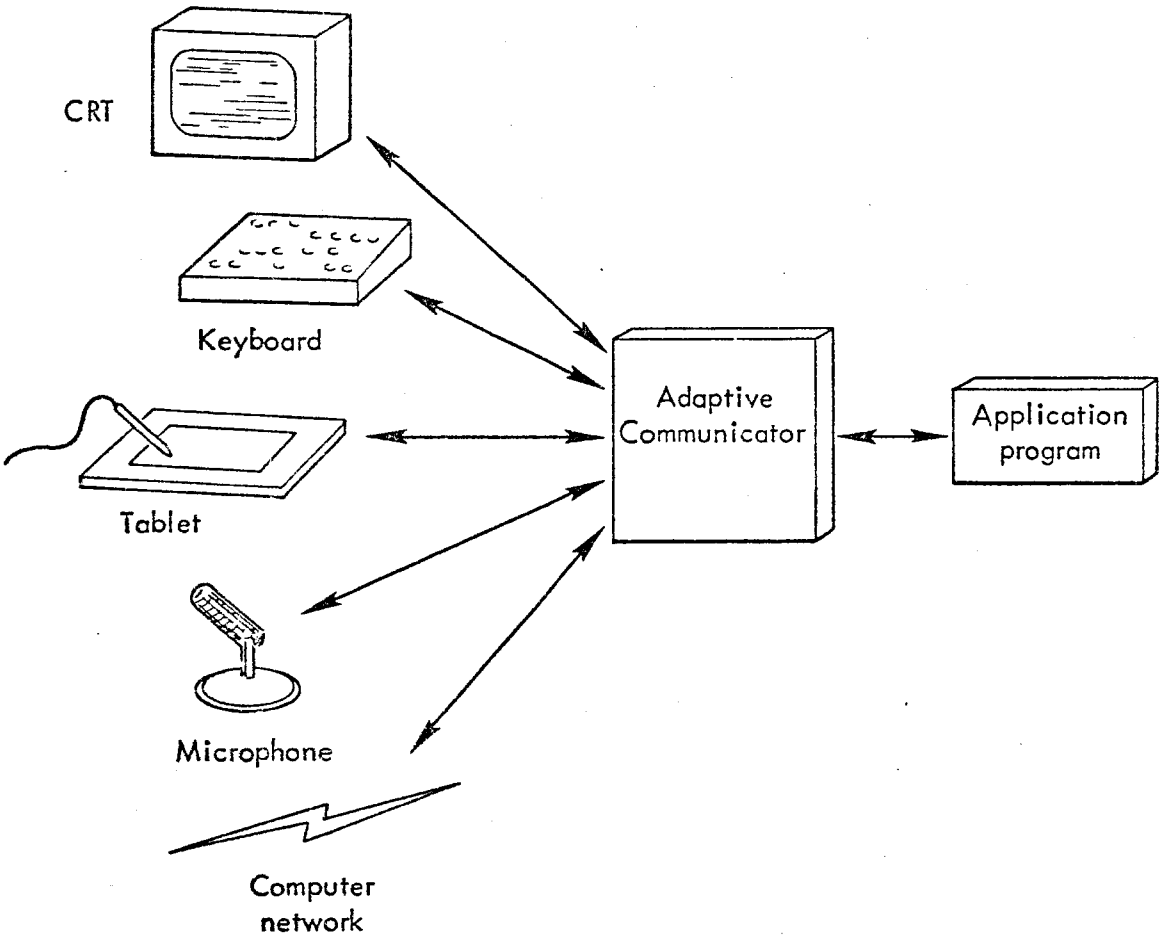


Fig. 1--An Adaptive Communicator as a Communication Interface to an Application Program

assemblies at that level. In turn, this third system would search for patterns constituting higher-level assemblies within its context. One can imagine a circuit-analysis program being sent descriptions of high-level assemblies drawn in this manner, rather than being required to recognize structural elements that are of no direct interest. It is important to realize that the communication between systems within such an adaptive communicator need not be strictly in order of ascending generality; logical objects discovered by "high-level" systems could be transmitted back to "low-level" systems to affect the recognition these low-level systems perform.

It is generally acknowledged that for flexibility and for representing the complex interrelationships among data items, lists and rings are useful data structures in interactive systems. However, the programming that produces changes in the data is most often performed using conventional languages. A more natural way of manipulating lists or rings as data structures would be to search directly for patterns or subgraphs within that data structure and to alter the data based upon the patterns found. The naturalness of this approach, i.e., directly modifying a directed-graph data structure by sets of pattern-replacement rules, was recognized by Christensen in his work on AMBIT-G [1-3]. Recent work on web grammars by Pfaltz and Rosenfeld [4] and by Montanari [5] has provided a formalism for performing pattern matching and replacement within data structures that essentially are directed graphs.

As an example of using a set of pattern-replacement rules to describe the interpretation of user actions (on a tablet or with a light pen), consider the four rules described below. The rules allow the user to create a set of links and nodes (without loops). (We assume a stylus action becomes mapped into a graph structure labeled "POINT," as discussed above.)

- If the user points at a location (x,y) where there is no node, create a node there.
- If the user points at a node, mark it active.
- If the user points at the active node, remove the "active" marker from that node.
- If the user points at a node, and another node is marked "active," create a link from the active node to the one pointed at; also, remove the "active" marker from the node so marked.

In the above cases, POINT is deleted from the graph structure.

Figure 2 shows these four rules displayed in the graph notation used throughout this report. The order in which the rules are shown is the inverse of the order listed above. Rules may be *stated* in any order, but the order in which they are *stored* is important, since they are tested for applicability in that order. (This organization of pattern-replacement rules is a restricted form of "programmed grammar" [6].)

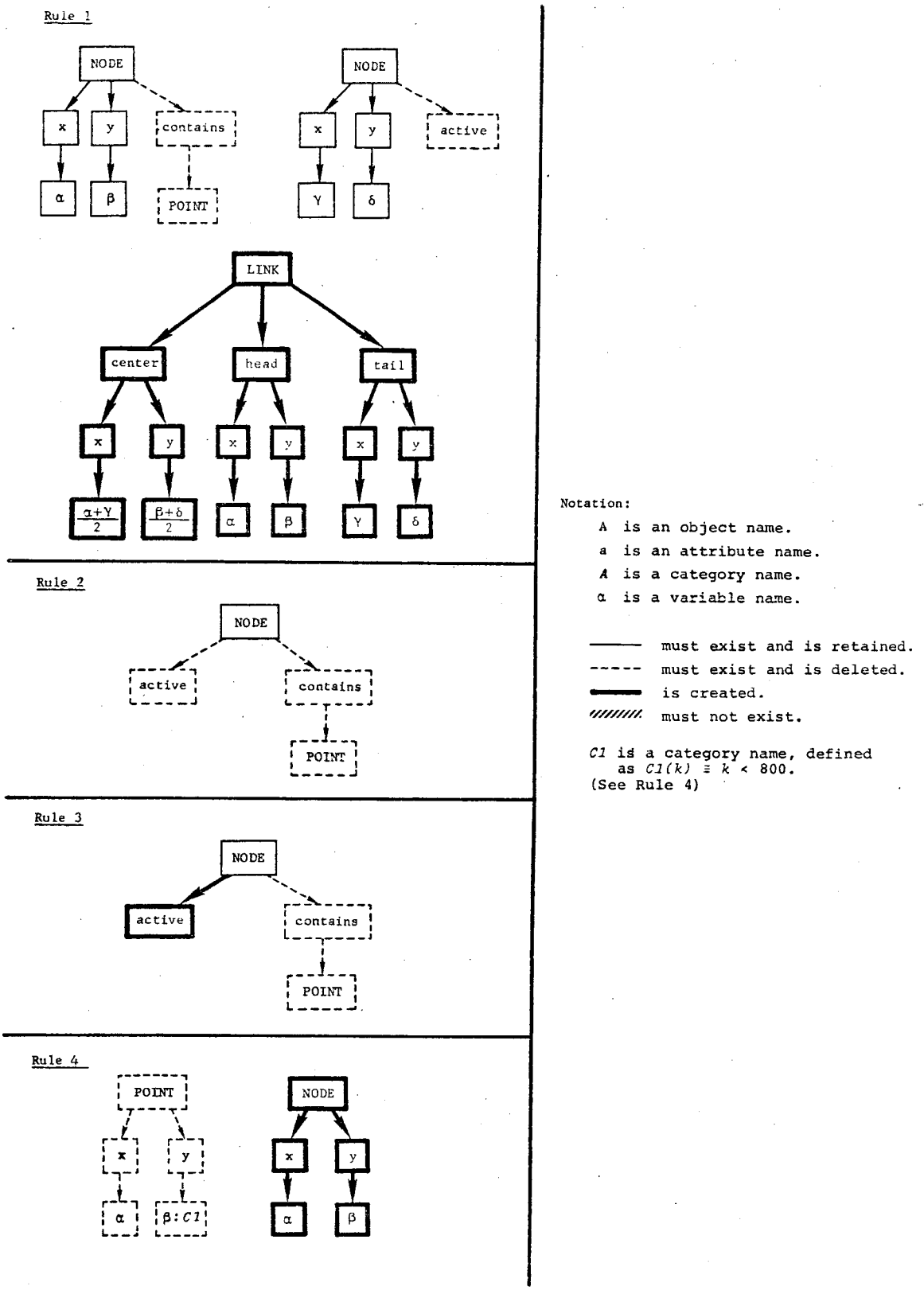


Fig. 2--A Set of Four Pattern-Replacement Rules

To illustrate the meaning of the notation, consider Rule 1, which is interpreted as follows:

> If there is a pattern consisting of two NODEs, one marked "active" and the other containing a POINT, (a) keep the structure drawn in solid lines; (b) delete the structure drawn in dashed lines; and (c) create the structure drawn in heavy lines.

The specifications for the pattern to be matched and the replacements to be made have been combined into one diagram (see key, Fig. 2). However, in the discussion below, we speak of the "pattern part" and "replacement part" of the rule as separate entities.

Several aspects of the notation should be considered. (The notation is detailed in Sec. II.) First, objects in the data base may have more attributes than are shown in the rule (see POINT in Rules 1 and 4); only the necessary ones need be mentioned. Second, the order of the rules is important (e.g., interchanging Rules 1 and 4 alters the interface). Third, information can be gathered for use in the replacement procedure as the matching procedure progresses. (Greek symbols are treated as variables that take on values from the context pattern.)

Attributes can be used to represent relations between NODEs and POINTs. (For example, see Fig. 2, "contains.") These relations and attributes can be discovered and manipulated by the standard operation of pattern-replacement rules.

Our approach has a parallel to Waterman's work [7] on generalized machine learning. Waterman points out that a pattern-replacement rule is analogous to a heuristic that is normally embedded within an artificial-intelligence program. The highly stylized form of a rule lends itself to machine learning. Also, this extreme stylization of the "programming language" permits a degree of independence of neighboring statements and allows a program flexibility not normally found in programming languages. With this approach, individual program statements can be added, modified, or deleted within the set, without affecting the pattern-replacement function of their neighbors.

This preliminary discussion raises questions about (1) the exact nature of the pattern matching and replacement process, and (2) the use of that process in the operation and modification of the man-machine interface.

In Sec. II, we discuss the extensions of web notation that are required in order for web-pattern-replacement rules to represent heuristics that govern the behavior of a context analyzer.

II. PATTERN-REPLACEMENT RULES AS A PROGRAMMING SYSTEM

A web is a directed graph, the nodes of which are labeled with elements of a finite vocabulary. More exactly, let V be a finite set, called the vocabulary; the elements of V are called symbols. A directed web, W, over V, is a triple (N, F, A), where N is a set of vertices, F is a labeling function from N into V, and A is a set of ordered pairs of elements of N, called arcs. If V has only one element, webs are clearly equivalent to graphs.

Below, we refer to the programming-system data base as a "context." This context web should be considered separate and distinct from the pattern-replacement-rule webs that operate upon this context.

In order to use web notation and concepts in a practical programming system, we must make explicit the notion of a pattern match between a pattern, P, and a context web, W_c. In particular, we need to include, in P, specifications of structures that *must not* occur, as well as structures that *must* occur in the context web being matched (see key, Fig. 2). Also, we find it necessary to enlarge the concept of matching between the labels on vertices. To this end, we define a pattern match between P and W_c as a relationship between the vertices in P and W_c. In order to explicate this relationship, we introduce the concepts of a *web map* between two webs over the same vocabulary; a *restriction* of a web to a subweb; and an *extension* of the web map to a superweb. These concepts are defined in the Appendix. In order to formalize the notion of prohibited structure within a pattern, we define a pattern as a *pair* of webs (see the Appendix). Here, however, we informally refer to the pattern part of a rule as a *single* web, which has some prohibiting conditions attached.

Some specialized subwebs are used to represent logical entities, called "objects." An object is a tree structure whose root vertex is labeled with an object name; branches of that root vertex, themselves roots of subtrees, represent attributes associated with the object. The roots of attribute subtrees are labeled with attribute names. Branches, if any, of attribute roots are vertices, which represent

values associated with these attributes. These value vertices are either:

- o Labeled with an atomic value;
- o The root of an attribute subtree; or
- o The root of a tree representing another object.

The vocabulary, V, from which vertex labels are taken, may therefore be thought of as consisting of the union of three sets: V_N, a set of object names; V_A, a set of attribute names; and V_V, a set of attribute values. An example of a context consisting of three objects (two NODEs and a POINT) is shown in Fig. 3.

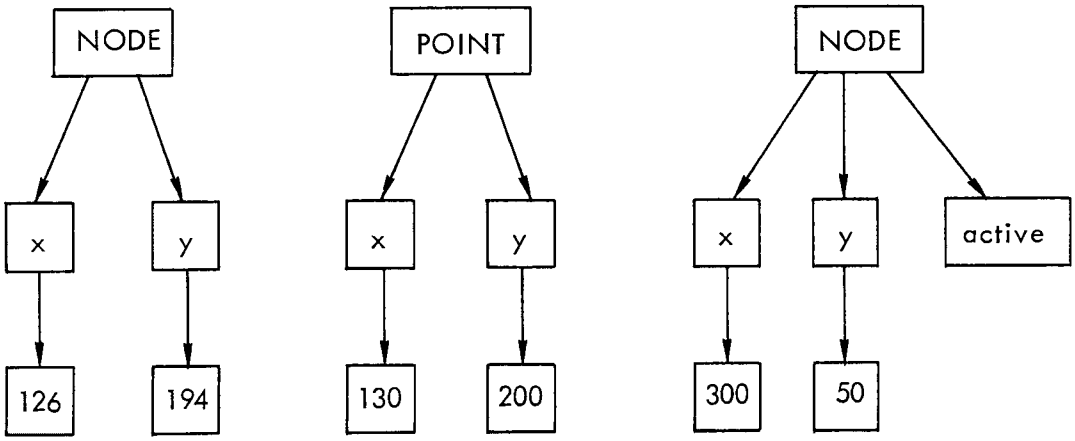


Fig. 3--A Context

In representing the pattern and replacement webs within rules, an additional notational convention, or extension, is used. Note that in Rule 1, Fig. 2, the NODE shown has the same structure as the NODEs in the context in Fig. 3. However, the values associated with the attributes x and y are labeled with Greek letters representing variable names, rather than with atomic values. The pattern shown in Rule 1 matches a corresponding structure within the context if all labels map onto corresponding labels in the context; however, vertices labeled with Greek letters (variables) may match any corresponding vertex in

the context. As a side effect, variables become bound to the values that label the corresponding vertices.

Note that variable names (e.g., in Rule 1, Fig. 2) also specify replacements to be made within the context. Values bound to variables during the pattern-matching process are used wherever these same variable names occur within the replacement specification. Below, we extend the notion of variables to include optional conditions.

A web-replacement rule formally consists of:

1. A pattern web;
2. A replacement web;
3. Specifications for embedding the replacement web in the context web;
4. A general condition that must be satisfied. (This is usually stated as a combination of words and predicates.)

In our extensions of web notation, we attempt to give highly stylistic, explicit notations for conditions in order to use rewriting rules as programming rules. For example:

1. Pattern matches (as formally defined in the Appendix) allow the description of web structure that must not occur during the match (see Fig. 4).
2. Within a pattern web, a vertex may be labeled by a pair, α:*C*; if α is present, the corresponding label in the context, which must be atomic, is bound to the variable α; if the *category* name *C* appears, the context label must satisfy *C*'s test for the pattern to succeed. For example, in Fig. 2, the category definition

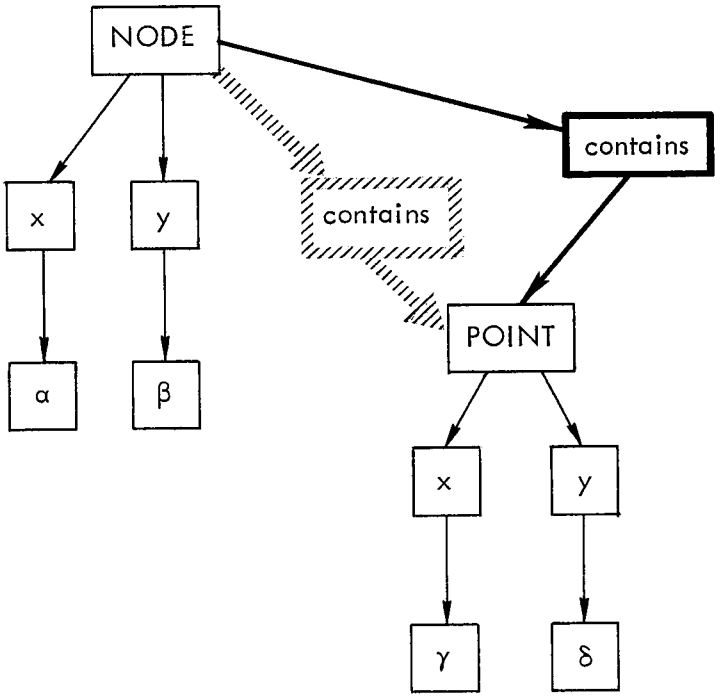
   $$C1(k) \equiv k<800,$$

   as used in Rule 4 ("β:*C1*"), means that both the variable β and the dummy variable *k* are bound to the corresponding value in the context; then the category test "*k<800*" is performed. It is possible that neither α nor *C* may appear, in which case the context label is not assigned to any variable, and that portion of the match is assumed to succeed (see Fig. 2). The above extends the vocabulary V to include V_α, the set of

variable-category pairs. Of course, if the label is in $V-V_\alpha$, then the match must be exact. Note that this paragraph defines the relation to be satisfied by the two labeling functions in the definition of a web map (see the Appendix).

3. Moreover, each rule has a predicate to be satisfied by values drawn from the context (the default value being true). (See Fig. 4.)

Rule 0



Pred: $(\alpha - \gamma)^2 + (\beta - \delta)^2 < 100$

Fig. 4--A Rule with Prohibited Structure and a Predicate

The use of category names is our parallel to what Waterman has called a backward form, or "BF," production rule. We assume that our categories are defined as they are in Waterman's BF rules.

The pattern part of the rule shown in Fig. 4 might be verbally stated as "there must exist two objects, a NODE and a POINT, such that the variables α, β, γ, δ bound by these objects satisfy the predicate and such that the NODE must not have an attribute labeled 'contains,' which in turn points to or has as its value the POINT object." The replacement part of the rule (Fig. 4) states that if all these conditions are met, a vertex labeled "contains" should be created and interconnected, as indicated in the figure.

As mentioned above, we have combined the pattern and replacement webs into a single diagram that specifies both the pattern conditions to be met and the replacement to be made if that pattern is successful. Combining both pattern and replacement into a single diagram makes explicit the embedding instructions associated with that rule. In our notation, the following embedding rule is assumed throughout:

- All shown vertices that must exist and that are retained have embeddings unchanged by application of the rule;
- All shown vertices that must exist, but are deleted, also have all emanating and terminating arcs deleted;
- All created vertices have no arcs connected to them within the context, except those explicitly shown in the creation rule.

One further notational extension is used in our pattern-replacement rule: if a rule specifies the creation of a vertex, then the vertex in the replacement part of the rule may be labeled with an expression containing variable names and atomic values (Rule 1, Fig. 2). The value placed as a label on the vertex created by this rule is the value of that expression at the time the replacement is made. This use of expressions as labels in the replacement part of a web rule corresponds to Waterman's use of expressions in the replacement part of his "action rule."

We have emphasized that another distinction between our use of the web notation and web grammars *per se* is the ordering we impose

on a set of rules. This ordering implies a flow of control through the set, depending on the success or failure of the various pattern matches. Section III examines our method of control flow, and the analogy between pattern-replacement rules (discussed above) and heuristics governing the operation of a program.

III. PATTERN-REPLACEMENT RULES AS HEURISTICS

Waterman described a system for encoding heuristics as an ordered set of pattern-replacement or production rules. He is concerned, as we are, with learning heuristics by their dynamic manipulation. He places the following requirements on the *representation* of heuristics:

1. Permits separation of the heuristics from the main body of the program;
2. Provides identification of individual heuristics and an indication of how they are interrelated;
3. Allows compatibility with generalization schemes.

When Waterman applied his theory to a system for learning the rules of poker, relatively short fixed-length parameter vectors were sufficient to encode the program status at any point in the game. Heuristics were represented by rules consisting of a pattern part and a replacement part, each symbolically representing these fixed vectors of parameter values. As mentioned in Sec. I, the complex needs of an interactive programming system are usually best represented by a graph structure. Therefore, we use directed graphs or webs as the pattern and replacement parts of a production rule that represents a heuristic. In Sec. II, we extended web notation to allow for Waterman's variable assignment and expression evaluation. Waterman, in his discussion of a poker-playing program, distinguishes among the three types of production rules: forward form, backward form, and action rule. His forward-form rule is a method of creating a synonym for an expression; our notation contains no parallel to this rule. His backward-form rule gives a symbolic representation to a particular atomic value, which is then tested symbolically by the rules. We incorporated a roughly equivalent notational device in our use of category names as valid labels within a pattern. His action rules correspond to what we call pattern-replacement rules.

In Waterman's description of a poker-learning program, the heuristic action rules are used as follows. Each successive pattern is tested against the context, from top to bottom, until the first successful

pattern match is found. At this point, the corresponding replacement specified by that rule is made; this constitutes a "play" by the program, and action stops until the other player "makes a play." Then, another attempt is made to apply the rules to the revised context, starting from the top. We vary from this approach by testing the pattern part of each successive rule, in order, until the first successful pattern match is made. We then make the corresponding replacement. Another attempt is made to apply the set of rules by testing each pattern (in order, from the beginning of the set of rules) until another successful pattern match is found. This process terminates only when no rule in the set is applicable to the current context. At this time, the program pauses until an external source changes the context.

An example of this flow of control through an ordered set of rules may be seen by considering a set of rules consisting of the rule in Fig. 4 followed by the rules in Fig. 2, applied to the context in Fig. 3. On the first "pass" through the rules, Rule 0 applies to the context. The corresponding context transformation made by an application of Rule 0 is shown in Fig. 5. On the second pass, which attempts to apply the set of rules to the revised context in Fig. 5, Rule 1 applies; its application produces the context shown in Fig. 6. On the third pass through the set of rules, no rule applies to the context shown in Fig. 6, and the system becomes quiescent. The net effect of repeated applications of the rules is the creation of a LINK from a previously "active" NODE to a NODE just pointed at.

A major contribution of Waterman's research is a discussion of program adaptivity, obtainable by modifications to existing sets of heuristics. (Also see Evans for a discussion of adaptivity in rule-directed programs [8,9].) Waterman has isolated three items of training information that should be extracted from a user whenever a program driven by these heuristics makes an erroneous decision. These items are used to construct a new heuristic, called a training rule, which is blended into the existing set to correct that erroneous decision. Waterman defines the following items of training information:

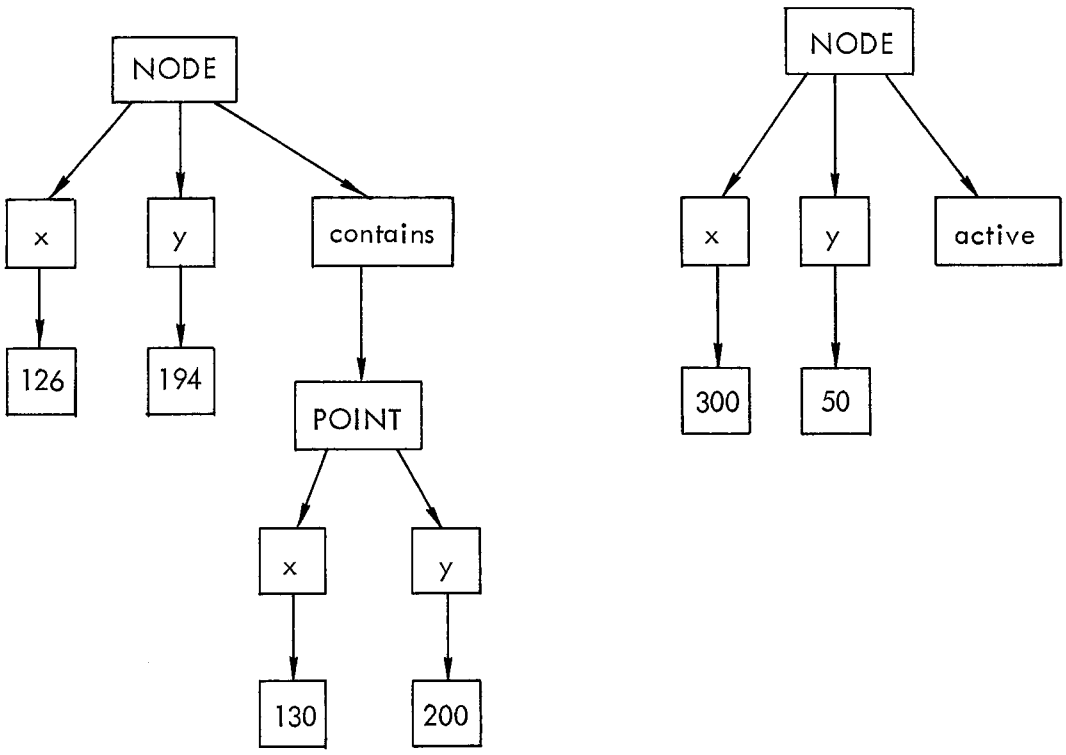


Fig. 5--The Context of Fig. 3 After
the Application of Rule 0

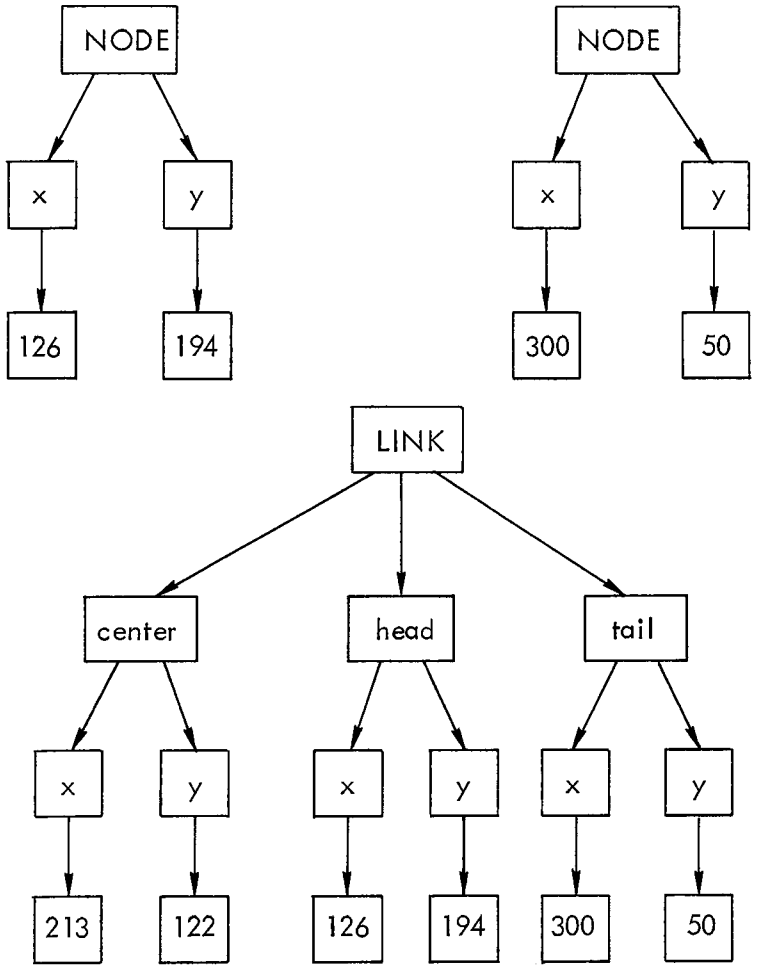


Fig. 6--The Context of Fig. 5 After
the Application of Rule 1

1. Acceptability information, i.e., a good or acceptable decision for the situation. The acceptability information forms the righthand side (or replacement part) of the new training rule.
2. Relevancy information, i.e., the situation elements that are relevant to making this decision.
3. Justification information, i.e., the reason for the decision, expressed as an evaluation of these relevant situation elements. The relevancy and justification information form the lefthand side (or pattern part) of the training rule.

Waterman discusses the process of "blending" a new training rule, in the form of a heuristic, into an existing set of rules. His algorithm for performing this blending is stated in terms of the training rule and a set of action rules based on a symbolic subvector that represents the context. Below, we paraphrase his algorithm in terms of our representation in an expanded web notation. In addition, we assume that whenever a heuristic applies during training, the system tentatively makes the corresponding replacement, and obtains feedback from the trainer as to the correctness of that decision step. This procedure simplifies the problem of assigning blame within the set of heuristics when an incorrect decision is made. The following steps define our algorithm:

1. Determine the first heuristic or production rule that applies to the current context, if in fact any rule applies. Perform the indicated replacement, if any. Determine from the system trainer whether or not that action was appropriate; if it was, repeat this step until no rule applies. If the decision was not acceptable, go to step 2.
2. Obtain the training information from the trainer and use it to construct the training rule. Use the justification information to change any category definitions implied by that information. If in fact a category definition is changed, go to step 3; otherwise, go to step 4.

3. Determine the first heuristic that applies to the current situation (using the revised category definitions) and perform the indicated replacement. If this decision is advocated by the acceptability information, go to step 1; otherwise, go to step 4.
4. Locate the error-causing rule, i.e., the pattern-replacement rule (or heuristic) responsible for the unacceptable decision made in step 3 or step 1.
5. Search the heuristic rules above the error-causing rule for a target rule, i.e., a rule *suitable for modification to apply* to the current context. If such a rule is found, modify it so that it applies to the current context, and go to step 3; otherwise, go to step 6.
6. Search the heuristic rules below the error-causing rule for a target rule. If (a) such a rule is found, (b) the error-causing rule is *suitable for modification so that it does not apply* to the current context, and (c) the rules between the error-causing rule and the target rule either do not apply to the current context or are *suitable for modification so that they do not apply*, then modify the target rule so that it applies to the current context, modify the error-causing rule so that it does not apply, and modify the rules between these two so that they do not apply to the current context; go to step 3. Otherwise, go to step 7.
7. Place the training rule immediately above the error-causing rule in the list of heuristics, and go to step 1.

If, at some point in the operation of the above algorithm, no heuristic rule applies to the current context, then for the purposes of the algorithm, all existing heuristics can be thought of as "above" the error-causing rule. In this case, if a rule is to be inserted "immediately above" the error-causing rule, it should be placed at the bottom of the existing list of heuristics.

Given a training rule, a particular heuristic rule is *suitable for modification to apply* to the current context if (a) the righthand side (or replacement part) of the training rule and the heuristic rule

are the same (within a mapping between variable names labeling corresponding vertices), and (b) the lefthand side (or pattern part) of the training rule and the heuristic rule both have the same structure--i.e., if they differ only in the labels on vertices representing atomic-attribute values. Variable names must correspond under the mapping in (a) above. Modifying the heuristic rule entails changing the labels on these attribute-value vertices either by creating new category names encompassing both the information in the training rule and the information formerly in the heuristic rule, or by changing the definitions of categories used for attribute values so that the training information is embodied in these definitions. An additional possibility for modifying a heuristic rule to embody training information is to simplify (generalize) the structure of the pattern part of the heuristic rule; however, we have not explored the ramifications of that possibility.

Given a training rule and a heuristic rule, *to modify* that heuristic rule *so that it does not apply* to the current context means that categories can be modified to exclude the values in the current context.

An interesting aspect of the above discussion is that the pattern-matching procedure used to compare a pattern web within a context can also compare, for structural similarity, a training-rule pattern web with a heuristic-rule pattern web. To gain this capability, a standard pattern match must be modified by redefining the relation to be satisfied by the labeling functions in the web map used during that pattern match. Additionally, we must take into account both the required and prohibited aspects of each pattern. Informally, this means that the prohibited structures must match and the required structures must match.

IV. GENERAL CONSIDERATIONS

We envision a pattern-replacement programming system as the basis for a flexible interface between a user and his application program. In such an environment, it is useful to think of the user as being in one of several states. Thus, it is convenient to organize the heuristic rules within a programming system into collections of rules called states. Section III describes the flow of control between rules within each state. This state organization allows the gross behavior of such a system to be that of a finite-state machine, with certain allowable transitions between states. If these states are named, the replacement part of a heuristic rule within a state may specify, as a side effect, a transfer of control to another state within the system. The use of state organization for interactive programs is discussed in depth by Newman [10-11].

Although we have emphasized the adaptive aspects of a pattern-replacement programming system, we have not ignored the fact that complex systems require considerable *a priori* knowledge of their subject area. By allowing the user to treat the sets of pattern-replacement rules as a succinct, high-level programming language, a pattern-replacement programming system can introduce such *a priori* knowledge in a form consistent with the adaptive mechanisms.

We have implemented a prototype pattern-replacement programming system, embodying the concepts presented here, in LISP 1.5. We are using this system to explore the techniques and concepts discussed. A difficulty with our prototype implementation of these concepts is that it does not take into account any of the special properties of context webs. In particular, a context consists primarily of logical objects represented by tree structures. A graph-matching algorithm that takes advantage of this knowledge would be more efficient than the generalized graph match currently used.

V. FUTURE WORK

Future work on the ideas discussed in this report will center on the following areas:

1. Making the pattern-match algorithm efficient so that large problems may be investigated;
2. Using the scheme in a substantial variety of applications to test the learning mechanisms and protocols required in these applications;
3. Imbedding a pattern-replacement programming system within the larger adaptive communicator framework in order to test it in situations involving multiple parallel channels and sophisticated user interactions.

Appendix

WEB DEFINITIONS

WEB

Let V be a finite set, called the vocabulary; the elements of V are called symbols. A directed *web* W over V is a triple (N, F, A), where N is a set of *vertices*, F is a *labeling function* from N into V, and A is a set of ordered pairs of elements of N, called *arcs*.

SUBWEB

Given a web W = (N, F, A) over a vocabulary V, the web S = (N_s, F_s, A_s) over the same V is called a *subweb* of W if N_s is a subset of N, if $F_s(n) = F(n)$ for all n in N_s, and if A_s consists of pairs in A whose terms are both in N_s. (The definition in Montanari requires a subweb to be a section graph. This is not required by the above definition.)

WEB MAP

Let W = (N, F, A) and W' = (N', F', A') be webs over the same vocabulary. The *web map* T:W→W' is an ordered pair of maps (M, M*), such that M:N→N' is 1:1, onto, and for all n in N, $F(n) \overset{R}{=} F'(M(n))$. Also, an induced map M*:A→A' defined so that for all (a, b) in A, M*((a,b)) = (M(a),M(b)) is 1:1 and onto A'. (The notation $\overset{R}{=}$ simply suggests some well-defined relation.)

WEB MAP RESTRICTION

Let T:W→W' be a web map between webs W and W' over the same V, where T = (M, M*). Let S = (N_s, F_s, A_s) be a subweb of W, and consider the restriction $M|N_s$ of M to N_s and the restriction $M^*|A_s$ of M* to A_s. The *restriction* T|S of T to S is defined to be the web map $(M|N_s, M^*|A_s)$.

WEB MAP EXTENSION

Let S be a subweb of the web W and T* be a web map whose domain is S. T is an extension of T* to W if $T|S = T^*$.

PATTERN MATCH

Let S be a subweb of the web W; the ordered pair P = (W,S) is called a *pattern*. Let W' be a web also over the vocabulary V of W. The pattern P *matches* the web W' if either of the following is true:

1. W = S and there is a web map T such that $T:S \rightarrow W'$.
2. $W \neq S$ and there is a web map T such that $T:S \rightarrow W'$ but no extension of T from S to W exists. (S is the required part and W-S is the prohibited part of the pattern.)

REFERENCES

1. Christensen, Carlos, "An Example of the Manipulation of Directed Graphs in the AMBIT-G Programming Language," *Interactive Systems for Experimental Applied Mathematics*, M. Klerer and J. Reinfelds, eds. Academic Press, New York, 1968, pp. 423-435.

2. Rovner, Paul D. and Henderson, D. A., "On the Implementation of AMBIT-G: A Graphical Programming Language," *Proc. of the AFIPS/ACM International Conference on Artificial Intelligence*, Washington, D.C., May 1969, pp. 9-20.

3. Christensen, Carlos, Wolfberg, Michael S., and Fischer, Michael J., *A Report on AMBIT-G* (in 4 volumes), Report CA-7102-2611, Applied Data Research, Inc., Lakeside Office Park, Wakefield, Mass., February 1971.

4. Pfaltz, John L and Rosenfeld, Azriel, "Web Grammars," *Proc. of the AFIPS/ACM International Conference on Artificial Intelligence*, Washington, D.C., May 1969.

5. Montanari, Ugo G., "Separable Graphs, Planar Graphs and Web Grammars," *Information and Control*, Vol. 16, 1970, pp. 243-267.

6. Rosenkrantz, Daniel J., "Programmed Grammars and Classes of Formal Languages," *JACM*, Vol. 16, No. 1, January 1969, pp. 107-131.

7. Waterman, D. A., "Generalization Learning Techniques for Automating the Learning of Heuristics," *Artificial Intelligence*, Vol. 1, 1970, pp. 121-170.

8. Evans, Thomas G., "A Grammar-Controlled Pattern Analyzer," *Proc. IFIP Congress 68*, North-Holland Publishing Co., Amsterdam, 1969, pp. 1592-1598.

9. Evans, Thomas G., "Grammatical Inference Techniques in Pattern Analysis," *Software Engineering*, Vol. 2 (Tou, Julius T., ed.), Academic Press, New York, 1971, pp. 183-202.

10. Newman, W. M., "A System for Interactive Graphical Programming," *Proc. AFIPS Spring Joint Computer Conference*, Vol. 32, 1968, pp. 47-54.

11. Newman, W. M., "A High-level Programming System for a Remote Time-shared Graphics Terminal," *Pertinent Concepts in Computer Graphics*, M. Faiman and J. Nievergelt, eds., University of Illinois Press, Urbana, 1969, pp. 200-223.