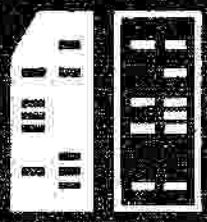
UIUCDCS-R-74-663

A COMPARATIVE DISCUSSION OF VARIABLE-VALUED LOGIC AND GRAMMATICAL INFERENCE

by

A. B. Baskin

July 1974



DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

UIUCDCS-R-74-663

A COMPARATIVE DISCUSSION OF VARIABLE-VALUED LOGIC AND GRAMMATICAL INFERENCE

by

A. B. Baskin

July, 1974

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

ACKNOWLEDGMENT

The author would like to thank Dr. R. S. Michalski for suggesting this line of inquiry and for his discussions concerning the material contained in this work. The author would also like to acknowledge Connie Slovak for her help in preparing the manuscript.

ABSTRACT

This paper reviews what is meant by grammatical inference and it discusses some of the currently used methods of grammatical inference. The application of a variable-valued logic system to this inference problem is explored and, where possible, direct comparisons between methods are discussed. Examples of a correspondence between a variable-valued logic inference process and a grammatical inference process are presented. Operations which increase the class of languages to which variable-valued logic can be effectively applied and operations which allow simplified variable-valued logic descriptions of classes of grammars are discussed. Type 3' and type 3" grammars are defined (both subsets of type 3 grammars) and the equivalence of variable-valued logic formulas and type 3' and type 3" grammars is illustrated.

TABLE OF CONTENTS

1.0 Grammatical Inference

In order to describe the process of grammatical inference, it is necessary to present a few definitions. Formal statements of the process of inductive inference [22] and specifically grammatical inference exist in the literature [5,9]. Since no formal theoretical comparisons will be included in this work, the appropriate constructs will be introduced only as needed to supplement the intuitive discussions to follow.

Feldman [9] has correctly observed that any discussion of grammatical inference can be generalized to include a much larger set of problems including the inference of functions, theories, and patterns. Because of the fact that the process of grammatical inference holds promise as a vehicle to study other forms of inference, more detail will be included in the discussion of the inference of a formal language grammar than might be otherwise justified.

A grammar, G, is a 4-tuple $G = < N, T, P, S >$, where N is a finite set of non-terminals, T is a finite set of terminals (the input alphabet), P is a finite set of productions or rules, and S is a non-terminal called the sentence symbol or the start symbol. It is necessary that N and T be distinct sets (i.e. their intersection must be empty). Lower case letters will be used to represent terminals, and capital letters will be used to represent non-terminals. A string is a concatenation of symbols from the sets N and T. A language is a set of strings where each string contains symbols from the input alphabet T only. The set of all strings over an input alphabet T will be denoted as T^*.

Thus, T* represents all strings which can be formed by the concatenation of elements from T selected with replacement.

The difference between terminals and non-terminals can best be demonstrated in terms of a simple example. The terminal is the atomic unit of a language. It is the unit below which it is not profitable to subdivide the language. Consider ordinary English; the logical terminals are letters. For information to be conveyed, it is not necessary to examine the structure of each individual letter to determine its component strokes. It is clear that a language could be formed by considering as terminals a set of strokes from which all letters could be formed. However, letters seem the most appropriate terminals for the English language. As examples of non-terminals, consider the concept of a word. It would be possible to describe English without the concept of a word, but it is simpler to group letters together into logical constructs which can be treated as a unit. Non-terminals serve just such a purpose. In addition to words, it is sometimes convenient to group words and letters (one letter words for instance) to form sentences. Thus the non-terminal "sentence" is formed from both terminals and non-terminals. The ideas of a paragraph and a chapter are also the use of non-terminals for the description of English. One speaks of the rules of grammar for English as the rules which specify the manner in which terminals and non-terminals can be put together to form statements in the English language.

As indicated by the example above, the use of a set of rules or productions is an essential part of a grammar. The set of rules specifies the valid substitutions allowed in forming strings of the language. The start symbol is required to be the first symbol in the

derivation of any string in the language. A derivation consists of the successive application of rules from the set of rules which change one string into another. The idea that a grammar can capture the pattern in a language is important to the discussions which will follow. Consider the following two grammars for the trivial language {abc, ab, cab}, where the input alphabet T = {a,b,c}:

S → abc		S → Dc		
S → ab	or	S → D	and	N = {D,S}.
S → cab		S → cD		
		D → ab		

The trivial set of productions on the left (and thus the grammar of which they are a part) defines the set of strings which constitute the language, but they do not demonstrate the pattern or structure that is present. The productions on the right specifically state that the string ab occurs in three places. In addition, in situations in which enumeration is not desirable, or even possible, the use of non-terminals can simplify the set of productions used to describe the language. The idea of simplicity has been studied along with the companion idea of cost [5,6,9]. For many applications, the measure of cost and complexity is the same. Several different measures of simplicity will be mentioned in section 2.

A string from the set T* which is an element of a given language, L, is said to be a positive instance of the language L. In a similar manner, a string from T* which is not in the language L is a negative instance of the language L. An information sequence or just a sequence is a group of strings presented in a specified order or sequence. A positive information sequence is a sequence in which all strings in the

sequence are also in the language L, and a negative information sequence is a sequence in which all strings are in the set T* but not in L.

With the above definitions, it is possible to specify what is meant by grammatical inference in more detail. Grammatical inference is the process of inferring the grammar which describes a positive information sequence while not including any strings from any given negative information sequence. The omission of a given positive instance of the language from the inferred language is considered an error. The inclusion of a known negative instance in the inferred language is also considered an error. Generally, errors are not allowed in the inference process. The inclusion of a string from T* about which nothing is known is called the introduction of a discrepancy in the inferred language. The inferred language is then greater than the initially given set of positive instances, but it does not contain any error. Cook [5] has noted that by increasing the discrepancy it is often possible to reduce the complexity of a grammar.

It is possible to classify grammars based on the strings used in the productions. The classification to be discussed below was introduced by Chomsky [3] and will be extended to include two interesting classes of grammars (and thus languages). For the purpose of the discussion below, a general production will be of the form:

$$\varphi A \psi \rightarrow \varphi \alpha \psi$$

where A is a non-terminal and α is a string of terminals and non-terminals. The interpretation of the rule above is that the string A may be replaced by the string α. The strings φ and ψ are the left and right context of A respectively.

Type 0 grammars are grammars with productions of the form described above. There are no other restrictions on the form of productions of type 0 grammars.

Type 1 grammars are grammars with productions of the form:

$\varphi A \psi \rightarrow \varphi \alpha \psi$ where α is not the null string.

Type 1 grammars are the so-called context sensitive grammars because the replacement must be made within the left and right context.

Type 2 grammars have productions of the form:

$A \rightarrow \alpha$ where α is not the null string.

Type 2 grammars are called context free because there is no specification of either left or right context in the productions.

Type 3 grammars, the so-called finite state grammars, are of the form:

$A \rightarrow \alpha$ where $\alpha = a$ or $= aB$

(a is an arbitrary terminal and B is an arbitrary non-terminal).

There are several important results from the formal theory of languages which will be important to the discussions which follow [13,14]. The most important idea is the idea of a machine which accepts a given language as defined by a grammar. This equivalence between machines (called acceptors) and grammars will be exploited when a comparison is made between grammatical inference and variable-valued logic.

Type 3 languages can be represented by a finite state machine (and thus the same finite-state grammar or language). Such a machine is totally specified by giving the states of the machine and a tabulation

of the state transition and output behaviors as a function of the input and the current state of the machine. The machine is seen as being presented the letters in a candidate string, one at a time, and the output of the machine after the input of the entire string indicates whether the string is in the language in question. In general, a different machine is used for each different grammar.

For the purpose of this work, type 3' grammars will be defined as type 3 grammars without recursion. This means that productions of the form:

$$A \rightarrow aA$$

(or sequences of productions which accomplish the same thing) are not allowed. This restriction can be expressed in terms of the machine acceptor for a type 3' grammar also. The state transition graph of the finite-state acceptor for a type 3' grammar is a directed graph with no loops.

As an additional extension to the formalism of Chomsky, type 3" grammars will be defined as type 3' grammars whose corresponding machines can be viewed as a tree with the root at the sentence symbol and no links that connect different levels of the tree. The corresponding restriction on the form of a grammar will be defined by example. The grammar below is type 3' but not type 3".

$T = \{a, c\}$

$N = \{B, D, S\}$

$S \rightarrow aB$
$S \rightarrow cD$
$D \rightarrow aB$
$D \rightarrow a$
$B \rightarrow a$
$B \rightarrow c$

The use of B on the right-hand side for two different non-terminals (S, D) distinguishes this grammar from type 3".

It is important to notice that, by construction, the types of grammars are related in the following way:

type 3" $\subset$ type 3' $\subset$ type 3 $\subset$ type 2 $\subset$ type 1 $\subset$ type 0.

This observation will be important later. An equally important observation is the fact that the machines necessary to accept the languages above increase in power from left to right. This means that a machine (or other formalism such as a grammar) which is sufficient for type 3 is not sufficient, in general to accept a type 2 language. Since subsets of type 3 are the simplest grammars and thus the simplest languages to represent, it is not surprising that early work in grammatical inference concentrated on type 3 languages and subsets thereof. The next section will present the general aspects of several different methods of grammatical inference. The range of applicability of each method will be indicated.

2.0 Methods of Grammatical Inference

Several surveys of results in grammatical inference exist in the literature [1, 5, 11]. Gold [12] presented a number of interesting results about grammatical inference which were extended by Feldman [9] and reviewed by Cook [5]. The details of the results are not of interest here because the problem was treated from an enumerative approach. Such an approach to the inference problem implies the enumeration of all possible grammars over a given input alphabet and the selection of the first grammar found to include all of the given positive instances and none of the negative instances. A more sophisticated problem, also treated by Feldman, is the problem of selecting the simplest grammar that meets the above requirements. An interesting result of these discussions is the fact that the selection of the least complex grammar from the set of all candidate grammars is unsolvable, in general, unless negative instances are given. This result combined with the fact that the number of candidates to consider for the enumeration grows combinatorially with the size of the language implies that the enumerative approach is impractical for any but simple examples.

As an alternative to the enumerative approach, several constructive approaches to the problem have been proposed. Most work that has been done involves finite-state grammars with some work being done for more powerful grammars [5]. Before discussing specific algorithms which have been suggested for grammatical inference, an intuitive example will be presented. Consider the set of all well-formed strings of parentheses. As a sample (a positive information sequence)

take all such strings of length six or less. The sample set is thus:

(), (()), ((())), (()()), (()), ()(()), ()()(), (())().

For the sets of productions below, the short hand notation S → a/b will be used to replace the more bulky S → a, S → b. The generalization is obvious.

The most trivial solution to the inference problem is the simple set of productions: S → ()/(())/((()))/(()())/()()/()(())/()()()/(())(). This set of productions portrays the sample with no discrepancy (the inclusion of strings not in the sample and not known as negative instances). It does not satisfy the desire to capture the structure of the language under study, and the method does not generalize to large and potentially infinite languages. Since we know that non-terminals are used to portray the structure of a grammar or to simplify a grammar, it seems reasonable to try to introduce some non-terminals into the trivial grammar in order to simplify it. Three different identifications of non-terminals will be demonstrated below:

1. Y → ()
   S → Y/(Y)/((Y))/(YY)/YY/Y(Y)/YYY/(Y)Y

2. Y → ((
   S → ()/Y))/Y()))/Y)())/()()/()Y))/()()()/Y))()

3. Y →))
   S → ()/((Y/(((Y)/(()(Y/()()/()((Y/()()()/((Y().

Other choices for Y are possible but they lead to less compact grammars than the above. On a purely intuitive basis, it appears that grammar 1 is the most compact representation of the sample and should be chosen as the best from this group. Can the reduction in complexity be improved? It would seem logical to take the new grammar and try to improve it by selecting a substitution for some string which might include Y and

assigning it to a new non-terminal. This iterative process forms the heart of many constructive approaches to the grammatical inference problem.

In general, the constructive approach to grammatical inference is the same as a heuristic search procedure. Such a procedure tries to make the best choice possible at any stage of the search. Since the objective of the constructive approach is to avoid an exhaustive search, the stopping condition becomes all important. Cook [6] disucsses several typical stopping criteria in the context of the inference of stochastic grammars, but many of his remarks are generally applicable. The first method for assuring the termination of a search process in an acceptable time is to restrict the set of allowed candidate solutions in such a way that an exhaustive search of this set can be performed in the worst case. This procedure is very similar to the total enumerative method and suffers from most of its problems.

Any constructive method starts with a candidate solution or a part of a candidate solution and tries to add to or subtract from the current best guess in order to improve the candidate solution. Implicit in such a process is the potential for backing up and trying another path which was previously rejected. At the point where a back up is suggested a decision must be made as to whether the amount of improvement to be gained by the retry is worth the effort. No definitive solution to this problem has been presented for the existing algorithms for grammatical inference. (More will be said about this problem when variable-valued logic is discussed.) The proposed stopping criteria in use at present are to limit the time or steps in the entire process, to avoid back up entirely, or to try to back up only when the discrepancy

of the new path proves to be less than the final discrepancy of the current best grammar. The last of these is the criterion used by Cook, and it is the most promising. At present it is not possible to prove it to produce a correct choice in the general case.

The first specific method of grammatical inference to be discussed, unlike those which will follow, does not involve the possibility of a back up. Biermann and Feldman [1] have reported a method of inferring finite-state grammars which creates and compares sublanguages. The original sample is divided into equivalence classes which share a common initial string. The resulting equivalence classes are identified with non-terminals and the grammar can be produced. The mechanism is best understood in its application by Biermann and Feldman [2] to the problem of the inference of finite-state machines from samples of their input and output behavior.

Biermann and Feldman define a relation which they use to partition the sample set of strings into equivalence classes. The relation uses only the first k terminals in a string. This means that the adjustment of the parameter k can cause the machine to pass from the universal acceptor (k=0) to the minimal deterministic finite-state acceptor for the sample (k$\geq$ the longest string's length). When used as an acceptor, the machine will have the desired behavior for the first k letters of a string and the behavior for longer strings is not assured to be as given in the sample. Thus, if an acceptor for only k letters were needed, the algorithm would be faster since it would not have to consider letters in strings in the sample after the k^{th} letter. For small values of k the machines are non-deterministic. Such a non-deterministic machine must be converted to a deterministic acceptor before it can be

converted to a grammar. Because of the equivalence between finite-state acceptors and type 3 grammars, the resulting grammar has the fewest possible number of non-terminals.

Biermann and Feldman have programmed the algorithm and they claim that typical constructions of machines with 10 to 20 states require only a few seconds of processor time. Since the process minimizes the number of states in the acceptor and thus the number of non-terminals, the process is not generalizable to alternate minimization criteria. A generalization of the algorithm has been made to infer productions of the form A → a or A → aBc, but it will not be discussed.

In order to simplify the example below, the algorithm will be used to find the minimal grammar for the sample set shown below:

{caaab, bbaab, caab, bbab, cab, bbb, cb}

where k>5. (Two sublanguages are considered equal if they are equal for the first k letters.) The first two sublanguages are:

S_c = {aaab, aab, ab, b} and S_b = {baab, bab, bb}

which are distinct sets (not equal and neither is a subset of the other). The two sets above each give rise to a non-terminal and the first production is:

A → cB/bD.

The two sets above will now be divided into sublanguages and all sublanguages which are distinct from all previous sets will give rise to new non-terminals. Sets which are subsets of previous sets or equal to previous sets will be identified with the non-terminal created by that set. Thus we have:

$$S_{ca} = \{aab,\ ab,\ b\} \quad \text{and} \quad S_{bb} = \{aab,\ ab,\ b\}$$

$$S_{cb} = \{b\}$$

where $S_{ca} = S_{bb} \subset S_c$ and thus no new non-terminal is needed. The set S_{cb} only contains a single terminal (all strings of length one) and thus no new other non-terminal is needed. The final set of productions is thus:

$$A \rightarrow cB/bD$$
$$B \rightarrow b/aB$$
$$D \rightarrow bB$$

which is the minimal grammar which produces the strings in the sample as required by the choice of k. Notice that the recursive production allows a language which is larger than the sample, but all strings of length 5 or less are properly described. The more general case of k less than the length of the longest string in the sample involves the conversion of a non-deterministic finite-state machine to a deterministic machine and it will not be treated further here. The method above is capable of inferring a type 3 grammar with the minimal number of non-terminals for a given sample set.

Feldman et al. [10] have implemented another algorithm for inferring finite-state grammars from a sample of a language. The process constructs a non-recursive type 3 grammar with residues which represents the sample. A residue is a production with a non-terminal on the left-hand side of the rewriting arrow and a string of terminals on the left. The grammar with residues is simplified to produce a finite-state recursive grammar. The resulting grammar is near minimal with no precise estimate of the distance from the minimum. As an example of the method consider the sample set:

{caaab, bbaab, caab, bbab, cab, bbb, cb}.

Strings are processed sequentially in order of decreasing length. The first string can be generated by the following:

S → cA
A → aB
B → aC
C → ab (a residue).

In order to generate the second string, bbaab, the following productions are needed:

S → bD
D → bE
E → aF
F → ab (a residue).

In order to generate the third string, caab, the production C → b must be added to the list of productions. If this process is continued, the set of non-recursive productions with residues found below will be generated.

S → cA/bD
A → b/aB
B → b/aC
C → b/ab (a residue production)
D → bE
E → b/aF
F → b/ab (a residue production)

A set of productions which generate the sample has been formed. It is now necessary that the residue productions be removed and, if possible, recursion be used in the replacements and simplifications. The residue production F → b/ab can be removed if the production E → b/aF is replaced by E → b/aE and all occurrences of F are replaced by E. A similar merger of the other residue production produces the following grammar:

S → cA/bD
A → b/aB
B → b/aB
D → bE
E → b/aE

where there are now no residue productions. The productions involving non-terminals A, B, and E are of the same form and can be merged together to form the set of productions below:

S → cB/bD
B → b/aB
D → bB

which is the same grammar which was obtained in the first example for the same sample set. As was pointed out in the first example, the inclusion of recursion automatically increases the discrepancy in the generated grammar, but the reduction in complexity appears to be worth it. The example above is presented by Cook [5] and Feldman [10].

Unlike the first example, the example above provided a large number of places to make a choice. The string to expand in productions must be chosen on the basis of other criteria if several strings are of the same length (the process normally processes strings in order of decreasing length). When residues are merged they may be merged in any order, and similar productions may be merged in any order. It is thus possible to apply other measures of cost or complexity at each choice. This ability and the number of choices to be made mean that the process cannot provide a guarantee of minimality, but a near minimal solution is possible using different measures of minimality than number of non-terminals.

Solomonoff [23], Chomsky [4], and Crespi-Reghizzi [7] have developed inference methods depending on an informant. The informant is used to indicate whether a string is in the language. The process might

try to generalize on the sample and in so doing create a string about which it has no information. The informant is used to answer the membership question about the new string. These processes have been applied with success in special cases [7] but it is generally as big a problem to act as an informant as it is to infer the grammar. Thus in order to solve the problem, it must first be solved.

The final method to be considered is the method due to Cook [6]. Cook has developed a cost and discrepancy measure for type 2 languages defined by type 2 stochastic grammars. A stochastic grammar is a grammar in which a probability is associated with any choice allowed in the production. The productions using / are those with choices. A stochastic grammar defines a stochastic language in which a probability can be associated with each string in the language. The probability associated with each string in the language is the sum of the probabilities associated with each possible derivation of the string. A derivation has the probability equal to the product of the probabilities of all productions used in the derivation. The method used by Cook is that which was described in the first example in this section. Cook makes an initial grammar consisting of the trivial grammar and the given (or assigned) probabilities. Specific kinds of simplifications are considered and the cost and discrepancy of each is computed. The alternative with the lowest cost is chosen and the discrepancy measure is used when the costs are nearly equal. Cost being the same, the lesser discrepancy alternative will be chosen. Just as in the last example, the process is a search and thus does not always find the minimum solution. There is no measure of the distance to the minimum.

Cook applied the algorithm above to the same example as the one used twice above. The intermediate steps are too lengthy to state here, but the algorithm converged to the following solution after 13 steps:

$X \rightarrow WY$	(1)
$Y \rightarrow b/aY$	(0.5,0.5)
$W \rightarrow c/bb$	(0.5,0.5)

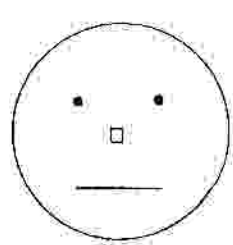
Recursion was not explicitly considered, but it was found to have the smallest cost and thus was chosen. Probabilities associated with the productions are shown in parentheses. When the probability information is available or can be generated, the algorithm is a more general solution to the problem than has been discussed. If the probabilities of sample strings must be arbitrarily assigned, the simplest solution that the algorithm could produce might not be found because of a poor choice of probabilities.

Other authors have approached the problem from the point of view of machines [21] or specialized applications. The algorithms discussed are a representative sample of those which exist.

3.0 Inference of Pattern Grammars

Evans [8] has reported an attempt to use the results of the inference of grammars for formal languages in the inference of descriptions of patterns in objects or images. The inference of a pattern grammar to describe a set of objects is more difficult than the problem of grammatical inference for formal languages.

The inference of a pattern grammar begins with the formulation of a pattern description. The first step in the formation of a description is the quantization of the object. In some instances the word 'digitize' is applicable, but in general the object is described in terms of a set of prespecified numeric quantities which are thought to have some relevance to the set of objects in question. Evans calls these numbers terminals and quite properly calls them lowest-level object types. After the object has been quantized a set of predicates is defined. The predicates are operators which form equivalence classes over the set of terminals in the pattern grammar. After the proper relations are chosen, it is possible to describe a pattern in terms of terminals and the relations which collections of terminals from the pattern obey. The description takes the form of a set of rules or productions which specify the description. An example will be used to define the form of the productions. Consider the figure shown below:



As Evans points out, the natural terminals to choose for such a pattern are circle, dot, line segment, and square. The relations are defined by the operators: above, inside, and left, each of which is binary. The description of the figure above is:

face → features, head: inside (features, head)
head → circle
features → eyes, nose, mouth: above (eyes, nose) ∧ above (eyes, mouth) ∧ above (nose, mouth)
eyes → dot, dot: left (dot, dot)
nose → square
mouth → lineseg.

This description does not precisely specify the image since no comment was made about the angle of inclination of the two dots or the line segment. It does contain the necessary information as specifiable by the terminals and the predicates chosen.

A statement of the algorithm for the inference of a pattern grammar is:

1. Quantize the pattern into a suitable set of parameters.
2. Define relations on the parameters which have structural meaning for the pattern (predicates).
3. Describe each instance of the pattern in terms of the relations which are true for that pattern. Express the description as a set of productions.
4. Form a grammar for the set of instances as the union of the grammars above.
5. Simplify the grammar.

The details of the various steps will be explained in terms of an example, Consider the three line drawings below.

a b c

The relevant terminals are circle, dot, triangle, and square. The first step results in the description of each of the three objects above in terms of the terminals. For step 2 the only relation between terminals that looks appropriate is the relation inside. The results of step 3 are shown below:

a. S → triangle, square: inside (triangle, square)
b. S → circle, square: inside (circle, square)
c. S → dot, square: inside (dot, square)

The union of the rules above creates a trivial grammar which completely specifies the set of images presented. This corresponds to the trivial grammar formed as a first step of the grammatical inference for formal languages. At this point, the simplification problem is the same as it was for the formal language case, and it should be hoped that many of the same tools could be brought to bear.

The example above points out a fundamental difference between the inference of pattern grammars and the inference of formal language grammars. The system of productions above is not capable of simplification with the set of terminals chosen. Evans indicates that at this point it is necessary to weaken some of the rules used to specify the initial set of objects. The significance of this observation is equivalent to the realization in the formal language case that the initially given input alphabet should be replaced with another. No inference procedure for formal languages known to the author has this provision. The example above should be restructured so that the simplification to the form below is possible.

S → any, square: inside (any, square)

The grammar above represents a generalization in order to simplify the

grammar. This concept will be reviewed again in the discussion of variable-valued logic as a vehicle for inference.

4.0 Variable-valued Logic

Detailed definitions of several forms of variable-valued logic may be found in the literature [15,16,17]. The system to be defined below will be a simple subset of the systems in current use and it will be expanded as more sophisticated operations are found to be needed. The operations defined below are sufficient to define a minimal subset of variable-valued logic system VL_1 [17].

A variable-valued logic system (a VL system) is a quintuple:

$$< X, Y, S, R_F, R_I >$$

where

X -- is a non-empty set of input or independent variables, whose domains, denoted by D_i i = 1, 2, 3, ... are any non-empty sets.

Y -- is a set of output or dependent variables, whose domains, denoted by ^jD, j = 1, 2, 3, ..., are any non-empty sets.

S -- is a set of symbols called connecting symbols. Initially the only connecting symbols we will use are : = ( ) $\wedge$ $\vee$ [ ].

R_F -- is a set of formation rules which define well-formed formulas (wff) in a VL system. A string of elements from X, D_i, Y, ^jD and S is a wff if and only if it can be derived from a finite number of applications of the formation rules.

R_I -- is a set of interpretation rules which give an interpretation to VL formulas. They specify the mapping from all wff to elements of the sets ^jD.

In the discussions which follow, the symbol x_i will be used to denote a variable which may take on values selected from the input set

D_i. Only one set of output variables will be needed and elements from it will be referenced by name where required. Allowable wff will consist of simple selectors:

$$[x_i = \text{a sequence of elements from the input set } D_i]$$

or more than one simple selector joined by the operators $\wedge$ or $\vee$.

The interpretation of VL formulas will be as follows:

A simple selector will have the value equal to the maximal element of the output set if the statement in it is true and the value zero otherwise.

Selectors joined by $\wedge$ will take on the value of the smallest selector so joined.

Selectors joined by the symbol $\vee$ will take on the value of the largest selector so joined.

Parentheses may be used to specify the order of evaluation in the normal way.

An example should help to indicate the nature of well-formed formulas and their evaluation. Consider three input sets:

$$X_1 = \{1,2,3\} \qquad X_2 = \{r,e,s,t\} \qquad X_3 = \{\text{cat, boy, dog}\}$$

and the output set: $D = \{0,1\}$. The following are well-formed formulas.

$$[x_1 = 1] \vee [x_1 = 2][x_2 = r] \vee [x_3 = \text{cat}]$$

$$[x_3 = \text{dog}][x_2 = e] \vee [x_2 = t]$$

$$[x_1 = 1][x_2 = s][x_3 = \text{cat}]$$

where $\wedge$ has been omitted where unambiguous and the order of evaluation is left to right with $\wedge$ given a higher precedence than $\vee$.

The three wff above can take on a value only after the values of each of the variables has been specified. The first formula will

have the value 1 if x_1 is assigned the value 1, or x_3 the value cat, or x_1 the value 2 and x_2 the value r. In all other cases the first formula will be assigned the value zero. The second wff will have the value 1 if x_3 has the value dog and x_2 has the value e, or x_2 has the value t; otherwise, the formula will take on the value zero. Formula two points out an interesting aspect of the evaluation of VL formulas. Since the operation $\vee$ selects the larger of the values it joins, the evaluation of formula two could terminate after the first condition was satisfied since it is not possible to find a larger value. Only if the first condition (x_3 = dog and x_2 = e) were not met would the second condition need to be evaluated. The third formula will be assigned the value 1 only if x_1 is 1 and x_2 is s and x_3 is cat. In all other cases the value will be zero.

The formulas above can be thought to select points from an event space. The event space has three dimensions because there were three input sets. The total number of distinct events in the space is the product of the cardinalities of the input sets. For the example above, this would be 36 events. The third formula above specifies a single event since it specifies a value for each of the input variables. The second formula specifies two groups of events sharing one of two properties (either x_3 = dog and x_2 = e or x_2 = t). A similar statement can be made about formula 1. The generalized logical diagram (GLD) has been proposed by Michalski [18] as a geometrical representation of a multi-dimensional space using the thickness of dividing lines to signify the dimensions. The diagram is constructed so that each event is represented by a single square in the diagram. The specification of a formula on such a diagram consists of putting a 1 in the squares of the

diagram represented by the formula. The GLD provides an excellent geometrical model for describing formulas and can be used to infer formulas from sets of events in simple cases.

Michalski [19] has reported an algorithm for inferring formulas based on sets of events to be included in the formulas. The algorithm has been used in a computer program, AQVAL, and has met with success in its applications. The details of the algorithm (based on the algorithm A^q) are beyond the scope of this work, but some general comments about the algorithm are appropriate.

The program AQVAL produces a near minimal formula for the representation of a set of events E^1 while not including any elements from the set of events E^0. The problem can be stated graphically on a GLD by placing a 1 in the squares which correspond to the elements of E^1 and a 0 in the squares of the diagram which correspond to the events in the set E^1. Those squares not marked are so called "don't care" conditions and may be included in the formulas if their inclusion will simplify the result. The inclusion of such don't care events in the formula corresponds to the introduction of discrepancy in grammatical inference.

In addition to near minimal formulas for the event set E^1, the program produces the maximum possible distance between the present formula and the minimal formula. The application of the algorithm is deterministic and does not require any backup. Repeated application of the algorithm to the problem can yield successively better formulas or better estimates of the maximum distance to the minimal solution. A minimal formula is the cheapest to evaluate using a cost function defined by the program user.

The estimate of the maximum distance between the present formula and the minimal formula produced by AQVAL appears to answer the backup question posed earlier for grammatical inference. After the application of AQVAL it is possible to decide whether the next iteration can produce a simplification which is worth the time it might take by examining the distance to the minimum. No such situation is known to the author in the case of currently applied methods of grammatical inference.

5.0 VL Systems and Grammatical Inference

Before attempting to apply the process of inference for VL formulas to the problem of grammatical inference, it is necessary to evaluate the applicability of the VL system as a means of expressing the problem and its solution. This comparison should yield interesting insights into the process of grammatical inference and into the structure of VL formulas. This section will discuss the formulation of the grammatical inference problem for formal languages in terms of the formalism of variable-valued logic. (Actually a subset of VL will suffice for the early discussions in which all input sets are of the same cardinality.)

As before, the problem is to infer from a positive information sequence and a (possibly empty) negative information sequence a grammar which describes the language of which the positive information sequence is a subset. This statement will be modified slightly to state that the desired result is a representation of the language not just a grammar. The nature of a VL representation of a language will be discussed at length in the next section.

For the discussions below, the simplest possible mapping from strings in the sample to VL variables will be made. All input sets will be of the same cardinality. The cardinality of the input sets will be that of the input alphabet as observed in the sample plus 1. The domains of the input variables will be the input alphabet and the previously unused symbol \$. The output set D will contain the values 0 and 1 and the meanings associated with these values will be 0 = not in language, 1 = in the language.

A number of input variables equal to the number of terminals in the longest string in the sample of the language will be needed. To simplify the constructions below, the special symbol $ will be used to signify the end of a string. That is the original positive and negative information sequences will be replaced by sequences in which $ has been added to the end of each string. A string will be represented in a VL system as a set of values of the input variables in positional notation. That is, the first input variable x_1 will be assigned the value of the first terminal in the string; x_2 the second and so on until the $ has been assigned. Input variables following the $ may be assigned arbitrary values as desired. For the discussion below the input variables after the $ will also be assigned the value $.

By using the procedure stated above it is possible to write a unique set of values for the input variables corresponding to any string over the alphabet used in the sample. This means that every string in the original alphabet (before the $ was added) corresponds to a unique square in the GLD.

The inference problem is thus stated: For each positive instance of a string in the sample place a 1 in the appropriate square of the GLD. For each negative instance of a string in the sample, place a 0 in the appropriate square of the GLD. All squares not marked correspond to strings about which nothing is known. The desired result is a formula which includes all of the events marked by a 1 and none marked by 0 (no error) and contains as few of the unmarked squares as possible (minimum discrepancy). In addition, the formula should be as simple as possible.

The tradeoff between complexity and discrepancy discussed by Cook [5,6] is clearly evident if the construction discussed above is used. Simplification is possible when specially selected don't care events are included. Having stated the problem in terms of a VL system, it is necessary to investigate the nature of possible solutions to the problem before attempting to infer descriptions for specific languages. The next section will discuss the representations of simple grammars in the VL system outlined above. The applicability and simplicity of VL descriptions of languages will be discussed.

6.0 VL Representations of Languages

Before the inference of VL_1 formulas can be applied to the inference of descriptions of languages, it is necessary to ascertain the applicability of VL systems to the representation of formal languages. It should be obvious from the definition of a VL system given above that the VL system as defined is not sufficient for recursion. There is no provision in the formalism to handle it. A less obvious result which will be demonstrated below is the fact that VL systems as restricted above are not sufficient to represent languages as well as type 3' grammars. The correspondence between VL_1 and grammatical representations of languages will be explored for simple classes of grammars below. New operations will be proposed for the VL system above and they will be used to extend the comparison.

The simplest grammar to be discussed is the type 3" grammar. In order to demonstrate that the VL system defined above is sufficient for languages described by type 3" grammars, algorithms for conversion between the two different formalisms will be outlined. In addition to the remarks of section 5 it is sufficient to specify the correspondence between productions and formulas.

Productions to VL_1 formulas.

1. Rewrite all productions with the same left-hand side (lhs) as one production using /.
2. Assign level number 1 to the production with the starting symbol on its lhs.
3. If no productions have been assigned then go to 6.
4. Write all productions whose lhs contain non-terminals referenced in the level above and assign this group of productions the next level value not assigned.

5. Go to 3.

6. Replace → with =.

7. Replace / with $\vee$.

8. Replace any single terminal, a, at level i with the form:

$$[x_i = a][x_{i+1} = \$]$$

9. Replace any terminal, a, next to a nonterminal at level i with:

$$[x_i = a]$$

where a may represent any terminal. The result is a system of VL_1 formulas which represents the grammar. An example will be presented below to demonstrate the process. Consider the type 3" grammar with start symbol S below:

E → a
D → c
E → c
S → aA
A → c
A → bD
A → cE

which can be rewritten in the form:

S → aA	level 1
A → c/bD/cE	level 2
D → c	
E → c/a	level 3

and the resulting VL_1 formulas are:

$$S = [x_1 = a]A$$

$$A = [x_2 = c][x_3 = \$] \vee [x_2 = c]E \vee [x_2 = b]D$$

$$D = [x_3 = c][x_4 = \$]$$

$$E = D \vee [x_3 = a][x_4 = \$]$$

The grammar and the VL_1 formulas above both represent the language:

{abc, ac, acc, aca}.

In order to show that the VL system defined above and type 3" grammars are equivalent it is necessary to show that for every VL_1 representation of a language there exists a corresponding type 3" language. Once again, the proof will be constructive.

VL_1 formulas to productions.

1. Rewrite the VL_1 formulas leaving out all selectors containing the symbol \$.
2. Rewrite the VL_1 formulas so that no two selectors are joined by the operation $\wedge$. Create new formula names if needed.
3. Remove the brackets and the x_i = from all selectors.
4. Replace = with $\rightarrow$ and $\vee$ with /.

The result is a set of productions which represent the same language as the set of VL_1 formulas. From the nature of the constructions it should be obvious that given any set of type 3" productions, if the productions are converted into a set of VL_1 formulas and then converted back to a set of productions, then the resulting set of productions will be the same as the original set. Since the inference of the simplest possible VL_1 expression for a language described by a type 3" grammar can be converted to the inference of a type 3" grammar which is just as simple, the inference of a VL_1 representation for a type 3" language is equivalent to the inference of a type 3" grammar directly.

A great deal can be learned from trying the conversion algorithms above on a grammar which is type 3' but not type 3". The examination of a few examples should be sufficient to convince the reader that the ordering

of productions used above is not possible for type 3'. If non-terminals which occur at more than one level are treated as a special case, then a set of VL_1 formulas which represent the grammar can be produced. The algorithm which was used to convert VL_1 formulas to a grammar can be used to convert back to a set of productions. An example will demonstrate the process. Consider the following type 3' grammar:

S → aA
A → c/bD/cE
D → c/bE
E → c/a

The reader can easily verify that the grammar above represents the language:

{abc, ac, acc, aca, abbc, abba},

and that the same language is represented by the following set of VL_1 formulas.

$$S = [x_1 = a]A$$

$$A = [x_2 = c][x_3 = \$] \vee [x_2 = c]E \vee [x_2 = b]D$$

$$D = [x_3 = c][x_4 = \$] \vee [x_3 = b]E'$$

$$E = [x_3 = c][x_4 = \$] \vee [x_3 = a][x_4 = \$]$$

$$E' = [x_4 = c][x_5 = \$] \vee [x_4 = a][x_5 = \$]$$

This example is simple enough that it can be shown that the VL_1 formulas above cannot be simplified. A most interesting situation arises when the set of formulas above is converted back to a set of productions. That set is shown below.

S → aA
A → c/cE/bD
D → c/bE'
E → c/a
E' → c/a

This grammar contains a non-terminal which was not found in the original set of productions. A moments thought will show that the new production is not necessary and can be removed to produce the grammar above. Thus in converting to a VL_1 representation and back something has been lost. The simplest VL_1 representation of the language is not as simple as the simplest type 3' grammar for the same language. This means that the inference problem for type 3' grammars is not solved if the simplest VL_1 description of the language can be found. The simplest VL_1 representation can be simplified after first being converted to a grammar.

In looking at the formulas for E and E' there is a distinct similarity in the structure of the formulas. In a type 3' grammar this similar structure could be portrayed by one single non-terminal E, but in the VL system defined above two formulas are required. In order to correct this situation, an addition will be made to the definition of the VL system above in order to allow the VL system to represent type 3' grammars. The addition proposed by Michalski [20] is to use selectors of the form:

$$[x_i, x_j, \ldots = a]$$

where a is an arbitrary terminal and the sequence of variables in the selector should be interpreted as follows: if x_i or x_j or ... is equal to the terminal a, then the selector takes the maximal value in the output set else it takes the value zero. As an example of its use, the set of formulas for the language above will be rewritten using the new selector.

$$S = [x_1 = a]A$$

$$A = [x_2 = c][x_3 = \$] \vee [x_2 = c]E \vee [x_2 = b]D$$

$$D = [x_3 = c][x_4 = \$] \vee [x_3 = b]E$$

$$E = [x_3, x_4 = c][x_4, x_5 = \$] \vee [x_3, x_4 = a][x_4, x_5 = \$]$$

The set of formulas above represents the language and can be converted into the original grammar by the algorithm stated below.

VL_1 formulas to productions.

1. Rewrite the VL_1 formulas leaving out all selectors containing the symbol \$.

2. Rewrite the formulas so that no two selectors are joined by the operation $\wedge$. Create new formulas if needed.

3. Remove the brackets, the sequences of variable references and the equal sign from the selectors.

4. Replace = with → and $\vee$ with /.

The conversion from a type 3' grammar to a set of VL_1 formulas can be accomplished by the following algorithm:

Productions to VL_1 formulas.

1. Rewrite all productions with the same lhs as one production using /.

2. Assign level number 1 to the production with the starting symbol as its lhs.

3. If no productions have been assigned level numbers and the previous level contained no non-terminals on the rhs of any productions, then go to 6.

4. Write all productions whose lhs contain non-terminals referenced in the level above and assign this group of productions the next level value not yet used.

5. Go to 3.

6. Remove all multiple copies of productions, but associate the level numbers of the productions removed with the copy left behind.

7. Replace → with = and / with $\vee$.

8. Replace any single terminal, a, in a production with associated level values i,j,k... with the form:

$$[x_i, x_j, x_k, \ldots = a][x_{i+1}, x_{j+1}, x_{k+1}, \ldots = \$].^{\dagger}$$

9. Replace any terminal, a, in a production with associated level values i,j,k,... with the form:

$$[x_i, x_j, x_k, \ldots = a].^{\dagger}$$

where a may represent any terminal.

As an example of the application of the conversion algorithm for the conversion from a grammar to a set of VL_1 formulas, the type 3' grammar above will be converted to a VL system. The result of the ordering of productions is:

S → aA	level 1
A → c/cE/bD	level 2
D → c/bE	
E → c/a	level 3
E → c/a	level 4

and the resulting set of formulas is the set of formulas used as an example of the definition of the new selector. It is simple to show that the set of formulas can be converted back to the same grammar for any type 3' grammar, so the equivalence between the inference problem for type 3' grammars and the extended VL system defined above is established. This means that the simplest result in one representation can be converted to an equivalent representation which is isomorphic to the original. The relationship between an extended VL_1 system, called VL_2 [20], and grammatical inference will be discussed in paper [24].

---

†In general, it may be necessary to add selectors of the form $[x_\ell = \$]$ for $\ell > i$ to eliminate the introduction of discrepancy.

7.0 Simplified VL Representations

The previous section presented an extension to the VL_1 system which served to increase the classes of languages which could be represented by a set of VL formulas. This section will explore the use of some operators in VL_1 which do not change the classes of languages which can be represented, but which allow a more concise representation of those already represented.

The comma was introduced as a way to indicate that for the purposes of the given selector a group of variables could be treated together, and the selector could be satisfied by any one of the variables. The existing VL_1 system allows the comma to be employed in a similar manner when specifying the values of the input variables for comparison. The meaning is the same as for the previous case. A selector with input values separated by commas is satisfied if any indicated input variable has the value of one of the values in the list. Using this feature, the first example of section 6.0 can be written:

$$S = [x_1 = a]A$$

$$A = [x_2 = c][x_3 = \$, E] \vee [x_2 = b]D$$

$$D = [x_3 = c][x_4 = \$]$$

$$E = [x_3 = a, c][x_4 = \$]$$

where the use of the comma is defined above. This equivalent representation contains only 8 selectors where the previous one contained 11. The selectors more compactly represent the same language. As a natural extension of this construct the use of : as defined by Michalski provides

an additional savings of expression. In addition, when the input set is ordered on the relation < (or mapped into a representation such as the positive integers for which the relation is defined), then the use of : to indicate a sequence of possible values represents a computational savings. It is no longer necessary to make the individual comparisons for each value in the interval defined by the :, but the question of whether the specified input variable has a value in the range can be answered by checking the end points only. The use of the comma for the specification of a sequence of values is the same as specifying an equivalence class of values which, for the process of evaluating this selector, may be treated as a unit. The use of the symbol : makes the manipulation of the specified sequence especially simple.

Michalski defines a number of other operators which can be used to simplify the expression and computation of the value of a VL_1 expression. The comparison $\neq$ is especially useful when the number of elements of the input domain to be excluded is less than the number of elements to be included. This simplicity of expression becomes most effective when combined with the use of sequences of values as defined above. For input sets which are ordered, the operations $<$, $>$, $\leq$, $\geq$ can provide simplifications of selectors. Since all of the above operations can be performed (although less concisely) with the test for equality, the addition of these operators does not change the class of languages for which VL is applicable, but they do allow a more concise manner of representation.

As an example of simplification, a grammar will be presented below and converted to a set of VL_1 formulas which will be simplified. The set of simplified VL_1 formulas will be used to answer the membership

question for the language defined by the grammar. The performance of the grammar will be compared to that of the VL_1 formulas on a production by production basis. The sample grammar is shown below:

$T = \{a, b, c, d\}$

$N = \{S, B, D, E, F\}$

$S \rightarrow aB/cB/bB/aD/bE$
$B \rightarrow a/b/d$
$D \rightarrow aF/cF$
$E \rightarrow bF$
$F \rightarrow c$

The order relations will test order in the alphabetic sense, thus the expression $a<b$ is true while $c<a$ is not. The VL representation of the grammar is shown below along with a simplified set of VL formulas.

$$S = [x_1 = a]B \vee [x_1 = b]B \vee [x_1 = c]B \vee [x_1 = a]D \vee [x_1 = b]E$$

$$B = [x_2 = a][x_3 = \$] \vee [x_2 = b][x_3 = \$] \vee [x_2 = d][x_3 = \$]$$

$$D = [x_2 = a]F \vee [x_2 = c]F$$

$$E = [x_2 = b]F$$

$$F = [x_3 = c][x_4 = \$]$$

and the simpler form:

$$S = [x_1 = a{:}c]B \vee [x_1 = a]D \vee [x_1 = b]E$$

$$B = [x_2 \neq c, \$][x_3 = \$]$$

$$D = [x_2 = a, c]F$$

$$E = [x_2 = b]F$$

$$F = [x_3 = c][x_4 = \$]$$

A good intuitive idea of simplicity of representation for a set of objects is the number of questions which have to be asked to determine if a given object is in the class. It seems intuitively well justified to say that a representation which requires that fewer questions be asked is simpler than one which requires more. If the

first production above were used to answer the membership question for a given set of strings (answer for each string whether or not the string was in the language), 10 questions would be asked. The questions would be of the form: Is the first letter an a and if so can the rest of the string be represented by B? The number of questions can be reduced to 8 if the first 6 take the form: Is the first letter a, c, or b, and can the rest of the string be represented by B? If the simpler VL_1 formula is used to answer the same questions, only 7 questions need to be asked. The reduction comes from the fact that no test for first letter of b need be made. As the productions become more complex and as the input alphabet grows in size, the savings can be expected to grow also. Similar comparisons are possible for the other productions with similar results. The explicit reference to $ in the VL case does not represent any extra work since the use of a terminal alone in a production carries an implicit check for the end of the string. The $ just makes the check explicit for the VL case.

Michalski presents other simplification operators which will not be discussed here but have similar effect. The distinction between operations which increase the classes of languages which a VL system can represent and operations which produce a simpler form of representation for the same class of languages, may be of help in designing new additions to the VL system or in evaluating the old.

8.0 Grammatical Inference and Pattern Recognition

As pointed out in section 3.0, the formalism of grammatical inference holds promise for application in the area of pattern recognition. The problem of the inference of pattern grammars as stated in section 3.0 is more complex than the inference of a formal language grammar from sets of positive and negative instances of strings of the language.

The first step in any pattern recognition process is the expression of the pattern as a set of numbers. In the general case, this could correspond to characterizing the pattern in terms of a set of primitives which are arbitrarily chosen. Numbers might be assigned to represent abstract constructs such as circle, line, and dot, or these might be represented by pairs of numbers such as radius and center, or end points.

Another problem to be treated is the problem of continuous variables. Since no computing machine allows the representation of continuous variables of infinite precision, it is necessary to quantize the continuous variables into discrete compartments. The selection of compartments may be conscious or it may be determined by the number of bits of a computer word assigned to represent a number. Thus, even continuous variables must be quantized.

For many applications it may suffice to use a rather coarse quantization of the continuous variable and thus simplify the number of different values to be treated. It might be sufficient to speak of human height as short, medium, or tall rather than inches measured from

0 to 100. By limiting the number of different values recognized, the description of any common properties can be more easily recognized.

The process that Evans calls the choosing of predicates is really the definition of operators which can be used to compare the primitive elements of his descriptions. In a very real sense, the specification of predicates is the specification of which operators are allowed in selectors in a VL system. The selection of input variables corresponds to the selection of the primitive properties of the pattern with which to express the pattern in numbers. In the case of grammatical inference for formal languages (at least for type 3' languages) the input alphabet is given and the only predicate needed is =.

If all of the above problems are overcome, the problem of inferring a simple representation for the pattern remains. At this point the desire for simplicity should be clearly specified. In the most general case, it would be appropriate to assign a cost to each of the primitive parameters to be associated by the inference process with the cost of evaluating that primitive. It would also be desirable to specify a cost of each comparison between primitives (as done in VL) and then minimize the total cost of the expression for the pattern.

Grammatical inference procedures generally do not allow the specification of specialized measures of cost or complexity. It is generally assumed that simplicity means the fewest possible number of productions or non-terminals. Cook [5,6] and Feldman [9] have discussed more generalized measures than productions or non-terminals, but neither has a fully generalized cost functional such as found in the inference of a VL system. Most grammatical inference processes have a prespecified

form for the complexity measure to be minimized and do not lend themselves to the minimization of a generalized cost functional. A VL inference process does provide such a capability.

Unlike the situation with the inference of grammars for formal languages, the solution is not uniquely defined. If a given cost functional is minimized for the inferred grammar then the solution is as good as any that can be found. In the case of the inference of a description of a pattern, it is possible that the initial choice of primitives was not optimal, or that the set of operations on the primitives was not complete, or that the quantization of the values was not optimal. Thus, unlike the formal language case, it might be possible to redefine the input alphabet (set of input variables or their domains) in such a way that a simpler description is possible.

There is no provision in grammatical inference for the inference of modifications in the input alphabet which can result in a simpler description. As noted in the case of VL systems, the use of the operators "," and ":" define classes over the input variables or their values. These constructs appear to be a first step toward inferring modifications to the input alphabet in order to produce a more efficient representation of a pattern. Consider the case of a quantization of a continuous variable which is finer than needed. Such a variable would always be compared against sequences using the : to specify the interval. If all references to values within the interval were always references to the same interval, then the interval could be replaced with a value on a coarser quantization of the variable. Such a substitution would leave the representation simpler and easier to use.

As another example of simplifications made possible by the VL formalism, consider the situation where a given collection of values separated by a comma is found repeatedly in the VL description. If the input set from which the values are selected is unordered, then it can be ordered so that the sequence separated by commas could be replaced by a sequence using the colon. After that substitution, simplifications such as those discussed above might apply. In addition, other patterns might emerge in the input sets in repeated applications of the process.

It is necessary to state that the above comments do not constitute a proof that a VL approach to pattern recognition is more powerful than an approach involving formal grammars, but some interesting new possibilities which the VL approach gives have been found.

NOTES

[1] Biermann, A. and J. Feldman, "A Survey of Grammatical Inference," in S. Wantanabe (Ed.), Frontiers of Pattern Recognition, Academic Press, New York, 1972.

[2] Biermann, A. W. and J. S. Feldman, "On the synthesis of finite-state machines from samples of their behavior," IEEE Trans. on Computers, June 1972, pp. 592-597.

[3] Chomsky, N., "On certain formal properties of grammars," Inf. and Control 2 (1959), pp. 137-167.

[4] Chomsky, N., Syntactic Structures, Mouton and Co., The Hague, 1957.

[5] Cook, C. M., "A cost function for concept formation," Technical Report 212, University of Maryland Computer Science Center, College Park, Maryland, December, 1972.

[6] Cook, C. M., "Experiments in grammatical inference," Technical Report 212, University of Maryland Computer Science Center, College Park, Maryland, August, 1973.

[7] Crespi-Reghizzi, S., M. A. Melkanoff and L. Lichten, "The use of grammatical inference for designing programming languages," Comm. ACM 16, 2 (Feb. 1973), pp. 83-90.

[8] Evans, T. G., "Grammatical Inference Techniques in Pattern Analysis," in J. T. Tou (Ed.), Software Engineering, Academic Press, New York, 1971.

[9] Feldman, J. A., "Some decidability results on grammatical inference and complexity," Inf. and Control 28 (1972), pp. 244-262.

[10] Feldman, J. A., J. Gips, J. T. Horning and S. Reder, "Grammatical Complexity and Inference," Technical Report No. CS125, Computer Science Department, Stanford University, June 1969.

[11] Fu, K. S., "A survey of grammatical inference," TR-EE 72-18, Purdue University School of Electrical Engineering, June 1972.

[12] Gold, M. E., "Language identification in the limit," Inf. and Control 10 (1967), pp. 447-474.

[13] Hopcroft, J. E. and J. D. Ullman, Formal Languages and their Relation to Automata, Addison-Wesley, Reading, Mass., 1969.

[14] Kain, R. Y., Automata Theory: Machines and Languages, McGraw-Hill Book Company, New York, 1972.

[15] Michalski, R. S., "A variable-valued logic system as applied to picture description and recognition," in Graphic Languages, Nake, F. and A. Rosenfeld, (Eds.), North-Holland Publishing Company, London, 1972.

[16] Michalski, R. S., "AQVAL/1--computer implementation of a variable-valued logic system VL_1 and examples of its application to pattern recognition," in Proceedings of the First International Joint Conference on Pattern Recognition, October 30--November 1, 1973, Washington, D.C., (73 CHO 821-9c).

[17] Michalski, R. S., "Variable-valued Logic: system VL_1," presented to the 1974 International Symposium on Multiple-Valued Logic, West Virginia University, Morgantown, West Virginia, May 29-31, 1974.

[18] Michalski, R. S., "A geometrical model for the synthesis of interval covers," University of Illinois Department of Computer Science, Report 461, c00-2118-0013, Urbana, Illinois, June, 1971.

[19] Michalski, R. S. and B. H. McCormick, "Interval generalization of switching theory," University of Illinois, Department of Computer Science Report 442, c00-2118-0008, Urbana, Illinois, May, 1971.

[20] Michalski, R. S., "Learning by inductive inference," Proceedings of the NATO Seminar on Computer Oriented Learning Processes, Bonas, France, Aug. 26 - Sept. 7, 1974.

[21] Miller, G. A., "Finite-state machine induction," M.S. Thesis, The Moore School of Electrical Engineering, University of Pennsylvania, May 1969.

[22] Solomonoff, R., "A formal theory of inductive inference," Inf. and Control 7 (1964), pp. 1-22, pp. 224-254.

[23] Solomonoff, R. J., "A new method for discovering the grammars of phrase structure languages," Information Processing, UNESCO, Paris, June 1959, pp. 285-290.

[24] Baskin, A. B. and R. S. Michalski, "Variable-valued logic and grammatical inference," in preparation.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-74-663	2.	3. Recipient's Accession No.

4. Title and Subtitle

A COMPARATIVE DISCUSSION OF VARIABLE-VALUED LOGIC AND GRAMMATICAL INFERENCE

5. Report Date: July, 1974

6.

7. Author(s): A. B. Baskin

8. Performing Organization Rept. No.

9. Performing Organization Name and Address

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

10. Project/Task/Work Unit No.

11. Contract/Grant No.

12. Sponsoring Organization Name and Address

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

13. Type of Report & Period Covered

14.

15. Supplementary Notes

16. Abstracts

This paper reviews what is meant by grammatical inference and it discusses some of the currently used methods of grammatical inference. The application of a variable-valued logic system to this inference problem is explored and, where possible, direct comparisons between methods are discussed. Examples of a correspondence between a variable-valued logic inference process and a grammatical inference process are presented. Operations which increase the class of languages to which variable-valued logic can be effectively applied and operations which allow simplified variable-valued logic descriptions of classes of grammars are discussed. Type 3' and type 3" grammars are defined (both subsets of type 3 grammars) and the equivalence of variable-valued logic formulas and type 3' and type 3" grammars is illustrated.

17. Key Words and Document Analysis. 17a. Descriptors

variable-valued logic
inference
grammatical inference

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages
	20. Security Class (This Page) UNCLASSIFIED	22. Price