

The University of Wisconsin
Computer Sciences Department
1210 West Dayton Street
Madison, Wisconsin 53706

FLEXIBLE LINGUISTIC PATTERN RECOGNITION

by

Leonard Uhr

Technical Report #103

October 1970

Flexible Linguistic Pattern Recognition

by

Leonard Uhr

OVERVIEW

A. INTRODUCTION

1. The spirit of linguistic pattern recognition is to examine the structures of interrelations in patterns.

Actual practice is to apply overly restrictive syntactic techniques, in particular phrase structure grammars. This paper describes a number of "flexible" extensions.

2. Phrase structure grammars generate and parse "grammatical" sentences by the application of a suitable sequence of rewrite rules. They operate on 1-dimensional linear strings of concatenated objects (words).

3. Phrase structure grammars can be used for linear connected strings in n dimensions so long as the beginning and the end is designated for each object in the string. Some patterns are such strings, or can be reduced to one linear string (in particular, an area can be reduced to its contour).

4. More elaborate pre-processing can often reduce a pattern to a set or a graph of linear strings. Phrase structure grammars can

be extended, by adding new operators to the standard concatenation operator, to handle connected graphs.

5. Techniques for describing graphs - the connection matrix and linear description - seem simpler and more structurally meaningful.

B. FLEXIBLE TECHNIQUES:

1. Objects can be connected at internal points and regions, as well as at their ends.

2. Objects can be connected at a distance. Distance and position can be an objective measure of the space, or the objects found in the space; or a subjective measure; or a set of bounds.

3. Specifications of positions and connections, or other relations, can be given loosely.

4. A technique is presented for successively more loosely applied specifications of position and relative distance.

5. Areas can be specified, and compounded into larger wholes.

6. Rewrite rules can be modified so that they succeed with less-than-perfect match, using a threshold, and counts or weights.

7. A rewrite rule can imply more than one replacement (which can include names of subsequent rewrite rules as well as pattern names), and a decision function can choose among multiple implications.

C. DISCUSSION

1. Real-world inputs are often defective and "ungrammatical" yet can be recognized. We should judge recognizers on efficiency and power.

2. We must relax the concept of a set of easily recognizable and unambiguous things (the words) concatenated together into a connected, bounded string (the sentence) that can be perfectly, completely, unambiguously parsed if we expect to recognize real world fields of objects in backgrounds.

3. Traditional syntactic pattern recognizers do poorly on real patterns, and need extensive pre-processing and feature extraction before the syntactic section takes over.

4. An important aspect of the syntactic approach in linguistics is to gain insight about the structure of language. Have we gained such insight from linguistic pattern recognition? How heavily should we rely upon phrase structure as opposed to transformational and semantic techniques?

5. The syntactic approach re-emphasizes the importance of structure. But it should be made more flexible, to handle the looser and more complex structure of real-world patterns; and it should be

combined with the parallel-serial, multiple-characterizer, probabilistic structure of the typical pattern recognizer.

BACKGROUND INFORMATION

A large number of papers have been written during the past few years on "linguistic" methods for pattern recognition. This work has been unusually well reviewed by Miller and Shaw (1968), Evans (1969), and Fu and Swain (1969). In almost all cases, "linguistic" has mean "syntactic" structure, despite the fact that several other things studied by linguists - notably the recognition and description of basic units (e.g. phonemes, morphemes, words), transformational grammars, and the study of semantics - are of central relevance to pattern recognition. (When it comes to recognizing semantic units, or even phonemes, linguists are confronted with deep problems of perception and cognition and, I suspect, could benefit from an examination of the research in pattern recognition and artificial intelligence.)

The Flavor of a Linguistic Approach

The spirit of the linguistic approach is, I think, to bring the idea of "pattern" back into pattern recognition research. A pattern is a complex structure of meaningfully interacting and interrelated

things, which in their turn are complex structures. Thus a "Boy" is a structure of head, trunk, arms and legs; a "Head" has hair, neck and face; a "Face" has eyes, ears, chin, mouth, cheeks; and so on. A boy and a girl in a certain orientation are a dancing couple and sometimes we might agree that the girl is "following" the boy.

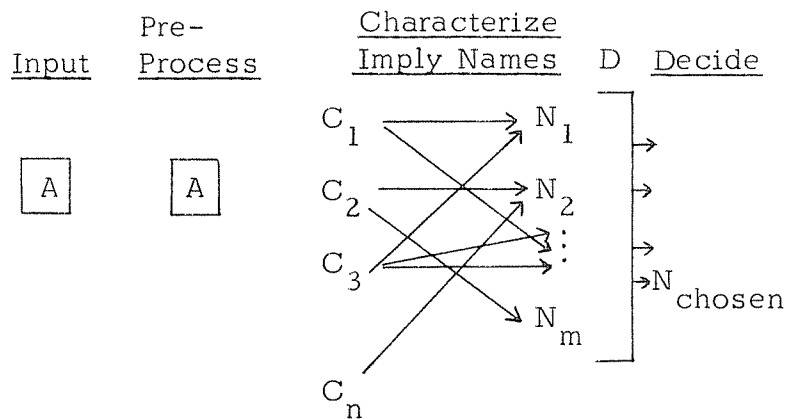
Ultimately we want pattern recognizers to transform inputs into meaningful descriptions of the sort indicated above. But the actual situation today is one in which they classify, choosing and outputting one from a mutually exclusive set of names that might be assigned to an input. The linguistic approach will play a vital role in the move toward description.

We typically think of a pattern recognizer as A) Pre-processing the input (e.g. smoothing and eliminating noise, translating and rotating) B) Characterizing (e.g. finding contours, strikes, angles and other information-rich features), and C) Deciding (choosing the single most highly implied name). Most research has concentrated on step C, looking for powerful decision procedures. And research on B, to get and use good features, has tended to consider features as independent things, each to be applied without respect to the others. This is illustrated in Figure 1a, which shows the typical

structure of pattern recognizers. But other structures are used, as shown in 1b, 1c, and 1d.

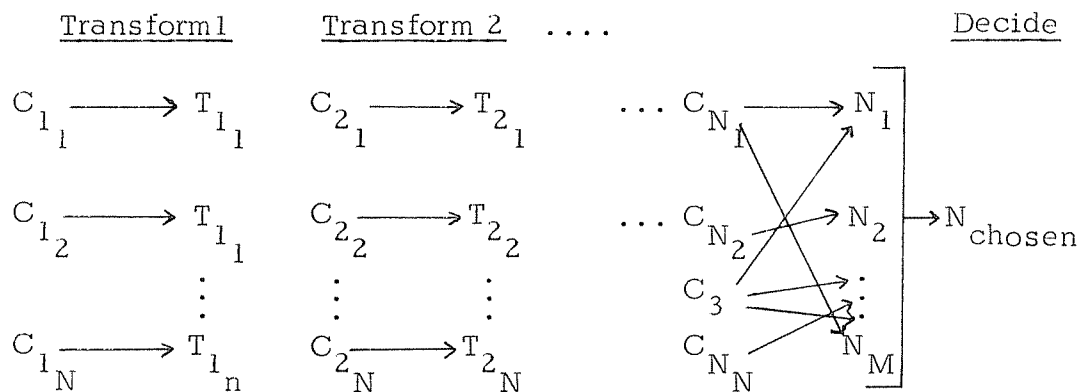
Figure 1. Some Pattern Recognizer Structures

- a) Parallel Characterizers (e.g. Bledsoe and Browning's, 1959, N-Tuples

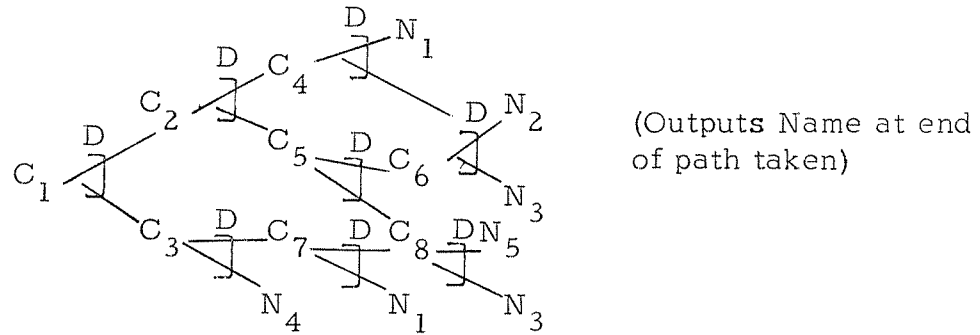


C = Characterizer
 N = Pattern Name
 D = Decision function

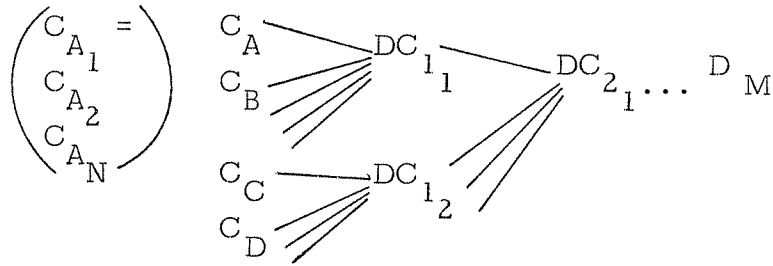
- b) Characterizers that Successively Transform, then Imply Names
 (e.g. Rosenblatt's, 1958, multi-layer, "perceptrons")



- c) Serial Characterizer Sort (e.g. Unger, 1959; Hunt, 1962; Towster's, 1969, Concept Formers)



- d) Parallel - Serial (eg. Selfridge's, 1959, "Pandemonium")



(Each C_A, C_B , etc. is a whole set of parallel characterizers, $C_{A_1}, C_{A_2}, \dots, C_{A_N}$. Each DC is a Demon that Decides, sending its Characterization to the next higher-level Demon. D_M is the Master Demon.)

The "linguistic" approach is an important corrective, since it focuses interest on the structuring of features into meaningful higher-level wholes. But it can easily become a straitjacket, if it insists upon a complete and exact description of the input, in the spirit of

the linguist's insistence upon a "grammatical" sentence. As we will see later, a good bit of non-linguistic pattern recognition research also examines structure.

The letter of the linguistic approach, at least to date, has been syntactic - applying and extending "phrase structure" grammars (developed by Chomsky; see e.g. Chomsky, 1957, 1963, 1965; Chomsky and Miller, 1965) to 2-dimensional patterns.

We will now examine phrase structure grammars, some of the attempts that have been made to extend them to 2-dimensional patterns, and some of the problems that arise. Then I will present some data structures and some techniques for "flexible" "syntactic" recognition that may overcome these problems. The syntactic approach is a good corrective, but it should be blended smoothly with the rather loosely structured probabilistic parallel structure that has given a number of pattern recognition programs a surprising amount of power.

Phrase Structure Grammars

A "grammar" for (linear) languages consists of a set of "rewrite rules" that indicate how one thing can be replaced by one or more strings of things. There are three kinds of things: 1) one root, 2) classes, and 3) terminals.

Let's look at a simple example of a grammar:

<u>Number</u>	<u>Rewrite Rule</u>	<u>Equivalent</u>	<u>Shortened Version</u>
R1	Sentence = Nounphrase+ Verbphrase	≡	S = NP VP
R2	Nounphrase = Article Adjective +Noun/Article+Noun	≡	NP = AR A N/A N
R3	Verbphrase = Intransitiveverb/ Transitiveverb+Nounphrase	≡	V = IV/TV NP
R4	Article = THE/A/THIS	≡	AR = AR1/AR2/AR3
R5	Adjective = RED/BLUE/BIG/ PRETTY	≡	A = A1/A2/A3/A4
R6	Noun = BOY/GIRL/DOG/BONE	≡	N = N1/N2/N3/N4
R7	Intransitiveverb = WALKS/RUNS	≡	IV = IV1/IV2
R8	Transitiveverb = KISSES/PATS/ EATS	≡	TV = TV1/TV2/TV3

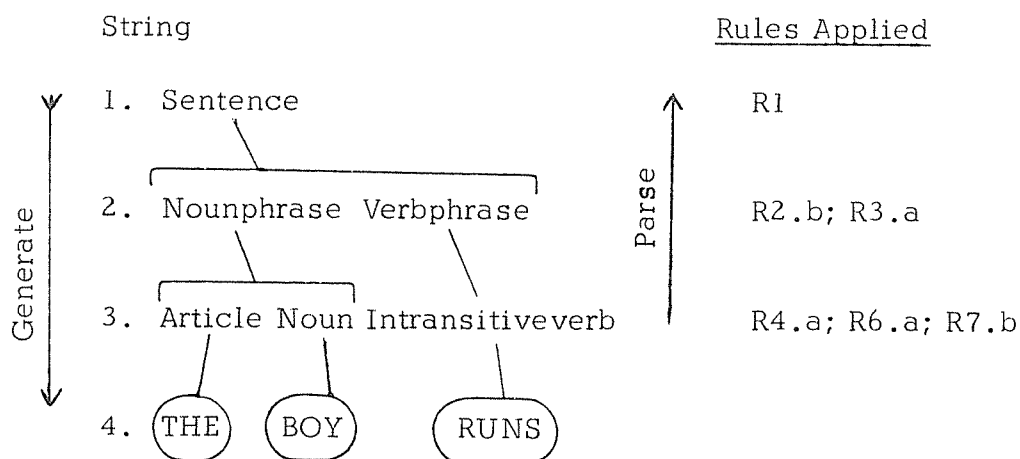
The "=" indicates "what's to the left is equivalent to one of the strings to the right", where the "/" denotes the end of each of the possible strings. The "+" (which is often absent) indicates concatenate (join together). "Sentence" is the root. Words in caps (e.g., RED,BOY) are the terminals found in actual sentences. The following are a few sentences this grammar will generate, or parse:

"THE BOY RUNS"

"A GIRL KISSES THE BOY"

"THE PRETTY BOY EATS THIS RED GIRL"

To generate a sentence, we start with the root, "Sentence". Successive transforms are effected by finding a left side of a rewrite rule in the string being transformed, and replacing it by the right side of that rewrite rule. When only terminals (that is, words) remain, the sentence is done. Thus "THE BOY RUNS" is generated in the following steps:



To parse a sentence, we start with the string of terminal words that form the sentence. Now right sides of rewrite rules are found and replaced by left sides. The string is considered parsed when the "Sentence" node is reached. Thus "THE BOY RUNS" parses, and is a legal sentence, by the reverse order, right-to-left application of the rewrite rules used for generation. (Parsing gets more complex, as sentences get more complex, and usually is not just a simple reversal of generation.)

If a sentence is successfully parsed it can be called "gram-matical", and its syntactic structure has been got by the parsing process. Thus "THE BOY RUNS" gets the structure

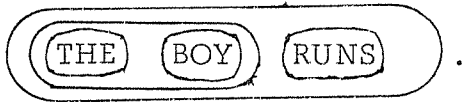
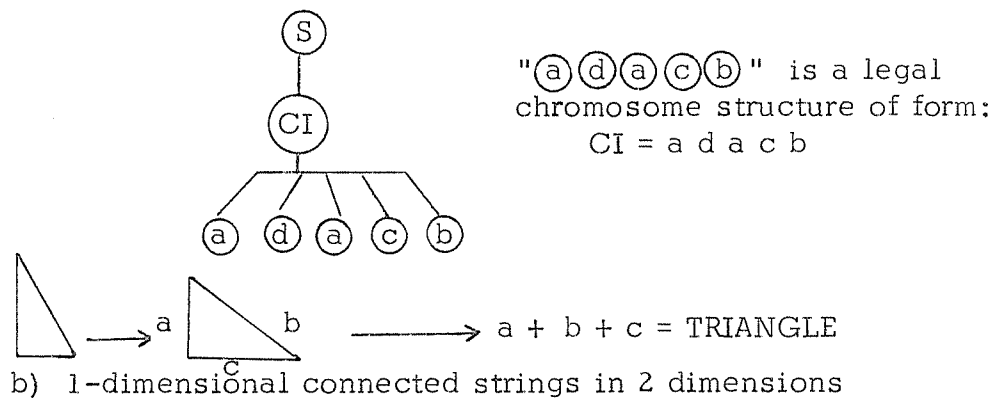
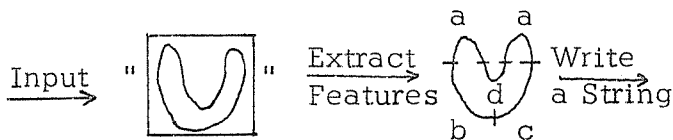
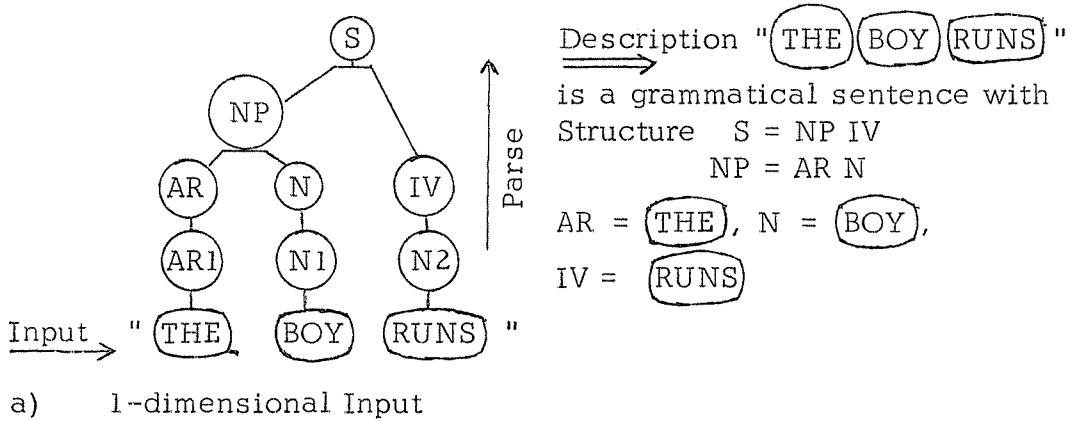


Figure 2. Parsing Linear Strings



Applying Phrase Structure Grammars to Patterns

Such a grammar can be successfully applied, without modification, to a few n-dimensional patterns - those that can be mapped onto the line in such a way as to preserve the basic units that the grammar must work with. In the syntactic analysis of language the basic units - the words - are given, and are unambiguously recognizable. In pattern recognition pre-processing and feature extraction are needed just to get these basic units.

There are of course a potentially infinite number of ways to map a finite set of points in an n-dimensional space into a linear string. Simply scan, as done by a tv camera or a computer when inputting a matrix. Or randomly choose points, without replacement. The problem to get the same string of features for all variations of the pattern. The linguist's syntactic analysis of language assumes such a string, that is, it assumes that the words (the features) have been perfectly extracted.

Linear Grammars for Contours of Areas

There is at least one kind of pattern that might be handled with a traditional phrase structure grammar - a pattern that is an area whose contour is a single line that can be decomposed into simple curve features; for example, a triangle, or a chromosome.

Ledley (Ledley et al, 1965; Ledley and Ruddle, 1965) has developed such a grammar for chromosomes (Figure 2). He uses a curve-follower to decompose a contour into simple curve features, and then describes the chromosome as a string of these features. Thus his "grammar" includes the several stroke features, and strings of these features to describe chromosomes.

Many other patterns could be similarly described. Triangles, rectangles, and other polygons can be described as linear strings of straight line segments of various lengths and slopes. Attneave and Arnoult (1956) proposed a simple system for describing contours, and describe their own faces as linear strings of these curves.

Looping Branching and Disconnected Figures

As soon as loops, branches, or disconnected figures occur, we can no longer make such a direct application of standard phrase structure grammars. Either a program must pre-process such patterns, to convert them to linear strings, or we must develop an extended phrase structure grammar.

Prather and Uhr (1964) first converted compounds of linear strokes, of the sort we find in letters, into thickened patterns - that is, into areas, not lines - and then got their contours. This gives a single contour string for linear letters such as C and M, and also

for some branching letters, such as E, G and H.

Patterns that contain loops, e.g. A and B, or are disconnected, e.g. i and =, will give several such strings. To handle this we must either link them into a single string, develop a grammar for sets of strings (similar to a grammar for paragraphs rather than sentences), or use more flexible syntactic techniques of the sort we will examine later.

Extended Linear Grammars for Graphs

Most syntactic pattern recognition techniques work with compounds of strokes, where thin lines are assumed and branches and loops (but not disconnected lines) are allowed. Either the patterns are made up of such thin strokes (as are the letters of the alphabet and spark and bubble chamber events), or a pre-processor is assumed that converts them into such strokes. In either case, the strokes must be recognized by feature extractors, and the input to the syntactic recognizer is a graph of stroke features, rather than a linear string.

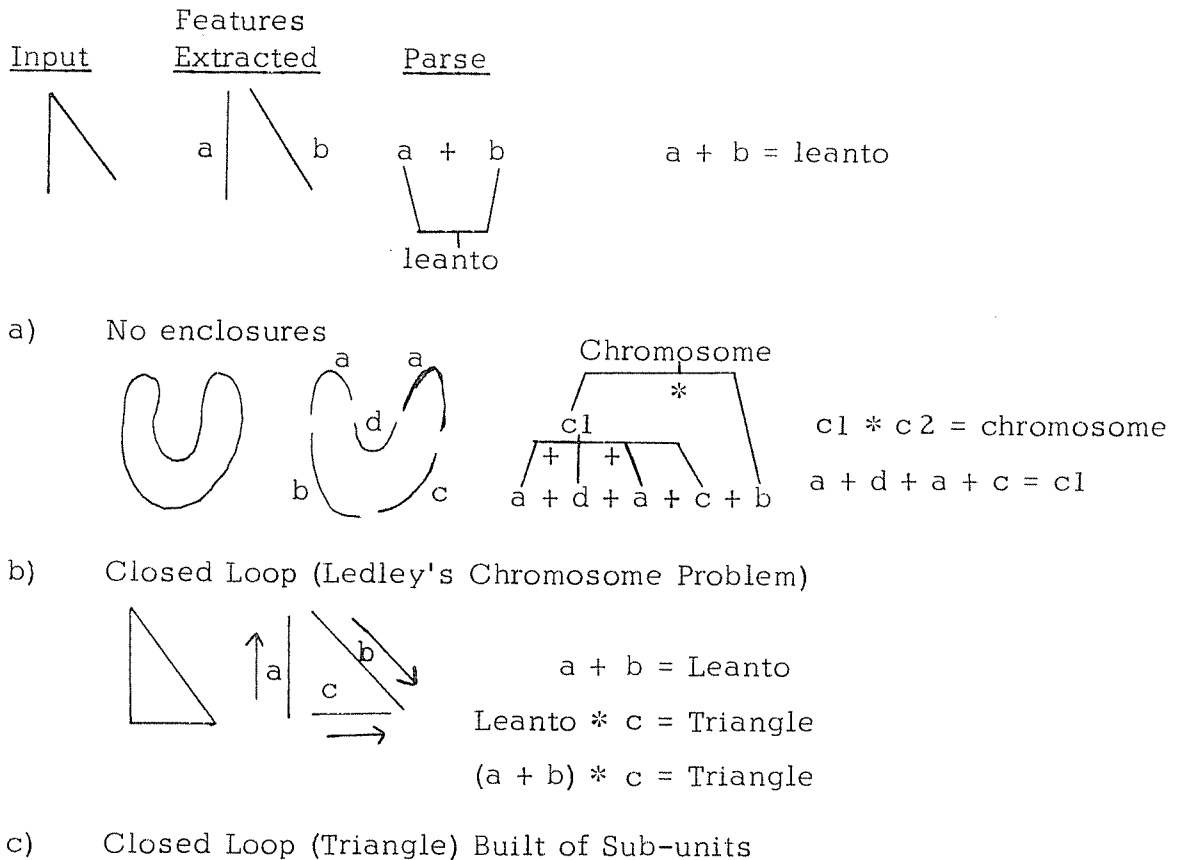
Shaw's (1969) Picture Description Language (PDL) is a good example of such an extended grammar. The standard phrase structure rewrite rule is of the form: $C1 = C2 + C3$ (e.g., Sentence = Noun-phrase Verbphrase), where each of the class elements (e.g. C2)

is assumed to be a linear string whose rightmost element is to be concatenated (that is joined) to the leftmost element of the next string. Because sentences are in 1-dimension their left-to-right representation in a line of print indicates the left-to-right order in which their words are joined. But in 2- and n-dimensional spaces a line can be oriented in an infinite number of ways. Shaw requires that the "tail" and "head" be specified for every line (equivalent to the "left" and "right" specification that is implicit for things ordered in a linear string). Now the concatenation operator in a statement like $C1 + C2$ means "attach the head of $C1$ to the tail of $C2$ ". Shaw introduces several additional operators to connect other combinations of heads and tails. For example, $C1 \times C2$ means "attach the tail of $C1$ to the tail of $C2$," $C1 * C2$ means "attach the tail of $C1$ to the tail of $C2$, and the head of $C1$ to the head of $C2$ (that is, form a loop).

In addition, Shaw's operators specify what are the tail and the head of the new combined string. For "+" (concatenation) this is simple - the tail of the first and the head of the second. For "x" it is arbitrary - the head of the first and the head of the second.

These extensions allow Shaw's PDL to describe any connected compound of strokes by a set of successive pairwise compounding operations, as illustrated in Figure 3. The general principle is to keep

Figure 3: Picture Description Language (PDL) Applied to Graphs



an end a "tail" or "head" until it has been joined to all the other strokes that connect to it. But the building up of the description seems somewhat unnatural and awkward. All strokes must connect at their "heads" and "tails." Two large units (e.g. the back and the seat of a chair) cannot be built up separately, and then connected. No variations will be tolerated. And something so simple as T or + can only be handled in what seems a rather ad hoc and contrived

way-by breaking the straight line at its join, orienting head and tail properly, and using an operator that outputs a tail in the middle of a line.

Describing Graphs and Patterns

Rather than pre-process the input down to one or more linear string(s), or try to extend an essentially linear grammar to handle graphs, we can use techniques that handle graphs directly.

Such techniques have a strong flavor of the syntactic, and some of the early attempts are usually referred to as syntactic (e.g. Grimsdale et al's, 1959, by Miller and Shaw, 1968). But they actually grew out of a completely different tradition, and owe little if anything to the linguistic approach that developed phrase structure grammars. They grow from a need to deal directly with graphs, rather than with strings, of connected objects.

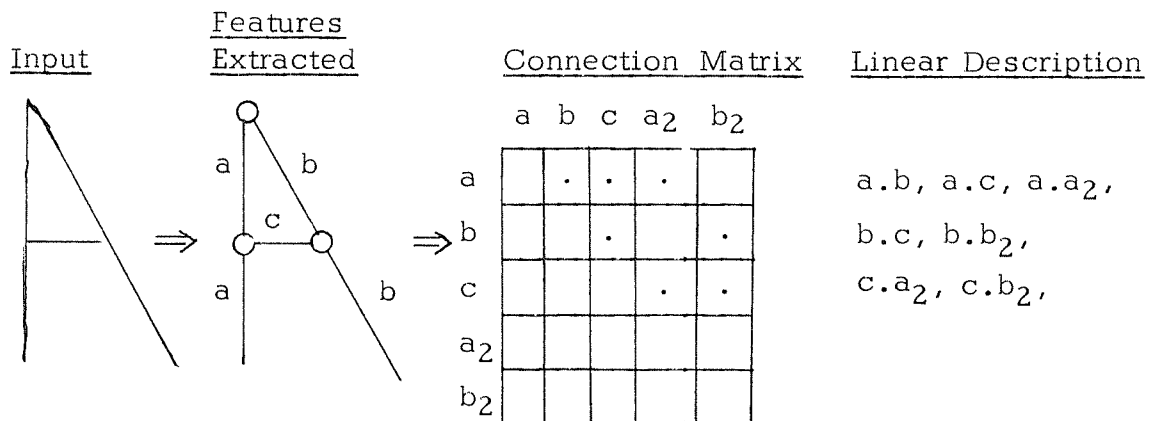
A standard way of describing a graph is with a connection matrix in which each node of the graph labels a row and a column, and an entry is made in each cell whose row and column names are connected in the graph. Standard graphs treat nodes as points, so we need only 1 symbol, say "." to indicate "connected" (Figure 4a). To handle line segments, we might, with Ledley, merely specify

"+" for concatenation, or use Shaw's technique, specifying a "tail" and a "head" for each node, and a set of compounding operators to connect (e.g. "+" indicates "head-tail", "x" indicates "tail-tail").

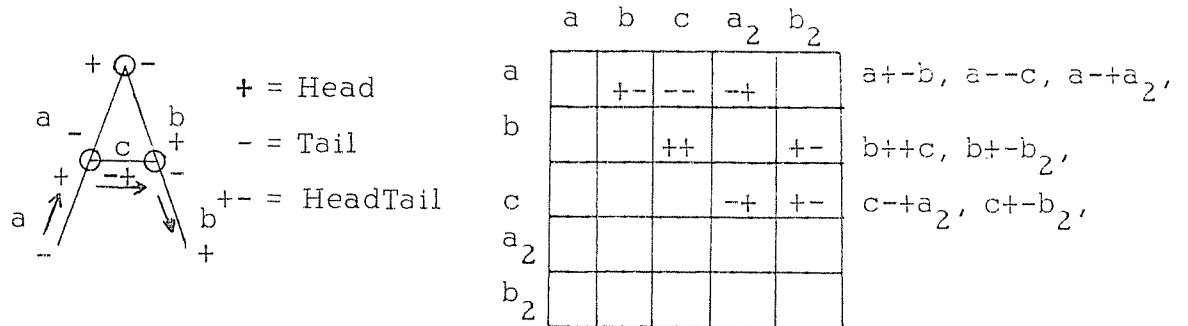
Alternately, (and, I think, more simply), we might use a 2-tuple to specify where on the row node and where on the column node the connection is made (Figure 4b). Thus Shaw's "+" becomes "+-", "x" becomes "--" and "*" becomes "--" and "++".

Grimsdale et al (1959) used such a connection matrix to describe the various figures their very sophisticated and complex program recognized. (Their program is of interest for the enormous amount of pre-processing it needed to reduce letters to lines, joins, and connected strokes. Figure 4 shows how a graph pattern can be represented by a connection matrix, and also by listing each thing-relation-thing.

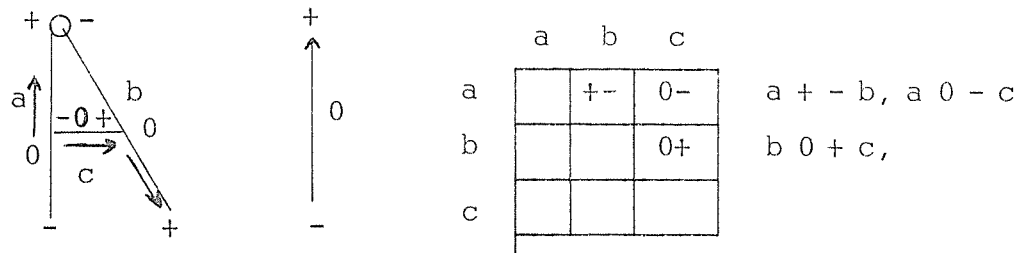
Figure 4. Describing Graphs



a) Strokes connected by '.' (signifying "connected")

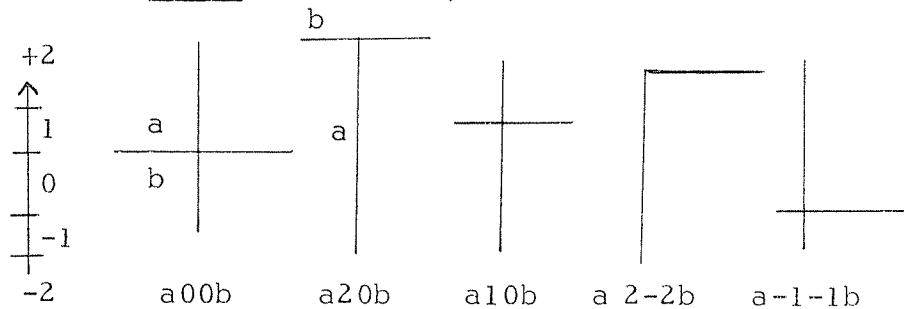


b) Strokes have Head (+) and Tail (-),



c) Strokes have Head(+), Tail(-) and Middle. Can Cross

Regions Pattern (Connection Matrix Description is Given Under Each Pattern)



d) Strokes have regions (+2, +1, 0, -1, -2)

The whole development of list processing languages is in large part the development of convenient representations for graphs.

Thus IPL's list structures and LISP's parenthesized notation form the basis for Evans' (1968) and Guzman's (1967, 1968) descriptions of objects.

FLEXIBLE TECHNIQUES FOR LINGUISTIC RECOGNITION

Flexible Connections

Why restrict connections to the two "ends" of an object?

Connections Within the String

Grimsdale et al (1959) specified connections at the tail, head, and middle (see Figure 4c). Uhr (1959) suggested connections at the ends, and in the regions near the ends and middle (see Figure 4d). Shaw must decompose every stroke into a set of strokes by cutting at every connecting point. E.g. a + becomes 4 strokes. But once we can specify the middle the + is described as 2 strokes, and their interrelations are clearly stated.

Uhr's regions can be either objective - of equal size - or subjective - where "end" is small, "near the end" is larger, and "middle" is largest, reflecting the way people perceive objects. Any number of regions can be specified. For more subtle descriptions of detail we can go one step further and give the program a set of procedures for computing relative positions.

Connections At a Distance

Up to now all connective have superimposed points of the two strings. But why not connect at a distance? This is quite commonly

done by "n-tuple" and "configurational" pattern recognizers, whose characterizers specify what to look for in specified disconnected regions of the input.

Without this (see Figure 5) a program cannot specify a pattern that is not a connected graph (one with a path between any two points). Thus = and i cannot be described by any of the grammars we have examined up to now. Nor can any other perfectly recognizable letter when one or two of its lines are broken or replaced by dashes.

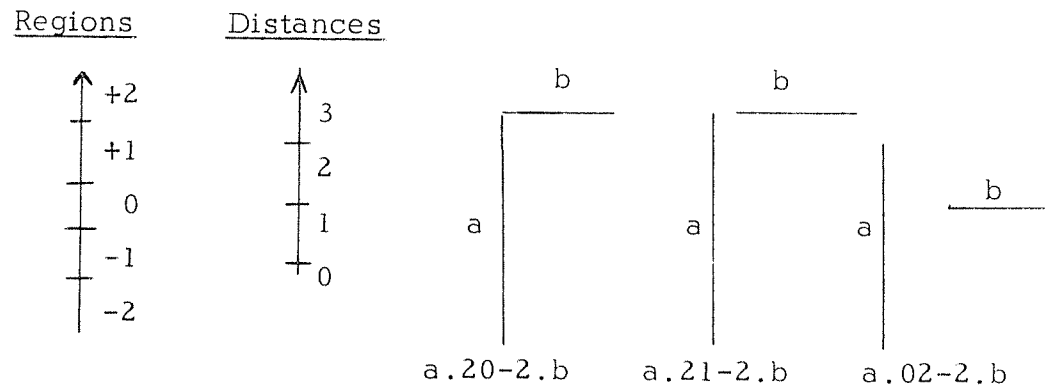
Uhr and Jordan (1969) give several techniques for connecting objects at a distance, and more are cataloged in Uhr (in press). First, let's note that a specification of a distance will be a vector of size N in N -dimensional space. Thus in a Chomsky grammar for strings in 1-dimension, or Ledley's grammar for strings in 2-dimensions, we need merely replace the single concatenation symbol "+" by a number to specify distance. The simplest distance is in terms of the units of the space - the symbols in a language string, or the cells in a picture matrix. Thus THE+DOG becomes THE.0.DOG and both will succeed in "SO+THE+DOG+EATS". But THE.3.DOG will succeed in "THE+BIG+DOG" and "THE+RED+DOG", and whenever THE and DOG are separated by exactly three letters. A variant subroutine can be easily coded to treat . N . as meaning " N or fewer letters to the right."

Rather than measure by counting letters, the program might count words, phrases, and/or other higher-level units. THE.W1.DOG will succeed in "THE+BARKING+DOG" (W1 means "1 Word distant") IF.PHRASE1.THEN will succeed in "IF+THE+RED+DOG+EATS+THEN..."

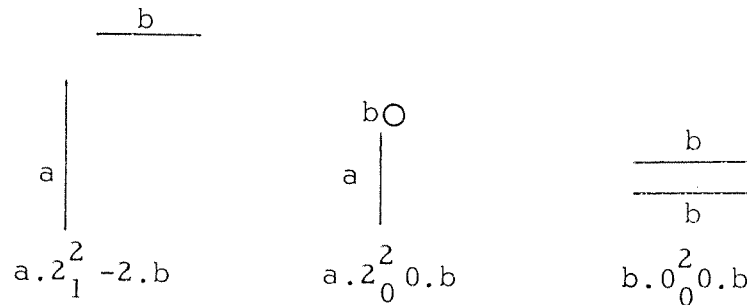
Figure 5. Connections at a Distance

a → b a b a b a b
a.0.b a.1.b a.2.b a.4.b

a) Objectively measured distances.



b) horizontal distances specified between joins (subjective distances shown)



c) x-y distances specified

Connections between features in 2 dimensions can be specified by 2 numbers, $\left[\begin{array}{l} y = \text{vertical distance up} \\ x = \text{horizontal distance to the right} \end{array} \right]$ (see Uhr, 1968, Sauvain and Uhr, 1969). Since the higher-level units are rarely cleanly given and extractable, as are the words and phrases in language sentences, and in any case empty background space can lie between them, distances in terms of array cells seem most appropriate. But it is also possible to specify subjective distances in the spirit of subjective joins. For example, a convenient set of subjective distances is SO = Touching, S1 = Close, S2 = Far, S3 = Distant (see Figure 5).

Subjective distances allow a single description to apply to a large number of variants - all combinations of elements of the description where each element can lie anywhere within the subjectively equivalent region. Uhr and Jordan (1969) give an alternate way of getting this kind of flexibility, by specifying bounds rather than distances. Eg. (giving bounds in objective distances), $T1 \cdot \begin{smallmatrix} 0,2 \\ 0,4 \end{smallmatrix} \cdot T2$ will succeed if T2 touches T1, and also in all cases where T2 is less than $\begin{smallmatrix} y=2 \\ x=4 \end{smallmatrix}$ away from T1. Both position on each part and distance between parts can (and usually should) be specified. (Fig. 5b)

Precise objective distances can be written in a subjective distance or a bounding system, which therefore include precise

distance systems. They can also effect enormous savings, since a single subjective distance or set of bounds defines an entire region of points, where a precise objective system would have to specify each of these points in a separate statement. Further, if the region contains an infinite set of points an infinite set of such statements would be needed.

Loosely Specified Connections

Rather than specify a distance, it often seems useful simply to specify co-occurrences, or directions. For example, we might write

$T1/T2$ to signify "T2 should co-occur anywhere with T1."

$T1 \rightarrow T2$ = "T2 should co-occur to the right of T1."

$T1 \downarrow T2$ = "T2 should co-occur to the right of and below T1."

General Relations

The joining and relative positioning of two things are just particular instances of the general problem of relating things together. We can use this same form, where $T1.RJ.TK$ specifies a relata, but the relation, RJ , can specify something like "above" or "encloses" or "is bigger than" or "kisses" or "likes". Now the relation must be defined in terms of usable procedures and tabled information.

Flexible Specification of Relations Among Things

Returning to relative positions (both within and between objects) let's examine one rather flexible way of allowing a program to notice any specified degree of precision it may find in a pattern, but to allow specifications to grow successively vague, until they fit.

Consider an interrelation among two thin strokes in two dimensions, eg:

$$\text{VERT2} \cdot \text{+-} \cdot \text{HOR1} = \text{GAMMA} = \begin{array}{|c} \hline \text{---} \\ | \end{array}$$

We can assume the strokes have some thickness (ie, are areas), and specify their join-positions in 2-dimensions (also assuming a range of regions 2, 1, 0, -1, -2), so that the above should be written:

$$\text{VERT2} \cdot \begin{array}{c} 2 \ 0 \\ 2 -2 \end{array} \cdot \text{HOR1} = \text{GAMMA} = \begin{array}{|c} \hline \text{---} \\ | \end{array}$$

Let's specify a third vector, for distance, which can range from 0 (= touch) through 1 (= close), 2 (= far), 3 (= distant). Now

$$\text{VERT2} \cdot \begin{array}{c} 200 \\ 222 \end{array} \cdot \text{HOR1} = \text{BROKENGAMMA} = \begin{array}{|c} \hline \text{---} \\ | \end{array}$$

Now let's substitute binary (or ternary) for the decimal numbers, where each successive digit is used to divide a line segment in half (or 3). Looking at relative position,

0 = CLOSE, TOUCH	00 = TOUCH
	01 = CLOSE
1 - DISTANT, FAR	11 = DISTANT
	10 = FAR

Now each successive digit makes relative position more precise . Each digit can be treated as another part of the characterizing description, specifying more precisely where to look , and adding a bit into the program's overall decision (of the sort to be discussed later) as to whether the characterizer has succeeded.

Describing and Compounding Areas

An area can be turned into its contour, which can be described as a concatenated string of curve segments (eg. Attneave and Arnoult, 1956; Ledley et al, 1965) But very little has been done to handle areas directly with syntactic techniques. Kirsch (1964) has developed a grammar for generating triangles. Dacey (1970) has generalized this grammar to handle any area that grows only at its bounds. (I suspect that this may be sufficient - in the real physical world objects either grow into "empty" space around them, or they don't grow.)

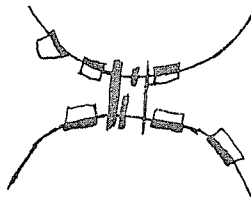
Patterns - even thinned strokes that result from pre-processing with a curve follower - always have area. Grimsdale et al (1959) did not really specify beginning-middle-end; actually they specified a 2-tuple, composed of Left, Middle, or Right and Top, Middle, or Bottom (except that Middle Middle never occurs with their decomposing scheme).

We generalized this in the previous section to the use of objective or subjective intervals in each dimension. A rectangle or some other hull might be drawn around a compound of strokes or some other irregular area. Now a connection is a triple, specifying the position on the first object, the distance, and the position on the second object.

But all these methods have difficulties: The scales and positions seem arbitrary. Hulls are hard to draw, and they are only approximations anyway. They seem too rigid and precise to be useful for a relation like "His head 'sits on' his shoulders" or "His head 'is tucked under' his arm."

The configurational characterizer (to be examined shortly), which rather loosely samples fragments of the areas involved, and is satisfied with only partial success, may well be the appropriate mechanism for compounding areas.

Figure 6. An example of a configuration Used to Join Patterns



For example, 4 or 5 of the pieces shown in Figure 6 would be enough to suggest a head above a shoulder. These could all be

described in a single n-tuple configuration, with a threshold high enough to insist that at least half of the pieces were found. Or they could be described by several configurations, so that, for example, a bottom contour of the face would have to be found along with a top contour of the shoulder. Or a more complex structure might insist that the parts of the configuration also belong to other configurations - so that the face parts belonged to a configuration that characterized the face, and the shoulder parts belonged to one that characterized the shoulder.

This, incidentally, is a way of handling joins (Shaw, 1969) and embeddings (Pflatz and Rosenfeld, 1969; Montanari, 1969) without any special mechanisms, by specifying a compound that contains one or more things from each of the compounds being joined. The compounding characterizer is a specification of the embedding: the program looks for the pieces in the pattern, and also checks that they were part of the configurations being compounded. When a well-articulated connection is desired the configuration will specify, eg. "Shoulder bone+Arm bone". When a looser relation is sufficient, or preferred, the specification can be of the sort shown in the head-neck-shoulder configuration of Figure 6.

LESS-THAN-PERFECTLY MATCHING REWRITE RULES

All syntactic techniques, from Chomsky through Ledley, Shaw, and Swain and Fu, insist that all parts of the rewrite rule be found before replacement can be effected. We have just examined techniques for relaxing the restriction that all parts must be concatenated, rather allowing parts to connect internally, at a distance, over some region, or within some bounds, or to cross. We will now examine mechanisms that relax the requirement that all parts must match.

A rewrite rule is of the form (reversing order so that parsing goes from left to right):

$$T1.R1.T2.R2 \dots RN.TN+1 = TNAME$$

A (linear representation of a) graph is of the form:

$$T1.R1.T2, T1.R2.T3, \dots T11.R1.T12, = TNAME$$

In both cases we traditionally insist that all Things be found as specified, over all Relations specified. That is, the rewrite rule specifies that all things be "and-ed" together.

As a program looks for the successive parts of a rewrite rule, it can count the parts found. If this count exceeds some specified THRESHOLD the program can decide the rewrite rule has succeeded - even though all parts were not found exactly as specified. A single

threshold can be stored for all rewrite rules, eg. $\text{THRESHOLD} = 60\%$ - meaning that at least 60% of the parts must be found for the rule to succeed. Or a threshold can be stored for each rewrite rule, expanding the right-hand side of the above statements to read:

$= \text{THRESHOLDN}, \text{TNAME}$

A threshold grammar includes grammars without thresholds, for they are simply the special case where $\text{THRESHOLD} = \text{Total Possible Count}$. (Note that a rule will succeed when any one of its "or-ed" parts succeeds if $\text{THRESHOLD} = 1$.)

Any threshold rule can be replaced by a set of rewrite rules: simply list all combinations of parts that will exceed the threshold. The set of threshold rules will never be larger than its equivalent set of rewrite rules. To the extent that a rule has many parts, and the threshold for acceptance is relatively low (and both of these conditions seem typical of the pattern recognition situation), a threshold grammar will effect significant savings, in both storage space for rewrite rules and processing time to apply them.

The count can be on any or all of the units in the rule: each Thing, each Relation, each Thing-Relation pair, and/or each Thing-Relation-Thing triple.

Rather than counts, the program could accumulate weights, if these were assigned to the parts of the rule, eg.:

$$T1*WtT1.R1*WtR1.T2*WtT2.R2.Wt*R2...RN*WtRN.TN+1*WtTN+1 =$$

THRESHOLDN, TNAME

$$T1.R1.T2*Wt1,T1.R2.T3*Wt2,...TI1.RI.TI2*WtI, = THRESHOLDN, \\ TNAME$$

Weights let each part of the characterizer play a role commensurate with its worth. They are especially important for learning (see eg. Uhr and Vossler, 1961; Nillson, 1965).

Note that a count rule is just the special case of a weight rule where all weights are 1.

If the Grimsdale program doesn't find a perfect match with one of the connection matrices that describe patterns, it then applies a number of complex procedures to get a "nearest" less-than-perfect match. This could be done quite simply with threshold rules, where each is actually a complete connection matrix for a pattern. The program would accumulate weights of the parts found for each characterizer. Each threshold would be the maximum possible weight. The program would then choose the name implied by the operator whose combined weights was closest to its threshold.

Prather and Uhr (1964) attempted to have their program learn configurations that were partial descriptions of the complete connection matrix, and combine the implications of these descriptions into probabilistic decisions of the sort we will examine in the next section.

Multiple Implications and Probabilistic Decisions

A phrase structure rewrite rule parses by replacing a set of things by a single thing. This is equivalent to a pattern recognition operator that replaces a characterization by a single implied name. Such a scheme either needs 1) some serial depth, where the implied name is itself a part of a subsequent operator's characterizer, (a heirarchical structure) or is the name of a subsequent operator (the sorting tree), or 2) a very large set of rather complete descriptions to characterize patterns, one such description for each variant pattern.

The typical pattern recognition program allows an operator to imply not just one, but rather a number of different names. Now a decision function is needed to choose among the names implied. (We must take care to distinguish the decision that an operator succeeds and the decision among the various names implied by a set of operators.)

This can easily be done with the rules we have written so far, by further expanding their right hand sides, to read:

$$\begin{aligned}
 &= \text{THRESHOLDN}, \text{TNAME1}, \text{TNAME2}, \text{TNAME3}, \dots \quad \text{or} \\
 &= \text{THRESHOLDN}, \text{TNAME1} * \text{WtTN1}, \text{TNAME2} * \text{WtTN2}, \text{TNAME3} * \\
 &\quad \text{WtTN3}, \dots \quad \text{or} \\
 &= \text{THRESHOLDTN1} * \text{TNAME1} * \text{WtN1}, \text{THRESHOLDTN2} * \text{TNAME2} * \\
 &\quad \text{WtN2}, \dots
 \end{aligned}$$

That is, each name can have equal weight, or its own special weight can be stored with it. One threshold can refer to all names, or a different threshold can be stored for each name.

Now a program also needs a decision function - that is, subroutines to combine the weights of the same name when it is implied by several different rules, and to choose among the several different names implied. Typically, the name whose combined weight is highest is chosen. Sometimes a program chooses and outputs the first name that exceeds some threshold for deciding. Sometimes the program insists that the chosen name be sufficiently higher than the next 1, or N, most highly weighted names and, if none meets that criterion, outputs "CAN'T DECIDE."

Rules that allow multiple (that is, one or more) implications include rules that allow only one implication as special cases.

A single multiple implications rule can be replaced by a whole set of traditional rules, each implying one of the implications. But standard phrase structure rules will not allow multiple implications. But standard phrase structure rules will not allow multiple implications, since they lead to "ambiguous" parses that cannot be handled without the decision function that makes a choice among probables. The restriction that only "grammatical" and "unambiguous" sentences should be allowed is not tenable for real world pattern recognition: Nature does not follow the rules of linguists.

DISCUSSION

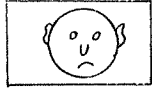
On the contrary, linguists, along with psychologists and computer modellers should be trying to discover the rules of nature. In the real world of language, just as in the real world of perceived patterns, people do a beautiful job of understanding broken, defective, and ungrammatical things. The pattern recognition structure of configurations of weighted related things multiply implying weighted things over thresholds gives greater power for pictures, and is quite likely to give greater power for language strings as well. By "power" I mean simplicity of the set of rules needed, which means efficiency in storage space and processing time, and size of the set of inputs that can be successfully recognized, described, or parsed. What

virtue is there to declaring a sentence "ungrammatical" and refusing to extract meaning from it that is evidently there? Should pattern recognizers throw out most letters because they have "ungrammatical" imperfections?

The Impossibility of Bounded, Complete, Unambiguous Covers

The traditional phrase structure grammar, and even the extended grammars, make several additional assumptions that are almost always violated by pattern recognition inputs.

The root "Sentence" node makes sense only when we can assume that an input is one meaningful, grammatical, connected unit, where every part serves some purpose, and should be accounted for.

"THE+BOY+RUNS" is such a unit. But a pattern like  is not. At the heart of the pattern recognition problem is the recognition of many variants as instances of the same pattern, including translated, noisy, broken variants, and variants in a noisy background. A sentence is tightly bounded; it has no background. A syntactic recognizer would throw out "XTHE BOY RUNS"

"TXHE BOY RUNS"

"THE X BOY RUNS"

"XYZZTHE BOY RUNSXXYZZX"

and similar strings as ungrammatical sentences. But these are only the simplest of the sorts of variations that we want a pattern recognizer to handle, by ignoring them.

The terminal "word" nodes make sense only when we assume that a word always looks exactly the same, and, further, is ended by a space. Thus a syntactic recognizer would throw out "THU BOY RUNS" because it would not find THU, and it would throw out "THE BOY RUNS" because it would not find THEBOY in its list of terminal words. But another central aspect of pattern recognition lies in the absence of primitive markers, in the mystery of the basic units. A pattern like an A is made up of some unknown set of features, each of which can merge into or even overlap the others - and we don't know what those features are, we don't know how to "spell" them. Worse, we have every reason to think that we must recognize features, as well as the pattern as a whole, over a wide variety of "misspellings" - that is, over a wide variety of variations.

Thus we must first have a set of pre-processors and feature extractors that are guaranteed to give an unambiguous linear connected covering of the pattern before we can apply a traditional grammar. That is, not only must the pattern be pre-processed into a

string, but it must also be decomposed into a unique and complete covering by the feature extractors. This is equivalent (in the far more complex 2-dimensional situation) to demanding that feature extractors regularize an input like

"THHE BXXXXXXOY RANS" or

"THE BOY RUNS" to

"THE BOY RUNS" so that the standard grammar can handle it.

But this merely pushes most of the problems to the earlier pre-processing stage⁵. For example, a standard production rule assumes concatenation; but BX^nOY is best recognized by a more typical pattern recognition n-tuple of the form $B \rightarrow 0 \rightarrow Y = BOY$ where the arrow (\rightarrow) signifies "anywhere to the right."

Phrase Structure Grammars and Pattern Recognition

Let's recapitulate the features of a phrase structure grammar, and the problems in applying them to pattern recognition:

A set of rewrite rules is used to parse an input, giving a single, complete and unambiguous covering if the input is "grammatical." But whereas in language the basic units - the words - are assumed to be unambiguous, clearly recognizable, and clearly bounded, in pattern recognition we don't know, and we have reason to doubt, that there are equivalent basic units. In any case we

don't know what they are, nor do we know how to extract them. Much of pre-processing and feature extraction serves that purpose.

The basic word units are concatenated in language strings. But as soon as we have 2-dimensional spaces where patterns move around and vary, we must extend our relations considerably. To the extent that patterns can be badly distorted and fragmented, yet still recognizable, we cannot insist upon perfection. To the extent that background exists (and without it there would be no foreground figure) we cannot expect a perfect, connected covering.

Thus 1) the inputs to the basic phrase structure rule may not be unambiguously given; 2) the rewrite rule must be extended to handle relations other than concatenation, and to succeed with only a fragmentary match; 3) it is unreasonable to require a perfect sequence of rewrite rules that make use of and account for all parts of the input, ending with a complete description of a "grammatical" instance of a pattern; and 4) the "parsed" input will not be completely covered and accounted for.

Characterizers and Rewrite Rules

The typical picture of a pattern recognition program is that a set of characterizers implies a set of names, among which one is chosen. The characterizers can be organized in parallel, series, or parallel-series, as indicated in Figure 1.

The serial organizations are similar to the organization of re-write rules in a grammar tree. But intermediate decisions can be made to choose the particular characterizers to apply next as well as what output name to vote for (eg. Uhr, 1969).

When we actually look in detail we find that the individual characterizer can have much of the structure of a rewrite rule. This is obvious for the compounds of strings and the graphs that we have just examined, where a characterizer combines a set of strokes. But many pattern recognition programs have combined other things, such as individual cells of the matrix (eg. Bledsoe and Browning, 1959). Uhr (in press) shows how the parts of a compound characterizer can be a mixture of any kinds of functions and tabled information. For example, a characterizer might specify that several strokes be present over a certain relation, that another stroke co-occur anywhere, and that the total area and total height meet certain specifications.

How Well Do Syntactic Recognizers Work?

Do linguistically motivated pattern recognizers give good recognition rates? I think the answer is, not very. Worse, they seem to be rather kludgy systems, hard to code and refine. When they work at all on real-world patterns they are only a part of a

larger system, which includes rather sophisticated pre-processing, feature extraction, and classification. There is no objective evidence to suggest that the syntactic portion of the recognizer has actually improved performance.

Narasimhan's very interesting work (1963, 1964, 1966) never resulted in an actual pattern recognition program. Grimsdale's (1959) program uses the connection matrix only incidentally, as a small part of a very large program. Much work has been purely theoretical, eg. Pflatz and Rosenfeld's (1969) development of Web grammars, and Montanari's (1969) extensions; and Swain and Fu's development of probabilistic programmed grammars (1970).

Eden (1961, 1968; Eden and Halle, 1961) suggested one of the most interesting syntactic schemes, for generating handwriting from a small alphabet of basic strokes.

Eden's students appear to have had a great deal of difficulty in using the generative technique that generates handwriting as the basis of a recognition program. Earnest (1963) had minimal success. Mermelstein (1964; Mermelstein and Eden, 1964) got better results, but only by introducing additional rather sophisticated techniques for identifying strokes in the first place, and for using contextual information about word constraints that limited the letter possibilities.

In contrast, more traditional pattern recognizers that have attempted to incorporate some structural features - in particular the compounding of several parts into a characterizer - have given good results (eg. Andrews et al, 1968; Rabinow, 1968; Uhr and Vossler, 1961, 1963; Zobrist, 1970).

What do Our Syntactic Languages Reveal About Structure?

Chomsky points out that the worth of a type of grammar lies in the way it reveals the structure of the language. But a grammar like Shaw's seems rather arbitrary. It works fairly well, but it reveals little. The "classes" are merely the different kinds of strokes (new lines and curves are defined as needed). What attaches to what in what order of embeddedness is sometimes dictated by structural difficulties in finding appropriate beginnings and ends for attaching, but usually it can go on in any order. The grammar is about as interesting as a grammar for a sentence that embedded THE BOY EATS RED MEAT into the form:

((((THE(BOY(EATS RED))))MEAT) or any other structure got by successively pairing words.

Why Syntax and Phrase Structure Rules?

If we really know that phrase structure grammars are inadequate for language, with all of its simplifications and abstractions

and regularizations, why should we expect it to work well for the far more difficult problem of perception of the real world? Chomsky and most structural linguists insist on the necessity of transformation grammars, to handle the underlying deep structure of a sentence. To quote Chomsky (Chomsky and Miller, 1963, p. 299) "As far as we know, the theory of transformational grammar is unique in holding out any hope that this end can be achieved" (handling a natural language like English).

Other linguists, notably Lamb (1969) and Fillmore (1968), are attempting to develop more semantically oriented systems. And several attempts have been made to apply pattern recognition and heuristic techniques, taking a semantic approach, to natural language processing (eg., Lindsay, 1963; Quillian, 1967; Raphael, 1964; Sikklossy, 1968; and Uhr, 1964).

Language and Objects

One can raise many doubts about the "linguistic" approach. But it has unquestionably served a crucial positive function, in reemphasizing the importance of structure. Most work in this field has focused on the decision among alternate possible classifications, as implied by the characteristics (features) of the input. Some work has concentrated on the getting of good features. Much

of that has tried to get structural features; but it is the linguistic school that has re-emphasized the importance of structure.

But we need a delicate balance between structure and flexibility. For an unknown amount of structure exists, and must be detected; but it is embedded in an unknown amount of non-structure. We cannot insist upon perfectly spelled and completely grammatical sentences in the real world. On the contrary, we want our recognizers to extract the same semantic meaning from widely varying and flawed fields of patterns.

After all, what do words like "syntax and "grammar" refer to if not the structure and pattern of language. We are fortunate to have so well developed a science of syntax, and we should try to apply it to sensory patterns. But we should apply it with discretion at the least, and with wisdom if we can.

It is just because of the one-dimensional character of language strings, and the man-made character of languages, that we have been able to make more progress in studying the syntax - the structure - of language. The sensory pattern recognition problem is far more difficult, and it would be foolhearty to try to apply easy solutions, without deepening them so that they contend with, rather than ignore, the difficulties.

The one-dimensional string eliminates most of the problems in connecting objects. In one dimension an object has two neighbors, to its left and its right. In two dimensions an object has, potentially, an infinite number of neighbors.

But that's not the only thing. People have made languages. People define and spell words, and legalize rules of grammar. A misspelled word or an ungrammatical sentence is thrown out of consideration by the linguist. This is, ultimately, foolhearty of him, and I hope there will be no tendency to do the same in pattern recognition. For we will always be embarrassed by the person (or pigeon) who recognizes the pattern perfectly well even though somebody has declared it ungrammatical.

The pattern recognizer is often described as having three stages: 1. a pre-processor, 2. a feature extractor, and 3. a classifier. But this is an oversimplification, and when one examines a program that actually tries to get good results on real-world patterns, one quickly finds a more complex structure, where the various steps merge together. When the classification is unsure, more features are examined; when a feature cannot be characterized nicely, more pre-processing is done.

In particular, it is extremely common to use a number of stages, in which the application of characterizers and the decisions are interspersed.

When a set of characterizers imply additional characterizers to apply, and this goes on for several layers, we get just the kind of heirarchical depth that we see in a phrase structure tree. When a characterizer is an n-tuple configuration, we get just the structure of a replacement statement. Actually, the structures are richer, and more powerful to the extent that thresholds, multiple implications, and a wider variety of positionings are allowed.

Can we not take advantage of linguistic methods, but without imposing upon the pattern recognition problem a stultifying rigidity that may be appropriate for the former, but seems antithetical to the latter?

The linguist exactly specifies his individual objects (terminal words in the vocabulary and internal class names of nodes in the syntax tree), and insists they appear exactly as specified. The pattern recognizer has no idea where his objects come from, or why, but he does know that they are a mess, that many unknown and complex non-linear transformations would be needed to group all members of the same pattern class - that is, all objects to which the same

name should be assigned - and that he does not know, and has no hope of knowing, these transformation. The pattern recognizer tries his best, or tries to get his program to learn as well as possible to do as well as possible. The difference is simply the difference between recognizing the string 'DOG' as punched 100 times unambiguously onto a punched card for input to a computer, and recognizing 100 different dogs.

SUMMARY

This paper has examined a number of aspects of "grammars" for pattern recognition, including several new mechanisms that I call "flexible:"

- A. Grammars for languages like English:
 - 1. Dimensional concatenation grammars (Chomsky's phrase structure grammars)
- B. Traditional grammars for pictures:
 - 2. N-dimensional concatenation grammars (Ledley)
 - 3. Head-tail grammars for graphs (Shaw)
 - 4. Graph grammars (Grimsdale, Pflatz and Rosenfeld)
- C. Flexible grammars for pictures:
 - 5. Head-middle-tail graph grammars (Grimsdale)
 - 6. Regions graph grammars
 - 7. Disconnected graphs grammars
 - 8. Flexible positions, distances, and relations grammars
 - 9. Threshold rewrite rules for describing, joining and embedding areas
 - 10. Threshold grammars for less-than-perfect match
 - 11. Multiple implication grammars with decision functions

Several additional variants have been examined, including the use of objective vs. subjective measures, and the use of counts vs.

weights. Combinations of some of the above can be effected, for example a disconnected graph grammar with thresholds and multiple implications.

BIBLIOGRAPHY

1. Andrews, D. R., Atrubin, A. J. and Hu, K. The IBM 1975 optical page reader: Part III: Recognition and logic development. IBM J. Research and Development, 1968, 12, 364-372.
2. Attneave, F. and Arnoult, M. D. The quantitative study of shape and pattern perception. Psychol. Bull., 1956, 53, 452-471.
3. Bledsoe, W. W. and Browning, I. Pattern recognition and reading by machine. Proc. East. Joint Comput. Conf., 1959, 16, 225-232.
4. Chomsky, N. Aspects of the Theory of Syntax. Cambridge: MIT Press, 1965.
5. Chomsky, N. Formal properties of grammars. In Handbook of Mathematical Psychology (R. D. Luce, R. R. Bush and E. Galanter, Eds.) New York: Wiley, 1963.
6. Chomsky, N. Syntactic Structures. The Hague: Mouton & Co., 1957.
7. Chomsky, N. and Miller, G. A. Introduction to the formal analysis of natural languages. In Handbook of Mathematical Psychology (R. D. Luce, R. R. Bush and E. Galanter, Eds.) New York: Wiley, 1963.
8. Dacy, M. F. The syntax of a triangle and some other figures. Pattern Recognition, 1970, 2, 11-31.
9. Earnest, L. D. Machine recognition of cursive writing. In: Information Processing, Proceedings of the IFIP Congress, 1962. (C. M. Poplewell, Ed.) Amsterdam: North Holland, 1963. 462-466.
10. Eden, M. On the formalization of handwriting. Proc. Symposia Applied Math. Amer. Math. Soc., 1961, 12, 83-88.
11. Eden, M. Handwriting generation and recognition. In M. Eden and P. A. Kolars (Eds.) Recognizing Patterns: Studies in Living and Automatic Systems. Cambridge: MIT Press, 1968. 138-154.
12. Eden, M. and Halle, M. The characterization of cursive writing. In Colin Cherry (Ed.), Fourth London Symposium on Information Theory. Washington, D. C.: Butterworth, 1961, 287-299.

13. Evans, T. G. A grammar-controlled pattern analyzer. Proc. IFIP Congress 68, H152-H157.
14. Evans, T. G. Grammatical inference techniques in pattern analysis. Paper presented at COINS Symposium, Florida, 1969.
15. Fillmore, C. J. The case for case. In: Universals in Linguistic Theory. (E. Bach and R. T. Harms, Eds.) New York: Holt, Rinehart and Winston, 1968. 1-88.
16. Fu, K. S. and Swain, P. H. On syntactic pattern recognition. Presented at Third COINS Symposium, Bal Harbour, Fla., December, 1969.
17. Grimsdale, R. L., Sumner, F. H., Tunis, C. J. and Kilburn, T. A system for the automatic recognition of patterns. Proc. Inst. Elect. Engineers, 1959, 106 (Pt. B), 210-221.
18. Guzman, A. Computer Recognition of Three-Dimensional Objects in a Visual Scene. Unpublished Ph.D. Dissertation, Project MAC (TR-59), Cambridge: MIT, 1968.
19. Guzman, A. Some aspects of pattern recognition by computer. TR-37, Project MAC, Cambridge: MIT, 1967.
20. Hunt, E. B. Concept Formation: An Information Processing Problem. New York: Wiley, 1962.
21. Kirsch, R. A. Computer interpretation of English text and picture patterns. IEEE Transactions on Electronic Computers, 1964, 13 363-376.
22. Lamb, S. M. Linguistic and cognitive networks. Paper presented at Symposium on Cognitive Studies and Artificial Intelligence Research, University of Chicago, 1969.
23. Ledley, R. S., et. al. FIDAC, In: J. Tippet et al (Eds.) Optical and Electro-Optical Information Processing Systems. Cambridge: MIT Press, 1965.
24. Ledley, R. S. and Ruddle, F. H. Chromosome analysis by Computer. Scientific American, 1965, 40-46.

25. Lindsay, R. K. Inferential memory as the basis of machines which understand natural language. In: Computers and Thought (E. A. Feigenbaum and J. Faldman, Eds.), New York: McGraw-Hill, 1963. 217-233.
26. Mermelstein, P. Computer Recognition of Connected Handwritten Words. Sc.D. Thesis, Dept. of Elect. Engin., MIT, 1964.
27. Mermelstein, P. and Eden, M. Experiments on computer recognition of connected handwritten words. Information and Control, 1964, 7, 255-270.
28. Miller, W. F. and Shaw, A. C. Linguistic methods in picture processing - a survey. Proc. Fall Joint Computer Conf., 1968, 33, 279-290.
29. Montanari, G. U. Separable graphs, planar graphs and web grammars. TR 69-96, Univ. of Maryland Computer Science Center, August, 1969.
30. Narasimhan, R. Labeling schemata and syntactic description of pictures. Information and Control, 1964, 7, 151-179.
31. Narasimhan, R. Syntactic descriptions of pictures and gestalt phenomena of visual perception. R 142, Digital Computer Lab., Univ. of Illinois, July, 1963.
32. Narasimhan, R. Syntax-directed interpretation of classes of pictures. Comm. ACM, 1966, 23, 166-173.
33. Nillson, J. J. Learning Machines, New York: McGraw-Hill, 1965.
34. Pfaltz, J. L. and Rosenfeld, A. Web grammars. Proc. Int. Joint Conf. on Artificial Intelligence, May 7-9, 1969, Washington, D. C., 609-619.
35. Prather, Rebecca, and Urh, L. Discovery and learning techniques for pattern recognition. Proc. 19th Annual Meeting of the Assoc. Comp. Machinery, August, 1964.
36. Quillian, M. R. Word concepts: A theory and simulations of some basic semantic capabilities. Behavioral Science, 1967, 12, 410-430.

37. Rabinow, J. The present state of the art of reading machines. In: L. Kanal (Ed.) Pattern Recognition. Washington: Thompson, 1968, 3-29.
38. Raphael, B. A computer program which understands. AFIPS Proc. FJCC, 1964, 26, 577-589.
39. Rosenblatt, F. The Perceptron: A Theory of Statistical Separability in Cognitive Systems. Buffalo: Cornell Aeronautical Labs Report VG-1196-G1, 1958.
40. Rosenfeld, A. Isotonic grammars, parallel grammars, and picture grammars. TR 70-111, Univ. of Maryland Computer Science Center, April, 1970.
41. Sauvain, R. W. and Uhr, L. A teachable pattern describing and recognizing program, Pattern Recognition, 1969, 1, 219-232.
42. Selfridge, O. G. Pandemonium: a paradigm for learning. In: Mechanization of Thought Processes. London: HMSO, 1959, 511-535.
43. Shaw, A. C. A formal picture description scheme as a basis for picture processing systems. Information and Control, 1969, 14, 9-52.
44. Siklossy, L. Natural Language Learning by Computer. Unpubl. Ph.D. Diss., Pittsburgh, Carnegie-Mellon Univ., 1968.
45. Swain, P. H. and Fu, K. S. Nonparametric and linguistic approaches to pattern recognition. TR-EE 70-20. Purdue Univ., 1970.
46. Towster, E. Studies in Concept Formation. Unpublished Ph.D. Dissertation, University of Wisconsin, Madison: 1969.
47. Uhr, L. Feature discovery and pattern description. In L. Kanal (Ed.) Pattern Recognition. Washington: Thompson, 1968, 159-181.
48. Uhr, L. Flexible Pattern Recognition. Computer Sciences Dept. Tech. Report, 56, Madison: University of Wisconsin, 1969.

49. Uhr, L. Machine perception of printed and hand-written forms by means of procedures for assessing and recognizing gestalts. Paper presented at 15th Annual Meeting of Assoc. Comp. Mach., Cambridge, 1959.
50. Uhr, L. Pattern Recognition, Learning and Thought. Englewood Cliffs: Prentice-Hall, in press.
51. Uhr, L. Pattern-string learning programs. Behavioral Science, 1964, 9, 258-
52. Uhr, L. and Jordan, Sara. The learning of parameters for generating compound characterizers for pattern recognition. Proc. Int. Joint Conf. on Artificial Intelligence, Wash., May 7-9, 1969, 381-415.
53. Uhr, L. and Vossler, C. A pattern recognition program that generates, evaluates and adjusts its own operators. Proc. West. Joint Comput. Conf., 1961, 19, 555-569.
54. Uhr, L. and Vossler, C. A pattern recognition program that generates, evaluates and adjusts its own operators. In: Computers and Thought (E. A. Feigenbaum and J. Feldman, Eds.), New York: McGraw-Hill, 1963, 251-268.
55. Unger, S. H. Pattern recognition and detection, Proc. IRE, 1959, 47, 1737-1752.
56. Zobrist, A. Feature Extraction and Representation for Pattern Recognition and the Game of GO. Ph.D. Diss., Computer Sciences Dept., Univ. of Wisconsin, 1970.