Syntax-Directed Recognition of Hand-Printed Two-Dimensional Mathematics

ROBERT H. ANDERSON[1]

Harvard University, Cambridge, Massachusetts

I. Introduction

Research in the real-time recognition of hand-printed characters [1–5] offers the possibility of drawing mathematical expressions on a RAND Tablet [6] or similar input device, and obtaining a list of the characters and their positions in an x-, y-coordinate system. This paper discusses the use of a set of replacement rules to recognize, or "parse," such two-dimensional configurations of characters. The replacement rules might be considered to be a generalization of the context-free Backus Normal Form rules used to describe a class of syntaxes for character strings.

Two syntaxes are mentioned in some detail: one for arithmetic expressions, and one for matrix descriptions.

The use of two-dimensional mathematics as computer input has several obvious advantages; it is more "natural," and therefore it is easier to learn, and it is more succinct (for example, the use of a summation sign with limits instead of a DO-loop). Klerer and May [7, 8] and Wells [9, 10] have described systems which use a modified Flexowriter for input of two-dimensional mathematical expressions. Their development of programming languages which use two-dimensional mathematics illustrates the clarity and brevity which are possible.

Martin [11] and Krakauer [12] at MIT have worked on the inverse problem: generating two-dimensional mathematical expressions as computer

[1] Present address: The RAND Corporation, Computer Sciences Department, Santa Monica, California.

output. Their program converts the tree structure representing a mathematical expression into a high-quality two-dimensional visual display of that expression, using a set of recursive rewrite rules to perform the transformation. There are some similarities between their work and the system discussed here. For example, both systems consider a mathematical expression as being enclosed in a rectangle with coordinates defining its size, location, and typographical center. However, fundamentally different problems are involved in the recognition and in the display of mathematical expressions. The display problem of converting a meaning into a pattern is basically one of transmitting information up and down within a tree structure until a harmony has been achieved among all nodes of the tree (i.e., until a set of mutually consistent spatial coordinates has been established for all displayed quantities). The recognition problem of converting a pattern into a meaning is one of building a tree structure from the information contained in an unstructured set of data. Rather than involving tree manipulations, the problem is one of determining the correct *type* of manipulations to be performed on a set of data so that these manipulations will result in a structure.

Narasimhan [13, 14], Shaw [15–17], Kirsch [18], and Ledley [19] have discussed two-dimensional syntactic analysis as a method of pattern interpretation. A survey of relevant literature is contained in Feder [20]. Most of this research has been concerned with languages whose terminal characters are line segments, and whose syntax describes possible connections between line segments. The syntax of mathematical expressions and matrices differs in several respects from the objects of the above efforts. In one respect, the problem has been simplified by the assumption that the character recognition problem is solved, and that each hand-drawn character has been recognized as an instance of one of the finite set of terminal symbols of the syntax. This assumption eliminates much of the syntactic processing required, for example, in analyzing bubble chamber photographs (cf. Narasimhan [14]). In another respect, however, the problem is more difficult; the characters in a hand-drawn mathematical expression are not connected to each other, and their positions and extent are not discretized into an array (except for the discretization arising from the number of significant digits to which their position is determined).

II. Top-Down Syntax-Directed Recognition

The recognition algorithm discussed here is *top-down*; that is, it starts with the ultimate syntactic goal and the entire set of input characters, and attempts to partition the problem into subgoals (and corresponding subsets of characters) until either every subgoal is reached, or else all possibilities

have failed. A syntax rule, then, will provide instructions for the partitioning[2] of a character set into subsets, and will assign a syntactic goal to each of these subsets.

A top-down parsing algorithm was chosen because it provides a natural way of hypothesizing the global properties of a configuration at an early stage of the recognition procedure. This feature is important because an operator symbol such as an integral sign affects the interpretation of the characters in its neighborhood ("dx" is a different syntactic unit in the configurations "$\int \sin x\, dx$" and "$cx + dx$"). I am sure that a bottom-up algorithm could be devised which would provide an equivalent recognition capability; it is not clear that it would be a more efficient recognizer.

A discussion of various parsing algorithms may be found in Griffiths and Petrick [21]. They use the terms "top-to-bottom" and "bottom-to-top" for the algorithms which are called "top-down" and "bottom-up" here.

The algorithm is also *syntax-directed*, meaning that all choices of subgoals and subsets are governed by syntax rules which are read as parametric data; the algorithm is independent of the content of these rules.

This recognition algorithm is considered to be a component of an on-line interactive computing system. In this environment, it would provide a very flexible recognition capability. In addition to the syntaxes for mathematical expressions and matrices mentioned in this paper, similar syntactic descriptions have been constructed and tested for the recognition of hand-drawn directed graphs and flowcharts. Since the syntax rules are used as parametric data, the user should be able to alter these rules and therefore modify and extend the input language to suit his individual requirements. Another advantage of syntax-directed recognition is that the syntax provides an explicit, concise definition of the interactive language.

A top-down syntax-directed recognition scheme has several disadvantages which must be considered. It is slower than a machine-language program tailored to a specific job, since the syntax rules must be continually interpreted, and also since many incorrect parsings might be attempted before the correct one (if any) is found. A second disadvantage concerns error detection; a purely top-down syntax-directed algorithm cannot pinpoint the error in an unsyntactic configuration; it can only reply, "It is not a valid configuration." Both of these disadvantages may be overcome to a large degree by modifications to the parsing algorithm or syntax rules. A discussion of the implementation of the algorithm and its efficiency will be presented at a later point in this paper. It is felt, however, that the disadvantages would be minimal in a highly interactive system in which the interpretation of each handwritten statement

[2] *Partition* is being used with the set-theoretic meaning of dividing a set into mutually exhaustive and mutually exclusive subsets.

would be shown to the user upon completion of its syntactic analysis. In this case, the number of characters being analyzed at any time would remain small, and any errors could be isolated and corrected as they appeared.

III. Characters

We assume that a character-recognition program provides the following five items of information about each character recognized: its *value* (in ASCII or some other code) and its extent, in the form *xmin*, *xmax*, *ymin*, *ymax*. The scale of the coordinate system used to determine this positional information may be arbitrary; all syntactic relationships are based on the positions of characters relative to each other.

For efficiency in the syntactic analysis for the particular syntaxes discussed here, characters will be preprocessed in the following manner.

By means of a table-lookup or similar procedure, each character will be given a *syntactic category* and two additional positional coordinates: *xcenter* and *ycenter*. These coordinates will reflect the typographical center of the character, as illustrated in Fig. 1. The xcenter for a character is always the

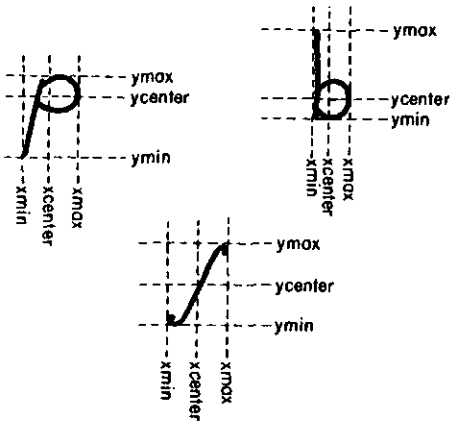


FIG. 1. Typographical center of characters.

average of xmin and xmax; ycenter is calculated from ymin and ymax by a function which depends on the particular character. These center coordinates could best be supplied by the character-recognition program, since relevant information about the internal configuration of each hand-written character is available to it.

The syntactic category of each character could be determined by using syntactic rules such as "$\langle$letter$\rangle \rightarrow a|b| \ldots | z$," but this would unduly

lengthen the syntactic processing. For the same reason, we also assume that the preprocessing will form the syntactic units "unsignedint" and "unsignedno" from hand-written strings of digits. If it were assumed that the digits comprising a number were always written consecutively left-to-right, the preprocessing program could be greatly simplified; it would need test only consecutive input characters for the proper values and relative position.

Table I gives the input characters needed for arithmetic and matrix syntaxes and the syntactic category which each of them is given by the preprocessing. It is assumed that each of these characters may be drawn in any size and at any location on the input surface. The terminal alphabet of the syntax, then, consists of the syntactic categories listed in the right-hand columns of Table I. In addition, the explicit letters *a*, *c*, *d*, *e*, *i*, *n*, *o*, *s*, *t* are needed for trigonometric and function names.

TABLE I

CHARACTERS AND THEIR SYNTACTIC CATEGORIES

Input character	Syntactic category	Input character	Syntactic category
a, *b*, ..., *z*	letter	=	=
integers	unsignedint	∞	∞
real numbers	unsignedno	(	(
horizontal line	horizline	)	)
vertical line	vertline	[	[
diagonal line with	diagline	]	]
negative slope		Σ	Σ
+	+	Π	Π
−	−	∫	∫
.	.	√‾	√‾
,	,		

IV. Syntactic Units

Just as each character's position is described by the six spatial coordinates xmin, xcenter, xmax, ymin, ycenter, and ymax, higher syntactic units composed of these characters also will be assigned coordinates. Although the number and content of the coordinates assigned to a syntactic unit may in general vary and be used to transmit complex information during the syntactic analysis (for example, by having a list structure as a value), for the syntaxes under discussion each syntactic unit will again have the six coordinates: xmin, xcenter, xmax, ymin, ycenter, ymax. Just as the ycenter of a character need not lie halfway between its ymin and ymax, the center of an

arithmetic expression, for syntactic purposes, need not be halfway between its extrema; the center of an expression is usually determined by the position of the principal operator in that expression. Figure 2 illustrates this point.

A syntax rule for two-dimensional analysis will specify replacements of syntactic categories contingent upon their correct relative placement, i.e., contingent upon an examination of their relative coordinates. The next section will describe how a syntax rule for 2D analysis might be generalized from ordinary BNF rules which operate on strings of characters.

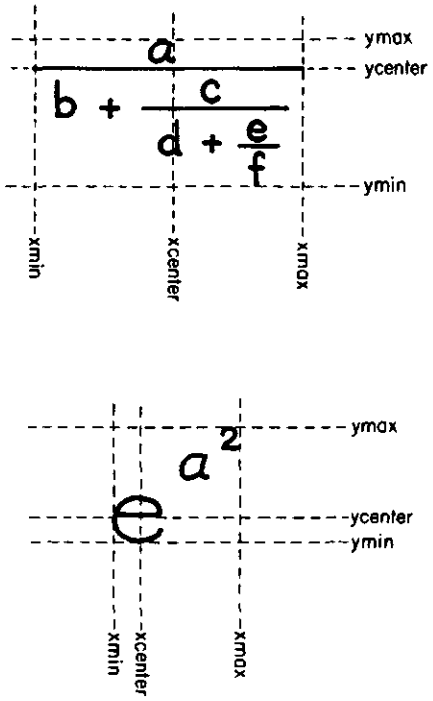


FIG. 2. Coordinates of syntactic units.

V. Syntax Rules for Two-Dimensional Character Configurations

Consider the following example of a context-free replacement rule for operating on a character string:

⟨expression⟩ → ⟨expression⟩ "+" ⟨term⟩

In English, this might be interpreted as:

Given the goal "expression" and a string of characters; try to partition the string into three substrings, where the first is an expression, and it is

followed by the second, which is a "+" sign, which is followed by the third, which is a term; if each of these subgoals is reached, report success; otherwise, report failure.

The analogous situation is more complicated in two dimensions. Consider the syntax rule we need to handle the following character configuration:

$$\frac{a^2 + b}{c}$$

In English, we might say:

Given the goal "arithmetic term" and a set of characters, where the position of each is described by a set of coordinates, try to partition the character set into three subsets $S1$, $S2$, and $S3$ such that the following conditions hold:

(1) $S1$ is an expression,
(2) $S2$ contains the single character "horizontal line,"
(3) $S3$ is an expression,
(4) $S1$ is above $S2$, and bounded in the x-direction by the extent of $S2$, and
(5) $S3$ is below $S2$, and bounded in the x-direction by the extent of $S2$.

If these tests are successful, assign a set of coordinates to the overall configuration, each of these being a function of the coordinates of $S1$, $S2$, and $S3$; report these coordinates along with "success"; if not successful, report failure.

Several important differences between the syntax rules for linear and two-dimensional character configurations should be noted:

1. the linear rule reports only "success" or "failure"; the 2D rule returns coordinate information in the event of success. These coordinates are necessary in the determination, at a higher level, of spatial relationships between syntactic units (just as conditions (4) and (5) in this rule tested relationships between "smaller" syntactic units).

2. in the linear case, the only relationship used between syntactic units is adjacency; this relationship is never explicitly tested, because this one-dimensional information is contained in the ordering of the input characters. In the 2D case, there are many possible relationships between syntactic units; e.g., one may be above, to the right of, or within the other. These relationships are part of the syntactic structure of the character configuration and must be tested explicitly.

In summary, a syntax rule for a 2D character configuration should contain the following information:

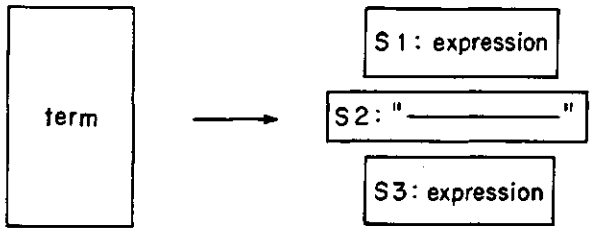
(1) the syntactic category on the "left-hand-side" of the rule,
(2) directions for partitioning a character set into subsets,
(3) syntactic goals associated with each of these subsets,

(4) relations to be tested among syntactic subunits, if they are successfully parsed, and
(5) the coordinate set of the left-hand-side, as a function of the coordinates of the right-hand-side subunits.

We will now develop a notation in which all of the above information may be encoded. As an example, consider again the replacement rule applicable to the term

$$\frac{a^2 + b}{c}$$

Figure 3a gives a graphical representation of the desired replacement rule. That is, we wish to partition the "term" space into three subspaces, $S1$, $S2$, $S3$, such that the second contains the single character "horizline." For this type of rule, in which a terminal character is one of the right-hand side components, the partitioning strategy will be as follows: if the set of characters under consideration does not contain the desired character (in this case, a horizontal line), then the rule does not apply; if there is exactly one instance of the desired character in the set, then the partitioning of the rest of the characters in the set will be based upon their position relative to that instance



(a)

term ⟶

S1:	expression	P1:	$c_{01} > c_{21}$ and $c_{03} < c_{23}$ and $c_{04} > c_{26}$	C1:	c_{21}
				C2:	c_{22}
S2:	horizline	P2:	$\emptyset$	C3:	c_{23}
S3:	expression	P3:	$c_{01} > c_{21}$ and $c_{03} < c_{23}$ and $c_{06} < c_{24}$	C4:	c_{34}
R:	$\emptyset$			C5:	c_{25}
M:	$(m_1)/(m_3)$			C6:	c_{16}

(b)

FIG. 3. Graphical and tabular form of replacement rule.

of the terminal character. In the example depicted by Fig. 3a, a character, c, is placed in subset $S1$ if and only if: *xmin*(c) > *xmin*(horizline) *and* *xmax*(c) < *xmax*(horizline) *and* *ymin*(c) > *ymax*(horizline). A similar predicate would test the character for placement in subset $S3$. If a character were found for which no predicate is true, the rule would be inapplicable to that character configuration.

If there are several instances of the desired terminal character in the set of characters under consideration, then these instances should be ordered and used successively as the basis for partitioning the other characters, until either an instance is found for which the partitioning is successful and all of the conditions in the syntax rule are satisfied by the resultant subsets, or else none is successful and the rule is therefore inapplicable.

It should be noted that the above partitioning strategy, used for rules with a terminal character on the right-hand side, places a restriction on the other (nonterminal) right-hand side syntactic units in the rule; it must be possible to delineate these units by mutually exclusive conditions which define each area by its spatial relationship to the terminal character(s) in the rule. This restriction considerably simplifies the partitioning algorithm, and does not seriously limit the descriptive power of the syntax.

The partitioning conditions and other relationships among syntactic units in the replacement rule illustrated graphically in Fig. 3a are presented in a tabular format in Fig. 3b.

The following notation is used in Fig. 3b:

Si: Each syntactic subcategory on the "right-hand side" of the rule is numbered $S1$, $S2$, etc. The digit in this label is used elsewhere in the rule to refer to that syntactic unit.

Pi: Corresponding to each Si is a boolean partitioning predicate Pi. Each syntactic subcategory is assumed to have the following six coordinates: (1) xmin, (2) xcenter, (3) xmax, (4) ymin, (5) ycenter, and (6) ymax. The subscripted variable "c_{ij}" refers to the jth coordinate of the syntactic subcategory labeled Si. For example, in Fig. 3b, "c_{21}" would be xmin of the terminal character "horizline." The notation "c_{0j}" in the partitioning conditions refers to the jth coordinate of an arbitrary character of the character set being partitioned. Each character of that set will be tested by the partitioning conditions, as described previously. The null symbol $\emptyset$ is used for partitioning predicates corresponding to terminal characters, since no predicate is needed for them.

R: The predicate labeled R tests the spatial relationships among successfully parsed subcategories. If this predicate is false, the rule is inapplicable to the character configuration. The null symbol in Fig. 3b means that no spatial relationship is to be tested; the

partitioning conditions are sufficient to insure the proper relative placement of the subcategories.

M: This item is used to perform an action or build a structure which reflects the successful application of the rule. When this syntax-directed recognition procedure is used as one part of an interactive mathematical system, the function M in the syntax rules would probably build a tree structure of the parse for subsequent use by a compiler or interpreter. However, in the samples of an arithmetic syntax given here, M uses string substitution to construct a string which gives the meaning derived by the parse in a combination of English and linear mathematics. The semantic string (i.e., the M value) derived from the parse of the syntactic unit labeled Si is substituted wherever the variable "m_i" appears in the string labelled M. The resulting string is the M value for the syntactic unit on the left-hand side of the rule. If Si is a terminal character, its M value is just that character itself (that is, a string of length 1).

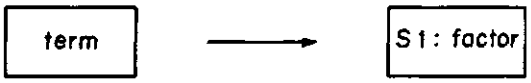
Ci: The six coordinates of the syntactic unit on the left-hand side of the replacement rule are specified in terms of the coordinates of the subunits on the right-hand side of the rule.

Using Fig. 3b as an example, the operation of the recognition algorithm as governed by this replacement rule would be as follows: given the category "term" and a corresponding set of characters, attempt to partition these characters into three subsets satisfying the conditions P; if such a partitioning is not possible, report "failure," otherwise attempt to parse each nonterminal subcategory Si, with its corresponding set of characters; if the parse of any subcategory fails, then if there is another possible partitioning (using another instance of the terminal character "horizline" as the basis), try it, otherwise report "failure"; if all subcategories are successfully parsed, calculate the six coordinate values $C1 \cdots C6$ and the semantic string M, and report "success" along with the six coordinates and semantic string for "term."

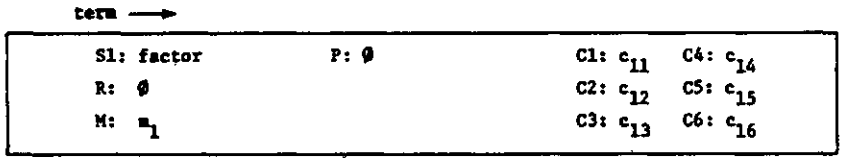
It should be noted that the recognition algorithm stops after the first successful parse. The syntax rules should be ordered so that the first interpretation encountered is the desired one. For example, they should test for the entity "sin" before adjacent letters are interpreted to mean implied multiplication of variables.

VI. Other Types of Replacement Rules

Up to this point, the only type of replacement rule which has been discussed is one which has a terminal character on its right-hand side. This character was used as a basis for the partitioning of the other characters.



(a)



(b)

FIG. 4. Replacement rule with right-hand side consisting of one nonterminal category.

A second type of replacement rule is one which has exactly one nonterminal category on its right-hand side. A rule of this type is used to change the syntactic category assigned to a character set. Figure 4 gives an example of such a rule. Since the character set is not partitioned, neither partitioning instructions nor relations between syntactic units are needed.

The only other possible type of replacement rule is one which has several subcategories on its right-hand side, none of which are terminal. For simplicity in the partitioning algorithm, rules of this type will be restricted to the following "normal form": only two subcategories are allowed, and it must be possible to draw a straight line on the plane in which the characters are drawn, such that the line partitions the characters into the two subsets corresponding to the two syntactic subcategories.

For this type of rule, the partitioning strategy is the following: let "$f(x, y) = \text{constant}$" be the equation of a straight line which would correctly partition the characters into the syntactic categories $S1$ and $S2$. The n characters are then ordered (using, for example, their center coordinates) by the function $f(x, y)$, producing the ordered set $(c_1 c_2 \cdots c_n)$; the $n - 1$ partitions

$$\begin{array}{ll} S1 = \{c_1 \cdots c_{n-1}\} & S2 = \{c_n\} \\ S1 = \{c_1 \cdots c_{n-2}\} & S2 = \{c_{n-1} c_n\} \\ \vdots & \vdots \\ S1 = \{c_1\} & S2 = \{c_2 \cdots c_n\} \end{array}$$

are then successively attempted until either one is found which meets all

(a)

term ⟶

S1:	term	P: f(x,y)=x	C1: c_{11}
S2:	factor		C2: avrg(c_{11},c_{23})
R:	$\lvert c_{15}-c_{25}\rvert$ < htol and ($c_{21}-c_{13}$) < hmax		C3: c_{23}
M:	$m_1 * m_2$		C4: min(c_{14},c_{24})
			C5: c_{25}
			C6: max(c_{16},c_{26})

(b)

FIG. 5. Replacement rule with right-hand side consisting of two nonterminal categories.

other criteria in the syntax rule (namely, the relation R) or else none are successful and the rule is therefore inapplicable.[3]

An example of this type of rule is given in Fig. 5. The rule in this example recognizes implied multiplication between two adjacent syntactic units. The partitioning condition P reflects the fact that the two syntactic units are separated by the line "x = constant." The relation R checks that the magnitude of the difference between the ycenter coordinates of the two units is less than the parameter htol and that the separation between the two syntactic units is less than the parameter hmax. By manipulating the values of these parameters, the system user should be able to "tune" the recognition algorithm to fit his printing, so that incorrect analyses are minimized. When not explicitly set by the user, parameters like htol and hmax would be given default values which are some function of the average size of the input characters.

[3] Although the stated partitioning strategy is adequate, in practice it is much more efficient to calculate the spatial extent of each character in the direction perpendicular to the line "$f(x, y)$ = constant." All partitions are then ignored between characters whose extents are overlapping.

FIG. 6. Several rules from a syntax for arithmetic expressions.

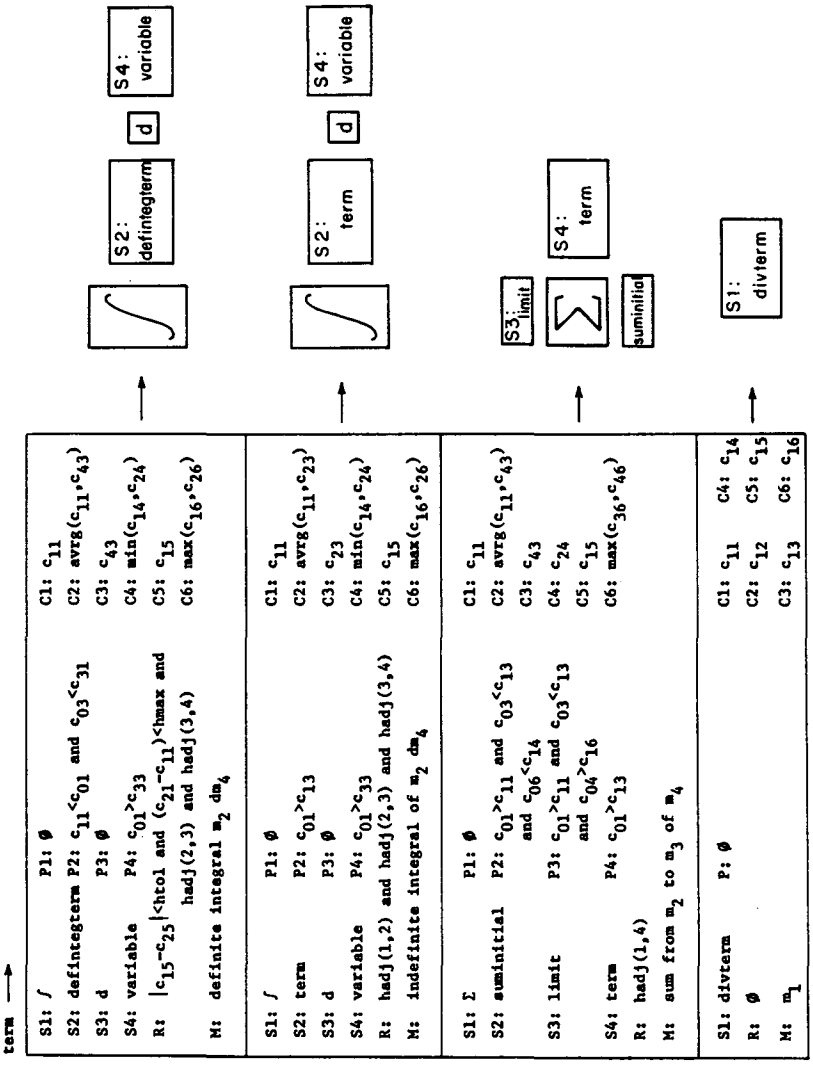


FIG. 6 (cont.).

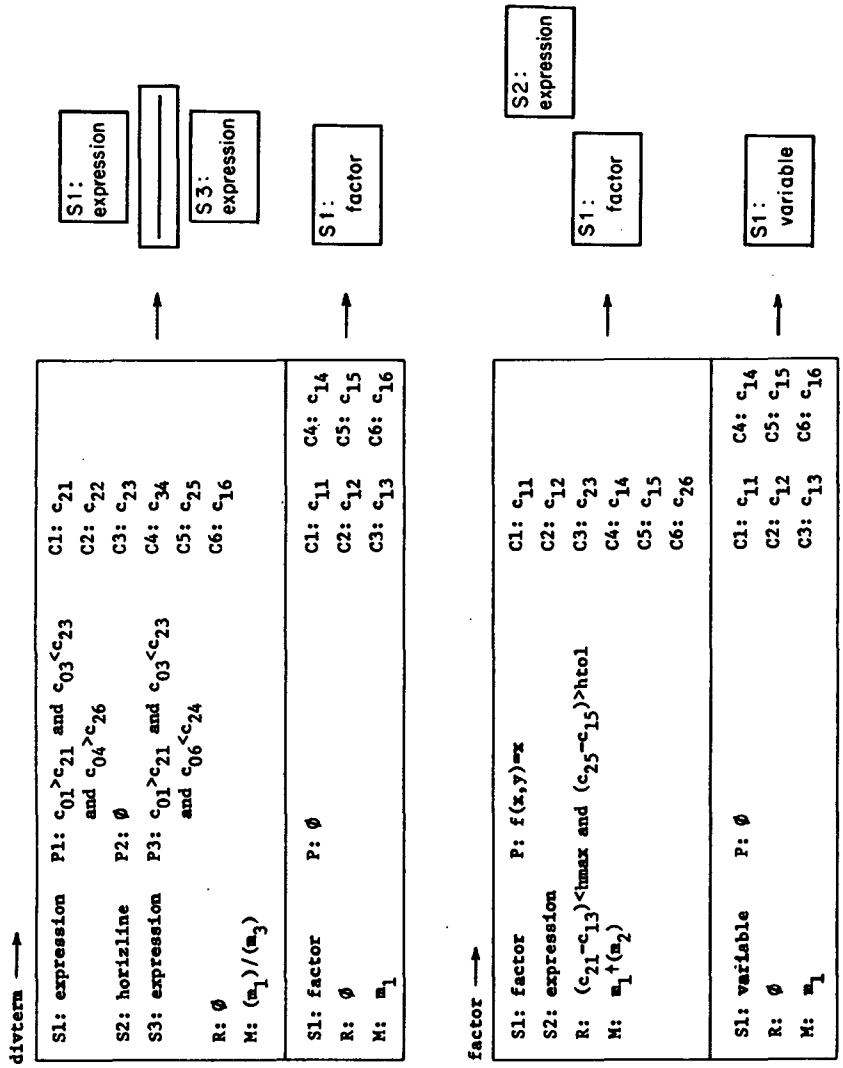

divterm ⟶
S1: expression
P1: $c_{01}>c_{21}$ and $c_{03}<c_{23}$ and $c_{04}>c_{26}$
C1: c_{21}
C2: c_{22}
C3: c_{23}
C4: c_{34}
C5: c_{25}
C6: c_{16}
S2: horizline
P2: ∅
S3: expression
P3: $c_{01}>c_{21}$ and $c_{03}<c_{23}$ and $c_{06}<c_{24}$
R: ∅
M: $(m_1)/(m_3)$
S1: expression
S3: expression
S1: factor
P: ∅
C1: c_{11}
C2: c_{12}
C3: c_{13}
C4: c_{14}
C5: c_{15}
C6: c_{16}
R: ∅
M: m_1
S1: factor
factor ⟶
S1: factor
P: f(x,y)=x
C1: c_{11}
C2: c_{12}
S2: expression
R: $(c_{21}-c_{13})<hmax$ and $(c_{25}-c_{15})>htol$
C3: c_{23}
C4: c_{14}
C5: c_{15}
C6: c_{26}
M: $m_1\uparrow(m_2)$
S2: expression
S1: factor
S1: variable
P: ∅
C1: c_{11}
C2: c_{12}
C3: c_{13}
C4: c_{14}
C5: c_{15}
C6: c_{16}
R: ∅
M: m_1

FIG. 6 (cont.).

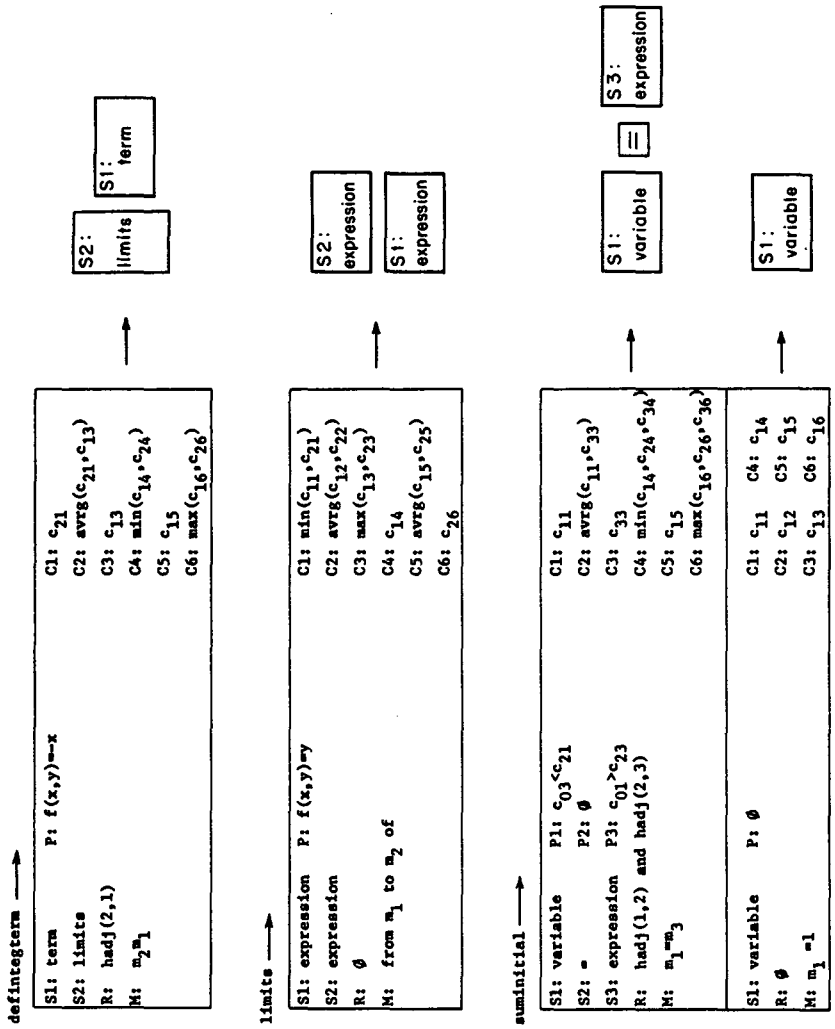
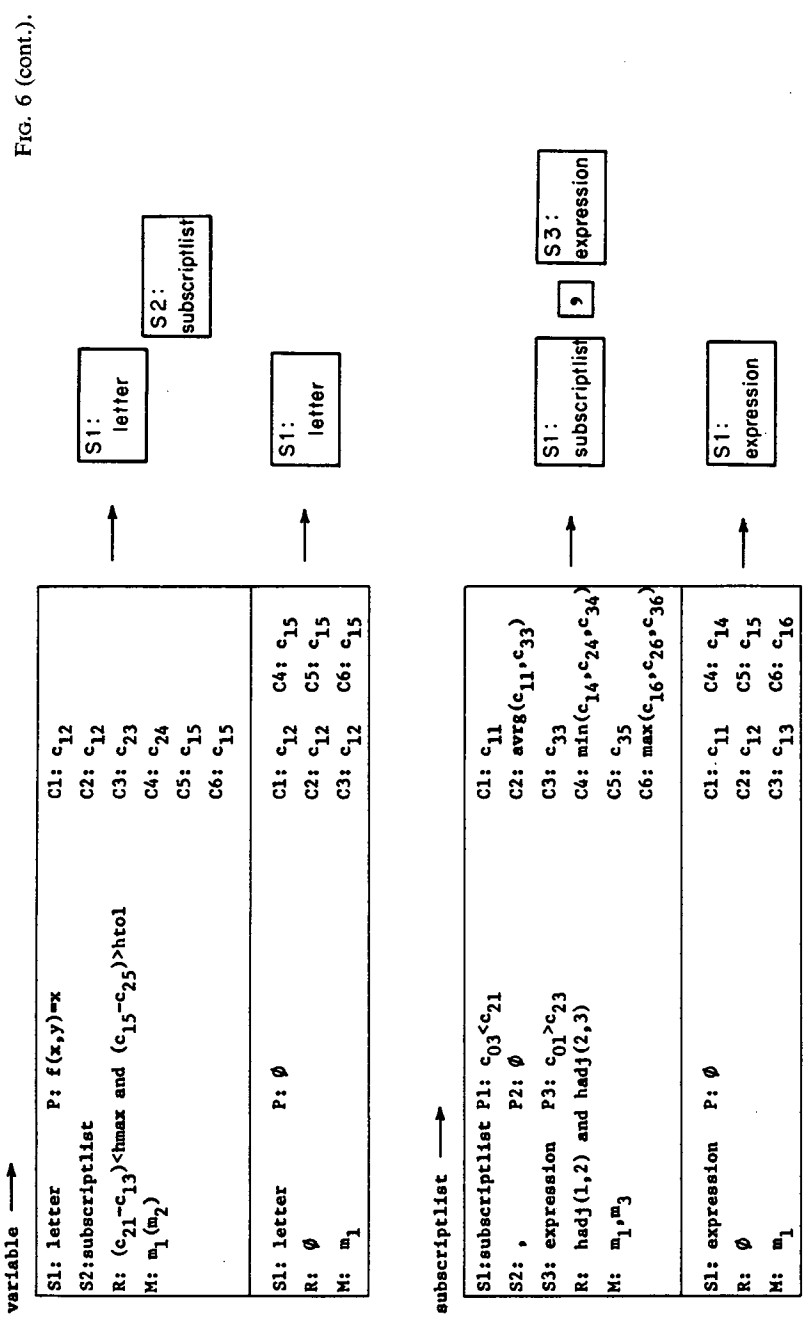

defintegterm
S1: term
S2: limits
R: hadj(2,1)
M: $m_2=m_1$
P: f(x,y)=x
C1: c_{21}
C2: avrg(c_{21},c_{13})
C3: c_{13}
C4: min(c_{14},c_{24})
C5: c_{15}
C6: max(c_{16},c_{26})
S2: limits
S1: term
limits
S1: expression
S2: expression
R: ∅
M: from m_1 to m_2 of
P: f(x,y)=y
C1: min(c_{11},c_{21})
C2: avrg(c_{12},c_{22})
C3: max(c_{13},c_{23})
C4: c_{14}
C5: avrg(c_{15},c_{25})
C6: c_{26}
S2: expression
S1: expression
suminitial
S1: variable
S2: =
S3: expression
R: hadj(1,2) and hadj(2,3)
M: $m_1=m_3$
P1: $c_{03}<c_{21}$
P2: ∅
P3: $c_{01}>c_{23}$
C1: c_{11}
C2: avrg(c_{11},c_{33})
C3: c_{33}
C4: min(c_{14},c_{24},c_{34})
C5: c_{15}
C6: max(c_{16},c_{26},c_{36})
S1: variable
=
S3: expression
S1: variable
R: ∅
M: $m_1=1$
P: ∅
C1: c_{11}
C2: c_{12}
C3: c_{13}
C4: c_{14}
C5: c_{15}
C6: c_{16}
S1: variable

FIG. 6 (cont.).


variable
S1: letter P: f(x,y)=x
S2:subscriptlist
R: (c21-c13)<hmax and (c15-c25)>htol
M: m1(m2)
C1: c12
C2: c12
C3: c23
C4: c24
C5: c15
C6: c15
S1: letter P: ∅
R: ∅
M: m1
C1: c12 C4: c15
C2: c12 C5: c15
C3: c12 C6: c15
S1: letter
S2: subscriptlist
S1: letter
subscriptlist
S1:subscriptlist P1: c03<c21
S2: , P2: ∅
S3: expression P3: c01>c23
R: hadj(1,2) and hadj(2,3)
M: m1,m3
C1: c11
C2: avrg(c11,c33)
C3: c33
C4: min(c14,c24,c34)
C5: c35
C6: max(c16,c26,c36)
S1: subscriptlist
,
S3: expression
S1: expression P: ∅
R: ∅
M: m1
C1: c11 C4: c14
C2: c12 C5: c15
C3: c13 C6: c16
S1: expression

VII. Scope of Recognition Capability

Figure 6 contains several rules from a syntax for recognizing two-dimensional arithmetic expressions. Each rule is presented in tabular form with an accompanying graphical "picture" as an aid for visualizing the relative spatial placement of the syntactic units.

The rules in Fig. 6 use the following global parameters:

htol Maximum vertical deviation for two syntactic units on the same typographical line

hmax Maximum horizontal separation between two adjacent syntactic units

The following boolean predicate is used:

$$\text{hadj}\ (i,j) \equiv 0 < (c_{j1} - c_{i3}) < \text{hmax} \quad \text{and} \quad |c_{i5} - c_{j5}| < \text{htol}$$

That is, the predicate "hadj(i, j)" is true for syntactic units *Si* and *Sj* if they are on the same typographical line, and *Si* is to the left of *Sj*, and they are sufficiently close together to be considered adjacent.

The following function "average" is used:

$$\text{avrg}\ (x, y) = 0.5 * (x + y)$$

In Fig. 6, it should be noted that the rules for the syntactic unit "variable" use only the center coordinates of "letter." This effectively contracts the spatial extent of each "letter" into a point. This is done so that a minor overlapping of adjacent handwritten characters will not cause a rejection of the configuration.

The design of the syntax occasionally relies on several features of the parsing algorithm.

1. Given a syntactic goal *G*, each rule with left-hand side *G* is tried *in the order given*, until either a "success" is reported, or else all rules have failed.
2. For rules with two nonterminal subcategories on their right-hand side the partitioning algorithm places the minimum possible number of characters in the *second* subcategory listed (i.e., in category *S*2).
3. For rules with a terminal character on their right-hand side, the predicates *P*1, *P*2, ... of the rule are evaluated *in order* for each input character in the set being parsed by the rule; a character is placed in the category corresponding to the first true predicate encountered.

The syntax could probably be designed in such a manner that it would be independent of the above features, but this would entail more rules and possibly additional syntactic categories.

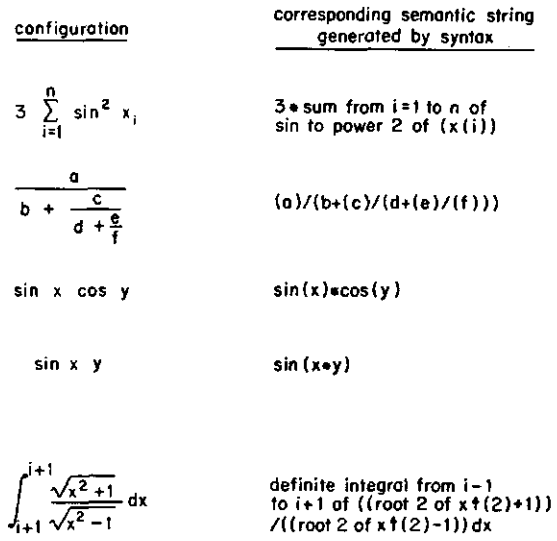


FIG. 7. Some recognizable mathematical expressions and their interpretation.

Figure 7 contains some hand-printed mathematical expressions which an implementation of a more complete syntax has been able to recognize (assuming reasonable values are assigned to parameters such as htol and hmax).

Two-dimensional mathematical notation contains some ambiguity. As one example, it must be decided whether the configuration

$$a_{ij}$$

means $a(i, j)$ or $a(i * j)$. Klerer and May have discussed the problem of ambiguity in two-dimensional mathematics at considerable length. Their system uses local context to resolve ambiguities. Using this philosophy, a_{ij} would be interpreted as $a(i, j)$ in the expression

$$\sum_{i,j=1}^{4} a_{ij}$$

or in similar circumstances where there is the equivalent of a double DO loop operating on it. However, this method of context-dependent interpretation is not foolproof; there is always the chance that the user means the given expression to be a way of writing

$$(a_1 + a_9 + a_{16}) + 2(a_2 + a_3 + a_6 + a_8 + a_{12}) + 3a_4$$

Context-dependent resolution of ambiguity could be introduced in the syntax-directed recognition scheme described in this paper by allowing the semantic

component of a rule, as one of its effects, to change other syntax rules. However, I would argue that the benefits of such manipulations probably would not justify the increased complexity. The alternative I propose is to keep the syntax of the interactive language constant, and design it to choose the most probable interpretation, such as $a_{ij} \equiv a(i,j)$. If a user wants an alternative interpretation for some specific application, he could alter the syntax accordingly. Such common alterations as the set of rule changes necessary for the interpretation $a_{ij} \equiv a(i * j)$ might be programmed as procedures which could be invoked by a simple declaration.

The same type of syntax rules used to define the syntax of arithmetic expressions may also be used to recognize many shorthand descriptions of matrices and vectors. Figure 8 shows some of the recognition capabilities of

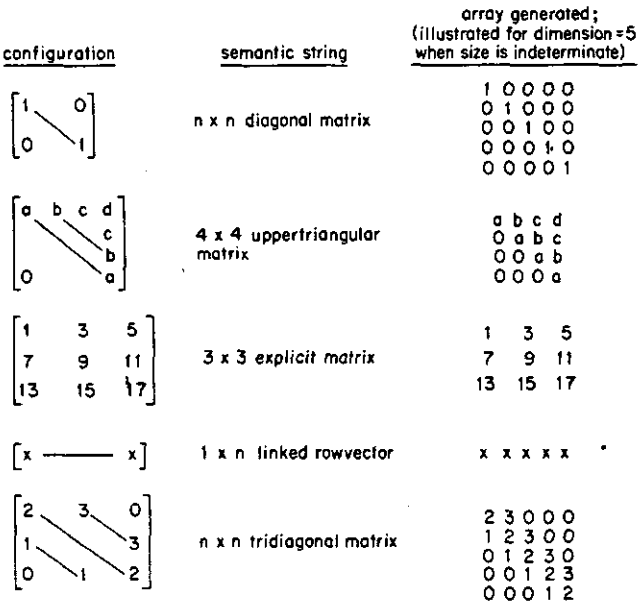


FIG. 8. Some recognizable matrix descriptions and their interpretation.

an implementation of a syntax for matrices. The implementation uses list processing rather than string manipulation for the semantic component (*M*:) of a syntax rule. In addition to a brief printed description of the recognized matrix, the semantic component creates an explicit array in the computer memory corresponding to the symbolic matrix description. If the size of the matrix is indeterminate, such an array will be created only if a dimension has been specified.

It should be clear that the information provided by the parsing of matrix descriptions (e.g., "uppertriangular," "diagonal," etc.) would allow an operational system to use specialized, more efficient storage and computational algorithms for certain classes of matrices.

VIII. Implementation and Efficiency

The recognition algorithm discussed in this paper was first implemented in LISP 1.5 on the CTSS time-sharing system of Project MAC at MIT. This implementation uses simulated graphic input: a typed list of characters and their coordinates. Experimentation with this program has shown that a "bare" top-down syntactic analysis is quite inefficient, especially in rejecting nonsyntactic character configurations. A major source of this inefficiency is the partitioning strategy used for rules with two nonterminal syntactic units on their right-hand side; up to $n - 1$ partitions may be generated by a set of n characters, and each of these partitions might require considerable processing.

However, by taking advantage of certain features of mathematical notation (e.g., its basically linear structure), and by the use of techniques employed in precedence analysis (cf. Wirth and Weber [22]) for programming languages, many "dead-end" analyses may be avoided. Two of the more important techniques used to gain efficiency are the following:

1. Although the order in which characters are given to the parsing algorithm is not important, it is presumed that a mathematical expression is written in a generally left-to-right manner. Also, for consistency, the rules in the syntax have been made left-recursive whenever possible. Therefore, if there are multiple instances of a terminal character which appears on the right-hand side of a syntactic rule, these instances are used in the *reverse* of the order in which they were received. This strategy permits expressions like

$$a + b + c + d + e,$$

if written from left to right, to be parsed by the left-recursive rule

⟨expression⟩ → ⟨expression⟩ "+" ⟨term⟩

without any backup or false analysis. Left recursion is also desirable for the partitioning strategy for rules with two nonterminal right-hand side categories, in which the minimum number of characters are placed in the second (rightmost) category.

2. Two sets are formed for each nonterminal category in the syntax: the set L of all possible characters which can occur leftmost in a valid character configuration for the category, and the set R of all possible rightmost charac-

ters. These sets may be calculated quickly from the syntax itself, and need be recalculated only when a change is made to the syntax. The majority of rules which have two nonterminal categories on their right-hand side in the syntax for mathematical expressions have these categories separated by the line "$x =$ constant." For these rules, the partitioning strategy entails ordering the character set by the x-coordinate. Using the L and R sets for the categories on a rule's right-hand side, it is easy to test possible partitions of the ordered character set for valid leftmost or rightmost characters. Invalid partitions may be discarded immediately. These tests are not employed for rules in which the two categories are partitioned by some line other than "$x =$ constant."

The LISP version of the parsing algorithm[4] was not written with processing speed as a primary criterion; however, with the above efficiency techniques it recognizes rather simple mathematical expressions, consisting of from about four to eight characters, in from two to five seconds (of CPU execution time on an IBM 7094). These numbers should not be taken too seriously, because the speed is highly dependent on the particular mathematical expression and the order in which the characters are written. However, it seems reasonable to assume from these results that a carefully coded machine-language algorithm, running on a computer at least as fast as a 7094, could bring the execution time down to about one second for reasonably small expressions (<10 characters).

The recognition algorithm is also implemented in PL/I on an IBM 360/40 computer with RAND Tablet and IBM 2250 display at The RAND Corporation, Santa Monica, California. Groner's [4] character recognition program is used to provide hand-drawn character input. As each character is drawn and recognized, it is displayed on a CRT in a canonical form in the same size and position in which it was drawn. When the desired mathematical expression has been drawn in this manner, the user signals that he has finished (by a button push or hand-drawn special character), and the parsing algorithm analyzes the configuration. Upon successful recognition, a box is drawn around the mathematical expression which has been recognized, and the meaning is displayed beneath the box. Figure 9 shows some examples of expressions which have been successfully recognized by this system.

The PL/I version of the recognition algorithm is again not optimized for speed. The mathematical expressions shown in Fig. 9 are recognized in about five seconds of execution time. This time includes the handling of CRT interrupts; much could be done to increase the efficiency of this implementation.

[4] The LISP functions are compiled; the syntax rules are read interpretively by the algorithm.

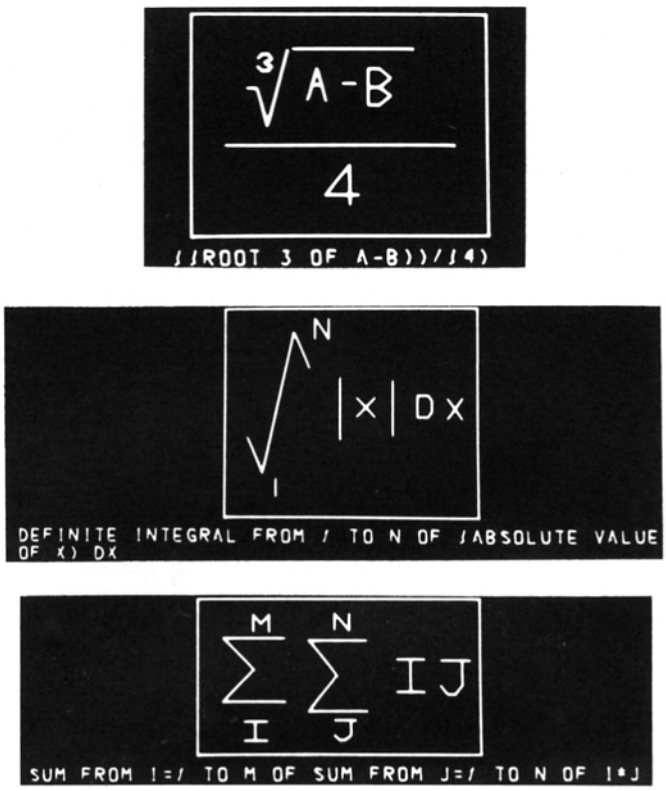


FIG. 9. Sample output from an implementation of the recognition algorithm.

Another possible source of efficiency would be parallel processing. A top-down parsing algorithm generates many subtasks, each of which must be evaluated independently, a situation which is ideally suited to parallel processing. An examination of traces of about a dozen typical analyses indicates that two processors sharing a common memory usually would perform a parse in about 60% of the time required for a single processor. For four parallel processors, this figure is approximately 45%. Use of more than four processors would not significantly further decrease the processing time.

IX. Summary

This paper has discussed a form of replacement rule and a parsing algorithm for handling two-dimensional character configurations. Two applications were illustrated: a syntax for recognizing arithmetic expressions, and

one for recognizing matrix descriptions. The recognition scheme is syntax-directed for flexibility; by modifying the replacement rules which define the interactive language, a user may incorporate additions or alterations to the language which were unforeseen by the system designer. The recognition algorithm is also flexible in that syntaxes have been developed for a variety of inputs, such as hand-written directed graphs and flowcharts, in addition to mathematical notation.

The problems of implementation and efficiency have been discussed briefly, and several methods given for considerably improving the efficiency of the parsing algorithm. Using these methods, a sufficient efficiency for interactive man–machine communication should be attainable.

Complete versions of all of the syntaxes mentioned in this paper may be found in Anderson [23]. The same reference also contains a formalization of the syntactic rules and parsing strategy which have been described.

ACKNOWLEDGMENTS

This work was supported in part by Project TACT at Harvard under Advanced Research Projects Agency (ARPA) Contract SD-265, by Project MAC at MIT under ARPA Contract Nonr-4102 (01), by The RAND Corporation under ARPA Contract DAHC 15 67 C 0141.

REFERENCES

1. BERNSTEIN, M. I., Computer Recognition of On-Line, Hand-Written Characters, RM-3753-ARPA, The RAND Corporation, Santa Monica, California, October 1964.
2. BERNSTEIN, M. I., An On-Line System for Utilizing Hand-Printed Input, TM-3052, System Development Corporation, Santa Monica, California, July 1966.
3. BROWN, R. M., On-Line Computer Recognition of Handprinted Characters, *Trans. IEEE, Electron. Comput.* **13**, 750–752 (1964).
4. GRONER, G. F., Real-Time Recognition of Handprinted Text, *Proc. Fall Joint Comput. Conf.* **29**, 591–601 (1966). Spartan Books, Washington, D.C., 1966.
5. TEITELMAN, W., Real-Time Recognition of Hand-Drawn Characters, *Proc. Fall Joint Comput. Conf.* **26**, Part 1, 559–575 (1964). Spartan Books, Washington, D.C., 1964.
6. DAVIS, M. R., and ELLIS, T. O., The RAND Tablet: A Man–Machine Graphical Communication Device, *Proc. Fall Joint Comput. Conf.* **26**, Part 1, 325–331 (1964). Spartan Books, Washington, D.C., 1964.
7. KLERER, M., and MAY, J., Two-Dimensional Programming, *Proc. Fall Joint Comput. Conf.* **27**, Part 1, 63–75 (1965). Spartan Books, Washington, D.C., 1965.
8. KLERER, M., and MAY, J., An Experiment in a User-Oriented Computer System, *Comm. ACM* **7**, 290–294 (1964).
9. WELLS, M. B., MADCAP: A Scientific Compiler for a Displayed Formula Textbook Language, *Comm. ACM* **4**, 31–36 (1961).
10. WELLS, M. B., Recent Improvements in MADCAP, *Comm. ACM* **6**, 674–678 (1963).

11. MARTIN, W. A., Symbolic Mathematical Laboratory, Chapt. 9, MAC-TR-36 (Thesis), Project MAC, MIT, Cambridge, Massachusetts, January 1967.
12. KRAKAUER, L. J., Syntax and Display of Printed Format Mathematical Formulas, Master's thesis, Elec. Engr. Dept., MIT, Cambridge, Massachusetts, 1964.
13. NARASIMHAN, R., Labeling Schemata and Syntactic Descriptions of Pictures, *Inf. Contr.* **7**, 151–179 (1964).
14. NARASIMHAN, R., Syntax-Directed Interpretation of Classes of Pictures, *Comm. ACM* **9**, 166–173 (1966).
15. SHAW, A. C., A Proposed Language for the Formal Description of Pictures, GSG Memo 28, Computer Group, Stanford Linear Accelerator Center, Stanford, California, February 1967.
16. MILLER, W. F., and SHAW, A. C., A Picture Calculus, GSG Memo 40, Computer Group, Stanford Linear Accelerator Center, Stanford, California, June 1967.
17. SHAW, A. C., A Picture Calculus—Further Definitions and Some Basic Theorems, GSG Memo 46, Computer Group, Stanford Linear Accelerator Center, Stanford, California, June 1967.
18. KIRSCH, R. A., Computer Interpretation of English Text and Picture Patterns, *Trans. IEEE, Electron. Comput.* **13**, 363–376 (1964).
19. LEDLEY, R. S., and WILSON, J. B., Concept Analysis by Syntax Processing, *Proc. Amer. Documentation Inst. Ann. Meeting*, **1**, 1–8 (1964).
20. FEDER, J., The Linguistic Approach to Pattern Analysis: A Literature Survey, Tech. Rept. 400–133, School of Engineering and Science, Dept. of Elec. Engr., New York Univ., University Heights, New York, February 1966.
21. GRIFFITHS, T. V., and PETRICK, S. R., On the Relative Efficiencies of Context-Free Grammar Recognizers, *Comm. ACM* **8**, 289–300 (1965).
22. WIRTH, N., and WEBER, H., EULER: A Generalization of ALGOL, and Its Formal Definition, Part I, *Comm. ACM* **9**, 13–25 (1966).
23. ANDERSON, R. H., Syntax-Directed Recognition of Hand-Printed Two-Dimensional Mathematics, Ph.D. dissertation, Div. of Engr. and Appl. Phys., Harvard Univ., Cambridge, Massachusetts, January 1968.