MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Cambridge, Massachusetts
Project MAC

Artificial Intelligence Project
Memo No. 145.

Memorandum MAC - M -360
October 19, 1967

A Fast Parsing Scheme for Hand-Printed Mathematical Expressions

by William A. Martin

ABSTRACT

A set of one-line text-book-style mathematical expressions is defined by a context free grammar. This grammar generates strings which describe the expressions in terms of mathematical symbols and some simple positional operators, such as vertical concatenation. The grammar rules are processed to abstract information used to drive the parsing scheme. This has been called syntax-controlled as opposed to syntax-directed analysis.

The parsing scheme consists of two operations. First, the X-Y plane is searched in such a way that the mathematical characters are picked up in a unique order. Then, the resulting character string is parsed using a precedence algorithm with certain modifications for special cases. The search of the X-Y plane is directed by the particular characters encountered.

I. Introduction

No satisfactory method of typing mathematical expressions in a linear string has as yet been devised. Chapter II of my thesis [13] shows the difficult notation I had to use. A good method for communicating two-dimensional expressions to the computer is needed. Klerer[10] has devised an algorithm for parsing two-dimensional expressions constructed on a slightly modified typewriter, but these expressions are not easy to type. One is therefore led to hope that the input of characters through a stylus device like the RAND tablet will some day be practical. Existing character recognition programs[9] are good enough to begin experiments. Anderson[1] has just constructed an algorithm for parsing mathematical expressions drawn on a RAND tablet. However, while Klerer's algorithm is quite fast, Anderson's takes many seconds to parse an expression of moderate size. There are several reasons for this; partly it is a matter of implementation. But a very important reason is that Anderson's algorithm is very general and has more power than is needed for most of the expressions we expect to encounter. For example, consider how Anderson's top down syntax directed algorithm would parse:

(1) $$x=\sum_{I=0}^{0} x^{2+I}$$

The program is given an ordered list of syntax rules. Those needed for this example would be:

1. S → E = E
2. E → T + T
3. E → T
4. T → 2

5. $T \rightarrow X^{E}$

6. $T \rightarrow X$

7. $T \rightarrow \sum_{S}^{E} T$

8. $T \rightarrow 0$

9. $T \rightarrow I$.

The spatial relationships in the exponent and summation are made explicit by giving X and Y coordinates. The parsing program must accept any expression which can be generated by starting with S and substituting the right side of any rule for its left side in the expression being formed. The parsing program must determine the sequence of rules used to form the input expression. Examining the rules in the order given, it would first try to apply rule 1. by partitioning the remaining characters on either side of an =. Choosing the rightmost = the program forms two possibilities:

$$x = _{I} \quad \text{and} \quad \sum_{0}^{0} x^{2+I}$$

which must be examples of expressions which can be formed by starting with E. Taking the right side first, the + indicates that it might be of the form:

$$\sum_{0}^{0} x^{2} \quad \text{and} \quad I$$

Proceeding in this manner, the original choice of = is found to be unacceptable. The parser then tries the second = and forms:

$$x \quad \text{and} \quad \sum_{I=0}^{0} x^{2+I}$$

which will prove to be correct. When an application of rule 7 is tried, the characters will be partitioned into groups depending on whether they are above, below, or completely to the right of the $\sum$.

Now let us see how Klerer's algorithm would handle this same example. The expression is considered to be a tree branching from left to right. The program will pick up the leftmost character, the X. Since this is a letter it will know from rule 5 that it might have an exponent, but a scan of the appropriate area shows none to be present. It next picks up the =, forming a string of the characters found. Here the program knows that only rule 1 can apply so it moves right again, adding the $\sum$ to the string. From rule 7 the program then knows to search in a similar manner, first below, then above, and then to the right of the $\sum$. It puts marking characters between the characters found in each area. In this manner the Klerer program forms the character string:

$$x = \sum (I = 0,0, x \nearrow (2+I))$$

without any false characters being picked up. This string is then parsed by an efficient method for linear strings. The Klerer method is superior on this example, but it can fail on: $\int_a^{b - \frac{x}{\frac{c}{d}}} dx$

Anderson will recognize this as

$$\int_a^b \frac{-x}{\frac{c}{d}}\, dx$$

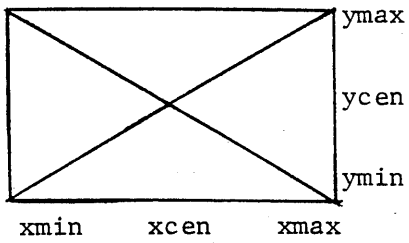
I think Anderson was led to his approach by a) the desire to handle a wide variety of notations, b) the belief that the characters would not be constrained in size and position as they are on the typewriter and c) the desire for a theoretically attractive syntax directed scheme. It

is certainly true that what little documentation Klerer has done makes his scheme seem very ad hoc.

In this memo I present my version of Klerer's scheme in a systematic manner, which shows its power and limitations. This makes more apparent where the power of Anderson's scheme is needed. Using as input a list of characters and the coordinates of each as shown in Fig. 1, the program appears to be about 20 times as fast as Anderson's on the examples shown in Fig. 2. Anderson's program will slow down more than linearly as the number of characters in the examples is increased, but the Klerer algorithm will not. On the other hand, Anderson's program contains many tests for correctly formed syntactic sub-units. Only after my program has been tested on a RAND tablet can a complete comparison be made between Anderson's scheme and mine.

```
(    (0   4   8   20   26   32   X)
     (12  16  20  33   39   45   2)
     (24  28  30  20   26   32   PLUSS)
     (36  40  45  26   26   26   QUOTIENT)
     (37  40  44  28   34   40   A)
     (37  40  44  12   18   24   B)
     (48  52  56  20   26   32   PLUSS)
     (60  64  68  20   26   32   X)        )
```

$\equiv \quad x^2 + \frac{A}{B} + x$



≡ (xmin xcen xmax ymin ycen ymax x)

Fig. 1

A Hand Coded Example

Example	Seconds to Parse, Compiled CTSS LISP
$x + 3 * I$	.2
$x^2 + \frac{A}{B} + x$	.3
$x! + \sum_{I=2}^{10} (x)^I$	.3
$\lvert x \rvert + y \rvert_{y=2}$	.2
$\frac{d^2}{dx^2} (x+y)$	.4
$f_i(x,y)$	.2
$\int_a^b x dx$	.2
$x - \text{SIN } x = 0$	.2

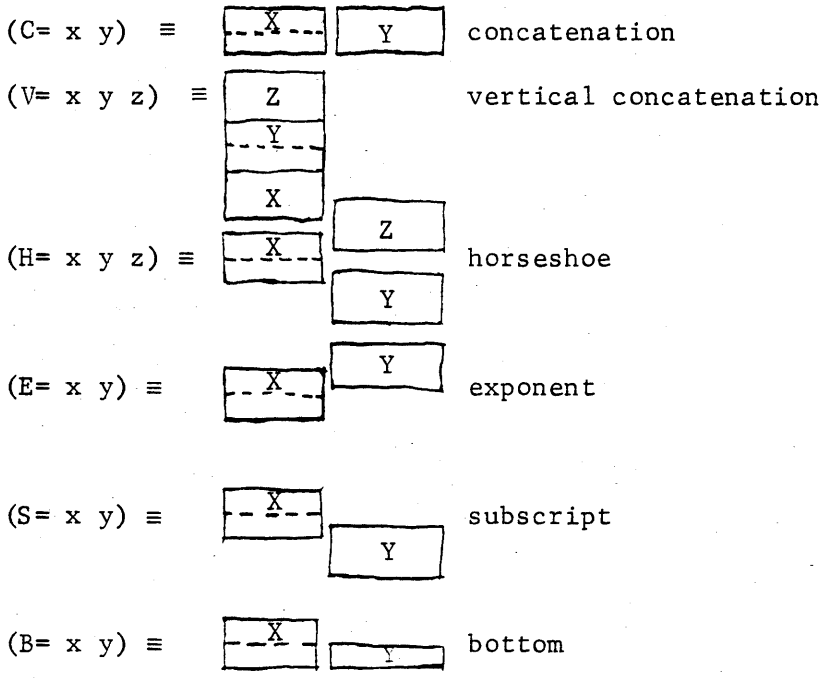
Fig. 2

Some Examples Parsed

II. The Grammar

The expressions to be parsed can be described by a context free grammar.[7] Such a grammar consists of a set of terminal characters, a set of non-terminal characters and a set of productions of the form $A \rightarrow \theta$, where A is a non-terminal character, and θ is a finite string of terminal and non-terminal characters. Starting with a specified non-terminal, S*, the mathematical expressions are described by the strings of terminals which can be formed by the repeated substitution of the right side of any production for its left side, until no non-terminals remain in the string.

Since we are describing two-dimensional mathematical expressions the terminal symbols consist of mathematical symbols, parentheses, and the following positional operators:



The center rectangle of V= is assumed to extend a character width to the left of the others; the reason for this will be explained later. C= can take an arbitrary number of arguments.

If one thinks of each mathematical symbol as being enclosed in a rectangle, then the positional operators show how to combine the rectangles to form larger ones. For example, the string of terminal symbols describing $\sum_{I=1}^{\infty} x^I$ would be:

(C= (V= (C= I = 1) $\sum$ ∞)(E= X I)).

We will call this positional operator notation. The positional operators defined here will form a left to right tree structure of symbols.

Before we can present our grammar two more complications must be introduced. First, we associate with each production a production in a parenthesis grammar which generates the internal LISP representation of the mathematical expressions. The use of production pairs is a simple example of the scheme proposed by Donovan.[3] For example, a simple grammar might be:

1.	S* → E*	S* → E*
2.	E* → (PWR H* E*)	E* → (E= H* E*)
3.	E* → X	E* → X
4.	H* → X	H* → X
5.	H* → E*	H* → (C= LPAR E* RPAR)

where PWR stands for exponentiation and LPAR and RPAR stand for left and right parenthesis. The generation of the strings to describe $(x^x)^x$ in the two languages would then proceed as follows:

LISP notation	Positional Operator Notation	
S*	S*	Rule
E*	E*	1
(PWR H* E*)	(E= H* E*)	2

LISP notation cont.	Positional Operator Notation cont.	Rule
(PWR H* E*)	(E= (C= LPAR E* RPAR)E*)	5
(PWR (PWR H* E*) E*)	(E= (C= LPAR(E= H* E*)RPAR)E*)	2
(PWR(PWR X E*)E*)	(E= (C= LPAR(E= X E*)RPAR)E*)	4
(PWR(PWR X X)E*)	(E= (C= LPAR(E= X X)RPAR)E*)	3
(PWR(PWR X X)X)	(E= (C= LPAR(E= X X)RPAR)X)	3

Whenever we substitute for a non-terminal in the positional operator string, we make the corresponding substitution in the LISP string. Our grammar now generates ordered pairs of strings. The first string represents the mathematical expression in our LISP notation and the second represents the expression in positional operator notation. Once the parsing program has found the series of productions which generate the positional operator notation for an input expression, the same series can then be used to generate its LISP representation.

The same non-terminal symbol might occur twice on the right side of a production. For example we might have:

E* → (PLUS E* E*) E* →(C= E* + E*).

In order to distinguish between these two instances when applying the same substitutions to both sides, they will be subscripted. This is just a notational convenience. The subscripted form of the rule above is:

E* → (PLUS (E* 1)(E* 2)) E* → (C= (E* 1) + (E* 2)).

Finally, the grammar can be further condensed without any change in its power if alternative choices and repeated substrings are introduced.[5] The

repeated strings are useful since functions such as PLUS and TIMES can take an arbitrary number of arguments. The expression x+x+x could be generated by the rules:

S* → (PLUS X S*) S* → (C= X + S*)

S* → X S* → X

But this leads to the unsimplified parsing (PLUS X (PLUS X X)). Instead we will write:

S* → (PLUS X (REPEAT 1 X)) S* → (C= X (REPEAT 1 (+ X)))

The 1 is just used to subscript the REPEAT expression itself for identification on both sides. Parentheses around the argument of the REPEAT on the right side indicate that it is a string of characters. Finally, we introduce OR, so that the rule which may be used to generate X + X - X is:

S* → (PLUS X (REPEAT 1 (OR X (MINUS X)))

S* → (C = X (REPEAT 1 (OR (+ X) (- X))))

The OR lets us represent all strings of terms connected by + or - signs with just one rule. The OR may have any number of arguments but the corresponding argument of a given OR must be taken on both sides.

The mathematical symbols and operators have been given the names shown in Fig. 3. The LISP notation is explained in Fig. 4. A grammar for the mathematical expressions used in my thesis is shown in Fig. 5.

$\sum$	SIGMA
=	EQSIGN
+	PLUSS
−	DASH
*	STAR
———	QUOTIENT
|	BAR
,	COMMA
!	EXCLAMATION
(	LPAR
)	RPAR
$\int$	INTEGRAL

Fig. 3

Symbol Names

(PLS A B C) $\equiv A + B + C$

(PRD A B C) $\equiv A \cdot B \cdot C$

(PWR A B) $\equiv A^B$

(DRV A B C D E) $\equiv \dfrac{d^{B+D}}{dA^B\, dC^D} E$

(ITG D A B C) $\equiv \int_A^B C\,dD$

(SUM A B C D) $\equiv \sum_{A=B}^{C} D$

(EVL A B C D E) $\equiv E \Big|_{A=B}^{C=D}$

(NAM A B C) $\equiv C_{A,B}$

(F A B) $\equiv F(A,B)$

(NAM A B (F C D)) $\equiv F_{A,B}(C,D)$

(FTL A) $\equiv A\,!$

(ABS A) $\equiv |A|$

(Most operators can take any number of arguments in the obvious manner.)

Fig. 4

LISP Notation

```
00020     E* (PLS(OR F* (PRD - 1 F*) NIL)(REPEAT 1(OR (PRD -1 F*) F*)))
00030        (C=(OR(OR F*(PLUSS F*))(DASH F*)NIL)(REPEAT 1(OR(DASH F*)(PLUSS
00040          F*))))
00060
00070     S* E* E*
00080     M* (EQN (E* 1)(E* 2)) (C=(E* 1) EQSIGN(E* 2))
00130     F* (PRD(P* 1)(REPEAT 1 P*)(OR C*(P* 2)))
00140        (C=(P* 1)(REPEAT 1(STAR P*))STAR(OR C*(P* 2)))
00150     F* P* P*
00160     F* C* C*
00170     C* (ITG V* (E* 1)(E* 2)(E* 3))
00180        (C=(H= INTEGRAL (E* 1)(E* 2))(E* 3)D V*)
00190     C* (SUM V* (E* 1)(E* 2) H*)
00200        (C=(V=(C= V* EQSIGN(E* 1))SIGMA(E* 2)) H*)
00210     H* (PWR R* E*) (E= R* E*)
00220     H* (ABS E*) (C= LBAR E* RBAR)
00230     H* (V* E*(REPEAT 1 E*))(C= V* LPAR E*(REPEAT 1(COMMA E*))RPAR)
00240     H* V* V*
00250     H* I* I*
00270     H* (FACTORIAL R*) (C= R* EXCLAMATION)
00280     H* (NAM E* (REPEAT 1 E*)V*) (S= V*(C= E*(REPEAT 1(COMMA E*))))
00290     H* (PWR(NAM(E* 1)(REPEAT 1 E*)V*)(E* 2))
00300        (H= V*(C=(E* 1)(REPEAT 1(COMMA E*)))(E* 2))
00310     H* E* (C= LPAR E* RPAR)
00320     Q* (PRD (E* 1)(PWR(E* 2) -1))(V=(E* 2)QUOTIENT(E* 1))
00340     H* (NAM(E* 1)(REPEAT 1 E*)(V*(E* 2)(REPEAT 2 E*)))
00350        (C=(S= V*(C=(E* 1)(REPEAT 1(COMMA E*))))LPAR
00360        (E* 2)(REPEAT 2(COMMA E*))RPAR)
00370     U* (OR(DRV(REPEAT 1(V*(OR 1 K*)))V*)
00380           (DRV(REPEAT 1(V*(OR 1 K*)))H*))
00390        (OR(V=(C=(REPEAT 1(D(OR V*(E= V* K*)))))
00400           QUOTIENT(C=(E= D(SUM/(K* 1 I)))V*))
00410           (C=(V=(C=(REPEAT 1(D(OR V*(E= V* K*)))))
00420           QUOTIENT(E= D(SUM/(K* 1 I))))H*))
00430     I* K* K*
00440     I* (FRT(K* 1)(K* 2)) (V=(K* 2)QUOTIENT(K* 1))
00470     E* (EVL(E* 1)(E* 2)(REPEAT 1((E* 1)(E* 2)))H*)
00480     (C= H*(B= BAR(V=(C=(E* 1)EQSIGN(E* 2))
00490     (REPEAT 1(C=(E* 1)EQSIGN(E* 2))))))
00500     C* (TRANSCENDENTAL(OR V* I* Q* E*))
00510        (C= TRANSCENDENTAL(OR(BLANK V*)(BLANK I*)
00520           (BLANK Q*)(LPAR E* RPAR)))
00530     K* INTEGER INTEGER
00540     V* LITER LITER
00550     P* U* U*
00560     P* Q* Q*
00570     P* H* H*
00580     R* (ABS E*) (C= LBAR E* RBAR)
00590     R* V* V*
```

Fig. 5.

III. Searching the X-Y Plane

I now want to describe how the parsing algorithm picks up the characters from the X-Y plane to form a linear string. The program is presented with a rectangle known to enclose all of the characters and a rule for selecting one character from the rectangle to be added to the linear string. For example, in Fig. 6 the program might be given the solid rectangle and instructions to find the leftmost character in it. The particular character found determines an ordered list of new rectangles, defined in terms of the dimensions of the original one and the dimensions of the character found. A character selecting rule is also associated with each rectangle. In Fig. 6 the divide bar would yield a list of rectangles 1,2, and 3 and the instructions to find the leftmost character in 1 and 2 and the leftmost character falling within the shaded tolerance area in 3. The program then calls itself recursively on each of these smaller rectangles. When no character is found in a rectangle, control returns to the next higher level. I call this a character directed search scheme.

The manner of defining the subrectangles and their associated selection rules depends on the particular set of positional operators. First, any given rectangle contains either a single symbol or is composed of smaller subrectangles related by one of the positional operators. (Requirement 1) Subrectanges must always be scanned in the same order. This requirement is not absolutely necessary, it is only necessary that we get the same interpretation of the linear string no matter what the order. The linear string parsing problem will probably be simpler, however, if requirement 1) can be met. Second, our approach will be to assign a choice of subrectangles and selection rules to each of the positional operators. When this is done

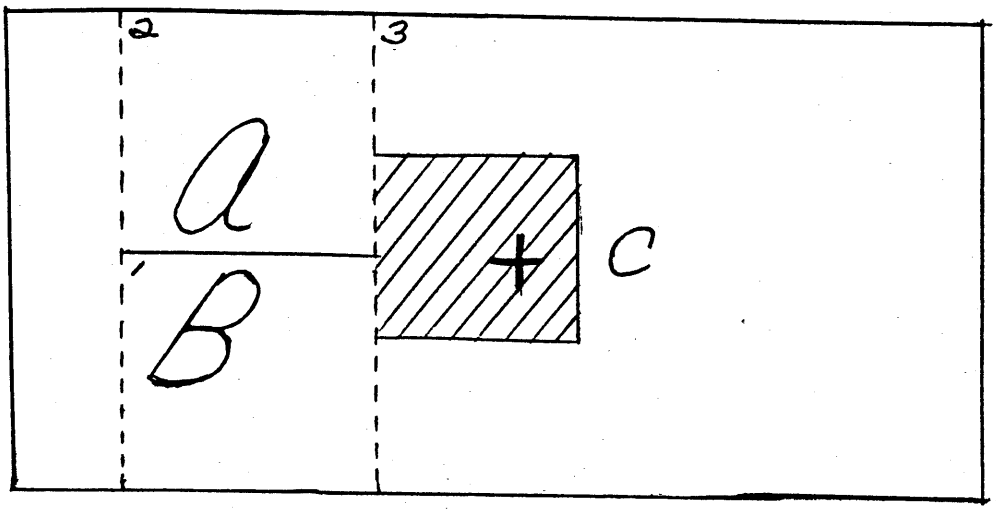


Fig. 6

A Step in the Character Search

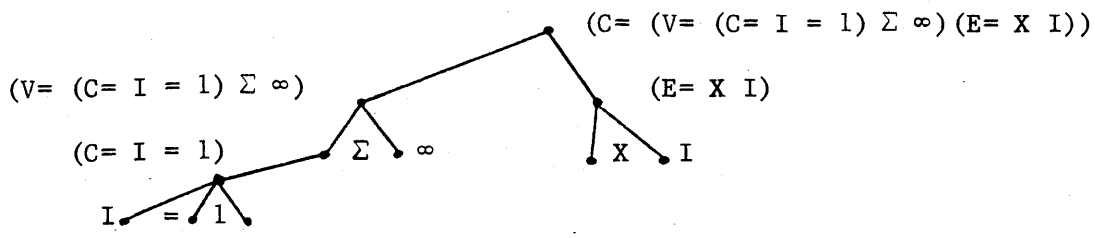
it is necessary that (Requirement 2) no grammatical string in positional operator notation can result in the picking up of some of the characters in a text-book expression defined by a second distinct grammatical string in positional operator notation. This requirement is to avoid ambiguous spatial parsings. For example, we do not want to allow an expression like X^2Y to have the legal parsings X^2 times Y or X times ^2Y.

Anderson also divides each rectangle into smaller ones, using the terminal characters as guides. Looking at the rules in his grammar and algorithm we note that he partitions the current rectangle in all appropriate ways. To be applicable, a rule may contain only those terminal characters which are in the current rectangle, and the characters which occur leftmost and rightmost in the subrectangles must be permitted by the rule. These tests certainly cut down the number of partitions tried. (Assertion A) However, for the grammar defined in the first section the names and dimensions of the characters which can occur leftmost, or rightmost in a rectangle completely determine the subrectangles to be tried. This is a fortunate situation! Requirement 2) then guarantees that if the associated selection rule picks any character in a given subrectangle, that subrectangle applies to the text-book expression at hand.

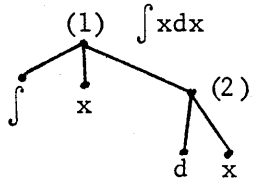
A little thought should convince the reader that a context free grammar will generate a tree structure of subexpressions. For example, the tree structure for the positional operator expression,

(C= (V= (C= I = 1) Σ ∞) (E= X I))

which was given in part 1 is:



A top down parsing scheme, of which Anderson's is an example, picks one of the forms which the top node can have. For this choice there are generally only certain possible choices for the lower nodes, and these can be dependent on the choice for the top node. This can be an advantage, as Anderson points out, if one of the lower nodes can be discovered very easily, thus constraining a higher node and reducing the possibilities for other nodes. For example, in $\int xdx$ which has the tree structure;



discovery of the integral sign gives us the form of node (1) and then we know that node (2) is not a product.

The task of parsing is sometimes simplified if the construction at any node depends only on the nodes below it. This is the situation in a precedence grammar, as defined by Floyd.[6] This property can be verified by examining the grammar in question. For the grammar defined in part 1, it can be verified that the choice of subrectangles is independent of context in which the main rectangle appears. Subrectangles and selection rules can be associated with certain characters without regard to the context in which the characters occur. These are the properties which make a character directed search scheme possible.

Assertion A) follows from a simple exhaustive argument for our particular grammar. First, the leftmost character of a rectangle must belong to its leftmost subrectangle for each positional operator except for vertical concatenation, which has no obvious leftmost subrectangle. Vertical concatenation is used only with the symbols Σ and ———. This is why we require that these symbols extend further to the left than rectangles with which they are vertically concatenated. We thus know for each positional operator which subrectangle is being searched first. Note that when we pick up a character we are starting simultaneously on all the rectangles in which it lies leftmost. Next, for each positional operator it will be possible to tell the positions of additional subrectangles whenever the first subrectangle is finished. To see this we list the characters which can occur rightmost in the first subrectangle of each positional operator.

operator	Characters
concatenation	C= (+ - LITER INTEGER | Σ $\int$) ! —— D
vertical concatenation	V= Σ ——
horseshoe	H= $\int$ LITER
exponent	E= LITER) D | INTEGER
subscript	S= LITER
bottom	B= |

In the case of B=, S=, H=, and V= the first subrectangle encloses a single character, so it is easy to get the dimensions of the other subrectangle from the dimensions of this character and those of the main rectangle. In the case of E= the rightmost character might be a) or | which is sym-

metrical with respect to the center line of the subrectangle, but may not extend to its complete height. Its vertical dimension will suffice if we allow the exponent subrectangle of the E= operator to be placed as low as the top of the rightmost character in the first subrectangle. Finally, the rightmost character of the first C= subrectangle is used to find the X- coordinate of the second subrectangle. The X- coordinate of the $n+1^{st}$ subrectangle is found from the rightmost character of the n^{th} subrectangle. The y- coordinate is found from the center of the leftmost character.

Notice that if we proceed in this way it is easy to meet requirement 1). The scheme will work if it satisfies requirement 2). Once again we use an exhaustive argument for the grammar at hand. Our method of starting the first subrectangle is the same for each positional operator. We must show that an additional subrectangle can't be started unless the positional operator in question applies. If a false rectangle were started, then the first character picked up in it must belong to a second grammatical construction. That is, there must be some previous character from which this character can be reached by two grammatical paths. We have already listed the rightmost characters which can initiate a second subrectangle; some of them can initiate more than one, either because they apply to an operator which initiates more than one or because they apply to more than one operator, or both. These are Σ, ——— , LITER, |,), D, and INTEGER. We must show that the same character can not be reached by taking more than one branch from a given character. In the case of Σ and ——— the plane is divided into non-intersecting areas above, below, and to the right of these characters. For the rest, the second subrectangle is to the right. To avoid the situation:

$$\int_{a}^{b - \frac{x}{c}} \frac{}{d}\, dx$$

we must make the objectionable requirement that every —— or Σ extend a full character width to the left of all symbols in the rectangles concatenated with it vertically. When this is done every rectangle will have its leftmost character at least a character width to the left of the others, then the operators H=, E=, S=, and C= can take second subrectangles only when the subrectangle's center lines are in the appropriate position. Under these conditions requirement 2) is met.

It will probably be necessary to relax requirement 2) in order to make the algorithm acceptable to users. We will do this after experience indicates the complete range of changes needed.

We have assumed that concatenation does not apply if the space for the next symbol is blank. Experimentation with tolerances will be required to determine when additional machinery will be needed to make this determination. One case which has already been handled is sin x, where a blank can be used to separate a transcendental function name from its simple argument. Since the n in sin alone is not sufficient to indicate that a blank can be added to the character string, this is a case where the search scheme can not be character directed. The whole phrase "sin" is required to make skipping a blank a legal operation. I call this an example of a phrase directed search scheme. It is especially simple because the dimensions of the phrase are not needed to guide the search and because the phrase can be found by a lexical parse.

A phrase directed search scheme is one where the characters are picked up in such an order that the search for the next character depends only on the grammatical phrases that can be formed using the characters already picked up. It might be necessary to compute the dimensions of the phrases, using the formulas which define the positional operators, in order to get the parameters needed to guide the search. If we relax this definition to allow back-up, then the expression

$$\int_a^{b - \frac{x}{c}} \frac{}{d}\, dx$$

which gave trouble above could be handled with a phrase directed scheme.

We should note that in order to formulate a grammar for matrices a more complicated search will have to be made when a new rectangle is entered.

IV. Parsing the String

When the mathematical symbols are picked up by the search algorithm it is necessary to add some additional symbols to indicate the positional relationships discovered. The exact symbols used should make the resulting string parsable by a simple algorithm if at all possible. The actual symbols used are:

(C= X Y) → X Y

(V= X ——— Y) → ——— X QMARK1 Y QMARK2

(V= X Σ Y) → Σ X SMARK1 Y SMARK2

(H= ∫ Y Z) → ∫ Y IMARK Z IMARK

(H= LITER X Y) → LITER SE= X SE=END NE= Y NE=END

(E= X Y) → X NE= Y NE=END

(S= X Y) → X SE= Y SE=END

(B= X Y) → X BMARK N= Y N=END

The use of QMARK, SMARK, and IMARK simplifies the parsing by making it unnecessary for the left-to-right parser to change state when a ——— , Σ, or ∫ is parsed. Applying the search scheme to each rule one can find the resulting linear string of terminal and non-terminal characters which would be produced. A parsing algorithm for this transformed set of rules must then be produced. These were found to form an operator precedence grammar[6] with two exceptions; | X | and $\frac{dx^i}{dy^i}$. Absolute value was handled by converting each | to either ⌈ or ⌉ as it was encountered. Absolute value thus becomes ⌈X⌉ . This can be done based on the preceding character, but it represents a change of state[11] since the pre-

ceding character may be either ⌜ or ⌝ . Derivatives were handled by us by looking two characters ahead to find them and then going into a special state for the duration of the derivative quotient.

It was then possible to calculate precedence functions for the rules not involving derivatives. Since the method of doing this is not usually stated, we will give it here. If one has an operator precedence grammar then each ordered pair of terminal symbols may have no relations or be in exactly one of the relations $>$, $<$, and $=$. Precedence functions f and g map the terminal symbols into the integers so that if xRy then $f(x)\ R\ g(y)$. Thus the precedence can be found by comparing the integers assigned to each terminal symbol. To find f and g, first associate with each terminal symbol x a second symbol x'. Define new relations G and E as follows. If $x > y$, then xGy'. If $x < y$ then $y'Gx$. If $x = y$, then xEy'. We have eliminated $<$, now we eliminate E. Arranging the symbols along the edges of a matrix, unprimed first, any two symbols can be in either relation E or G. Moving down the rows, if xEy then $xRz \rightarrow yRz$. Use this to copy the row for x into the row for y and eliminate the row for x. If xEy and yGx occurs, f and g do not exist. When the relation E has been eliminated the elements form a lattice under the relation G. One must now complete the transitive relation G.[14] If xGx occurs for some x, then f and g do not exist. Otherwise, the symbols can easily be ordered. Taken in any order the symbols are added one at a time to a list, being moved to the right until they reach an element with which they are in relation G. The elements are now assigned an integer corresponding to their distance from the right end of the list. The integers assigned to unprimed

elements form f and those assigned to primed elements form g. A small grammar with its precedence table and precedence functions is shown in Fig. 7.

	LISP	Positional Operator
E*	(PLUS E* T*)	(C= T* + E*)
E*	T*	T*
T*	(TIMES T* A*)	(C= A* * T*)
T*	A*	A*
T*	A*	(C= LPAR A* RPAR)
A*	NUMBER	NUMBER
A*	LITER	LITER

f \ g		3 +	9 *	8 NUM	7 LITER	6 (	0)
2	+	L	L	L	L	L	
5	*	G	L	L	L	L	
11	NUMBER	G	G				G
10	LITER	G	G				G
0	(			L	L		=
4	)	G					

Fig. 7

A Small Grammar with its Precedence

Table and Functions

V. Conclusion

The speed of this program indicates that a more complex one which overcomes its weaknesses will run in a practical time. The approach taken here should lead to an efficient result and one which can be understood. The next theoretical step is to construct a parser which will explore the left to right tree, only backing up in truly difficult cases.

Bibliography

1. R. H. Anderson, "Syntax-Directed Recognition of Hand-Printed Two-Dimensional Mathematics", presented at ACM Symposium on Interactive Systems for Experimental Applied Mathematics, Aug. 1967.

2. A. Colmerauer, "Relations de Precedence Totale", Institut de Mathematiques Appliquees, Universite de Grenoble, April 1967.

3. J.J. Donovan, "Canonic Systems and Their Application to Programming Languages", MAC-M-347, M.I.T. April 1967

4. T.G. Evans, "A Formalism for the Description of Complex Objects and its Implementation", Air Force Cambridge Research Lab., Bedford, Mass., Oct. 1967.

5. R.W. Floyd, "The Syntax of Programming Languages - A Survey", IEEE Transactions on Electronic Computers, Vol. EC-13, No.4, Aug. 1964.

6. _____, "Syntactic Analysis and Operator Precedence", JACM, Vol. 10, No. 3., July 1963.

7. S. Ginsburg, The Mathematical Theory of Context Free Languages, McGraw Hill, 1966.

8. T.V. Griffiths and S.R. Petrick, "On the Relative Efficiencies of Context-Free Grammar Recognizers", CACM, Vol. 8, No. 5, May 1965.

9. G.F. Groner, "Real-Time Recognition of Handprinted Text", Memorandum RM-5016-ARPA, The RAND Corporation, Oct. 1966.

10. M. Klerer, Unpublished Notes at Hudson Laboratories, Columbia University, 1965.

11. D.E. Knuth, "Translation of Languages from Left to Right", Information and Control 8, 1965.

12. S. Kuno, "Computer Analysis of Natural Languages", Preceedings of the Symposium on Mathematical Aspects of Computer Science, the American Mathematical Society, New York, 1966.

13. W. A. Martin, "Symbolic Mathematical Laboratory", MAC-TR-36, M.I.T. Jan. 1967.

14. M. Minsky, discussion.

15. N. Wirth and H. Weber, "Euler: A Generalization of ALGOL, and its Formal Definition: Part I", CACM, Vol. 9, No. 1, Jan. 1966.