COMPUTATION CENTER

UNIVERSITY OF NORTH CAROLINA AT CHAPEL HILL

CHAPEL HILL, NORTH CAROLINA

****

RESEARCH STUDY TO DEVELOP METHODS FOR SOLVING PROBLEMS ASSOCIATED WITH SPACE VEHICLES

****

FINAL REPORT

by

BETTY GAIL WOODWARD

Control Number: 1-5-75-00105-01 (1F)
1-5-75-00105-01 S1 (1F)

MONTHLY PROGRESS REPORT, CONTRACT NO. NAS8-20106

FEBRUARY 22, 1966

PREPARED FOR GEORGE C. MARSHALL SPACE FLIGHT CENTER, NASA

HUNTSVILLE, ALABAMA

I. INTRODUCTION

The need for algebraic simplification by computer was realized when the output from an algebraic differentiation program became unmanageable. Algebraic strings, when differentiated, can become extremely long and difficult to evaluate due to uncollected terms and common factors. Investigations of methods to simplify algebraic strings were in order. Manipulating a fairly short algebraic string by hand is fairly simple since a person can get a global view of the string and determine which algebraic laws need to be applied. As the string gets longer, the manipulations become tedious and the chance of error increases. A machine cannot concurrently look at the entire string but only at one character at a time. Therefore the algorithms must scan the string in a manner that places the individual characters in context in order to apply the algebraic laws needed for simplification. The application of the distributive law to the string to get it in a factored or more highly parenthesized form with fewer operations to do in evaluating it, and the grouping and deleting of like elements when possible, were the initial goals. But this does not necessarily result in the simplest form of an expression. The question of what is the simplest form of an expression is subjective. It depends on the expression and the way it is to be used. In some cases the expanded or less highly parenthesized form is desirable. This requires

an algorithm to multiply out parenthesized expressions and expand powers of functions. In expressions involving trigonometric functions, it is sometimes desirable to express the product of trigonometric functions as the sum or difference of other trigonometric functions. This requires an algorithm for trigonometric substitutions. Which in turn required an additional algorithm to substitute for trigonometric functions of negative arguments, ones with positive arguments, and to substitute the values for trigonometric functions with zero argument. Algorithms for these were developed. In all, the following nine algorithms were developed during the course of the investigations on algebraic simplification. This paper describes them in detail.

1. An automatic simplification algorithm which collects like terms and deletes them when possible
2. A factoring algorithm which expresses sums of products of terms as products of factors
3. A multiplication algorithm which expresses products of factors as sums of products of terms and products of a constant and a term as iterative sums of the terms
4. A grouping algorithm which expresses the iterative sum of a term as a product of a constant and the term, and which expresses the iterative product of a term as a power of this term
5. A power expansion algorithm which expresses the power of a term as an iterative product of the term

6. A trigonometric multiplication algorithm which expresses the product of trigonometric functions as the sum of trigometric functions
7. A trigonometric sign algorithm which expresses trigonometric functions of negative arguments as functions with positive arguments
8. A trigonometric value algorithm which substitutes the values for trigonometric functions of zero, and
9. A zero-one simplification algorithm which simplifies expressions containing zero or one.

All of the above algorithms were developed as subroutines to an algorithm published by Ershov [1] (which will henceforth in this paper be referred to as the Ershov algorithm), which transforms a parenthesized expression into a parenthesis free form. Many of the algorithms depend on applying, within proper precedence levels, the commutative and distributive laws of algebra; so a way of looking at each character in the context of its precedence level is needed. The Ershov algorithm is a natural way to do this since it decomposes the string by precedence levels and maintains a current description of these levels in an L-list (to be defined later).

The chief aim of this thesis is to furnish and explain the tools for algebraic simplification. It is easily seen that some of the algorithms are inverses of others; therefore the choice of which tools to apply and the order in which they are applied is very important. Since both of these depend on the expression to be simplified and the desired result, the choice is left up to the user.

DEFINITIONS In this thesis, the representation follows precisely that defined in the article Hanson, Caviness, and Joseph [2]. That definition is:

> In this discussion, all expressions will be assumed to be written in a simplified, algebraic compiler-like language. This language is summarized in the following table:

Operand or Operation	Symbolic Representation
Addition	+
Subtraction	-
Multiplication	*
Division	/
Exponentiation	P [$x^2 \equiv$ xP2]
Left Parenthesis	(
Right Parenthesis	)
Variable ...	Any ... alphabetic character except the symbol "P".
Constants	Remaining unassigned alphabetic characters and the single decimal digits, 0-9.
Transcendental functions	Represented by the string "NAME.(α)". Where "NAME" represents any one of the alphabetic strings in the following list and "α" is an expression (to be defined later).

Function	Name
Exponential	EXP
Logarithm	LOG
Sine	SIN
Cosine	COS
Tangent	TAN
Cotangent	COT
Secant	SEC
Cosecant	CSC
Arcsine	ARCSIN
Arccosine	ARCCOS
Arectangent	ARCTAN
Arcotangent	ARCCOT
Arcsecant	ARCSEC
Arccosecant	ARCCSC
Hyperbolic Sine	SINH

Function	Names (con't from previous page)
Hyperbolic Cosine	COSH
Hyperbolic Tangent	TANH
Hyperbolic Cotangent	COTH
Hyperbolic Secant	SECH
Hyperbolic Cosecant	CSCH
Hyperbolic Arcsine	ARSINH
Hyperbolic Arccosine	ARCOSH
Hyperbolic Arctangent	ARTANH
Hyperbolic Arccotangent	ARCOTH
Hyperbolic Arcsecant	ARSECH
Hyperbolic Arccosecant	ARCSCH

For example, the expression $ax^2 + bx + \sin cx$ would be written as A * XP2 + B * X + SIN.(C*X) in this compiler-like language.

The class of expressions that can be [simplified] by this program are explicitly defined as follows:

1. If Γ is a variable or a constant, then Γ is an expression.
2. If Γ is an expression, then (Γ), $+\Gamma$, and $-\Gamma$ are expressions.
3. If Γ and Δ are expressions, then $\Gamma + \Delta$, $\Gamma - \Delta$, $\Gamma * \Delta$ and Γ/Δ are expressions.
4. If Γ and Δ are expressions, then $\Gamma P \Delta$ is an expression.
5. If Γ is an expression and "NAME" represents one of the allowable transcendental functions of the language, then "NAME".(Γ)" is an expression.
6. The above are the only allowable expressions.

By augmenting our input language with a set of symbols used only within the algorithms, [the process of putting the input string into a parenthesis free form] can be simplified and facilitated. The symbol . will be used as a pseudo-operation. It will signify that the transcendental function specified by NAME, immediately preceding the symbol . has, as a functional argument, the quantity immediately following the . symbol. The symbol ⊟ will be used to represent the unary minus operation. The symbols ⊢ and ⊣ will be used as left and right expression delimiters, respectively, and will also be classed as operators.

All operators are assigned a precedence level. If $\phi(\theta)$ represents the precedence of an operator θ, then the precedence levels are

$$\beta(.) > \beta(P) > (\boxed{}) > \beta(*) = \beta(/) > \beta(-)$$
$$= \beta(+) > \beta(\underline{(}) = \beta(\underline{)}) > \beta(\vdash) = \beta(\dashv).$$

Normal parenthesis conventions are followed within expressions written in the input language. If parentheses are omitted from an expression, then the interpretation of the expression is in accordance with the precedence of the operators. Operators of higher precedence will be executed before operators of lower precedence. Operators on the same precedence level will be executed from left to right in the order of their occurrence in the expression.

The operands, which are represented by the digits 0 through 9 and the alphabetic characters excluding P, can be any variables and/or constants which are a members of some field K with the operators + and *. This restriction must be imposed so that the commutative laws of addition and multiplication, and the distributive law can be applied.

NOTATION In this paper the following notation is used. η_i and R_i refer to an operand and an operator respectively in a changing index system where the index i is defined as follows:

η_α refers to the current operand

$\eta_{\alpha-1}$ refers to the operand to the left of η_α in the input string or below η_α in a list called the L-list (to be defined later)

$\eta_{\alpha+1}$ refers to the first operand above η_α in the L-list if η_α is in the L-list, if not, $\eta_{\alpha+1}$ is the last operand entered in the L-list

R_β refers to the current operator, or the operator to the left of η_α in the input string

$R_{\beta-1}$ refers to the next operator to the left of R_β in the string

$R_{\beta+1}$ refers to the operator most recently entered in the L-list, the operator below $h_{\alpha+1}$ in the L-list.

Each of the permissable operators has a precedence. The precedence of R_i is denoted by $b(R_i)$ and the precedence value is denoted by b-value. The following notation concerning the precedence of operators is also used:

a b-lower operator refers to an operator of lower precedence

a b-higher operator refers to an operator of higher precedence

a b-equal operator refers to an operator of equal precedence.

ERSHOV ALGORITHM An algorithm published by Ershov [1] is used as the basic algorithm. It transforms the input expression into a Polish prefix or parenthesis-free form. The input expression is scanned from right to left and each character is entered into a list which will be referred to as the L-list. The elements of this list are then transferred to a table of triples, where each triple is composed of an operand, operator, operand. This table shall be referred to as the M-matrix. This resulting M-matrix is the parenthesis-free form of the input expression. The algorithm scans the expression character by character and operates on each in the following way. When it encounters an operand h_α, it enters h_α in the L-list and proceeds to look at the next character. When it encounters an operator, R_β, it compares $b(R_\beta)$ with $b(R_{\beta+1})$. If

$\phi(R_\beta) \geq \phi(R_{\beta+1})$, R_β is entered into the L-list and the algorithm proceeds to look at the next character in the string. If $\phi(R_\beta) < \phi(R_{\beta+1})$, a triple consisting of the last three elements in the L-list, $h_\alpha R_{\beta+1} h_{\alpha+1}$, is removed from the L-list and entered into the M-matrix in the first available row, say j. The operand, R_j, which represents this triple and specifies the row position in the M-matrix, is then entered into the L-list in place of the deleted triple. The algorithm then compares $\phi(R_\beta)$ with $\phi(R_{\beta+1})$, where $R_{\beta+1}$ is now the operator which preceeds R_j in the L-list, and proceeds as above. When the algorithm encounters a right parenthesis, it is entered into the next available location on the L-list. A left parenthesis causes everything in the L-list below the lowest corresponding right parenthesis to be transferred to the M-matrix as sets of triples. The right parenthesis is then deleted from the L-list and the algorithm proceeds to the next character to the left of the left parenthesis and ignores the left parenthesis. The pseudo-operator,".", signals that a function name is to its left. The algorithm collects the characters of this name and treats the name as a single operand. When the left-hand end symbol, "|-", is reached, the remainder of the L-list elements is transferred to the M-matrix resulting in the parenthesis-free form of the string.

ADDITIONAL DEFINITIONS Now that the Ershov algorithm has been discussed, a few additional definitions that will be needed can be made. A string consisting of operands connected by"*"and/or"/"will be referred to as a *-string (read times-string) and similarly one

in which the operands were connected by "+" and/or "-" will be referred to as a +-string (read plus-string). The Ershov algorithm enters *-string and +-strings as open backward chains: the first of the rows representing the string contains the first two operands of the string; each succeeding row contains a chain pointer in the left column which designates the preceding row. For example, in the M-matrix a *-string appears as follows:

a * b / c * d	⇒	R_i	a	* b
		R_{i+1}	R_i	/ c
		R_{i+2}	R_{i+1}	* d

A +-string in the M-matrix has the following form:

a + d - c + d	⇒	R_i	a	+ b
		R_{i+1}	R_i	- c
		R_{i+2}	R_{i+1}	+ d

The group or rows which represent a *-string will be referred to as a *-pattern and the group representing a +-string will be referred to as a +-pattern.

The following discussions of the developed algorithms are in one to three levels. Each lower level will be a more detailed description. The first-level portions, read consecutively, explain the functions of the algorithms. The first and second levels, read together, explain the application of the mathematic relationships and introduce the mechanisms of the algorithms. The third level explains, with examples, the details of the algorithms, including

data representation, and serves as a guide for the program listings which are separately available. In the examples an arrow is used to designate a character of the string as that currently under inspection by the Ershov algorithm.

II. THE AUTOMATIC SIMPLIFICATION ALGORITHM The automatic simplification algorithm transforms the input string into a shorter equivalent string, by applying the commutative laws of addition and multiplication, and the definitions of identity elements and inverses with respect to addition and multiplication.

Let K be a field such that all the operands connected by "+", "-", "*", or "/" in the L-list during the execution of the Ershov algorithm are elements of K. The two operations in this field are "+" and "*". For all a, b elements of K, a - b will be considered to be equivalent to a + (-b) where -b is the additive inverse of b. For all a, b, elements of K such that b is not zero, a/b will be considered to be equivalent to a*(1/b) where 1/b is the multiplicative inverse of b. The identity elements of K, with respect to "+" and "*", are "ø" and "1".

Since K is a field, the operations "+" and "*", are commutative. The algorithm repeatedly applies the commutative law of addition to the operands in the L-list which are connected by "+" and "-", (treating the operand that is above the minus as the additive inverse of the operand) in order to group common operands, or to group an operand with its additive inverse if it occurs. When an operand, a, is grouped with its additive inverse, -a, the triple, a-a, is replaced by ø. Similarly, the algorithm recursively applies the commutative law of multiplication to the common operands in the L-list which are connected by "*" and "/". When an operand, a, is grouped with its multiplicative inverse, the

triple, a/a, is replaced by 1. Usually the identity laws a-ϕ = a and A*1 = a are also applied.

SECOND LEVEL This algorithm is used as a subroutine within the Ershov algorithm. It is called as the Ershov algorithm enters an operand into the L-list.

Since all of the permissable operators, with the exception of $\boxminus$, are binary operators, η_α, the current operand, is bound by R_β or $R_{\beta+1}$, depending on which has the higher precedence. The binding operator will be denoted by R_b. In order to see whether the commutative law of addition or multiplication, if either, can be applied, $\flat(R_\beta)$ and $\flat(R_{\beta+1})$ are checked. If $\flat(R_\beta)$ or $\flat(R_{\beta+1})$ is greater than $\flat(*)$ or if $\flat(R_{\beta+1})$ is less than $\flat(R_\beta)$, neither of the laws can be applied.

If $\flat(R_\beta)$ is $\flat(+)$ and $\flat(R_{\beta+1})$ is $\flat(*)$, η_α is bounded by $R_{\beta+1}$ so $\flat(R_b)$ is $\flat(*)$. The value of $\flat(R_b)$ determines which commutative law will be applied. If $\flat(R_\beta)$ is less than $\flat(+)$ (i.e., when R_β"(" or "$\vdash$"), $\flat(R_\beta)$ is treated as $\flat(+)$, the lower of the two $\flat$-values of interest.

The algorithm scans up the L-list searching for another occurance of η_α until a $\flat$-lower operator than R_b is encountered. A $\flat$-higher operator will not be encountered since the Ershov algorithm would have previously caused the triple containing this operator to be transferred to M-matrix. Therefore, this scan checks only the operands that could be commuted with η_α. If another occurance of η_α is found, it is commuted with the operands below it in the

L-list so it will be in position for grouping or deletion.

THIRD LEVEL In this level the algorithm will be discussed in the following phases:

1. The entry into the algorithm
2. The scan for common expressions
3. The "commuting" routine
4. The "deleting" routine

1. THE ENTRY INTO THE ALGORITHM

This algorithm is entered when an operand, h_α, be it a variable or a row reference, is about to be entered in the L-list. Two tags γ and Δ, are set as follows, depending on the operator, R_β:

1. if R_β = +, (, or $\vdash$, set γ = + and Δ = - (see examples 2.1, 2.2, 2.3)
2. if R_β = - set γ = - and Δ = + (see example 2.4)
3. if R_β = * set γ = * and Δ = / (see example 2.5)
4. if R_β = /, set γ = / and Δ = * (see example 2.6)

Otherwise an exit from this routine is made.

Input String

$\vdash$ a + b - c $\dashv$
↑

L-list

$\dashv$ h_α = b and R_β = +
c $\therefore$ γ = + and Δ = -
-

Example 2.1 Setting γ and Δ when R_β = +

Input String

$\vdash$ a * (b + c) $\dashv$

L-list

$\dashv$

c

+

h_α = b and R_β = (

$\therefore$ γ = + and Δ = -

Example 2.2 Setting γ and Δ when R_β = (

Input String

$\vdash$ a + b - c $\dashv$

L-list

$\dashv$

c

-

b

+

h_α = a and R_β = $\vdash$

$\therefore$ γ = + and Δ = -

Example 2.3 Setting γ and Δ when R_β = $\vdash$

Input String

$\vdash$ a - b + c $\dashv$

L-list

$\dashv$

c

+

h_α = b and R_β = -

$\therefore$ γ = - and Δ = +

Example 2.4 Setting γ and Δ when R_β = -

Input String

```
⊢ a * b * c ⊣
    ↑
```

L-list

```
⊣        η_α = b and θ_β = *
c        ∴  γ = * and Δ = /
*
```

Example 2.5 setting γ and Δ when $\theta_\beta = *$

Input String

```
⊢ a/b * c ⊣
   ↑
```

L-list

```
⊣          η_α = b        θ_β = /
c          ∴  γ = / and Δ = *
*
```

Example 2.6 Setting γ and Δ when $\theta_\beta = /$

Δ can be considered as the "inverse" of γ since if for some operand η_i, we have $\gamma\ \eta_i\ \Delta\ \eta_i$, this would be equal to $\emptyset$ or 1 depending on whether $\gamma = +$ or $-$, or $\gamma = *$ or $/$, (e.g. $-a + a = \emptyset$ and $* a/a = 1$). In effect, this pair of operands with their respective operators would be deleted from the L-list.

2. THE SCAN FOR COMMON EXPRESSIONS

After γ and Δ have been set, a scan is made up the L-list, comparing the operators with γ and Δ. If the operator $\theta_i = \gamma$, the operand η_i and θ_i in the L-list is checked. If $\eta_i = \eta_\alpha$ the "commuting" routine is entered (see Example 2.7).

If $\eta_i \neq \eta_\alpha$, the scan of the operators in the L-list is continued until a γ-lower operator than γ is encountered. In this case an exit is made, (see Example 2.8).

Input String

$\vdash$ f + a + b - c + d + a $\dashv$

L-list

$\dashv$	h_α = a and R_β = +, $\therefore$ γ = + and Δ -
a	During the scan it is found that:
+	$R_{\beta+1}$ = + = γ but $h_{\alpha+1}$ = b $\neq$ h_α = a $\therefore$ scan continues
d	$R_{\beta+2}$ = - = Δ but $h_{\alpha+2}$ = c $\neq$ h_α = a $\therefore$ scan continues
+	$R_{\beta+3}$ = + = γ but $h_{\alpha+3}$ = d $\neq$ h_α = a $\therefore$ scan continues
c	$R_{\beta+4}$ = + = γ and $h_{\alpha+4}$ = h_α $\therefore$ the commuting routine
-	is entered
b	
+	

Example 2.7 Case where commutative law can be applied.

$\vdash$ f * a + b + c $\dashv$

L-list

$\dashv$	h_α = a and R_β = *, $\therefore$ γ = * and Δ = /
c	During scan it is found that $R_{\beta+1} \neq \gamma$ or Δ
+	and $\flat(R_{\beta+1}) < \flat(\gamma)$, $\therefore$ an exit is made
b	
*	

Example 2.8 Case where commutative law is not applicable

Now if $R_i = \Delta$, the operator, h_i, above R_i in the L-list, is checked. If $h_i = h_\alpha$ the "deleting" routine is entered and Δ h_i is

effectively removed from the L-list. If $h_i \neq h_\alpha$ the scan for comparing operators with γ and Δ is continued. (see Example 2.9).

Input String

$\vdash$ f + a + b - c + d - a $\dashv$

L-list

$\dashv$	h_α = a and R_β = +, $\therefore$ γ = + and Δ = -
a	During the scan of the L-list it is found that:
-	$R_{\beta+1}$ = + = γ but $h_{\alpha+1}$ = b $\neq h_\alpha$, $\therefore$ the scan is continued
d	$R_{\beta+2}$ = - = Δ but $h_{\alpha+2}$ = c $\neq h_\alpha$, $\therefore$ the scan is continued
+	$R_{\beta+3}$ = + = γ but $h_{\alpha+3}$ = d $\neq h_\alpha$, $\therefore$ the scan is continued
c	$R_{\beta+4}$ = - = Δ and $h_{\alpha+4}$ = a = h_α, $\therefore$ the deleting
-	routine is entered
b	
+	

Example 2.9 Case where deletion of operands is possible.

If $R_i \neq \gamma$ or $R_i \neq \Delta$ but R_i =) or $\dashv$, an exit is made from the scan back to the Ershov algorithm which then enters the operand h_α in the L-list. Now if $R_i \neq \gamma$ or Δ, and R_i is the last operator entered in the L-list, and R_i = * or /, set γ = * and Δ = /. Then repeat the above procedure for these new values of γ and Δ , (see Example 2.10).

If R_i does not satisfy any of the above conditions, an exit is made from the scan back to the Ershov algorithm which enters h_α in

the L-list.

Input String

$\vdash$ a + b * c / b + d $\dashv$

$\uparrow$

L-list

$\dashv$	h_α = b and R_β = +
d	$\therefore$ γ = + and Δ = -
+	$R_{\beta+1}$ = * $\neq \gamma$ or Δ
b	$\therefore \gamma \Leftarrow$ * and $\beta \Leftarrow$ /, and repeat scan with
/	new γ and Δ
c	
*	

Example 2.10 Case where γ and Δ must be reset.

3. THE "COMMUTING" ROUTINE

The "commuting" routine is entered whenever $R_i = \gamma$ and $h_i = h_\alpha$. First, this routine checks for additional occurances of pairs, $\gamma\ h_\alpha$, immediately above the first pair encountered by the scan. The operator, R_{i+1} above h_i in the L-list is compared to γ. If $R_{i+1} = \gamma$, the operand, h_{i+1}, is compared with h_α. If $h_{i+1} = h_\alpha$, this check for additional pairs, $\gamma\ h_\alpha$, is extended to h_{i+2}, (see Example 2.11).

Whenever $h_{i+1} \neq h_\alpha$, the elements below R_i, the operator below the first equivalent operand found, are moved up in the L-list, to effectively delete the operands equivalent to h_α and along with their operators, R_i, from this part of the L-list. Then the operands

equivalent to h_α and the operator γ are added alternatingly to the bottom of the L-list, until all the operands have been entered . An exit is then made to the algorithm, (see Examples 2.11 and 2.12). The grouping algorithm will collect this +-string or *-string of common variables and express them as a product or power, respectively, of the variable.

Input String

$\dashv$ d + a + b + a + c + a $\dashv$

L-list		L-list
$\dashv$	$h_\alpha = a \quad \Re_\beta = +, \therefore \gamma = +$ and $\Delta = -$	$\dashv$
c		c
+	$\Re_{\beta+2} = + = \gamma$ and $h_{\alpha+2} = a = h_\alpha$	+
a	$\Re_{\beta+3} = + = \gamma$ and $h_{\alpha+3} = a = h_\alpha$	b
+	$\Re_{\beta+4} = + = \gamma$ but $h_{\alpha+4} = c \neq h_\alpha$	+
a	$\therefore$ L-list becomes $\Rightarrow$	a
+		+
b		a
+		+

Example 2.11 Multiple occurances of an operand that can be commuted.

Input String

$\vdash$ a + b - c + b - d $\dashv$

Initial L-list

$\dashv$
d
-
b
+
c
-

$\eta_\alpha = b$

$\Re_\beta = +$

$\eta_{\alpha+2} = \eta_\alpha$

and $\Re_{\beta+2} = \Re_\beta$

$\Rightarrow$

Resultant L-list

$\dashv$
d
-
c
-
b
+

Example 2.12 Commuting an operand and operator

4. THE DELETING ROUTINE

The deleting routine is entered whenever $\Re_i = \Delta$ and $\eta_i = \eta_\alpha$. $\Re_i$ and η_i are deleted from the L-list and when required, the triple, $\eta_\alpha \Delta \eta_i$, is replaced in the L-list by the operand $\emptyset$ or by the operand 1. The elements in the L-list below the operator $\Re_i = \Delta$ which is immediately below the operand $\eta_i = \eta_\alpha$, are moved up so as to delete $\Re_i$ and η_i from the L-list, (see Examples 2.13 and 2.14).

Input String

$\vdash$ a + b + c - b + d $\dashv$ $\qquad$ $\vdash$ a + b + c - b - d $\dashv$

L-list $\qquad$ L-list

$\dashv$ $\qquad$ $\dashv$

d $\qquad$ $\Rightarrow$ $\qquad$ d

\+ $\qquad$ $\eta_\alpha = b$ and $\mathcal{R}_\beta = +$ $\qquad$ +

b $\qquad$ $\therefore \gamma = +$ and $\Delta = -$ $\qquad$ c

\- $\qquad$ $\eta_{\alpha+2} = b = \eta_\alpha$ and $\mathcal{R}_{\beta+2} = - = \Delta$

c

\+

Example 2.13 Deletion of an operator and operand from the L-list

Input String

$\vdash$ a * b * c / b + d $\dashv$

Initial L-list $\qquad$ Resultant L-list

$\dashv$ $\qquad$ $\eta_\alpha = b$ $\qquad$ $\mathcal{R}_\beta = *$ $\qquad$ $\dashv$

d $\qquad$ $\therefore \gamma = *$ and $\Delta = /$ $\qquad$ d

\+ $\qquad$ $\eta_{\alpha+2} = b = \eta_\alpha$ $\qquad$ +

b $\qquad$ and $\mathcal{R}_{\beta+2} = / = \Delta$

/ $\qquad$ $\therefore$ L-list becomes $\Rightarrow$ c

c

Example 2.14 Deletion of an operator and operand from the L-list

If $\gamma = *$ and $\mathcal{R}_\beta \neq \gamma$, (i.e. if the initial γ and Δ were changed), η_i and $\mathcal{R}_i$ were the last operand and operator to be entered in the L-list, and $\flat(\gamma) > \flat(\mathcal{R}_{i+1})$, then the operand "1" is added to the L-list. An exit is then made to the Ershov algorithm which then compares the $\flat(\mathcal{R}_\beta)$ with $\flat(\mathcal{R}_{i+1})$, (see Example 2.15).

Input String

$\vdash a + b / b + d \dashv$ $\qquad\qquad$ $\vdash a + b/b + d \dashv$

Initial L-list $\qquad\qquad$ Resultant L-list

$\dashv$		$\dashv$
d	$\Rightarrow$	d
+		+
b		1
/		

$R_\beta = +$ and $h_\alpha = b$

$\gamma = *$ and $\Delta = /$

$R_{\beta+1} = / = \Delta$

$R_\beta \neq \gamma$

$p(\gamma) > p(R_{i+1})$ where $R_{i+1} = +$

$\therefore \vdash a + b / b + d \dashv \Rightarrow \vdash a + 1 + d \dashv$

Example 2.15 Deletion of an operator and operand from the L-list and replacing the triple by "1".

If $R_\beta = \vdash$ or (, the procedure to be followed will depend on $R_{\beta+1}$. If $R_{\beta+1} = -$, change it to $\boxminus$, the unary minus, and continue the Ershov algorithm by comparing $p(R_{\beta+1})$ with $p(R_{\beta+2})$, (e.g. $\vdash A - B - A \dashv \Rightarrow \vdash \boxminus \beta \dashv$). If $R_{\beta+1} = /$, add the operand 1 to the L-list and continue the Ershov algorithm by comparing $p(R_\beta)$ with $p(R_{\beta+1})$, (e.g. $\vdash A/B/A + C \dashv \Rightarrow \vdash 1/B+C \dashv$). If $R_{\beta+1} =)$, erase $R_{\beta+1}$ from the L-list and then compare the $p(R_{\beta-1})$ with the precedence of the new $R_{\beta+1}$. If $p(R_{\beta-1}) > p(+)$, add the operand $\emptyset$ to the L-list and continue the Ershov algorithm by comparing $p(R_{\beta-1})$

with that of the operator above $\emptyset$ in the L-list, (e.g., $\vdash$ A* (B-B)+C -$\dashv$ $\Rightarrow$ $\vdash$ A*$\emptyset$+C $\dashv$). If $\flat(R_{\beta-1}) > 3$, continue the Ershov algorithm by looking at the element to the left of $R_{\beta-1}$ in the string, (e.g., $\vdash$ A+B(B-B)+C$\dashv$ $\Rightarrow$ $\vdash$ A+C $\dashv$). If $R_{\beta+1} \neq$ -, /, or), i.e., $R_{\beta+1}$ = + or *, $R_{\beta+1}$ is erased from the L-list and the Ershov algorithm is continued by comparing $\flat(R_\beta)$ with that of the last operator in the L-list, (e.g., $\vdash$ A*B/A+C $\dashv$ $\Rightarrow$ $\vdash$ B+C $\dashv$ and $\vdash$ A*(B+C-B) $\dashv$ $\Rightarrow$ $\vdash$ A*C $\dashv$).

If γ = * and $R_\beta \neq \gamma$ (i.e., if the initial γ and Δ were changed) and $\flat(\gamma) = \flat(R_{\beta+1})$ erase $R_{\beta+1}$ from the L-list and continue the Ershov algorithm by comparing $\flat(R_\beta)$ with the precedence of new $R_{\beta+1}$, (e.g., $\vdash$ A+B/B*C $\dashv$ $\Rightarrow$ $\vdash$ A+C $\dashv$).

If γ and Δ were not changed and $R_\beta \neq \vdash$ or (, the Ershov algorithm is continued by looking at $R_{\beta-1}$.

III. THE FACTORING ALGORITHM

The factoring algorithm transforms the input string into a shorter equivalent string by application of the distributive law of algebra. In the present formulation the distributive law is first applied to factor out the trigonometric functions and then the remaining factors in the remainder of the input string. Within the general algorithm any designated classes of expressions such as a particular variable raised to a power, a trigonometric function, an exponential function, a logarithm function, etc.—could be factored out in a specified order.

SECOND LEVEL By the definition of the distributive law, for all a,b,c, elements of the field K, $a * b \pm a * c = a *(b \pm c)$. The factoring routine is basically an algorithm for the recursive application of this law to the input string. As a result of the particular method used there are some limitations to the generality achieved by the algorithm.

The factoring routine is a subroutine within the Ershov algorithm. It is entered whenever a *-string is about to be entered into the M-matrix. The routine applies the distributive law to this *-string and the first *-pattern it encounters which contains at least one of the operands of the *-string. The selection of the variables to be factored out of *-strings is a function of the

ordering the *-string as it appears in the input string. e.g.,

a * b * c + a * d * e + f * b * c

$\Rightarrow$ a * (b * c + d * e) + f * b * c

$\not\Rightarrow$ b * c * (a+f) + a * d * e

but a * b * c + f * b * c + a * d * e $\Rightarrow$ b * c * (a+f) + a * d * e.

A second limitation of the algorithm is that the nth power of an element of the field K is treated as a distinct element of K and not as the product of n-terms. For this reason a variable can not be factored out of a power of the element and a *-string containing the variable, e.g., xP2+a*x $\not\Rightarrow$ x*(x+a).

THIRD LEVEL The factoring algorithm is entered whenever the Ershov algorithm is about to enter a triple, consisting of operand, "*", operand, into the M-matrix. First, the factoring routine scans up the L-list for a row reference which is connected to the *-string or a "+" or "-". If the row reference does not refer to a *-pattern, (see R5 in Example 3.1), the scan is continued; if it refers to a *-pattern, (see R4 and R2 in example 3.1), the operands of the pattern are compared with those of the current triple. If none of the operands of the first *-pattern encountered are equivalent to those of the triple, (see R4 in Example 3.2), the scan of the L-list is continued. When one of the operands of a *-pattern is found to be the same as one of those of the triple, it is entered into an I-list. Thus the I-list becomes a list of those

operands of a *-string and a *-pattern that are identical, and can therefore be factored out. The row reference to the *-pattern is saved in a temporary location, γ, and is deleted from the L-list, (see Example 3.2).

Input String	L-list	M-matrix
⊢ n+k*a*b*c+d+zP2+e*f*g-a*h*c ⊣	⊣	R1 a * h
↑	R2	R2 R1 * c
	-	R3 e * f
	R4	R4 R3 * g
	+	R5 z P 2
	R5	
	+	
	d	
	+	
	c	
	*	
	b	
	*	
	a	
	*	
	k	

Example 3.1 The L-list and M-matrix, upon entering the factoring routine, displaying a *-string in the L-list and the *-patterns in the M-matrix.

Input String

⊢ n + k * a * b * c + d + zP2 + e * f * g - a * h * c ⊣

↑

L-list		L-list	I-list	
⊣		⊣		
		R4	a	γ = R2
R2		+		
-		R5		
R4	⇒	+		
+		d	M-matrix	
R5		+	R1 a * h	
+		c	R2 R1 * c	
d		*	R3 e * f	
+		b	R4 R3 * g	
c		*	R5 z P 2	
*		a		
b		*		
*		k		
a				
*				
k				

Example 3.2 Factoring "a" from the *-string and the *-pattern referred to by "R2".

The operand that is to be factored out (now in the I-list) is deleted from the L-list and from the *-pattern. The deletion from the *-pattern is accomplished in one of two ways:

(1) if it is in the bottom row of the *-pattern, the row reference, γ, which refers to the *-pattern is changed to the other operand in that row which is a reference to the remainder of the *-pattern,

e.g., to delete "c" from

R1	a	* h
R2	R1	* c

$$\gamma = R2 \Rightarrow \gamma = R1$$

(2) if it is in any of the other rows of the *-pattern the other operand of that row is used to replace the reference to that row, i.e., the linking of the chain is changed so as to exclude the operand from the *-pattern.

e.g., to delete "a" from

R1	a	* h
R2	R1	* c

R1	a	* h		R1	a * h	
			$\Rightarrow$			
R2	R1	* c		R2	h * c	where γ = R2

The other operands of the *-string are also compared with those of the *-pattern. If a second operand is found to occur in both, it is also entered in the I-list and deleted from the *-string and from the *-pattern as described above, (see Example 3.3).

L-list	I-list		L-list	I-list
⊣	a	γ = R2	⊣	a γ=h
R4			R4	c
+			+	
R5			R5	
+			+	
d	M-matrix		d	M-matrix
+	R1 a * h	⇒	+	R1 a * h
c	R2 h * c		b	R2 h * c
*	R3 e * f		*	R3 e * f
b	R4 R3* g		k	R4 R3* g
*	R5 z P 2			R5 z P 2
k				

Example 3.3 Deletion of a second operand from the *-string and the *-pattern.

When all the operands that occur in both the *-string and the *-pattern have been deleted and put in the I-list, the remainder of *-string is then entered in the M-matrix. Next a triple, consisting of the reference to the remainder of the *-string now in the M-matrix, the operator "+" or "-", and the reference to the remainder of the *-pattern, is entered in the M-matrix. The operator, R_o, of this triple is determined by the operator, R_p, preceding the *-pattern reference and the operator, R_s, preceding the *-string. If R_p and R_s are both "+" or both "-", R_o is "+", otherwise, R_o is "-", (see Example 3.4).

The reference to this triple is entered in the L-list followed alternately by "*" and the elements of the I-list, (see Example 3.5). After these have been added an exit is made from this routine.

L-list	M-matrix	$\gamma = h$	L-list	M-matrix
R4	R1 a * h		R4	R1 a * h
+	R2 h * c		+	R2 h * c
R5	R3 c * f	⇒	R5	R3 e * f
+	R4 R3 * g		+	R4 R3 * g
d	R5 z P 2		d	R5 z P 2
+			+	R6 k * b
b				R7 R6 - h
*				
k				

Example 3.4 Transfer of the remainder of the *-string to the M-matrix and the entry of the final triple on the M-matrix.

I-list	L-list		L-list
a	R4		R4
c	+		+
	R5	⇒	R5
	+		+
	d		d
	+		+
			R7
			*
			c
			*
			a

Example 3.5 Entry of the factored form on the L-list.

IV. MULTIPLICATION ALGORITHM

The multiplication algorithm transforms the input string into an equivalent expanded string by a recursive application of the distributive law—for all a,b,c elements of the field K, a * (b±c)= a*b±a*c. The power of a variable, xPn where n is an integer such that $0 \leq n \leq 9$, is expanded into a product e.g., x P 3 ⇒ x * x * x.

SECOND LEVEL The multiplication routine is used as a sub-routine within the Ershov algorithm. It can be entered each time a triple with "*" as the operator is about to be entered into the M-matrix. Each triple is checked and if neither of its operands is a reference to a +-pattern, an exit from this routine is made. When exactly one of the operands is a reference to a +-pattern, the resulting expansion is such that each operand of the +-pattern is multiplied by the other operand of the current triple, (see Example 4.1). When both operands of the current triple are references to +-patterns, the resulting expansion from one application of this routine is such that each operand of one of the +-patterns is multiplied by the reference to the other pattern.

Input String

```
|- c + a * (b-c+d) -|        ⇒    |- c + a * b - a * c + a * d -|
     ↑                                ↑
```

L-list	M-matrix		L-list	M-matrix
-|	R1 b - c	⇒	-|	R1 b - c
R2	R2 R1 + d		R3	R2 R1 + d
*			+	R3 a * d
a			R4	R4 a * c
			-	R5 a * b
			R5	

(where the current triple is: a * R2

Example 4.1 Multiplication of a variable and a +-string.

When a *-string contains two or more +-strings as operands, the multiplication routine is applied two or more times to yield the expanded string.

e.g., (a+b) * (c-d) * (e+f) ⇒ a * (c-d) * (e+f) + b * (c-d) * (d+f)

⇒ a * c * (e+f) - a * d * (e+f) + b * c * (e+f) - b * d * (e+f)

⇒ a * c * e + a * c * f - a * d * e - a * d * f + b * c * e + b * c * f - b * d * e - b * d * f

(These are 3 +-strings, therefore 3 passes are made through the multiplication routine.)

If the initial scan of a triple containing a "*" operator encounters a reference to a +-pattern, then the +-pattern in turn is scanned and each of its operands along with the appropriate operator is transfered to the E-list--a temporary list used to hold the +-string while it is multiplied by the other operand of the current triple. The operator $\mathcal{R}_E$ in the expanded expression, corresponding to the individual operands in the E-list, are determined by the operator $\mathcal{R}_\beta$ that precedes the representation of the current triple in the input string and the operators $\mathcal{R}_S$ that precede the corresponding operands in the +-string. If $\mathcal{R}_\beta$ and $\mathcal{R}_S$ are both "+" or both "-", $\mathcal{R}_E$ is "+"; otherwise $\mathcal{R}_E$ is "-", (see Example 4.2).

After all the operands of the +-string are transfered to the E-list, the remainder of the *-string of which the +-pattern reference was a part, is put in the M-matrix and deleted from the L-list. The reference to this *-string is recorded in $\in$, (see Example 4.2). Next, the triples consisting of $\in$ * and each operand from the E-list are entered in the M-matrix. The references to these triples, along with the corresponding operators from the E-list, are entered in the L-list, (see Example 4.3). When this has been completed an exit is made.

Input String

|- c - a * (b - c + d) * e * f + g -|
  ↑

L-list	M-matrix	E-list
g	R1 b - c	d
+	R2 R1 + d	-
f		c
*		+
e		b
*		
R2		
*		
a		

	L-list	M-matrix	E-list	∈ = R4
	-|	R1 b - c	d	
	g	R2 R1 + d	-	
⇒	+	R3 a * e	c	
		R4 R3 * f	+	
			b	

Example 4.2 The operands of the +-pattern (referred to by R2) and the appropriate operators are put in the E-list. The elements of the *-string, excluding "R2" are entered in the M-matrix and ∈ is set.

Input String

⊢ c - a * (b - c + d) * e * f + g ⊣
  ↑

L-list	M-matrix	E-list	∈ = R4
⊣	R1 b - c	d	
g	R2 R1 + d	-	
+	R3 a * e	c	
	R4 R3 * f	+	
		b	

	L-list	M-matrix
	⊣	R1 b - c
	g	R2 R1 + d
	+	R3 a * e
	R5	R4 R3 * f
⇒	-	R5 R4 * d
	R6	R6 R4 * c
	+	R7 R4 * b
	R7	

(The result of this upon the input string

c - a * (b - c + d) * e * f + g would be

c - a * e *f * b + a * e * f * c - a * e * f * d + g)

Example 4.3 Entering the triples in the M-matrix and their references in the L-list to yield the expanded result.

The routine to expand a power of an operand is also a subroutine within the Ershov algorithm. The power expansion subroutine is entered whenever a triple, $h_\alpha P h_{\alpha+1}$, is about to be entered in the M-matrix. If $h_{\alpha+1}$ is an integer this triple is expanded; otherwise an exit is made. When $h_{\alpha+1}$ is an integer, the following *-string is formed and entered in the L-list:

$$h_\alpha * h_\alpha * \ldots . * h_\alpha$$

$h_{\alpha+1}$ times

(see Example 4.4).

Input String

|- a + b * c P3 * d + e -|

L-list ⇒ L-list

L-list		L-list
-|		-|
e		e
+	⇒	+
d		d
*		*
3		c
P		*
c		c
		*
		c

Example 4.4 Expanding the power of a variable.

V. THE TRIGOMETRIC ALGORITHMS

The trigonometric simplification algorithms express the product of two or more trigonometric functions as the sum or difference of appropriate trigonometric functions, substitute trigonometric functions with positive arguments for those with negative arguments, and substitute values for the trigonometric functions of zero by application of the following rules.

1. $\sin. x \sin. y = \frac{1}{2}\left[\cos. (x - y) - \cos. (x + y)\right]$

   $\sin. x \cos. y = \frac{1}{2}\left[\sin. (x + y) + \sin. (x - y)\right]$

   $\cos. x \sin. y = \frac{1}{2}\left[\sin. (x + y) - \sin. (x - y)\right]$

   $\cos. x \cos. y = \frac{1}{2}\left[\cos. (x + y) + \cos. (x - y)\right]$

2. $\sin. (-x) = -\sin. x$ $\quad\quad$ $\cot. (-x) = -\cot. x$

   $\cos. (-x) = \cos. x$ $\quad\quad$ $\sec. (-x) = \sec. x$

   $\tan. (-x) = -\tan. x$ $\quad\quad$ $\csc. (-x) = -\csc. x$

3. $\sin. \emptyset = \emptyset$

   $\cos. \emptyset = 1$

   $\tan. \emptyset = \emptyset$

   $\sec. \emptyset = \emptyset$

If "cot. ∅," "csc. ∅," or "log. ∅" is found, an error is indicated and the processing of the string is terminated. If "exp. ∅" is found, the value "1" will be substituted.

SECOND LEVEL OF TRIGONOMETRIC MULTIPLICATION ALGORITHM The algorithm for expressing the product of two or more trigonometric functions as the sum or difference of trigonometric functions is

a subroutine with the Ershov algorithm. After a trigonometric function has been entered in the M-matrix, the trigonometric multiplication algorithm is entered. A check is made to see if the reference to this function, which is now to be entered on the L-list, will be connected by a "*" to the row-reference to another trigonometric function. If so, the rules for the multiplication of these functions are applied. The triples representing the resulting sum or difference are entered in the M-matrix and their row-reference replaces the row-reference to the trigonometric function of the product now in the L-list, (see Example 5.1).

Input String:

⊢ a + b * sin. x * cos. y * c + d ⊣

L-list	M-matrix				L-list	M-matrix			
⊣	R1	cos.	y		⊣	R1	cos	.	y
d	R2	sin.	x		d	R2	sin	.	x
+					+	R3	x	+	y
c					c	R4	x	-	y
*				⇒	*	R5	1	/	2
R1 (tagged)					R1Ø	R6	sin	.	R3
*						R7	R5	*	R6
						R8	sin	.	R4
						R9	R5	*	R6
						R1Ø	R7	+	R9

The output string is as follows:

a + b * (½ * sin. (x+y) + ½ * sin.(x-y)) * c + d.

Example 5.1 The replacement of a product of trigonometric functions by a sum of trigonometric functions.

If the reference to the trigonometric function just entered in the M-matrix will not be connected by a "*" to another trigonometric function in the L-list, one of two things is done. If the operator preceding the current trigonometric function in the input string is p-lower than "*", the reference to the function is entered in the L-list and an exit is made, (see Example 5.2). Otherwise this reference to the trigonometric function is tagged so it will be easily recognized if another trigonometric function, connected to it by "*", is encountered. This tagged reference is entered in the L-list, (see Example 5.3).

If no other trigonometric function is encountered in the same *-string, no trigonometric multiplication is possible, so the tag is removed from the row-reference before the *-string is entered in the M-matrix.

Input String

⊢ a + sin. x * c + d ⊣
  ↑

L-list		M-matrix
⊣		R1 sin . x
d		
+	$p(R_\beta) < p(*)$	
c	∴ R1 is entered in the L-list	
*	and an exit is made.	
R1		

Example 5.2 When no trigonometric multiplication is required and $p(R_\beta) < p(*)$, a normal exit is made.

Input String

```
⊢ a + b * sin. x * c * cos. y + d ⊣
                          ↑
L-list                L-list              M-matrix
 ⊣                     ⊣                  R1  cos . y
 d          ⇒          d
 +                     +
 y          (tagged)   R1
 .
cos
```

Example 5.3 When no trigonometric multiplication is needed now but there is a possibility that there may be, the tagged reference is entered in the L-list.

SECOND LEVEL OF TRIGONOMETRIC SIGN ALGORITHM The algorithm for the substitution of trigonometric functions with positive arguments for those with negative arguments is also a subroutine within the Ershov algorithm. It is entered when a trigonometric function with a negative argument is about to be entered in the M-matrix. First, the sign of the argument is made positive. If the trigonometric function is cosine or secant, an exit is made, (see Example 5.4). Otherwise, the sign of the function must be changed, (see Example 5.5). If $\not\flat(R_\beta)$ is equal to $\not\flat(+)$, R_β is changed to the inverse operation.

Input String

```
⊢ a + cos. (-b) + c ⊣
    ↑
```

L-list	M-matrix		L-list	M-matrix
⊣	R1 ∅ - b	⇒	⊣	R1 ∅ - b
c			c	
+			+	
R1			b	
.			.	
cos			cos	

Example 5.4 Substituting for a trigonometric function with a negative argument, the function with a positive argument.

Input String

```
⊢ a + sin. (-b) + c ⊣
    ↑
```

L-list	M-matrix		L-list	M-matrix
⊢	R1 ∅ - b	⇒	⊣	R1 ∅ - b
c			c	
+			+	
R1			b	
.			.	
R_β = + sin			R_β = - sin	

$\therefore$ a + sin. (-b) + c ⇒ a - sin. b + c

Example 5.5 Substitution for a trigonometric function with a negative argument, the negative of the function with a positive argument.

If $p(R_\beta)$ is equal to $p(*)$, the trigonometric function is an operand in a *-string. A tag, T_S, is set. This in turn will cause the operator preceding the *-string to be changed before the string is entered in the M-matrix. At that time the tag, T_S is cleared, (see Example 5.6). Or, finally (i.e., when $p(R_\beta) < p(+)$ or $p(*)$), trigonometric function with a positive argument is entered in the M-matrix and a unary minus is inserted below the row-reference to the function in the L-list, (see Example 5.7).

Input String

$\vdash$ a + b * sin. (-c) + d $\dashv$

L-list	M-matrix		L-list	M-matrix
$\dashv$	R1 $\emptyset$ - c		$\dashv$	R1 $\emptyset$ - c
d			d	
+		$\Rightarrow$	+	
R1			c	
.			.	
sin			sin	
	R_β = *			T_S = 1, R_β = *

L-list	M-matrix		L-list	M-matrix
$\dashv$	R1 $\emptyset$ - c		$\dashv$	R1 $\emptyset$ - c
d	R2 sin . c		d	R2 sin . c
$\Rightarrow$ +			+	
R2		$\Rightarrow$	R2	
*			*	
b			b	
R_β = +	T_S = 1		R_β = -	T_S = 0

$\therefore$ a + b * sin. (-c) + d $\Rightarrow$ a - b * sin. c + d

Example 5.6 Setting and clearing T_S to change the sign of a *-string

Input String

⊢ a + b * (sin.(-c) + d) + e ⊣

L-list	M-matrix	L-list	M-matrix
⊣	R1 ∅ - c	⊣	R1 ∅ - c
e		e	R2 sin . c
+		+	
)		)	
d		d	
+		+	
R1		R2	
.		⊡	
sin			

R_β = (R_β = (

∴ a + b * (sin. (-c) + d) + e ⇒ a + b * (-sin. c + d) + e

Example 5.7 Use of unary minus in replacing the trigonometric function of a negative argument with the negative of the trigonometric function with a positive argument.

SECOND LEVEL OF TRIGONOMETRIC VALUE ALGORITHM The trigonometric value algorithm is also a subroutine within the Ershov algorithm. It is entered whenever a trigonometric, logarithmic, or exponential function with zero argument is about to be entered in the M-matrix. If the function is a cosine or an exponential, the value 1 is substituted for the function (see Example 5.8). If the

function is sine, tangent, or secant, the value ∅ is substituted for the function. After these substitutions are performed, an exit is made. If the function is cotangent, cosecant, or a logarithmic function, an error is indicated and the simplification is terminated.

Input String:

```
|- a + cos. ∅ + c -|
     ↑
```

L-list	L-list
-|	-|
c	c
+	+
∅	1
.	
cos	

Example 5.8 Substituting a value for a trigonometric function of zero.

VI. THE ZERO-ONE STAPLIFICATION ALGORITHM

The zero-one simplification routine is entered whenever a triple, $\eta_\alpha R_{\beta+1} \eta_{\alpha+1}$ where $R_{\beta+1}$ is "+", "-", "*", "/", or "P", is to be entered in the matrix. If neither η_α nor $\eta_{\alpha+1}$ is a zero or one, an exit is made.

If η_α is a zero and $R_{\beta+1}$ is p-higher than "+", the triple is replaced by $\emptyset$. If $R_{\beta+1}$ is "+", the triple is replaced by $\eta_{\alpha+1}$ and if $R_{\beta+1}$ is "-", by, $\emptyset \boxminus \eta_{\alpha+1}$. The identities which are applied are as follows:

for all a, elements of field K.

1. $\emptyset * a \Rightarrow \emptyset$
2. $\emptyset / a \Rightarrow \emptyset$ $\quad R_{\beta+1}$ is higher than "+"
3. $\emptyset \; P \; a \Rightarrow \emptyset$
4. $\emptyset + a \Rightarrow a$ $\quad (R_{\beta+1}$ is "+")
5. $\emptyset - a \Rightarrow -a$ $\quad (R_{\beta+1}$ is "-").

Whenever η_α is a one, if $R_{\beta+1}$ is p-higher than "*", the triple is replaced by one, $R_{\beta+1}$ is "*", by $\eta_{\alpha+1}$. Otherwise an exit is made. The identities are as follows:

for all a, elements of field K,

1. $1 / a \Rightarrow 1/a$
2. $1 + a \Rightarrow 1 + a$
3. $1 - a \Rightarrow 1 - a$
4. $1 * a \Rightarrow a$ $\quad (R_{\beta+1}$ is *)
5. $1 \; P \; a \Rightarrow 1$ $\quad (R_{\beta+1}$ is p-higher than "*")

Next, if $\eta_{\alpha+1}$ is zero and $R_{\beta+1}$ is $\flat$-equal to "+", the triple is replaced by η_{α}. If $R_{\beta+1}$ is $\flat$-higher than "*", the triple is replaced by one and if $R_{\beta+1}$ is "*" the triple is replaced by zero. If $R_{\beta+1}$ is "/" an error is indicated and the processing of the string is terminated. The appropriate identities are as follows:

for all a, elements of field K,

1. $a + \emptyset \Rightarrow a$ ($R_{\beta+1}$ is equal to "+")
2. $a - \emptyset \Rightarrow a$
3. $a \; P \; \emptyset \Rightarrow 1$ ($R_{\beta+1}$ is $\flat$-higher than "*")
4. $a * \emptyset \Rightarrow \emptyset$ ($R_{\beta+1}$ is equal to "*")
5. $a / \emptyset \Rightarrow \infty$ ($R_{\beta+1}$ is equal to "/"

(∞ is not an element of the field K, therefore an error is indicated).

Finally, if $\eta_{\alpha+1}$ is one and $R_{\beta+1}$ is $\flat$-equal to "+", no simplification is possible, so an exit is made. Otherwise the triple is replaced by η_{α}. The appropriate identities are as follows:

for all a, elements of field K.

1. $a + 1 \Rightarrow a + 1$ ($R_{\beta+1}$ is $\flat$-equal to "+")
2. $a - 1 \Rightarrow a - 1$
3. $a * 1 \Rightarrow a$
4. $a / 1 \Rightarrow a$
5. $a \; P \; 1 \Rightarrow a$

VII. OTHER THINGS TO BE DONE IN THIS AREA There are many directions along which future work in this area can proceed. Some of the things yet to be seem to fall into three classifications: (1) an extension to the existing algorithms, (2) an algebraic integration algorithm, and (3) an executive program to tie together the different algorithms for factoring, multiplication, etc., to give the user the option of which algorithms to apply and the order in which they are to be applied; and to evaluate the numerical value of the resulting expression.

Some of the extensions that need to be added are: (1) the ability to substitute an expression for a variable, (2) the simplification of logorithmic expressions by application of the identity, log (a*b) = log a + log b, (3) the simplification of exponential expressions such as $e^x * e^y \Rightarrow e^{x+y}$, (4) the simplification of expressions involving imaginary numbers, (5) the selection as to the order of factoring, (6) an extension to take care of infinity where possible and (7) the extensions of the existing trigonometric algorithms. The substitution of an expression for a variable can easily be done by first entering the expression in the M-matrix and then substituting its reference for the variable each time the variable is encountered during the processing of the input string. The simplification of logarithmic and exponential expressions would entail some additions to the trigometric multiplication routine. These can easily be made by inserting a few checks for "EXP" and

"LOG" in addition to the rules for handling the products and/or powers of these expressions. The simplification of expressions involving imaginary numbers would require that the imaginary unit have a unique representation and an algorithm to recognize and substitute for the powers or products of the imaginary unit the appropriate values, -1, +1, +i, or i.

An extension of the selection of the order of factoring requires more work and thought. A check for a general class of expressions with a specified operator is relatively simple, but an algorithm to check for a smaller sub-class of this general class requires more thought. The extension to take care of infinity requires a unique representation for infinity and an additional algorithm to substitute infinity for any sum or product of terms including infinity and also to set an expression with infinity in the denominator to zero. The extensions to the existing trigonometric algorithms are straightforward. At present the trigonometric multiplication algorithm processes only products of sines and cosines; so an extension for the other trigonometric functions is needed. Since the functions cotangent, secant, and coscant can be expressed in terms of sines and cosines, this extension can be accomplished simply by a temporarily representing these functions in terms of sine and cosine. An extension for the hyperbolic trigonometric functions involves the rules for the products of these functions and a few other additions to the existing trigonometric multiplication algorithm. This should be very easily accomplished. The existing trigonometric value and the trigonometric sign algorithms are concerned

only with curcular trigonometric functions. The extension of these to take care of hyperbolic trigonometric functions would be very simple.

Most present approaches to algebraic integration have used a "table-look-up" technique. Another method of approach which has been under preliminary investigation is the use of operational techniques for the antiderivative operator, D^{-1}. In this method the operational formulae are applied to the expression to be integrated and the resulting expression will contain the operators D^k where k is a positive integer. The parts of the expression operated on by D^k are then differentiated k times yielding the algebraic expression for the integral (antiderivative) of the original expression.

This method can be applied to a function, F, of the form: "$F = \sum_{p=1} G_p H_p$ where G_p is a polynomial, for each p, of degree M_p, and where H_p is a function whose first $M_p + 1$ integrals are known", [3] to obtain an exact solution.

When a function can be approximated by a function F of the form described above, to within a specified degree of accuracy, an inexact integrated expression can also be obtained by applying this method to the approximating function. Some examples of these functions are "sin kx, cos kx, sinh kx , cosh kx...for $k < 1$, and log kx for $x > 1$, and e^{kx} for $k < 1$."

The development of an algorithm for algebraic integration by this method will require a tremendous amount of work. Investigating further the classes of expressions that can be handled in this

way would be the first step. The executive program will entail much additional work and thought. Currently, there are essentially two effective ways to incorporate these algorithms into a user's package. The preferred package depends on the individual installation. If the installation is such that the user can interact with the machine by a console typewriter or similar device, an executive program of the following type would be best. The executive program would contain all the algorithms and would execute one or more of them according to parameters set by the user. The notation for specifying these parameters should be kept as simple as possible. With this type of set up, the user could apply one or more algorithms to his input string, look at this result, and then decide what to do next. The second way to tie the algorithms together would be to incorporate them into a compiler. The user could then specify which algorithms to apply and the order in which to apply them during the course of a computer program. The user should also have the option of how many times an algorithm is applied to an expression or of checking the result of each application and determining if addition applications are needed. This is particularly important in the use of the multiplication algorithm. The user might want it completely multiplied out without having to determine how many applications this would require or he might desire only partial multiplication. Another option that should be included is the ability to apply two or more algorithms in a specified order a specified or computed number of times. This is important for expressions

envolving many products of trigonometric functions. The ideal combination and order would probably be (1) the trigonometric multiplication algorithm, (2) the automatic simplification algorithm, (3) the trigonometric sign algorithm, (4) the trigonometric value algorithm, (5) the zero-one simplification algorithm, and (6) the automatic simplification algorithm. An algorithm to substitute the numerical values for the operands and to evaluate the resulting expression is also needed. When these algorithms have been incorporated into an algebraic compiler, these options are immediately available.

REFERENCES

1. ERSHOV, G. P. Programming Program for the BESM Computer. Translated from the Russian by M. NADLER and Edited by J. P. CLEAVE. Pergamon Press, London-New York-Paris (1959).

2. HANSON, JAMES W., CAVINESS, JANE SHEARIN, and JOSEPH, CAMILLA. Analytic Differentiation by Computer. Comm. ACM 5 (June 1962).

3. Private communication with D. SHOWALTER, May 1965.