SCRATCHPAD/1

AN INTERACTIVE FACILITY FOR SYMBOLIC MATHEMATICS

J. H. Griesmer *
R. D. Jenks
IBM Thomas J. Watson Research Center
Yorktown Heights, N. Y. 10598

Summary. The SCRATCHPAD/1 system is designed to provide an interactive symbolic computational facility for the mathematician user. The system features a user language designed to capture the style and succinctness of mathematical notation, together with a facility for conveniently introducing new notations into the language. A comprehensive system library incorporates symbolic capabilities provided by such systems as SIN, MATHLAB, and REDUCE.

0. INTRODUCTION

The major goal of the SCRATCHPAD project has been the design and implementation of a symbolic mathematics facility which provides a mathematician user with a powerful algebraic capability, and which at the same time is as convenient to use as his pencil and paper. To achieve this goal, principal effort has been expended in four areas:

(i) The implementation of an experimental LISP system for both interactive and batch use permitting simultaneous access to a large number of LISP-based algebraic facilities;

(ii) the building of a library of symbolic facilities taking maximum advantage of developments in symbolic computations originating elsewhere;

(iii) the design and implementation of an extensible user language enabling a user to state his problem using notations approaching that of conventional mathematics;

(iv) the development of a flexible evaluator giving the user maximum control over the substitution and simplification mechanisms of the system.

SCRATCHPAD has been implemented in the LISP programming language using an experimental System/360 LISP system. The major features of this LISP system that enhance its capability for symbolic and algebraic computations are provisions for unlimited precision integer arithmetic and for accessing a sizable number of LISP compiled and assembled programs.

This latter capability has enabled significant portions of the following systems to be incorporated into the library so as to be simultaneously available to the SCRATCHPAD user:

MATHLAB [4, 5, 16] - Carl Engelman, MITRE Corporation

REDUCE [8 - 11] - Anthony Hearn, Stanford University and the University of Utah

On-line Simplification System [14, 15] - Knut Korsvold, Stanford University and the Norwegian Defence Research Establishment

Symbolic Mathematical Laboratory [17] - William Martin, MIT

Symbolic INtegration (SIN) [19] - Joel Moses, MIT.

An extensible language approach was adopted in the design of the SCRATCHPAD user language. The initial language presented to each user is called the "base language" and contains a set of basic syntactic constructs described by notations resembling those in conventional mathematics. These basic constructs may then be extended by an individual user in order that he may tailor the system to his particular needs.

The objectives in the design and implementation of the SCRATCHPAD evaluator were to achieve efficiency comparable to that of REDUCE2 yet to provide the generality and understandability of the FAMOUS system [7]. The evaluator design is built upon the concept of "replacement rules" and provides a basis for the manipulation of not only algebraic expressions but also inequalities and logical expressions.

The remainder of this paper is divided into four sections plus an appendix. Section 1 presents an overview of the system organization and discusses its library facilities and current input/output capabilities. The section concludes with a capsule view of the design and implementation of the evaluator.

Section 2, the main section in the paper, sketches the syntax and semantics of the base language. Other topics include user control of the evaluator and the manipulation of expressions.

Section 3 describes the design of the syntax extension feature and gives a brief description of the underlying facilities used for its implementation. The final section describes some future directions of the project. The appendix contains a sample conversation of the SCRATCHPAD/1 language in the EBCDIC alphabet, and includes a complete syntactic description of the base language.

* on leave during 1970-71 at the University of California at Berkeley.

1. SYSTEM ORGANIZATION AND CAPABILITIES

A user session with SCRATCHPAD consists of the sequential processing of user commands by the SCRATCHPAD supervisor. During such processing, four major component parts of the system are utilized:

an input translator to convert input strings into a form suitable for interpretation;
an output translator to convert expressions in an internal form to a two-dimensional format for output to the user;
a library accessible by the evaluator containing the bulk of the special purpose algebraic manipulation functions; and,
an evaluator to carry out the appropriate action indicated by the command.

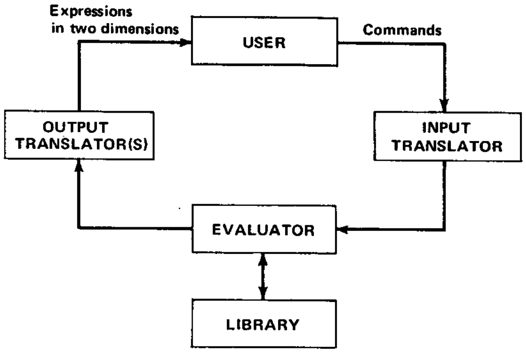

Figure 1. System Overview

(i) The input translator. User commands in the source language of SCRATCHPAD are accepted by the input translator and translated into a form suitable for interpretation by the evaluator. The current mode of input for interactive use is via an alphanumeric keyboard, either on an IBM 2741 terminal or on an IBM 2250 graphic display console connected to an IBM 1130 computer.

The SCRATCHPAD language described in Section 2 permits objects with associated two-dimensional graphics similar to that in customary mathematics, e. g.

$$x_i\,(t) \qquad {}_1y^{i}_{j,k} \qquad \Sigma_i\ f_i$$

For keyboard input all two-dimensional forms are linearized in a straight forward manner (Figure 2).

(ii) The output translator. A modified CHARYBDIS (Jonathan Millen, MITRE) [18] program produces line-by-line character output to form a two-dimensional image of a mathematical expression. The CHARYBDIS output translator may be used by a person accessing the system from either the 2741 terminal or the 1130/2250 display system.

The Picture Compiler (William Martin, MIT) [17] is also available with the 1130/2250 system; this system creates 2250 display commands, in the form of vector strokes and branches to character subroutines, to produce a two-dimensional display of mathematical expressions. The revision of the Picture Compiler for use in SCRATCHPAD was effected with the consulting help of Professor Martin and yielded an improved facility for breaking multiline expressions.

(iii) The library of SCRATCHPAD/1. The following symbolic capabilities, most of which were originally written for other systems, are currently available in SCRATCHPAD/1:

SIMPLIFICATION (R, M, K, S, N)
POLYNOMIAL GREATEST COMMON DIVISOR (R)
DIFFERENTIATION (R, N)
INTEGRATION (M, S)
POLYNOMIAL FACTORIZATION (M)
LAPLACE TRANSFORMS (M, K)
INVERSE LAPLACE TRANSFORMS (M)
SOLUTION OF LINEAR DIFFERENTIAL EQUATIONS (M, K)
MATRIX OPERATIONS (R, N)
SOLUTION OF SYSTEMS OF LINEAR EQUATIONS (N)

The letters "R, M, K, S" refer, respectively to REDUCE, MATHLAB, Korsvold, and SIN. These facilities are described in the documentation of the parent systems. The letter "N" refers to newly created facilities.

The following is indicative of the variety of problems for which the library has been utilized:

(a) generation and study of polynomials arising in graph theory and sorting;
(b) inversion of transition matrices resulting from a problem in data compression;
(c) symbolic differentiation and substitution required in a study of wave propagation in an elastic media;
(d) symbolic triple integrations arising in queuing theory;
(e) solution of an eight-dimensional system of linear symbolic equations arising from research on optimal difference formulae;
(f) investigation of subdeterminants derived from transformations applied to systems of nonlinear ordinary differential equations.

(iv) The SCRATCHPAD evaluator. The design of the evaluator, which is enlarged upon in Section 2, is based on the following concepts:

Replacement rules, created by user commands, are name-value relationships between such entities as variables (or, functions) and expressions;
The environment denotes the set of all replacement rules in effect at a particular instant in time;
Evaluation consists of a systematic transformation of an entity such as an expression, as governed by the environment.

These concepts provide a foundation for both the manipulation of expressions and the definition of functions and variables. The manipulation of expressions in SCRATCHPAD consists of their evaluation in a changing environment, as when replacement rules for variables affecting expansion, factoring, etc. are altered. On the other hand, definitions of functions are formed by combining replacement rules into LISP-like conditional expressions.

Two representations of expressions are handled by the evaluator. The "external form" (LISP prefix form) is used to communicate with the various subsystems in the library and to represent expressions in replacement rules. In addition, expressions which are output to the user are automatically labeled and saved in an external form which mirrors the output format.

All algebraic computations are carried out in "internal form", the 'standard form' of REDUCE2, in order to gain efficiency in both speed and storage. On evaluation, a substitution obtained from a replacement rule is converted from external to internal form. In the case of a substitution for a variable, the internal form is stored in an "already-simplified" cell associated with the variable. The evaluator maintains a representation of a directed graph describing the dependency of variables and forms on one another in order that already-simplified forms may be used as long as possible.

Two-dimensional format	Input format	Rules for linearization
$x(u,v)$	x(u, v)	Functional arguments are separated by commas and enclosed in parentheses
x_i	x[i]	Scripts (if any) are enclosed in brackets with subscripts given first
$x_{i,j}$	x[i, j]	Multiple scripts at the same location are separated by commas
${}_{p}x^{k}_{i,j}$	x[i, j;k;;p]	Superscripts, presuperscripts, and presubscripts follow subscripts in that order with a semicolon used to separate each group from the next
$x^{j}_{i}(t)$	x[i; j] (t)	Functional arguments follow scripts
$x_{i_{k_{l}}}$	x[i[k[l]]]	Scripting may occur to any depth

Figure 2. Examples of FORMs and rules for linearization.

2. THE SCRATCHPAD BASE LANGUAGE

The SCRATCHPAD base language is designed primarily for interactive use by a mathematician unskilled as a programmer. In particular, it is not a programming language or a language for the specification of algorithms.

For its purpose, the language features the desirable characteristics of simplicity, uniformity, and succinctness. The language is experimental and in many ways unconventional. In its interactive design, it resembles APL [6]. Syntactically, it combines a natural style of notations from conventional mathematics with features derived from JOSS [21] and SYMBAL [3]. The semantic model bears closest resemblance to that in FAMOUS. The language has wide applicability; its design currently provides a framework for manipulation of all of the following:

(a) finite and infinite sums, products, and sequences
(b) relations such as equations and inequalities
(c) arbitrarily indexed variables, functions, and operators
(d) sets
(e) arrays of arbitrary dimension.

The language is described in the following nine sections. Section 2. 1 presents the hierarchy of syntactic constructs ending with COMMAND, the name given to each distinct message issued from the user to the system. Sections 2. 2 and 2. 3 elaborate on these constructs and present an intuitive meaning of a COMMAND as that of an "assertion". In Section 2. 4, the notion of an assertion is made more precise by the description of the concepts of replacement rule, environment, and evaluation. Sections 2. 5 through 2. 8 describe the manipulation of expressions, iteration of COMMANDS, and user control over evaluation. Section 2. 9 describes a procedural language facility which enables symbolic subroutines to be created.

2. 1 Syntactic Constructs

Our purpose here is to present an overview of the syntax of the language, not to discuss inessential syntactic details. Accordingly, we will present all examples in their two-dimensional format using an expanded character set. For actual input to the system, the syntactic constructs are one-dimensional strings currently using the EBCDIC alphabet as described in Table I in the appendix.

PRIMITIVEs are the building blocks of the language; they are of five types:

INTEGERs	31	12345678909876543 21
VARIABLEs	x	i
FORMs (Figure 2)	$x(i)$	x_i
VECTORs	$\{1,2,1,2\}$	$\{\{1,2\},\{1,2\}\}$
SETs	$\{i \mid x(i)=1\}$	$\{i \mid x(i) = 1 \;\&\; i > 2\}$

An OPERATOR-FORM contains one of the four prefix operators (Figures 3): Σ, Π, $\int$, ∂ .

	two-dimensional format	input format
indefinite integral	$\int_x f(x)$	int [x] f(x)
definite integral	$\int_{x=0}^{1} f(x)$	int [x=0; 1] f(x)
summation	$\Sigma_{i=0}^{\infty} f_i(x)$	sum [i=0; inf] f[i] (x)
product	$\Pi_{i=0}^{\infty} f_i(u)$	prod [i=0; inf] f[i] (u)
ordinary derivative	$\partial_x f(x)$	df [x] f(x)
partial derivative	$\partial_{x,y,y} f(x,y)$	df [x, y, y] f(x, y)
or:	$\partial_{x,y}^{1,2} f(x,y)$	df [x, y; 1, 2] f(x, y)

Figure 3. Examples of OPERATOR-FORMS

EXPRESSIONs are the strings manipulated by the user. An EXPRESSION is either a PRIMITIVE, an OPERATOR-FORM, or any syntactically allowed combination of PRIMITIVEs, OPERATOR-FORMs, operators (Figure 4), parentheses, and quote marks, e. g.

$$x + y*(1 - 'u(t)'^2)^{1/2} \qquad \sum_{i=1}^{n} \frac{h^i}{fact(i)} *\partial_x u$$

Ellipses (...) may also be used in sums, products, and sequences to indicate missing terms, e. g.

$$1+2+...+n \qquad x_1 * x_2 *...*x_n \qquad \{1, 2, ...\}$$

STATEMENTS are the fundamental constructs for making definitions and declarations. They consist of

(1) a left-part: usually a VARIABLE or a FORM;
(2) a relator: >, ≥, =, ≤, <, ∊ ; and
(3) a right-part: an EXPRESSION, e. g.

$$f_i(x) = x^i \qquad x > 0 \qquad i \in \{1, 2, ...\}$$

Boolean operators:

∨	boolean-or	(n-ary)
∧	boolean-and	(n-ary)
¬	boolean-not	(unary)

Relational operators:

<, ≤, =, ≥, >, ∊ (binary)

Arithmetic operators:

+	unary plus; n-ary plus
-	unary minus; binary minus
*	multiplication (n-ary)
/	division (binary)
//	integer-division (binary)
**	exponentiation (binary)

Figure 4. Primitive operators in the system

A COMMAND is normally an "assertion" consisting of one or more STATEMENTs: a main STATEMENT optionally followed by one or more of qualifying STATEMENTs with commas separating, e. g.

$$f_i(x) = x^i, \qquad x > 0, \qquad i \in \{1, 2, ...\}$$

Assertions are always associated with the left-part of the main statement in the COMMAND, e. g. the above COMMAND is called an "assertion on $f_i(x)$".

Other types of COMMANDs such as "syntax extension commands" discussed in Section 3 are not further discussed here.

2.2 The PRIMITIVEs

INTEGERs

INTEGERs consist of a string of digits of arbitrary length. The numbers used in the SCRATCHPAD language are always rational numbers. Numerical calculations on rational numbers use unlimited precision integer arithmetic and therefore always remain fully accurate within the storage limits of the machine.

The INTEGERs are basic constants in the SCRATCHPAD language with 1 and 0 denoting the boolean constants "true" and "false" respectively. The number 0 also denotes the empty SET and the empty VECTOR.

VARIABLEs

VARIABLEs serve several purposes. They may denote user's variables and constants; they may also be regarded as names for equations, relations, or sets of objects. In addition, several VARIABLEs play special roles in the evaluation mechanism (Section 2.6).

An assertion on a VARIABLE is used to assign a range of values to that VARIABLE, e. g.

(a) $x > 0$	asserts x is greater than 0
(b) $y = 2$	asserts y equals 2
(c) $i \in \{1, 2, ...\}$	asserts i is a positive integer
(d) j ∊ integers	asserts j is an arbitrary integer.

An assertion has a global effect, e. g. the assertion "x > 0" holds until a subsequent assertion is given on x. If otherwise undeclared, a variable is understood to range over the set of all EXPRESSIONs. An assertion on a variable is removed by a statement with an empty right-part, e. g.

$$x =$$

A VARIABLE is treated as a constant (e. g. for differentiation) if its range is restricted to a single value. Three ways of asserting that p is a constant are:

(e) p = 2 asserts p denotes the constant 2
(f) p = "p" asserts p denotes itself
(g) p constant also asserts p denotes itself (a built-in extension to the base language)

FORMs

The notation f(x) is called a FORM with parameter x. In general, a FORM may have any combination of the following five types of parameters: functional arguments, subscripts, superscripts, pre-superscripts, and/or pre-subscripts (Figure 2). Any two forms which have the same leading variable name and the same combination of parameter types, are associated with the same "function descriptor", e. g. all of

$f_0(0)$ $f_0(1)$ $f_1(1)$ $f_i(j)$

refer to the same function descriptor $f_*(*)$, and are independent of the following FORMs:

f(0, 1) f(*, *)

which are associated with function descriptors:

$f_{0,1}$ $f_{*,*}$

FORMs may be used to represent parameterized objects, such as f(i), f_i, log(x), sin(x), etc. Certain special FORMs however are reserved for directly accessing the SCRATCHPAD library functions, e. g. "integrate (u, x)" may be issued to calculate the indefinite integral of u with respect to x.

Assertions on FORMs are used to describe function definitions, e. g.

(h) i(x) = x asserts i(a) = a for all a
(i) h(x, y) = (x>y) asserts h(a, b) = 1 if a>b else 0 if $a \leq b$

The "a" and "b" were chosen here arbitrarily to represent the dummy variables used by the system to store the above assertions. In SCRATCHPAD/1, assertions on FORMs are restricted to those having "=" as the RELATOR between the FORM and the EXPRESSION. Thus, for example, the assertion

g(x) > f(x)

is not currently allowed.

By convention, the range over which the assertion on a FORM is valid is indicated by the use of NUMBERs and VARIABLEs with appropriate preassigned ranges as parameters. In (h) and (i) above, for example, we have assumed that x and y have unrestricted range. As a more illustrative example, given the assertions "x > 0" and "i in {1, 2,... }":

(j) $f_i(x, x) = x^i$ asserts that $f_a(b, c)=b^a$ when b>0, b=c, and a is a positive integer

The ranges of parameters may also be given by qualifying statements following the main statement, e. g.

(k) g(0, x)=x, x > 0 asserts that g(a, b)=b when a=0 and b>0

Qualifying statements may be regarded as assertions themselves except for one important difference: qualifying assertions only affect the statements to their left in the COMMAND in which they appear. In particular, the qualifier "x > 0" in COMMAND (k) above has no effect on the evaluation of x before or after COMMAND (k) is issued.

Successive assertions on FORMs are "stacked" with new assertions given precedence. Through assertions the user is able to give piecewise definitions of functions:

k(x, y)=0, x < y asserts k(a, b) = 0 when a < b
k(x, y)=1, x > y asserts k(a, b) = 1 when a > b

and, recursive definitions of functions:

f(0)=1 asserts f(a) = 1 when a=0
f(1)=1 + x asserts f(a) = 1 + x when a=1
f(p)=x*f(p-1)-y*f(p-2), $p \in \{2,3,...\}$
asserts f(a) = x*f(a-1)-y*f(a-2) when $a \in \{2,3,...\}$

Assertions may be either selectively removed:

f(0) =

or totally removed:

f(x) = , x =

VECTORs

VECTORs serve two general needs: (1) to denote finite and infinite sequences (Figure 5, Examples 1-6), and, (2) to denote the mathematical objects of vectors, matrices, and tensors (Figure 5, Examples 7-10).

The ellipsis "..." may be used between elements according to rules which will be incompletely described here. The sequence denoted by {i, j,... , k} is i, i+(j-i), i + 2*(j-i), etc. up to k=i + n*(j-i) for some numeric or symbolic quantity n. The only restrictions here are that i, j, and k be distinct and that k-i be some multiple of the step j-i. The sequence {i,... , k} is understood to have i+1 as its second member (Examples 2-4). Any number of ellipsis expressions may be given (Examples 4, 5). Arbitrarily complicated EXPRESSIONs are allowed in ellipsis expressions provided they mutually exhibit one linearly varying quantity, e. g. in the current system, an ellipsis expression such as that in Example 6 is recognized whereas one such as

$$\{a^2_{x_1}, \ a^3_{x_2}, \ ...\}$$

is not.

Sequences with non-linearly varying indices may be given by use of a function valued subscript, e. g. if x_i denotes the Fibonacci sequence 1, 2, 3, 5, 8,... for i=1, 2,... respectively; the successive elements in

Example 6 are then

$$a^1_1,\ a^2_2,\ a^3_3,\ a^4_5,\ a^5_8,\ \text{etc.}$$

Examples 7-10 illustrate how VECTORs are used to describe vectors and matrices.

The elements of VECTORs are consecutively "indexed" normally starting with 1. For example, the first element of $\{1,3,12,13,\ldots\}$ is 1, the second element is 3, and its nth element $(n \geq 3)$ is 9+n.

It is also possible to give an arbitrary initial index and to leave VECTOR elements unspecified (Examples 8-10). Various examples and uses of VECTORs are described throughout the appendix (in particular, see Sections 8 and 11).

Example 1. A sequence of 5 elements:

$\{u/x,\ 1+x,\ -5,\ 12,\ p(x)\}$

Example 2. The sequence of the first five positive integers:

$\{1,2,3,4,5\}$ $\{1,2,\ldots,5\}$ $\{1,\ldots,5\}$

Example 3. The infinite sequence of positive integers:

$\{1,2,\ldots,\infty\}$ $\{1,2,\ldots\}$ $\{1,\ldots\}$

Example 4. The sequence of integers from 1 to k excluding j:

$\{1,2,\ldots,j-1,j+1,\ldots,k\}$

Example 5. The sequence u/v, -5, the positive integers exceeding 10, and the negative integers below -43:

$\{u/v,\ -5,\ 11,\ 12,\ \ldots,\ \infty,\ -44,\ -45,\ldots\}$

Example 6. An infinite sequence of multi-indexed forms (assume i to range over $\{1,2,\ldots\}$):

$\{a^1_{x_1},\ a^2_{x_2},\ldots\}$ $\{i{:}a^i_{x_i}\}$

Example 7. A vector of two elements each of which is a vector of 4 elements, i.e., a matrix of 2 "rows" and 4 "columns":

$\{\{a,b,c,d\},\ \{1,\ -12,\ u/v,\ p\}\}$

Example 8. A vector of length 2n+1, starting index -n, and all but 0 element unspecified:

$\{-n{:},\ 0{:}1,\ n{:}\}$

Example 9: An infinite vector with initial index -2, and whose elements with index -2, 5, 13, 15, ... are 1; all other elements are undefined:

$\{-2{:}1,\ 5{:}1,\ 13{:}1,\ 15{:}1,\ldots\}$
$\{k{:}1\}$ where $k \in \{-2,\ 5,\ 13,\ 15,\ \ldots\}$

Example 10: Three n by n matrices (assume i and j have been asserted to range over $\{1,2,\ldots,n\}$) (n may be left unspecified):

$\{i{:}\{j{:}\}\}$ (all elements unspecified)
$\{i{:}\{j{:}i{=}j\}\}$ (unit matrix)
$\{i{:}\{j{:}1/(i+j-t)\}\}$ (generalized Hilbert matrix)

Figure 5. Examples of VECTORs

SETs

A SET may be used to make more complicated assertions on VARIABLEs, e.g.

$S=\{x \mid x>0 \,\&\, f(x)>0\}$ asserts S is the set of all x such that $x>0$ and $f(x)>0$
y in S asserts $y>0$ & $f(y)>0$

Operations on and between SETs such as union, intersection, and complementation are planned for a later version of the system. Additional details on SETs and their evaluation appear in Section 10 of the appendix.

2.3 Expressions and Pattern Matching

EXPRESSIONs denote the most general right-part of a statement. They are built up from the following hierarchy of constructs: PRIMARY, FACTOR, TERM, ALGEBRAIC EXPRESSION, and CONDITIONAL EXPRESSION (Figure 6).

(1) PRIMARY h $\quad fact(i)$ $\quad u(t)$ $\quad \partial^i_t f(t)$ $\quad (1-u(t)^2)$

(2) FACTOR h^i $\quad t^{-1}$ $\quad t^2$ $\quad u(t)^2$

(3) TERM $\dfrac{h^i \partial^i_t f(t)}{fact\,(i)}$ $\quad t^{-1} * (1-u(t)^2)$

(4) ALGEBRAIC EXPRESSION $\dfrac{h^i \partial^i_t f(t)}{fact\,(i)} + t^{-1} * (1-u(t)^2)$ $\quad at^2+bt+c$

(5) CONDITIONAL EXPRESSION $\dfrac{h^i \partial^i_t f(t)}{fact\,(i)} + t^{-1} * (1-u(t)^2)$ if $t>0$

(6) EXPRESSION $\dfrac{h^i \partial^i_t f(t)}{fact\,(i)} + t^{-1} * (1-u(t)^2)$ if $t>0$ where $u(t)=at^2+bt+c$

Figure 6. The parts of speech associated with EXPRESSIONs, listed in order of increasing generality

The most general left-part of a statement currently handled is a TERM. For example, all of the following are legitimate statements:

```
(z + phi (x) )     = psi (z)
sin (x) **2        = 1 - cos(x)**2
'a*(b+c)**3'       = d
'a*(b+c)**3'*x     = d*x
```

Here x and z are assumed to have arbitrary range. These statements are again assertions but in this case are placed in the special category called "pattern matching rules". Pattern matching rules are applied independently of other types of assertions in a separate phase of evaluation (see below).

Quote marks are used to surround phrases which are to match exactly. In the third example above, a match will be found in

$$\ldots + a(b+c)^3 + \ldots \quad \text{but not in} \quad \ldots + a\, c^2 (b+c)^3 + \ldots$$

To cause a match in the second case, the assertion in the fourth example must be used.

With some exceptions, arbitrary EXPRESSIONs are allowed as parameters in FORMs on the left-hand side of an assertion, e. g.

$$f(g(h(x)), x) = 0, \qquad x > 0$$

will match any FORM of f(*, *) for which the parameters have the explicit form "g(h(b))" and "b" where b denotes any expression which can be shown to be positive. Pattern matching is applied recursively, e. g. given the assertion:

$$\cos(x + \mathrm{phi}(y)) = \mathrm{mu}(y)$$

then the EXPRESSION: "cos (phi (cos(r + phi(s))) + t)" will evaluate to:

$$\mathrm{mu}(\mathrm{mu}(s))$$

One restriction is that only binary EXPRESSIONs in + and * are allowed: e. g. "g(a+b+c) = 5" is not allowed.

2. 4 Evaluation and Environment

The meaning of an assertion is embodied in the notion of a "replacement rule", e. g.

x = 2 means: "replace x by 2"

Replacement rules are stored internally with a VARIABLE or function descriptor referenced in the left-part of a STATEMENT. For example, the COMMAND "$f_i(x)=x^i$, x>0, i∈ {1, 2, . . . }" creates a replacement rule for $f_*(*)$ in the form of a 3-tuple:

(= (b>0 & a is a positive integer) b**a)

The general form of a replacement rule is:

(RELATOR CONDITION EXPRESSION)

The RELATOR is the top-level infix operator of the assertion. The CONDITION part is either empty or contains a representation of a "when" phrase describing conditions on standardized "dummy parameters". The EXPRESSION part corresponds to the right-part of the assertion and represents a substitution in terms of these dummy parameters.

The totality of replacement rules which are applicable at any one instant in time is called the "environment of evaluation". At the beginning of a user's session, VARIABLEs and FORMs have no preassigned replacement rules and evaluate to themselves. As the session progresses, the user builds the environment by the introduction of replacement rules through assertions.

Evaluation of VARIABLEs, FORMs, and EXPRESSIONs involves their transformation according to the current environment. Evaluation of VARIABLEs is carried out by recursive examination of replacement rules, e. g. given the assertions:

x = y; y = z; z = w; w = 2;

the value of x is obtained by evaluating y, etc. until the 2 turns up on evaluation of w. This method of evaluation thus consists of continuous substitution until no more substitutions can be made.

The evaluation of FORMs is similar except that the actual parameters of the FORM are first evaluated, then substituted into the replacement rules for corresponding dummy variables. Each rule is examined in turn until one is found for which the "CONDITION" part is satisfied;; the "EXPRESSION" part of that rule is then evaluated and returned as a substitution for the FORM. If no such rule is found, the value of the FORM is taken to be the original FORM with its parameters evaluated.

The evaluation of EXPRESSIONs is similar to that of corresponding prefix forms in LISP. For example, the evaluation of

$$x + y + z$$

(where x, y, and z may be any syntactically allowed EXPRESSIONs) consists of evaluating x, y, and z to obtain values x', y', and z', then applying the system function, in this case SIMPPLUS, associated with the infix operator "+ ". The function SIMPPLUS carries out the symbolic sum of x', y', and z' and applies certain built-in replacement rules for simplifying the result. In general, all operators listed in Figure 4 have corresponding system functions which are applied to the arguments in a similar manner.

The EXPRESSION in the right-part of a COMMAND is evaluated twice before use: once on input when the replacement rule is created and then again when the replacement rule is applied.

An EXPRESSION surrounded with quote marks (') evaluates to itself; quote marks are used to delay or inhibit evaluation, e. g. given the assertion y=2,

the statement:	creates the replacement rule:	for which on later application:
x=y	replace x by 2	x evaluates to 2
x='y'	replace x by y	x evaluates to 2
x=''y''	replace x by 'y'	x evaluates to y

2. 5 Manipulation of EXPRESSIONs

Thus far we have shown only how the SCRATCHPAD language may be used for making definitions and declarations. In this and the following three sections, we will show how the language is also a convenient language for manipulation. The key idea throughout is the following:

> EXPRESSIONs are manipulated by their evaluation in a dynamically varying environment of replacement rules.

A COMMAND consisting of an EXPRESSION standing alone is evaluated, printed out, and assigned to the special VARIABLE WS called the "workspace". Thus the COMMAND

f(x) is equivalent to: ws = f(x)

and results in the evaluation of f(x) and the display and assignment of the result to the workspace. The contents of the workspace are not affected by other types of COMMANDs.

All non-trivial EXPRESSIONs entered into the workspace are given consecutive integer labels beginning with 1. For example, the above COMMAND might result in the printout

(101):

$$\left(\frac{1}{X\,(1 - AX)} + \frac{1}{(1 - X)\,(1 - AX)}\right) *G(X) + \frac{H(X)}{X\,(1 - BX)}$$

All labeled EXPRESSIONs are automatically saved on secondary storage and may be referenced by the special FORM WS(n), where "(n)" is a previous label issued by the system.

The contents of the workspace may be manipulated in whole or in part. The variable WS always refers to the entire workspace. In addition, the user may select, set, or modify any subexpression of the workspace without affecting the remainder of the workspace. For example, given the expression (101) in the workspace,

alter (1, 1), mcd = 1

causes the first term, first factor to be evaluated in the environment with MCD ("make common denominator") set to 1; the effect is to rewrite this term with a common denominator:

(102):

$$\frac{G(X)}{X\,(1 - X)\,(1 - AX)} + \frac{H(X)}{X\,(1 - BX)}$$

2. 6 User Control

We highlight the facilities for user control over the evaluation mechanism of SCRATCHPAD by describing the use of flags, special VARIABLEs, the EVMODE feature, and WHERE-clauses.

Flags and Special VARIABLEs. Flags are VARIABLEs referenced by the evaluator. Although no flags have initial values, all have initial meanings. For example, given no replacement rule for the flag XPS (expand powers of sums), all powers of sums of two or more terms are expanded out in full. On the other hand, given the assertion

XPS = n

where n is a positive integer,

$$(a+b+\ldots+c)^{m} \qquad (m > 0)$$

is expanded out for $m \leq n$. With a few exceptions such as XPS, the flags in SCRATCHPAD are presently identical to those in REDUCE2.

In addition, certain special VARIABLEs are used to control the formatting of expressions as in REDUCE2, e. g.

order={x, z, y} is used to order x ahead of z ahead of y in the output of polynomial products

factors={u, v} indicates that powers of u and v are to be factored out of expressions on output

The EVMODE feature. Thus far, we have in fact described only one of the phases of evaluation, namely the substitution phase. The entire evaluation of an EXPRESSION normally consists of the application of five independent phases in a specified order (Figure 7).

Phase Number	Description
1	Simplification with no substitution
2	Simplification with substitution
3	Expansion under flag control
4	Expansion in full
5	Pattern matching (outside/in)
6	Pattern matching (inside/out)
7	Rational simplification
8	Restructuring

Figure 7. Phases of Evaluation. Normal evaluation consists of the successive phases 2, 3, 5, 7, and 8.

The special variable EVMODE may be used to change the normal mode of evaluation. For example, the COMMAND

ws, evmode = {1, 4, 8, 1}

causes the current EXPRESSION in the workspace to be (a) simplified, (b) expanded out in full, (c) restructured, and (d) resimplified.

WHERE-clauses. We have already seen how qualifying statements alter the environment used in evaluating the main statement in a COMMAND. In addition, any EXPRESSION may be qualified by a WHERE clause in order to change the environment for the evaluation of that EXPRESSION, e. g.

... *(x where x = 0)* ...

evaluates to 0 regardless of the value of x in the environment. The WHERE-clause may also contain a single variable, e. g.

u where r

In this case, r is understood to be a name of a "rule-vector": a VECTOR of replacement rules to be invoked solely for the evaluation of u, e. g. r might have been previously defined by the assertion

$$r = '\{x=1, y > 0, i \in \{1, 2, \ldots\}\}'$$

or by the assertion

$$r = '\{ra,\ re,\ x > 0\}'$$

where each of RA and RE likewise evaluate to rule vectors. As another example, the WHERE-clause may be used to set the variable EVMODE in order to temporarily alter the evaluator, e. g.

$$\ldots +(u \text{ where } evmode = \{1, 2, 4, 2, 8\}) + \ldots$$

2.7 Iteration

We have seen above how the actual parameters used in an assertion on a FORM may be used to describe the range over which the assertion is to be valid. For example, if i ranges over $\{1, 2, \ldots, 10\}$ then the assertion:

$$f_i = x^i$$

is assumed to hold over all $i \in \{1, 2, \ldots, 10\}$. It is natural and convenient to extend this range convention as follows.

Any assignment to the workspace containing a VARIABLE which ranges over a VECTOR is iterated over the successive elements of that VECTOR. For example, if i ranges over the sequence $1, 2, \ldots, 10$ then, to display the first ten members of the sequence $\{p_1, p_2, \ldots\}$, one simply issues the COMMAND

$$p_i$$

If more than one variable is to be iterated, iteration variables are ordered lexicographically, e. g. given the assertions "$j \in \{1, 2, \ldots, i\}$" and "$i \in \{1, 2, \ldots, n\}$, the command

$$a_{i,j}$$

may be used to print out a lower triangular array represented by $a_{i,j}$, i. e. $a_{1,1}$, $a_{2,1}$, $a_{2,2}$, $a_{3,1}$, etc.

2.8 Operator-Forms

Four basic operators are provided in SCRATCHPAD/1: differentiation, integration, sum, and product.

The differentiation facility is substantially that of REDUCE2. Here the system provides the basic mechanisms for calculating total and partial derivatives of EXPRESSIONs with respect to variables and forms. Rules for partial derivatives of functions may be completely or incompletely specified, e. g. given the assertion

$$\partial_x f(x, y) = y$$

then the EXPRESSION: "$\partial_t f(x, y)$" will evaluate to:

$$y\ \partial_t x + (\partial_y f(x, y))\ (\partial_t y)$$

Replacement rules may also be given for partial derivatives of any order, e. g.

$$\partial_{x,y,y} f(0, 1) = phi(x,y) \text{ (or, } \partial^{1,2}_{x,y} f(0, 1) = phi(x, y))$$

Manipulation of integrals may be performed in one of two modes. In one mode, the user builds his own table of integrals in the form of replacement rules through assertions:

$$\int_{x=0}^{\infty} \frac{J_1(x\ t) * \sin(x)}{x} = t^{-1}(1-(1-t^2)^{1/2}) \text{ if } t>0\ \&\ t<1$$
$$\text{else } t^{-1} \text{ if } t > 1$$

On the other hand, with the special variable SIN=1, the evaluator will issue a call to the SIN facility in the SCRATCHPAD library to evaluate the integral. Replacement rules are normally scanned first before the library facility is called.

Sums may be finite or infinite and transform according to replacement rules:

$$(1) \qquad \Sigma_{i=0}^{n}\ c_{x,i} * c_{y,n-i} = c_{x+y,n}$$

The normal notation for products and sums is that which describes the total range of summation as in (1). Here, the variable i serves as a dummy index for the summation. Had i been asserted to range over $\{0, 1, \ldots, n\}$ then

$$\Sigma_{i=0}^{n} \text{ may be replaced by } \Sigma_i$$

In general, if i had been asserted to range over any VECTOR then the single subscript i may be placed on sum OPERATOR-FORMs to cause summation over the successive VECTOR elements. This convention provides a more efficient notation and is a device which allows sums over much more complicated ranges. For example, given the assertion

$$i \in \{1, 2, \ldots, j-1, j+1, \ldots, k\}$$

then

$$\Sigma_i f_i \text{ is equivalent to: } \Sigma_{i=1}^{j-1} f_i + \Sigma_{i=j+1}^{k} f_i$$

Sum OPERATOR-FORMs may be used interchangeably with corresponding ellipsis constructs, e. g.

$$f_1 + f_2 + \ldots + f_{j-1} + f_{j+1} + \ldots + f_k \text{ and } \Sigma_{i=1}^{j-1} f_i + \Sigma_{i=j+1}^{k} f_i$$

are equivalent.

Product OPERATOR-FORMs have similar properties as sum OPERATOR-FORMs .

2.9 Procedures

While the majority of the user's interaction might be carried out in a step-by-step manner, there are times when it is desirable to be able to execute a block of COMMANDs over and over. For this purpose, a facility is also provided which enables the user to create subroutines called "procedures" for subsequent execution by the system evaluator. Any COMMAND prefixed by a label of the form n.m for n and m integers, e.g.

1.40 s=y where x=a

is neither translated nor interpreted but rather stored by the system for later reference. The numbers n and m are called respectively the block and COMMAND numbers. After all constituent COMMANDs for a block have been issued, a subroutine may be created by the use of the special FORM "procedure(n)" in the right-part of a statement. The effect of this COMMAND is to copy, collate, and translate the constituent COMMANDs from block n into an executable subroutine. For example,

taylorseries (y, x, a, n)=procedure (1), n $\in$ {0, 1, ... }

is used to create the replacement rule:

replace taylorseries (a, b, c, d) by (contents of block 1) when d in {0, 1, ... }

On interpretation of a procedure, COMMANDs are executed in numerical order with two exceptions: the special statement "GO n" where n is a COMMAND number causes the interpreter to go to COMMAND n for the next step: (2) the statement "RETURN x" causes the interpreter to exit from the block; in this case, x is evaluated and returned as the value of the procedure.

The user may insert, delete, or change any COMMAND of a block as follows:

1.62 $y=x^2+2$ inserted between COMMANDs 61 and 63
1.30 deletes an existing COMMAND 30
1.40 s=s+h*(y where x=a) replaces COMMAND 40

Also,

procedure (1) = deletes all COMMANDs of block 1

Editing of blocks has no effect on previously created procedures. Examples of procedures are given in Section 13 of the appendix.

3. SYNTAX EXTENSIONS

It seems evident that no single computer language could ever hope to realize the requirements for mathematicians working in diverse areas. Their use of specialized notations is often extremely important and perhaps essential. It is therefore an important goal to enable the user to introduce and use his own specialized notations on-line.

To accomplish this objective, a translator writing system META/LISP [12] has been developed together with an extendable language facility META/PLUS [13]. META/LISP is another in the family of syntax-directed translator writing systems based on the META II model of V. Schorre [20]. Through META/LISP, input translators for several languages resident in the system have been produced; these include SCRATCHPAD; LPL, a higher-level LISP language; and META/LISP itself. The META/LISP facility enables sophisticated users to interactively modify existing input translators (Figure 8).

The extensible language facility, META/PLUS, enables a SCRATCHPAD user to effectively and conveniently extend the base language at the SCRATCHPAD source level, and without any of the formalisms required by META/LISP. For example, the notation in the base language for the absolute value of an EXPRESSION x is "absval (x)". If, however, the user wishes to use the notation |x|, then he may issue the command:

(2) "|x|" = "absval (x)", x expression

Here the x is used as a dummy variable to denote any EXPRESSION. Syntax extension commands produce compiled incremental changes to the base trans-

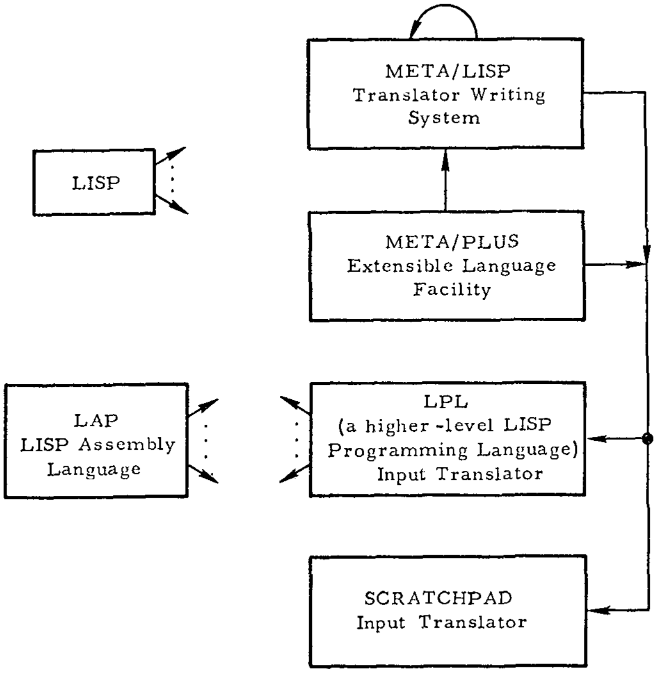

Figure 8. Components of the extendable system
(Arrows show direction of modification)

(a) META/LISP can alter any translator it produces including its own.
(b) META/PLUS can extend any translator produced by META/LISP.
(c) LISP, LAP, and LPL (a higher-level LISP Programming Language) may be used to define system functions and hence to extend the system.

lator and are removable at any later time:

"|x|" = , x expression

Effective use of the syntax extension feature requires knowledge of the constructs (parts of speech) of the SCRATCHPAD language (Table I). The most general form of a syntax extension command is

"N" = "D", <qualifier>, <qualifier>, ...

where N is a new notation, and D, its definition in terms of known constructs in the base language plus all extensions to date. Each <qualifier> consists of a dummy variable followed by the name of a part of speech: N (resp. D) contains exactly one (resp. at least one) occurrence of a dummy variable. Each dummy variable is assumed to range over all strings of the type with which it is associated.

All symbols and variables which do not appear in <qualifier>s stand for themselves. In particular, if no qualifiers are given, then the extension is called a "text-to-text extension" and is handled directly by the input translator.

· Syntax extensions are similar to replacement rules not only in their syntactic form, but also in the fact that they are "stacked" with new extensions having precedence over old ones.

Extensions never add parts of speech to the language; rather, they augment existing ones. The part of speech chosen by the system to be augmented is that which as near as possible will allow the new notation N to be given in precisely the same context as D.

Two significant features of META/LISP are (1) the ability to detect looping in the translator induced by an extension, and (2) the ability to add an extension of the type "x y" to part of speech x for any string y whatsoever. These features are both needed in order to handle some of the more useful extensions to the base language. For example,

"x!" = "fact (x)", x primary

requires that any PRIMARY x followed by a "!" will translate as "fact(x)". As "fact(x)" is a FORM, the system initially makes the extension to FORM. But this extension introduces a loop in the translator. The extension is then removed and replaced by an extension of the above special type on PRIMARY.

The system is able to guarantee that an extension will work immediately after completion. But since extensions are stacked, the system cannot guarantee that old constructs will not take on new meanings. For example, the extension

(3) "||x||" = "norm(x)", x expression

will lead to an extension to the part of speech FORM. If given following (2), both |x| and ||x|| will have their desired meanings. If, however, (3) is given before (2) then (3) is effectively covered up since "||x||" will then translate as "absval (absval(x))".

4. FUTURE DIRECTIONS

Input/Output Facilities

The chief lack in capability in SCRATCHPAD/1 is a graphical facility commensurate with that offered by the language and symbolic facilities. For indeed, the emulation of the user's scratchpad is impossible without a sophisticated facility for tablet input and manipulation of expressions on a display. Tablet input allows one to input two-dimensional expressions in random order, not strictly left-to-right as required by other devices. A graphical manipulation capability with a facility for pointing to subexpressions suggests a seemingly ideal way of handling manipulations which cannot be succinctly dealt with in any other reasonable way (Figure 9).

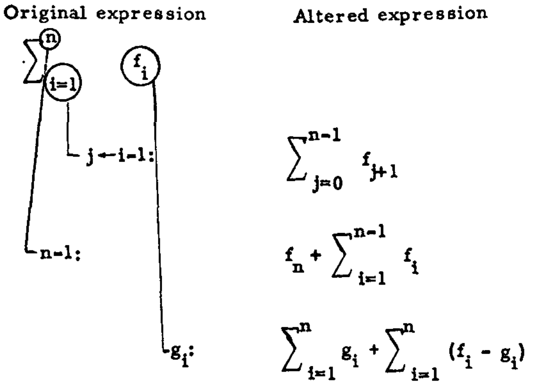
Original expression
Altered expression
$\sum_{i=1}^{n}$
f_i
$j \leftarrow i-1$:
n-1:
g_i:
$\sum_{j=0}^{n-1} f_{j+1}$
$f_n + \sum_{i=1}^{n-1} f_i$
$\sum_{i=1}^{n} g_i + \sum_{i=1}^{n} (f_i - g_i)$

Figure 9. A needed graphical capability. Given the original expression displayed on a screen, a user may wish to modify it in any number of ways. Changing the circled expression to an indicated expression alters the form of the expression without changing its mathematical content.

Manipulation of Inequalities and Boolean Expressions

One chief area heretofore ignored by algebraic manipulation systems is that of manipulation of inequalities. The SCRATCHPAD language, however, is designed to handle any type of two-sided relation. The implementation of appropriate facilities for manipulation of inequalities as well as for sets is planned for a later version of SCRATCHPAD.

The SCRATCHPAD evaluator model also suggests a basis for theorem proving. The manipulation of boolean expressions may be carried out in exactly the same way as the manipulation of algebraic expressions. Boolean expressions are transformed to 0 or 1, or to another boolean expression as a result of applications of system functions and replacement rules as shown in Sections 9 and 10 in the appendix. The proof or disproof of a theorem might be therefore established by stating a conclusion as a boolean expression, then manipulating the environment representing the hypothesis until the conclusion is reduced to 1 or 0 by successive transformations.

Extensions

The value of the syntax extension facility in the SCRATCHPAD system can surely be gauged only after substantial interaction with users of the system. The current design is adequate for allowing new commands to be introduced into the system, as, for example, to change the input language to that of REDUCE2. However, for extensions such as (2) for absolute value, it is also desirable to have an extendable output translator which will make the appropriate inverse transformation on output as well as to select the appropriate output format. The feasibility of extending META/LISP to deal with such extensions will be studied.

ACKNOWLEDGEMENTS

The dependence of the SCRATCHPAD/1 system on the work of others has already been mentioned in Sections 0 and 1 of this paper. Special acknowledgement must be made of the benefits of a continuing interaction with Professor A. C. Hearn of the University of Utah. In particular, Professor Hearn developed additional REDUCE2 facilities which were required in SCRATCHPAD/1.

The design and implementation of the language, evaluator, and syntax extension feature is due to R. D. Jenks. The collection and integration of the numerous sybsystems into the SCRATCHPAD library and the implementation of the unlimited precision arithmetic package are due to J. H. Griesmer. Modifications to the CHARYBDIS system were made by F. W. Blair. F. W. Blair is also responsible for the design of the experimental System/360 LISP system. The implementation and continuing improvement to the LISP system is the work of F. W. Blair, J. H. Griesmer, J. E. Harry and M. Pivovonsky.

REFERENCES

[1] Blair, F. W., Griesmer, J. H., and Jenks, R. D., "An interactive facility for symbolic mathematics," Proceedings of the International Computing Symposium, Bonn, Germany, 1970, pp. 394-419.

[2] Blair, F. W., and Jenks, R. D., "LPL: LISP programming language," IBM Research Report, RC 3062, September 23, 1970.

[3] Engeli, M. E., "User manual for the formula manipulation language SYMBAL," Computation Center, University of Texas at Austin, March 1968.

[4] Engelman, C., "MATHLAB: A program for online assistance in symbolic computations," Proc. 1965 Fall Joint Computer Conference, Volume 27, Part 2, Thompson Book Co., Washington, D. C. and Academic Press, Inc., London, 1967, pp. 117-126; also Proc. 1965 Fall Joint Computer Conference, Volume 27, Part 1, Spartan Books, Washington, D. C., 1965, pp. 413-421.

[5] Engelman, C., "MATHLAB 68," in Information Processing 68, A. J. H. Morrell, ed., North Holland Publishing Company, Amsterdam, 1969, pp. 462-467.

[6] Falkoff, A. C. and Iverson, K. E., "APL/360: user's manual," IBM Thomas J. Watson Research Center, August 1968.

[7] Fenichel, R. R., "An on-line system for algebraic manipulation," Project MAC Report MAC-TR-35 (Thesis), Massachusetts Insitute of Technology, Cambridge, Mass., December 1966.

[8] Hearn, A. C., "A user-oriented interactive system for algebraic simplification," in Interactive Systems for Experimental and Applied Mathematics, M. Klerer and J. Reinfelds, eds., Academic Press, New York, 1968, pp. 79-90.

[9] Hearn, A. C., "The problem of substitution," in Proceedings of the 1968 Summer Institute on Symbolic Mathematical Computation, R. G. Tobey, ed., IBM Boston Programming Center, Cambridge, Mass., June 1969, pp. 3-19.

[10] Hearn, A. C., "REDUCE2 user's manual," Stanford Artificial Intelligence Project Memorandum No. 133, Stanford University, Palo Alto, California, October, 1970.

[11] Hearn, A. C., "REDUCE2: A system and language for algebraic manipulation," these proceedings.

[12] Jenks, R. D., "META/LISP: An interactive translator writing system," IBM Research Report, RC 2968, July 1970.

[13] Jenks, R. D., "META/PLUS: The syntax extension facility for SCRATCHPAD," submitted

[14] Korsvold, K. "An on-line algebraic simplify program," Stanford Artificial Intelligence Project Memorandum No. 37, Stanford University, Palo Alto, California, November 1965.

[15] Korsvold, K., "An on-line program for non-numerical algebra," (abstract), Communications A. C. M., 9, (August 1966) p. 553.

[16] Manove, M., Bloom, S., and Engelman, C., "Rational functions in MATHLAB," in Symbol Manipulation Languages and Techniques, D. G. Bobrow, Ed., North-Holland Publishing Company, Amsterdam 1968, pp. 86-97.

[17] Martin, W. A., "Symbolic mathematical laboratory," Project MAC Report MAC-TR-36 (Thesis), Massachusetts Institute of Technology, Cambridge, Mass., January 1967.

[18] Millen, J. K., "CHARYBDIS: A LISP program to display mathematical expressions on typewriter-like devices," in Interactive Systems for Experimental and Applied Mathematics, M. Klerer and J. Reinfelds, eds., Academic Press, New York, 1968, pp. 79-90.

[19] Moses, J., "Symbolic integration," Project MAC Report MAC-TR-47 (Thesis), Massachusetts Institute of Technology, Cambridge, Mass., December 1967.

[20] Schorre, D., V., "META II, A syntax-directed compiler writing language," Communications A.C.M., 8, (10), 1965, p. 605.

[21] Shaw, J. C., "JOSS: A designer's view of an experimental on-line computer system," Proc. AFIPS 1964 Fall Joint Computer Conference, Spartan Books, Baltimore, Md., pp. 455-464.

APPENDIX

Sample Conversation Using the IBM 2741 Typewriter Terminal

Table I describes the input syntax for input to SCRATCHPAD from an IBM 2741 keyboard with an EBCDIC selectric ball. The following character substitutions are necessary for transliterating notations appearing in Section 2:

		substitutions
braces	{ , }	(,)
brackets	[,]	<,>
member symbol	∈	IN
relators	≤, ≥	<=, >=
boolean operator	∧, ∨	&, OR
prefix operators	Σ, Π, ∫, ∂	SUM, PROD, INT, DF

In addition, a blank must precede < and > when used as a relator or part of a relator, e. g.

x less than i is written: x < i not x<i

TABLE I

INPUT SYNTAX FOR SCRATCHPAD/1 BASE LANGUAGE
(for use with IBM 2741 and EBCDIC Selectric Ball)

1	INTEGER	d d*
2	VARIABLE	a a*
3	FORM	2 s [f] | 2 f
4	VECTOR	(5 [, {5 | ... }]*)
5	VECTOR ELEMENT	[[-] v:] 19
6	SET	(2 | 19)
7	PRIMITIVE	1|2|3|4|6
8	†OPERATOR-FORM	{DF|INT|PROD|SUM} s 11
9	PRIMARY	'19' | 7 | 8
10	FACTOR	9 [** 9]*
11	TERM	10 [t {10 | ... }]*
12	SIGNED TERM	[p] 11
13	ALGEBRAIC EXPRESSION	12 [p {12 | ... }]*
14	CONDITIONAL EXPRESSION	13 [IF 19 [ELSE 13]]
15	†EXPRESSION	14 [WHERE 20]
16	RELATION	15 [r 14]
17	NEGATION	[¬] 16
18	CONJUNCT	17 [& 17]*
19	BOOLEAN EXPRESSION	18 [OR 18]*
20	STATEMENT	[11 = | 2 r] 15
21	ASSERTION	20 [, 20]*
a	ALPHABETIC	A|B|C|D|E|F|G|H|I|J|K|L|M N|O|P|Q|R|S|T|U|V|W|X|Y|Z
d	DIGIT	0|1|2|3|4|5|6|7|8|9
f	FUNCTIONAL ARGUMENTS	(19[, 19]*)
p	PLUS-OPERATOR	+ | -
r	RELATOR	<|<=|=|>=|>|IN
s	†SCRIPTS	<{21 | ; }*>
t	TIMES-OPERATOR	*|/|//
v	SVAR	1 | 2

Notation:

terminal symbols appear in capital letters
braces { } are used for grouping;
vertical bar | is used for alternation;
square brackets [ ] are used to indicate optionality;
A* or [A]* means zero or more A's
† somewhat simplified

1) CONVERSATION

A conversation with SCRATCHPAD consists of the user issuing commands to the system; user input in lower case begins in column 5; system responses in upper case begin in column 1

```
    x=1                  commands confined to a single line need
                              no termination symbol
    x=1;x=1;x=1          otherwise, commands may be separated by
                              semicolons
    x   _                an underscore is used
    =   _                     to continue a command
    1                         to the next line
    x                    typing the name of VARIABLE or FORM
X:  1                         results in the display of its value
    x=&                  a syntactically incorrect command is typed
X=&                           back to the user with an underscore
  _                           under the offending character
    y                    VARIABLEs and FORMs initially evaluate to
Y                             themselves
    ws                   expressions standing alone are evaluated and
Y                             put into the workspace: ws
    y=2                  typeout occurs only when  ws  is set
    y                    typeouts of VARIABLE values are labeled
Y:  2                         if VARIABLE has substitution
    x=y                  the r.h.s. of a statement is evaluated
    x                         in the current environment
X:  2
    'y'+2                the value of a quoted expression is the
Y + 2                         expression itself
    ws                   further evaluation yields the value
4                             of the expression
    x='y'+2              quoting delays evaluation on assignment
    x
X:  4
    y=3
    x
X:  5
    x where y=1          WHERE-clauses alter the environment
3                             for evaluation
    r='(x=1,y=2)'
    x+y where r          when a WHERE-clause contains a single
3                             VARIABLE that VARIABLE is assumed
                              to evaluate to a VECTOR of statements
```

2) REPLACEMENT RULES

Commands create relationships between VARIABLEs (or FORMs) and EXPRESSIONs called "replacement rules". Replacement rules will be described here by the following prototype:

"(r) replace f by e when c"

where:

- r is one of the relators: <, ≤ , = , ≥ , > , and IN;
- f is a standardized representation of the VARIABLE/FORM;
- e is an expression, a substitution for f; and
- c is a conditional-expression which describes conditions under which a substitution will occur

The r is used to indicate the mode of evaluation in which replacement will result. Evaluation is normally in the (=)-mode; see Section 6 for examples of evaluation in other modes.

```
     x=0                   "(=) replace x by 0"
     x
X:   0
     y(1)=1                "(=) replace y(a) by 1 when a=1"
     y(1)                  replacement rule exists for argument 1
Y(1): 1
     y(0)                  but not for argument 0
Y(0)
     y(1)=                 rules are cleared by
     y(1)                      issuing a statement
Y(1)                           with an empty r.h.s.
     x > 0                 "(>)replace x by 0"; declares range of x ;
     X                     no substitution since evaluation is in (=) - mode
X
     f(x)=x                "(=) replace f(a) by a when a > 0"
     y=1                   with y=1:
     f(y)=0                "(=) replace f(a) by 0 when a=1"
     f(1)                  new rule takes precedence
F(1): 0
     f(2)                  but old rule remains
F(2): 2
     f(0)                  no rule for f(a) when a=0
f(0)
     x=                    clear x giving x arbitrary range
     f(x)=2*x              "(=) replace f(a) by 2*a"
     f(1)                  all previous rules overridden
F(1): 2
     x in (1,2,...,n) "(in) replace x by (1,2,...,n)"
     f(x)=x                "(=) replace f(a) by a when a in
     f(2)                       (1,2,...,n)" but not known if n > 2
f(2)                            so replacement rule is not used
     n > 2;f(2)            range of n known, so rule applies
F(2): 2
```

3) COMMANDS WITH QUALIFIERS

The general form of a command is that of a main statement optionally followed by a number of qualifying statements with commas separating. Statements are evaluated from right to left with each qualifying statement affecting only the environment of evaluation for statements to its left. Only the main statement creates a replacement rule; this rule is added to the environment at the conclusion of the evaluation of the command.

```
     x=1
     f(x)=3*x, x > 0 "(=) replace f(a) by 3*a when a > 0"
     f(2)                  left-most statement creates permanent
6                               replacement rule;
     x                     the x > 0 has no global effect: qualifiers
1                               create temporary replacement rules
     f(i)=g<i>-g<i-1>,_    which only affect the environment of
        g<i>=u(x-i*h), _   evaluation for statements to their
        i in (1,2,...)     left: "(=) replace f(a) by u(x-a*h)
                           =u(x-(a-1)*h) when a in (1,2,...)"
     f(1)                  the ordering of qualifiers in above example
U(X - H) - U(X)                 was essential
     y<i>=i, i in _        qualifiers are used to indicate range over
        (-5,12,18,20,...)  which the rule is valid
     p*q*r+p*r*s,_         or, to temporarily alter the environment
        factors=(p,q)           of evaluation
P R (Q + S)                but not to indicate global dependencies:
     x=0, t > 10                means: "(=) replace x by 0"
     t=-5                       not: "(=)replace x by 0 if t > 10"
     x                     To create the latter type of
0                               rule, IF-clauses must be used
```

4) IF-CLAUSES

An IF-clause is used to give conditional expressions or to specify additional conditions for replacement rules.

```
     x=                    clear x
     x=1 if t > 10         "(=) replace x by 1 when t > 10"
   x, t = 12               evaluate x in the environment where
1                               t equals 12
     f(x)=x                parameters of FORMs on l.h.s. cannot
CAN'T DO                        have environment-dependent ranges
     t > 10
     x=-1 if t < 10 "(=) replace x by -1 if t > 10"
     x                     old rule remains
1                               "replace f(a) by a when
     f(t)=t if t in s           t > 10 & t in s"
     f(x,y)=x**y if x > y,  x > 0, y > 0
     f(1,1)                "(=) replace f (a,b) by a**b
F(1,1)                     when a > b & a > 0 & b > 0"
     x=                    let x have arbitrary range
     sin(pi)               no trigonometric functions initially
SIN(PI)                         known to the system
     sin(x)=0 if absval(x/pi) in (0,1,...)
                           "(=) replace sin(a) by 0 when
     sin(2*pi)                  absval(a/pi) in (0,1,...)"
SIN(2 PI): 0
```

5) COMPOUND STATEMENTS

This section discusses several useful constructions which are provided as built-in extensions to the base language.

```
     a=1;b=1;c=1           multiple rules are separated by semicolons
     a=b=c=1               equivalent to the above
     b
B:   1
     (x,y,z)=2             the left-hand side of rule may be a
     x                          VECTOR of TERMs all receiving
X:   2                          identical treatment
     (x,y,z)>=0            equivalent to: x ≥ 0;  y ≥ 0;   z ≥ 0
     y                     no existent (=)-replacement rule for y
Y
     x>=y>=z>=0            equivalent to z ≥0; y≥ z;   x ≥ y;
                                the value of a statement p ≥q is the
                                left hand side p
```

6) INEQUALITITES (these facilities are not operable in SCRATCHPAD/1)

```
     t=u=v=                let t, u, v, be arbitrary
     (x,y) > 0             let x and y be positive
     x                     no "=" replacement rule exists for x
X
     >) x                  evaluation in the ( >)-mode is signaled by
0                               a > ) preceding the command
     t+x                   "(=) replace ws by t+x"
T + X
     >) ws                 "(=) replace ws by t"
T
     't=x'                 "(=)replace ws by t=x"(enter equation into ws)
T = X
```

```
     >) ws                 evaluating equality in the (>)-mode may turn
T > 0                              the equality into a relation
     (x=t)                 "(=) replace ws by t=x" (enter equation into ws)
X = T
     >) ws                 evaluation in (>)-mode only affects the
X = T                              r.h.s. of a relator
     't < x'
T < X
     >) ws                 (>)-evaluation will not work when substitution
T < X                              is not appropriate
     reverse(ws)
X > T
     >) ws where t >1
X > 1
     f(t) <g(t), t > 0  "(<) replace f(a) by g(a) when a > 0"
     <) f(t)
F(T)                       the sign of t is unknown
     <) f(x)               x is known to be positive
G(X)
     f(u*v) <=f(u)*f(v)
     <=) f(2*w)            "( ≤ ) replace f(a) by f(b)*f(c) when a=b*c"
F(2) * F(W)
     f(u+v)>=u+v
     cascade=0             force substitution to go one step at a time
     >) f(x+y)
X + Y
     >) ws                 x, y replaced by 0 since each
0                                  has been asserted positive
     u='g(x)=f(x+y)+y'     enter equation into u
     u                     load equation into workspace
U:   G(X) = F(X + Y) + Y
     >) alter(2,1)         selectively evaluate the 1st part of the
G(X) > X + 2Y                      2nd part of the workspace, i.e. f(x+y)
     >) ws
G(X) > 0
     cascade=              remove value from cascade
```

7) MORE GENERAL STATEMENTS

The most general left-hand side of a statement currently allowed is that of syntactic type TERM (roughly speaking, product of powers of VARIABLEs, FORMs, and expressions enclosed in prens).

```
     x=y=                  let x, y have arbitrary range
     sin(x)**2=1-cos(x)**2
     sin(x)*cos(x)=1/2*sin(2*x)
     (sin(u)+cos(u))**2
1 + SIN(2U)
     df<x>sin(x)=cos(x)
     df<t>sin(t)**2
SIN(2T)
     (p,q,r) constant
     p+q=5                 left-hand side is most generally a TERM
CAN'T DO
     (p+q)=2
     xps=0                 turn off "expand powers of sums" flag
     2 + (p+q)**2
6
     (2+p+q)**2            p+q must appear in parentheses for
               2                   substitution to occur
(2 + P + Q)
     r*(p+q)**4*x=10*x
     (r+(p+q)**4)**2       expansion occurs before matching
 2
R  + 276
     xps=                  turn on expand flag
     (p+q)**2
 2         2
P  + 2PQ + Q               prens lost hence no substitution
     f(g(x),x)=x           "(=) replace f(a, b) by b when a='g' (b)"
     f(u(x,y),g(x),h(y))=0, x > 0
                           "(=) replace f(a, b, c) by 0 when a='u' (d, e) &
                                   b='g'(d)&c='h'(e)&d > 0"
```

8) SEQUENCES AND ITERATION

The following range convention has been adopted for SCRATCHPAD/1: if an expression standing alone contains a VARIABLE ranging over a VECTOR then that expression is iterated over the successive elements of the VECTOR.

```
     i in (1,2,3)
     2*i
2                          "(=) replace ws by 2"
4                          "(=) replace ws by 4"
6                          "(=) replace ws by 6"
     i+j where j in (1,2,...,i)
2                          i=1; j=1

3                          i=2; j=1
4                          i=2; j=2   iteration VARIABLEs are
                                      ordered lexicographically
4                          i=3; j=1
5                          i=3; j=2
6                          i=3; j=3

     dots=1
     (a<1>,a<3>,...,a<2*n+1>)
A
 1                         with "dots = 1", the first, second,
                           and last elements of an ellipsis
A                          sequence are given
 3


...


A                          Otherwise, if the limit is symbolic or
 2 N + 1                   infinite, the user gives a carriage
     dots=                 return to obtain successive values
     (i:a<i>), i in (1,2,...,2*n+1)
A
 1
                           (carriage return)

A
 3

     .                     (stop iteration)

     p<1>=1                recursive definitions
     p<2>=1 + x
     p<i>=x*p<i-1>-d*p<i-2>, i in (3,4,...)
     i in (1,2,...)
     p<i>                  iteration on an infinite sequence
P : 1                      user gives carriage return to obtain
 1                                 successive values

P : X + 1
 2                         (carriage return)


   2
P : X  + X - D
 3

     .                     (stop iteration)
     j in (p<1>,p<2>,...)

     a<j>                  non-linear iteration results
A
 1

                           (carriage return)
A
 X + 1

                           (carriage return)
A
  2
 X + X - D

     .                     (stop iteration)
     j**2
1

                           (carriage return)
           2
1 + 2X + X
     .                     (stop iteration)
```

9) BOOLEAN EXPRESSIONS AND THEIR EVALUATION

A relation is a simple example of a boolean expression. The evaluation of relations containing one of the operators ≤, <, > , ≥ is exemplified by that of > .

```
     1 > 0                        this is an illegal STATEMENT
CAN'T DO
     (1 > 0); (0 > 1)   these are relations
1                                  1 means "true"
0                                  0 means "false"
     x=y=; (x > 0) clear x and y; is x > 0?
x > 0                          relation not reducible to 1 or 0
     x=y;   (x > 0) set x to y; is x > 0?
y > 0                          relation reduces to another relation
     x = 1; (x > 0)declare range of x
1                              x is > 0 if it evaluates to number > 0
     x > 0; (x > 0)       or, if it is declared to be so
1
     x > y; y > 0; (x > 0) ot, if it is declared to be > a
1                                    VARIABLE and that VARIABLE>0
     x >= y; y > 2; (x > 0) etc.
1
     x < y; y <=-1; (x > 0) but not >0 if it can be shown to
0                                     be  ≤ 0;
     x < 1; (x > 0)       if it is undecidable, relation remains
x > 0                             unchanged
     x >=y; y > -1; (x > 0)
x > 0
```

A relation (x in S) evaluates to 0, 1, or another logical expression derived thereof.

```
     (5 in (1,2...))     a VECTOR is permitted as the r.h.s.
1                                of an IN relation
     x integer           let x range over all integers
     (x in (1,2,...))    the relation: is x in (1, 2, ...))?
(x >= 1)                         reduces to: is x ≥1?
     (x in (1,5,7))      when the VECTOR is not an
(x in (1,5,7))                   ellipsis expression involving an
     (x in (1,5,7)),x=4          arithmetic progression of integers,
0                                the relation reduces to 1, 0, or
     (x in (1,5,7)),x=1          the original relation with left
1                                and right sides evaluated
```

More general logical expressions are formed from relations by using the operators ¬, OR, and & in the customary way

```
    (¬12)                 any non-zero number is equivalent
0                                to 1 when used with boolean operators
     x=0; (12 & x); (12 or x)
0
1
```

10) SETS

For the evaluation of (x in S) where S evaluates to a set, the value returned is that of the defining predicate of set evaluated in the environment where its dummy parameter is bound to the value of x.

```
    N=(1,2,...)                            the set of positive integers
    N=(i|i integer & i > 0)                SET definition of same
    (10 in N)                              The relation (x in S) for set
1                                          S reduces to 0 or 1,
    x=
    (x in N)                              .or, to another expression
(X INTEGER & X > 0)                           derived thereof
    S=(x|f(x) > 0)                        the set of x such that f(x) > 0
    f(y)=1, y > 10        define f(a) for all parameters
    f(y)=0, y < 10             a except a=10
    (1 in S)              The value of (1 in S) equals the
0                              value of (f(1) > 0) = 0
    (11 in S)             The value of (11 in S) equals the
1                              value of (f(11) >0) = 1
    (10 in S)             The value of (10 in S) equals the value
f(10) > 0                      of (f(10) > 0) which is itself
```

11) VECTORS

```
     (1,2,3)           elements of VECTORs are printed out
1                              one per line
2
3
     n > 2                    establish range of n
     i,j in (1,2,...,n)       establish range for i and j
     h=(i:(j:1/(i+j-t)))      generalized Hilbert matrix(order n)
     (h)<2,2>

  1
-----
4 - T

     (h)<3,2>                 range of n must be properly established
DON'T KNOW THAT n >= 3
     (h)<n-1,n>               (1, 2, ..., n) must have at least 2
                                       elements if n > 2
     1
----------
2N - T - 1

     (h)<0,1>              value of elements outside range is 0
0
     h<1,2>                H    is independent of H
H                           i, j
 1,2
     h<i,j>=(h)<i,j>       defines h     for all i, j in (1, 2, ..., n)
                                    i, j
     h<1,2>

          1
H    : -----
 1,2  3 - T

     A=(i: (j: a<i;j>))  let A have elements a^i_j

     a<i,j>=i+j if (i+j)//2=0, _    i, j integer
         i,j integer
     A , n=2                     Give replacement rule for a <i;j>
                                      for integer index values i, j where
 1                                    i+j is even
A : 2                            Then print A for n=2
 1

 2
A
 1

 1
A
 2

 2
A : 4
 2
     delta<i,j>=(i=j)               define Kronecker delta
     delta<i,j>=1 if i=j else 0       same
     I=(i:(j: delta<i,j>))          define unit matrix
     I=(i: (j: i=j))                a more convenient way
     H*I, n=2                       Arithmetic expressions are
  1                                     given for vector(or matrix)
-----                                   expressions in exactly the
2 - T                                   same way as for scalar
                                        expressions.
  1
-----
3 - T

  1
-----
3 - T
  1
-----
4 - T
```

12) SUM OPERATOR FORMS

```
    u=sum<i=-n;n>f(i)        summation indices are given directly,
    i in (-n,...,n)          or, indirectly through VECTORs
    u=sum<i>f(i); u              (n assumed > 0)

---N
/        f(i)                Sums with a FORM as a summand
---i=-N                      are decomposed into sums over the
                             various domains of definition
    u, f(i)=1, i in (-n+1,...,n-1)
F(-N) + F(N) + 2 N - 1          (f undefined at end points)
    u, f(i)=1, i <= 0

---N
/      F(i) + N                 (f undefined on positive integers)
---i=1

    u, f(0)=0

--- -1          ---N            (f defined only at 0)
/       F(i) + /       F(i)
---i=-N         ---i=1
```

13) PROCEDURES

Commands beginning with a numerical label of the type n.m' are not interpreted but stored by the system to be later combined to form a procedure by means of the special FORM procedure (n)". A procedure is a block of commands to be executed as a group. The number n refers to the block, the number m to the command. Command numbers are treated as integers when collated to form a procedure.

The following additional statements are allowed in a procedure:

a) the statement "GO n" is treated as a jump to step number n

b) the statement "RETURN n" causes an exit from the procedure; n is then the value of the procedure.

```
    1.10  m=(u)<1>               load first element into m
    1.20  i=1                    (should be i=2)
    1.30  return m if i>length(u)
    1.40  s=(u)<i>               load next element into s
    1.50  go 60 if m > s         if m > s go to 60
                                 (missing statement)
    1.60  i=i+1                  advance index
    1.65  go 30                  go to step 30
    1.70  hoho                   (junk)
    max(u)=procedure(1), _
        (m,i,s) local            declare m, i, and s local
    1.20 i=2                     change step 20
    1.55 m=s                     insert between step 50 and 60
    1.70                         delete step 70
    max(u)=procedure(1), _       try again
         (m,i,s) local
    x=(1,4,3)
    max(x)
4
    procedure(1)=                erase block 1
    1.10 i=0                     Taylor-series expansion of y
    1.20 h=1                         about x=a in powers of h
    1.30 s=y where x=a               to order n
    1.40 i=i+1
    1.50 return s if i > n
    1.60 h=h*(x-a)/i
    1.70 y=df<x>y
    1.80 s=s+h*(y where x=a)
    1.90 go 40
    taylorseries(y,x,a,n)= _
      procedure(1), y=x=a=,_
      (s,i,h) local, n in (0,1,...)
                         "(=) replace taylor series (a, b, c, d) by
                              BEGIN s, i, h; ... END when
                              d in (0,1,...)
```

14) SYNTAX EXTENSIONS

```
    |-2|                         Notation |x| is not part of the
|-2|                             base language but may be added by
-                                means of syntax extension command.
    "|x|" = "absval(x)", x expression
EXTENSION OF TYPE FORM
    |-2|                               try out
2
    "|x|" =     , x expression   remove extension
ALL EXTENSIONS REMOVED FROM FORM
    "n mod p" = "n-p*(n//p)",_   define "mod" for n, p
       n,p svar                  variables or numbers
EXTENSION OF TYPE ALGEBRAIC EXPRESSION
    13 mod 4
1
    E=(n | n mod 2=0)        define E = set of even integers
    "even" = "in E"
TEXT TO TEXT EXTENSION
    f<i>=i**2, i even        define f over even integers i
    "(n*x| )"="(x|x mod n=0)",_
        n integer, x variable
EXTENSION OF TYPE SET
    F=(3*n|)                 set of integers congruent to 0 mod 3
    "(n*x|r)"="(x|r&x mod n=0)",_
        n integer, x variable, r relation
EXTENSION OF TYPE SET
    (5*m| m > 0)             set of positive integers congruent
                                 to 0 mod m

    "integrate f wrt x"="int<x>f",_
       x variable, f term      (INTEGRATE command)
EXTENSION OF TYPE OPERATOR-FORM
    "let x=y" = "x='y'",_     (LET command)
       x term, y expression
EXPRESSION OF TYPE STATEMENT
    "consider x" = "(x)",_    (CONSIDER command)
       x expression
EXTENSION OF TYPE PRIMARY
    "put x into y" = "y=x",_  (PUT command)
       x expression, y variable
EXTENSION OF TYPE STATEMENT
    "substitute x=e into y"=_  (SUBSTITUTE command)
     "y=y where x=e",       _
     (x,y)variable, e expression
EXTENSION OF TYPE STATEMENT

    let y=g(x,t)                y = 'g(x, t)'
    consider 2*x=y              (2*x= y)
 2 X = G(X,T)
    integrate ws wrs x           int < x >ws

  2       /
 X    =   |  G(X,T)
          /X

    put ws into u               u = ws
    substitute _                u=u where g(x, t) = 4*x*t^2 - 1
      g(x,t) = 4*x*t**2 -1 into u
    u

     2      2 2
U:   X  = 2X T  - X
    "set" = "substitute"
TEXT TO TEXT EXTENSION
    "in" =  "into"
TEXT TO TEXT EXTENSION
    set x=1 in ws               ws= ws where x=1

      2
1 = 2T  - 1

    set t=1 into ws             equation 1=1 reduces to
1                                   1 (meaning "true")
```