The SCRATCHPAD Language

R. D. Jenks
Mathematical Sciences Department
IBM Thomas J. Watson Research Center
Yorktown Heights, New York

"By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems..."
—Alfred North Whitehead

ABSTRACT. **SCRATCHPAD is an interactive system for symbolic mathematical computation. Its user language, originally intended as a special-purpose non-procedural language, was designed to capture the style and succinctness of common mathematical notations, and to serve as a useful, effective tool for on-line problem solving. This paper describes extensions to the language which enable it to serve also as a high-level programming language, both for the formal description of mathematical algorithms and their efficient implementation.**

1. INTRODUCTION

SCRATCHPAD ([3],[8],[9],[10]) is an experimental interactive system for on-line symbolic mathematics under development at the IBM Thomas J. Watson Research Center, Yorktown Heights. This paper describes ongoing work on the design and implementation of the user language and interface to the SCRATCHPAD system.

The SCRATCHPAD language has been described as a two-dimensional language designed for interactive symbolic computation, and for eventual use in on-line exploratory research with graphical input/output. Work has recently begun on a compiler for the language and on new facilities for types and extensions. These new features have been added so that SCRATCHPAD may also be used as a language for the formal description of algorithms as well as a source language for their efficient implementation. The language design presented here is an attempt to unify 5 language concepts within a single language.

1. *A declarative language suitable for the interactive definition and manipulation of symbolic formulae and expressions.* A conversation with SCRATCHPAD consists of a sequence of commands issued by the user. Each command is processed before the next is read. There are basically two uses of commands: (1) to create transformational rules (intuitively speaking, "definitions"), and (2) to manipulate expressions. Commands consist of a 'main' statement optionally followed by one or more 'qualifying'statements grouped to the right. Statements have the format: L r E, where L and E are expressions and r is a relational operator. The following 3 commands, for example, define a recurrence relation for the Legendre polynomials:

$$p_0=1$$
$$p_1=x$$
$$p_n=((2n-1)\times x\times p_{n-1}-(n-1)\times p_{n-2})/n \qquad (n>1)$$

Here, definitions are stated in an incremental ("piecewise" or "recurrence relation") style frequently used in mathematics. This style puts the "main thought" first and trailing thoughts afterward. The "main thought" is the definition, the trailing thoughts are qualifiers which restrict the range over which the definition is valid.

As a language for manipulation, the concept of the *workspace* (abbreviated by "ws") is very useful. If the user wishes to display an expression, say E, he issues a command to load the workspace with that expression. This is written "ws=E", or simply, "E". Once an expression is in the workspace, it may be operated on, e.g.

$$\int_t ws \qquad (u=f(t)\times\cos(t),\ exp=1)$$

means: "integrate the current expression in the workspace with respect to t where u=f(t)×cos(t) and exp=1; then put the result back into the workspace and display". The statement "exp=1" causes expressions to be expanded during evaluation. Here the "main" thought is the manipulation to be performed, and the trailing thoughts are local modifications to the environment to be used in evaluation.

Each new value of the workspace is assigned a consecutive integer and is stored into a history file from which it may be later retrieved. This history file provides a "conversational backtracking" facility which, among other benefits, enables a user to completely restore the environment which existed prior to any previously issued command [9].

2. *A pattern matching language with evaluation described by a Markov algorithm.* From the SCRATCHPAD user's viewpoint, symbolic computation simply involves the transformation of expressions from one form into another. Transformations are governed by a Markov algorithm operating on a *stack* of "replacement rules" (see Figure 1). The stack has two parts: a system part and a user part (initially empty). The system stack consists of "built-in" transformation rules which the user normally wants applied automatically, e.g. x×1 → x and 0+x → x. An expression is evaluated by scanning the stack from front to back in search for an appropriate rule. If a rule causes a substitution to be made, the transformed expression is then evaluated, etc. until no further changes are possible. The resulting expression is called the *value* of the original expression. For example, with the above two rules, the expression 0+1×x has value x.

All user commands with L r E statements add rules to the user part of the stack. The L part denotes a pattern, the E part, a replacement. For example, if the user issues the above recurrence relation, then the stack in Figure 1(a) is augmented to that in Figure 1(b). Commands consisting of simple one-line L r E statements as above are typical. On the other hand, the E part may consist of a multi-line procedure, or, the left part L may specify a pattern for an integration rule or combinatorial identity, e.g.

$$\Sigma_{0\le k\le n} c_{k,n}\times\cos(k\times x)=2^n\times\cos(n\times x/2)\times(\cos(x/2))^n$$

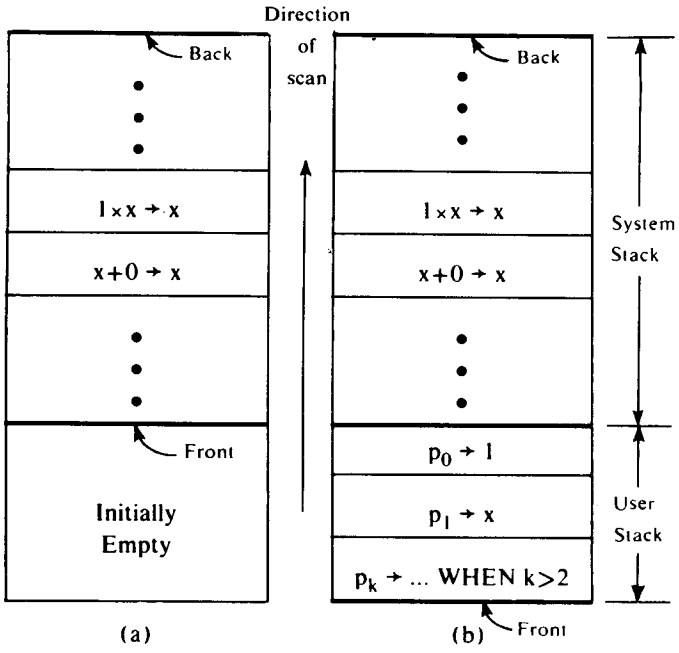

Figure 1. Simplified Stack Model

By means of user commands, the interactive user is able to directly introduce into the system special purpose transformation rules necessary for the solution of his problem. New rules take priority over existing rules. Rules may also be selectively "cleared" (removed from the stack) or "frozen" (user rules become system rules).

One principal value of this Markov algorithm approach is that it provides a simple model for symbolic mathematical computation. A second is that it results in a non-procedural style of programming: the order in which transformation rules with non-overlapping domains are created is irrelevant to subsequent computational results. Another virtue of this approach is that it unifies the notions of procedural extensions and pattern-match extensions. Indeed there is no semantic distinction between statements of the form $cos(x)^2=1-sin(x)^2$ and a procedure definition. Finally, the use of patterns frequently enables one to *explicitly* describe a transformation or computation without stating *how* it is to be done, e.g.

$$\text{reverse}([x_1,\ldots,x_n])=[x_n,\ldots,x_1]$$

Pattern-matching may also be indicated by an **is** relator, e.g. u **is** x+y, used either explicitly at the top level or inside conditional expressions, or implicitly in **cases** expressions. Consider, for example,

$\partial_t u$= **cases:** u
x+y: $\partial_t x+\partial_t y$ $(x,y\neq 0)$
...

On evaluation of an expression of the form $\partial_t u$, the pattern x+y is applied to u. If a match occurs, x and y are bound to the matched subexpressions and the right member of the pair is evaluated to become the value of the conditional expression.

3. *An extensible language.* The *extensible* approach to system design is essentially that of providing a "core" system which is relatively easy to learn together with a means of extending that core in a variety of ways. We believe the arguments for such an approach to be especially valid for a symbolic mathematical system expected to provide a natural on-line interface to people with widely varying interests, backgrounds, and problem requirements.

Extensions to SCRATCHPAD are made in three ways: (1) through L r E statements described above which provide for both pattern-match and procedural extensions; (2) syntax extensions and generalized transformations which permit new notations to be both introduced into the language and transformed (Section 4); and, (3) "type" extensions which allow new data types to be added (Section 5).

SCRATCHPAD is described here as two systems: a "basic" system and a "standard" system. The basic system contains only basic transformations (such as those for manipulating and simplifying polynomial and rational function expressions, but, e.g. no knowledge of trigonometric functions), a basic language (which enables the user to state commands, form expressions, but which has no specialized notations such as ' | x | '),and primitive types (such as integer, real, and sequence, but not, e.g. sets, matrices, and complex numbers).

The standard system is the one generally available to users. It is derived from the basic system almost entirely through compiled extensions of the above 3 types. Extensions are stored in files which may be modified to suit an individual user's needs or tastes. In addition, a library of "pre-programmed" extension modules are available. These allow such special symbolic packages as matrix manipulation, truncated power series, and gaussian integers to be loaded into the system on request.

The extensible approach is a useful one for reasons of flexibility, simplicity, and convenience. It is also valuable as a pedagogic device. The new user first learns only the basic language, its relatively simple syntax and semantics. Once learned, the user is then taught a hierarchy of notations derived from basic notations. For example, although 'plus(x,y)' is the primitive form for addition, extensions in standard SCRATCHPAD allow plus to be expressed in a variety of ways, e.g. 'x+y' for 'plus(x,y)','+/u' for the APL operation 'reduction(u,plus)', 'Σ u' for '+/u','$\Sigma_{\ldots}$' for '+/[u **for** ...]', '$\Sigma^n{}_{i=m}u$' and '$\Sigma_{m\leq i\leq n}u$' for '$\Sigma_{i\in[m,\ldots,n]}u$', '$\Sigma_{m\leq i}u$' for '$\Sigma_{i\in[m,\ldots]}u$', etc.

4. *A formal description language allowing data representation free programming.* The SCRATCHPAD language is called *formal* in the sense that transformations may be expressed by mathematical notations in machine independent form. Transformations can be described separately for each operator, argument type, and pattern. A significant feature is that transformations may be given in *data representation independent form.* Like LISP, data and programs in SCRATCHPAD have a similar syntax. Unlike LISP, however, programs generally describe transformations on expressions independent of their *internal* representation. Declara-

tions of data type intended for compiler consumption may be placed *outside* of the program description itself. For example, in the above **cases** example, the program is given in the context where argument t is declared as an 'identifier' and argument u, a rational function. These declarations are often irrelevant to the understanding of the program, and only serve to give efficient object code on compilation. When order is important, conditional expressions may be used.

5. *A high-level implementation language.* The language is intended to serve both as a language for the formal description of algorithms and as a language for interactive manipulation of mathematical expressions. It is also intended to be the source language for the implementation of most of the underlying system. This combination of uses offers many advantages. For example, since program descriptions are formal, the source code itself would serve as a reference manual for the system.

The problem of compiling efficient code involves obtaining an efficient implementation of the Markov model. A key point in its design is that no rules on the stack can control how the scanning is done. As a result, the Markov model can be implemented without the use of a stack at all. Rules are simply converted into conditional expressions which are hung on identifiers representing variable and operator names. Successive rules for the same operator cause a conditional expression representing the set of transformations to incrementally grow. When a "compile" command is issued, user-defined rules are compiled and become part of the system stack. When new user-defined rules are created for operators for which there already exist system-defined rules (such as +,×, etc.), the compiled code representing the system-defined rules is embedded in new code to first check the new rules. On compilation, the source code for the system-defined rules is retrieved, combined with the code for the user-defined transformations, optimized, then recompiled.

The compilation of the **cases** statement has an unusual requirement. Here the code used to do the pattern matching is clearly dependent on the type of the arguments, in particular, a canonical form chosen for representation of the expressions. The choice of an appropriate canonical form for a computation can dramatically effect the performance, and SCRATCHPAD has several. For each such canonical form, specially prepared text describes the meaning of each canonical form in terms of LISP prefix form. The compiler then references this text to produce appropriate code for pattern matching.

The SCRATCHPAD evaluation model while providing a simple explanation for evaluation, is nevertheless LISP-like evaluation in disguise. If one regards expressions as operator-operand trees, operands are generally evaluated first then the operator is applied to the operands as in LISP. The significant difference here is that command-level evaluation is to be idempotent: evaluation involves continuous transformation of an expression until no further transformations are possible. Evaluation thus produces a constant relative to the environment of evaluation.

2. LANGUAGE OVERVIEW

The SCRATCHPAD language has four major parts of speech: Primitives, Expressions, Statements, and Commands. Primitives are Constants, Designators, and Sequences (note: throughout the remainder of the paper we will use capitalized nouns to denote parts of speech).

Constants. Constants are Numbers and Strings. Numbers are of two types: Integer and Real. Integers consist of strings of digits of indefinite length. Symbolic calculations generally use unlimited precision rational arithmetic to ensure that numerical coefficients remain fully accurate. Reals are written in a conventional way. Strings are strings of characters delimited by enclosing quotation marks: 'abc'.The symbol '' denotes the empty string.

Designators. Designators are used to "name" Expressions and serve as denotations for symbolic constants and indeterminates in the program. Two types of Designators are: Identifiers and Forms. Identifiers are strings of letters and digits beginning with a letter. Certain "special" Identifiers[9] have special meanings, e.g.:

Special Identifier:	Effect:
exp	if exp=1, Expressions are fully expanded during evaluation
gcd	if gcd=1, greatest common divisors of rational expressions are removed during evaluation
factors	if factors=$[x_1,...,x_n]$, products of powers of the x_i's are factored out on output

A Form may have subscripts, superscripts, pre-subscripts, pre-superscripts, and functional arguments, in any combination and to any depth, e.g.

$$x_i \quad x_{i,j} \quad x^i_{j,k} \quad x_{i,j_k}(u,v) \quad ...$$

Forms are used to represent parameterized objects(such as x_i, x(i), log(x), etc.). Certain special Forms[9] denote system provided transformations, e.g.:

Special Form:	Use:
factor(e)	factors polynomial e over the integers
solve(e,x)	solves e=0 for x (e.g. e=$[e_1,...,e_n]$ linear in x=$[x_1,...,x_n]$)
coeff(x,n,e)	returns coefficient of x^n in e (when x,n, e are Sequences, the result is a higher-dimensional Sequence)

A third type of designator, the Hyphenation, is discussed in Section 5.

Sequences. A Sequence is an ordered list of heterogeneous Expressions, separated by commas and enclosed in square brackets, e.g. S=$[e_1,...,e_n]$. The Expressions e_i are called the "elements" of S. The symbol [] denotes the empty Sequence. Sequences are of four categories. (1) An *Explicit*-Sequence is a series of Expressions separated by commas, e.g. [1,2,3]. (2) An *Implicit*-Sequence is similar but is one which contains at least one ellipsis Expression (...), e.g. [1,2,...]. A Sequence of one of the above two categories is also called an *Enumerated*-Sequence. The remaining two categories are

those which contain a **for** or a **s.t.** (such-that) clause: (3) *Virtual*-Sequences are those which contain an explicit iteration clause introduced by a **for**, e.g., [x(i) **for** i ε S]. Finally, (4) *Predicate*-Sequences are those which contain no explicit iteration clause but contain a predicate preceded by **s.t.** (written here " | "), e.g. $[x_i \mid i>0]$.

Expressions. Expressions represent the objects manipulated by the user. An Expression is either a Primitive or a syntactically allowed combination of Primitives, operators, ellipses, and brackets. Ellipses are used in patterns to indicate missing parts. Within Sequences the Expressions bordering the ellipsis are assumed to indicate an evident arithmetic progression, e.g. $[x(1),x(3),...]$, $[x_i(t),x_i(t^3),...]$, $[x_{1,2}(t_2),x_{3,2}(t_4),...]$. Operators are of three types: infix, prefix, and suffix. Some operators (e.g. + and $\times$) have associated n-ary prefix operators, (e.g., Σ and Π) in which case they have a variety of acceptable formats, e.g. for Σ:

$$\Sigma_{i=1}^{n} \quad \Sigma_i \quad \Sigma_{i\varepsilon S} \quad \Sigma_{1\le i\le n} \quad \Sigma_{1<i\le n}$$

Other prefix operators include $\forall$, $\exists$, ∂, $\int$, $\cup$, $\cap$, and various operators for searching over aggregate objects.

The APL convention for reduction over a sequence may be used in standard SCRATCHPAD to convert an infix operator to an n-ary prefix operator, e.g. if $x=[x_1,...,x_n]$ where each of the x_i is scalar (not a Sequence, Set, etc.) then $+/x$ will evaluate to $x_1+...+x_n$. If at least one x_i is not scalar, then $+/x$ transforms to $[+/x_1,...,+/x_n]$. In addition, $+[n]/x$ is used to force reductions over the n^{th} dimension of the Sequence.

Two other types of Expressions are Conditionals and Blocks. A Conditional Expression is expressed using a) **if** as an infix operator(the basic form), e.g. 'e_1 **if** p_1 **else** ...', (b) by the conventional '**if** p_1 **then** e_1 **else** ...', or, c) by a **cases** expression, e.g. '**cases**:$(p_1:e_1;...)$'. In addition, the Expression '**cases**: u; $(a_1:e_1;...;a_n:e_n)$' is an allowed abbreviation for '**cases**: (u **is** a_1: e_1;...;u **is** a_n:e_n)'.

A Block Expression consists of one or more Commands separated by semicolons and enclosed between **begin** and **end**, or between round parentheses when no ambiguity arises. Commands are executed in sequence until a **return** expression is encountered which causes an immediate exit from the Block. Gotos are not allowed.

Statements and Commands. An entire SCRATCHPAD interactive conversation, e.g., consists of a sequence of Commands, and thus might be regarded as a Block. A Command consists of one or more Statements separated by commas and optionally qualified or iterated as described below. Statements within a Command are executed in parallel.

The general form of a Statement is: L r E , where L and E are Expressions, and r is one of the ten relators:

$$= \quad \neq \quad < \quad \le \quad \ge \quad > \quad \varepsilon \quad \notin \quad \textbf{is} \quad \textbf{not}$$

More precisely, L may be an Expression containing operators outside parentheses with precedence exceeding that of the relators. Other operators may appear in L and E but must be enclosed in parentheses.

In addition to the above Commands which are called "Rule-Commands", are "Special-Commands" (e.g. **special**, **type**, **syntax**), and "System-Commands", identified by the prefix ")". System-Commands are either utility commands (e.g. **)edit**), commands to alter or clear the history file (e.g. **)freeze** and **)clear**), or commands to initiate backtracking(e.g. **)revert**). System-Commands are otherwise distinguished from the first two categories for two reasons. First, System-Commands may only be issued at the conversational level whereas Rule- and Special-Commands may appear within Blocks in the E part of a Statement. Second, Commands in the first two categories may be backtracked; incremental changes to the system which result from their execution are appropriately recorded on the history file so as to make their effect reversible. System-Commands are never recorded on the history file and are not reversible.

We now describe two aggregates in standard SCRATCHPAD which are derived from Sequences, and then "iteration" and "qualification".

Sequences and Derived Aggegrates. In SCRATCHPAD, the Sequence is the fundamental aggegrate object. All other aggregate types required in the system must be derived from Sequences through type extensions. Two such types provided in the standard language are Sets and Maps. In both of these cases, the type further breaks down into four categories of Sequences.

Sets. A Set S is a sequence of distinct heterogeneous Expressions separated by commas and enclosed in braces, e.g. $S = \{e_1,...,e_n\}$. The Expressions e_i are called the *elements* of S. The symbol {} denotes the empty Set. Explicit-Sets are treated as explicit-Sequences in which elements are canonically ordered and where duplicate elements are removed. One useful category of Sets is the Predicate-Set which has the form {x | p} where x is generally a Identifier and p is some predicate usually involving x.

Maps. A Map M is a sequence of pairs s_i:e_i separated by semi-colons and enclosed in angle brackets, e.g.

$$M=<s_1:e_1;...;s_n:e_n>$$

where the s_i and e_i are Expressions. The Expressions s_i, called the *selectors* of M, must be distinct. The Expressions e_i, called the *components* of M, are distinct from special value NIL but otherwise arbitrary. The symbol <> denotes the empty Map.

A Map denotes a mapping with domain(M) $=\{s_1,...,s_n\}$, and range(M)$=\{e_1,...,e_n\}$. Infix operator (.) is used to apply Maps to domain Expressions, i.e. $M.s_i$ evaluates to e_i for all $i\varepsilon$ [1,...,n]. The value of M.s where s $\notin$ domain(M) is NIL. When M.s is given on the left hand side of a Statement, an element will either be replaced by or inserted into M according as s ε or $\notin$ domain(M). The Statement M.s=NIL is used to remove a pair from the Map.

A few conventions are used in specifying Maps. When successive components are the same, only the last component need be written, e.g. if $e_1=e_2=...=e_m$ then M may be expressed as $<s_1,s_2,...,s_m:e_m...>$. Omitted selectors take on assumed values. If s_1 is missing, it is defined as 1. A missing s_i, $i > 1$, is taken to be $s_{i-1}+1$. Thus, $<e_1;...;e_n>$ is short for $<1:e_1;...;n:e_n>$, and $<m:e_1;...;e_n>$ means $<m:e_1,m+1:e_2;...;m+n-1:e_n>$, etc.

Maps are extremely useful data structures in symbolic manipulation. One relevant special Form is comap(x,e) which on evaluation "decomposes" a polynomical e in x to a Map (Examples 3c) and 3d) in Figure 2).

1. Examples of Sequences
   a) [a,b,u/v,[u,v]] seq. of 4 elements
   b) [x_1,x_2,...,x_n] seq. of n elements (n symbolic)
   c) [x_i **for** i ε S] =b) with S=[1,2,...,n]
   d) [x_i] (i ε S) =c) (using range convention)
2. Examples of Sets
   a) {1,2,4,5} set of 4 elements
   b) {x_1,...,x_{j-1},x_{j+1},...,x_n} =a) with x_i=i,n=5,j=3
   c) {x_i **for** iεS | i≠j} =b) with S={1,2,...,n}
   d) {x | f(x)=0} a predicate-set
3. Map representation of polynomial u=($x^5-5x^3y^3+2y^3$)
   a) <0:$2y^3$;3:$-5y^3$;5:1> with lead variable x
   b) <0:<5:1>;3:<0:2;3:−5>> with lead var. y then x
   c) <x^5:1;x^3:$-5y^3$;1:$2y^3$> result of comap(x;u)
   d) <x^5:1;x^3y:−5;y^3:2> result of comap([x,y],u)
4. Maps as structures
   a) <real:5; complex:1> complex integer
   b) <comp:Bach;opus:32;title:'Tocatta in G';inst:organ>

Figure 2. Examples of Sequences, Sets, and Maps

Iteration. A sequence of Expressions or Statements separated by commas is optionally followed by an "iterator" construction introduced by the infix operator **for**. Two meaningful forms of the iterator are: '**for** iεS', to iterate over a single sequence, and '**for** ($i_1εS_1$,...,$i_nεS_n$)', to iterate over several in parallel. In addition, an iterator may contain a selection predicate introduced by a such-that clause, e.g. '**for** iεS | p(i)'.

If S denotes an Explicit-Sequence, then the evaluation of 'e **for** i ε S' causes e is to be repeatedly evaluated as i runs over the successively generated elements of Sequence S. When the iterator has a selection predicate, e.g. 'i ε S | p(i)' then p(i) is a predicate local to the iterator and acts as a filter. The predicate is evaluated for each value of the index i. If p(i) is FALSE for some value of i, that value of i is not generated. When S is not an Enumerated-Sequence, then the iteration is not carried out and the sequence remains in virtual or predicate form. In the standard language, other forms of iteration (using **while**, **until**, **unless**, **repeat**, and **do** constructs) are provided. Typically these are displayed after the body of the iteration, and result in a format similar to that suggested by Anderson in [1].

Convention. With special Identifier, range-convention=1, Expressions enclosed in brackets which contain free Identifiers ranging over a Sequence are agreed to have implicit iterators, e.g. using this convention with iεS, {x_i} and [x_i | j≠i] expand to {x_i **for** iεS} and [x_i **for** iεS | j≠i] respectively. This convention is assumed throughout this paper.

Qualifiers. A sequence of Expressions or Statements separated by commas can be optionally 'qualified' by a **where**-clause consisting of the infix operator **where** followed by an Expression, normally a Block, as its right argument, e.g.

u=ws−beta×f(y) **where** (s_k=k×c, exp=1)

The **where**-clause causes the left argument of the **where** to be evaluated in a local environment created by executing the right argument of the **where**. Operator **where** is left-associative so that multiple qualifiers are executed from right to left. The **where** operator may replaced by two or more blanks when no ambiguities arise; this convention gives rise to the format used in the examples in section 1. An Identifier may be used as the right argument of the **where**-clause in which case it is assumed to have a Block as its transform; in this way, a set of transformation rules may be "stored" as well as invoked by use of a single name, e.g. 'ws **where** trigtoexponential'.

If b is the Block (s_1;...;s_n) and s is a Statement, then 'b **where** s' may be written as '**let** s; s_1;...;s_n'. The effect of the **let** statement, e.g. '**let** x=y; b' is to make x a local variable of Block b.

3. SEMANTICS

Environment. The effect of evaluating a Statement of the form: *L r E* is to create a replacement rule having the form of a sequence of four elements:

[L*(pattern)*, C*(condition)*, r*(relator)*, E*(replacement)*]

where L and E are expressions corresponding to the *L* and *E* parts of the Statement; and, C is a predicate describing conditions on the pattern and environment under which the replacement rule is applicable.

The ordered set of replacement rules on the stack at any one instant in time is called the "environment of evaluation". At the beginning of a user's session, only certain special Identifiers and Forms which denote special system names, operators, and functions have associated replacement rules; these make up the frozen portion of the stack called the "system stack". As the user session progresses, the user builds an environment through the use of Commands containing Statements. Every such Command creates one replacement rule which is added to the front of the stack.

Evaluation. Evaluation means the transformation of one Expression into another through the continuous application of rules in the current environment until no further transformations can be made. The resulting Expression is called the *value* of the original Expression. Ordinarily, the issuance of a command L r E does not cause the evaluation of L or E or any of their constituent subexpressions. In order to cause evaluation of an expression, the expression must be prefixed by the meta operator μ.

The process of evaluation involves the use of a "scanning function" for scanning the stack from front to back in search of appropriate replacement rules. This function takes two arguments: first an expression to be transformed (the *subject*) and secondly, a set of permissible relators (the *relatorset*). As each rule is encountered during the scan, the function determines whether or not the *relator* part of the rule is a member of *relatorset*. If it is not, the rule is bypassed and

scanning continues to the next rule up the stack. Otherwise, the *pattern* part of the rule is compared to the *subject*. If the pattern has the same syntax as the subject except at formal parameter positions, a match is said to occur. A match having occurred, the *condition* expression is then evaluated with formal parameters bound to corresponding argument expressions in the *subject*.. Three outcomes are then possible: (a) the resulting value is TRUE, and the rule is then said to "apply"; (b) the resulting value is FALSE, the rule is then said "not to apply", and scanning continues up the stack at the next rule; (c) the value is neither TRUE nor FALSE, the applicability of rule is then said to be "uncertain", and scanning is terminated. If a rule R is found to "apply", the scanning function returns both R and a form which may be evaluated to scan the stack upwards from R. The "transform" of the *subject* is then obtained by substituting actual parameters for formal parameters into the E part of R and "simplifying"(see below). If no rule is found to "apply" or if scanning was terminated, the scanning function returns NIL (indicating "no transform").

Evaluation of Primitives and Expressions may then be described as follows. The value μc of a Constant c is c itself. The value μx of a Identifier x is obtained by first scanning the stack with *subject* x and *relatorset* {'='}.If x has transform E, μx is μE; otherwise, μx is x itself. The value $\mu f(a,b,...,c)$ of almost all Forms f(a,b,...,c) is described as above except with x replaced by $f(\mu a,\mu b,...,\mu c)$. Infix and prefix expressions are evaluated as if represented by equivalent Forms, e.g. $\mu(x+y)=\mu plus(x,y)$.

Consider the following example set of Commands:

command:	rule no:	replacement rule created:
$i > 0$	1	[i,TRUE, '>', 0]
$f_0=1$	2	[f_a, a=0, '=', 1]
$f_i=2f_{i-1}+1$	3	[f_a,a>0, '=', $2f_{a-1}+1$]
$i=1$	4	[i, TRUE, '=', 1]

These 4 rules leave the user stack (read "bottom up") in the configuration shown on the right. Consider now the evaluation of f_i with *relatorset* ={'='}(the usual case). As described below, first the parameter i is evaluated. The stack is scanned for a transform for i. Rule 4 is first encountered; the pattern matches, the condition expression is TRUE and the relator belongs to *relatorset*; therefore the transform of i is 1. Thus μi is 1. Next, $\mu 1$(a Constant) is 1. The stack next is scanned for f_1. Rule 3 applies and delivers the transform $2f_0+1$, which is then further evaluated. First, $2f_0$ is evaluated. $\mu 2$ is 2. $\mu 0$ is 0. Next, the stack is scanned for f_0. This time a match occurs at rule 3, but μ of the *condition* part is FALSE. Scanning therefore continues to rule 2 where a match again occurs but this time with the *condition* part evaluating to TRUE. Since '=' ε *relatorset*, the transform of f_0 is 1. The stack is then scanned for a transform for 2×1. A match occurs in the system part of the stack and delivers the (simplified) transform 2. Similarly, $\mu 1$ is 1 and then $\mu(2+1)$ is 3. Thus μf_i is 3.

When the applicability of a rule is uncertain, no transform is given. For example, if rule 4 were replaced by $i\geq 0$, μf_i would be f_i. This is explained as follows. (a) μi is i. (A match for i occurs at stack rule 4 but the relator '$\geq$' is not a member of *relatorset* ={'='} and so the scanner returns NIL). (b) the stack is scanned for f_i. A match occurs at rule 3. The rule associated with '>' in the user stack then attempts to show i>0 is TRUE or FALSE by scanning the stack for rules on i with *relatorset* ={'$\geq$','>'}. Rule 4 applies. But since $i\geq 0$ does not imply i>0, the *condition* part of rule 3 evaluates to neither TRUE nor FALSE; scanning therefore terminates and NIL becomes the transform of f_i.

The evaluation of other syntactic forms is as follows. The value μC of Conditional $C=(e_1$ **if** p_1 **else** ... **else** e_n **if** $p_n)$ is μe_k if there exists a k ($1\leq k\leq n$) such that $\mu p_1 = ... = \mu p_{k-1}$ = FALSE and μp_k = TRUE. μC is NIL in all other cases. Value NIL may mean that some μp_i was neither TRUE nor FALSE and that the correct choice of e_k is regarded as uncertain. The value of a Statement L r E is μL in the environment which exists after its rule has been created. The value of a Block $b=(s_1;...;s_n)$ is obtained by computing μs_1 then μs_2, etc. until a **return** e is encountered; the value μb is then μe. If no **return** Expression is found, μb is μs_n. The value μb where b is $(s_1,...,s_n)$, n>1, is undefined. The value of $\mu(s_{n+1}$ **where** $(s_1;...;s_n))$ is equivalent to $\mu(s_1;...;s_n; s_{n+1})$ except that rules created by the first n Statements are removed from the stack following the evaluation of s_{n+1}.

The evaluation of iterated forms is handled uniformly by the mechanism of *generators* (following "streams" in [5]). Generators are functional representations of Enumerated-Sequences and may be "passed" as arguments in Forms. If g_1 is a generator for an Explicit-Sequence $[a_1,a_2,...,a_n]$, then there exists generators $g_2,...,g_n,g_{n+1}$, such that μg_1 is $[a_1,g_2]$, μg_2 is $[a_2,g_3]$,..., μg_n is $[a_n,g_{n+1}]$, and finally μg_{n+1}=NIL signifying that all elements have been exhausted. If g_1 represents an Implicit-Sequence then μg_k, k>1, is similar except that a designation for "..." may be returned as the a_k part of its value.

Ordering. All Expressions are canonically ordered on evaluation. Ordering is handled by procedures stored as the replacement parts of the replacement rules associated with each primitive operator. Sums, for example, are ordered so that, e.g. $(x+1)^2$ displays as: x^2+2x+1. This ordering on evaluation allows one to regard two scalar Expressions as equal if they have the same syntax. Canonical ordering is generally unspecified except with respect to integers where it has the usual meaning. The relative ordering of specific identifiers however may be prescribed using a special **order** command[9].

Construction of Replacement Rules. The interpretation of a Statement at the top level has the side effect of adding one replacement rule to the front of the stack. The first step in its interpretation is to evaluate all subexpressions in L and E preceded by a μ operator; the resulting expressions for L and E are then free of μ. The second part of its interpretation creates the replacement rule to be added to the stack. If L is a Identifier x, then the rule created is generally [x,TRUE,r,E,]. On the other hand, if L is not a Identifier, then it is regarded as having the form f(u,v,...,w) where the Identifiers placed in expressions u,v,...,w are called

"statement parameters" and, by agreement[10], indicate the range over which the replacement rule is to be applicable. The determination of the rule to be added to the stack in this case involves five steps:

1. Determination of statement parameters. The left hand side of the rule is scanned for determining a set S of *statement parameters* and a preliminary form of the condition part C of the rule to be created. The scanning of L begins with $S=\{\}$, and C=TRUE. As each Identifier x in L is encountered, the stack is searched to obtain all replacement rules for x(one for each relator r). If the set is non-empty, then x is added to S. Each rule found is converted to a predicate x r' e' and added to C (i.e. *and*-ed into C). Scanning then continues into e' for new statement parameters and conditions. Variables such as gcd, exp, ... which have been declared **special** are exceptions to the above and never considered as statement parameters.

2. *Definition of statement parameters.* The L part of the rule is expressed in the form f(a,b,...,c) where a,b,...,c are unique formal parameters used in place of u,v,...,w. A subset S' of statement parameters may be expressed as linear combinations of formal parameters, by first (a) equating the formal parameters a,b,...,c to u,v,...,w, then (b) solving a linear subsystem system of equations for statement parameters in terms of formal parameters. This system of equations may be overdetermined or underdetermined. When overdetermined (Figures 3b,d), certain extra equations expressed entirely in terms of formal parameters are selected as conditions to be added to C. When underdetermined (Figures 3c,d) an **is** predicate is formed to define members of S − S' and is added to C.

3. *Qualification by statement parameters.* If S' is non-empty, the the E part is augmented by a **where**-clause defining parameters x' for each x' ε S' (Figure 3b).

4. *Substitution and Simplification.* Next, a new set of unique formal parameters is substituted into [L, C, r, E] for the distinct members of S. The final new rule [L' , C' , r, E'] is obtained by simplifying the resulting C and E parts of the rule.

5. *Inconsistency Check.* Finally, the stack is scanned with *subject*=L' and *relatorset*={r' } for each r' ≠r. Each applicable rule found is compared with the new rule to determine if the two rules are inconsistent. If so, an error is signalled and the new rule is discarded.

Statement:	Replacement Rule:
(a)f(x,x+y,x)=y	[f(a,b,c), a=c, '=', b−a]
(b)f(x+y,x−y,2x)=g(x,y)	[f(a,b,c),c=a+b,'=',(g(d,e) **where** (d=c/2,e=(b−a)/2))]
(c)f(x+y)=f(x)+f(y)	[f(a), a **is** b+c, '=', f(b)+f(c)]
(d)f(x,x,y+z)=g(x,y,z)	[f(a,b,c), a=b ∧ c **is** d+e, '=', g(a,d,e)]

Figure 3. Examples of rules created by Statements (assume 'x,y,z arbitrary')

4. SYNTAX EXTENSIONS.

Syntax extensions are used to introduce new notations into the language. These are special Commands of the form "**syntax** p:s" where p is an Identifier denoting a new or an existing part-of-speech (non-terminal of the grammar), and s is a string expressing a syntactic construct. The simplest example of its use is that of adding a new operator to the language:

syntax infix-op: '+' 700 701

Here the left and right precedences of the operator are written following the string to describe how argument Expressions are implicitly grouped and whether the operator is left-, right-, or non-associative.

Syntax extensions may also be used to introduce entirely new constructs into the language, e.g.

syntax expression: '| e |' (e expression)

Once introduced into the language, the new construct may be used in ordinary Statements, e.g.

| s | = length(s) (s sequence)
| x | = absval(x) (x scalar)

The effect of syntax extensions is to add a production to the grammar. Like rules, extensions are "stacked" with more recent extensions taking precedence over previous ones. Also like rules, extensions may be selectively cleared or frozen.

Syntax extensions allow new constructs to be introduced into the language which in general cannot be transformed using Statements of the customary form L=E, e.g.

syntax command: '... **for** m≤i≤n ...'

In order that these new constructs may be transformed on evaluation as well, a generalized form of Statement is introduced.

Generalized Statements. Generalized Statements have the form L→R and are used to describe transformations on more general syntactic constructs, such as those introduced through **syntax** Commands. Generalized Statements L→R are equivalent to ordinary Statements when both L and R are Expressions.

Several additional notations are provided for convenience. The Statement L←R adds the reverse transformation R→L to a special list of transformations performed to Expressions prior to their display. In addition, the Statement L←→R combines two Statements into one; its net effect is to create a new "external" notation for an equivalent "internal" notation, e.g. *

x+y ←→ plus(x+y)

Finally, **syntax** commands may be combined with user commands as illustrated by:

syntax expression:
'**let** s;s_1;...;s_n' → '(s_1;...;s_n) **where** s'
(s,s_i statement (1≤i≤n, n integer))

* The requirements of x and y which match x+y when '+' has left precedence(lp) 700 and right precedence(rp) 701 is as follows. If ι, ρ, and σ denote infix, prefix, and suffix operators respectively, and a and b are expressions, then x is aιb implies rp(ι)>700; x is ρa implies rp(ρ)>700; similarly, y is aιb implies lp(ι)>701; y is aσ implies lp(σ)>701.

5. TYPES.

Every Expression has an associated attribute called its *type*. Primitive types include the parts of speech (uncapitalized) of the language (e.g. integer, real, constant, identifier), denotations for various canonical forms used for representing expressions (e.g. ran (rational number), poly(polynomial), raf (rational canonical form)), and special classes such as literal (meaning "self-denoting"), scalar (meaning "not an aggregate"), and arbitrary (union of all types). Built-in non-primitive types correspond to non-primitive parts of speech such as expression, statement, block, etc. A type consists of an Identifier called its "name", an expression called its "syntax", a set of Expressions called "selectors", a set of "coercion rules" which describe how expressions of other types are to be converted to the given type, and an "automatic coercion list" (see below).

A new type is introduced by a special **type** command which identifies the type name, syntax, and the set of selectors. For example, the statement

type complex **is** <re:raf, im:raf> (re,im literal)

defines a new type complex as a Map containing two elements with selectors re and im. The syntax derived from the Expression on the right serves as a predicate for screening candidates to receive type complex on coercion.

Expressions of a given type can be created only through application of coercion rules. Coercion rules for a new type are introduced through Statements using the one-argument form of the type on the left-hand side, e.g.

complex([a,b])=<re:a,im:b> (a,b raf)

In general, if *foo* is a type, then *foo*() denotes its coercion function. The evaluation of the form *foo*(x) where *foo* is a type is handled in a special way. If *foo*(x) has transform e then e is checked for having the requisite syntax for type *foo* objects. If it does, e is returned with its type changed to *foo*. For example, having issued the above coercion rule, the expression complex([a,b]) will then produce an expression of type complex on evaluation. By agreement, the rule '*foo*(x)=x' if x is already of type *foo* is always assumed.

If x is an Identifier or a Form, then the type of x unevaluated is always 'identifier' or 'form' respectively. Declarations may be given however to force replacement values associated with x to be of a given type. This is done by issuing the statement: 'x **is** *foo*', or 'x *foo*', for short, at the top level. Such declaration causes the coercion function for *foo* to be applied to all non-NIL transforms of x with an error signaled should coercion be unsuccessful.

When a function expecting an argument of type *foo* receives an argument of type *fum*$\neq$*foo*, then the coercion function for *foo* may be applied to the argument in an attempt to coerce it to type *foo*. This so-called "automatic coercion" from *fum* to *foo* will occur only if *foo* is a member of the "automatic coercion list" associated with *fum*.

The "automatic coercion list" is a Sequence of types to which a given type may be automatically coerced. This list is initially empty for new types and is created or modified through statements of the form 'coerce(*foo*)=E', where E is a Sequence of known types, e.g.

coerce(complex)=[scalar, sequence]

indicates that *only* scalar Expressions and Sequences are to be automatically coerced to type complex (but not, e.g. Maps, Sets, matrices, etc). When the automatic coercion list is empty for a given type (as is the case for the examples below), coercion can take place only through explicit use of the coercion function in rules, e.g. 'g(x)=g(complex(x)), x **not** complex'.

The coercion list also affects the evaluation of 'x *foo*' at the inner level where it is interpreted as a predicate with side-effect. This predicate evaluates to TRUE if x evaluates to an expression e which *can* be coerced to type *foo*, and FALSE otherwise. More specifically, the evaluation of predicate 'x *foo*' involves the following cases: (1) if e is of type *foo*, then the value is TRUE with no side effect; (2) if (a)e is of type *fum*$\neq$*foo*, (b)*foo* is a member of the automatic coercion list for *fum*, and (c)a coercion rule for *foo* successfully coerces e into expression e', then the value is TRUE with the side-effect that x is locally bound to e'. (3) in all other cases, the value of the predicate 'x *foo*' is FALSE with no side effect.

A Hyphenation (a hyphenated word) may be used to indicate categories of given types, e.g. enumerated-sequence, predicate-set, as introduced earlier. In addition, the Hyphenation may be used to introduce parameterized types, e.g. matrices-over-*t* where *t* stands for some arbitrary type (Figure 5). The kludge "x **is** type-*t*" is used to declare x to be of type *t* (for some *t*) for the purpose of creating replacement rules.

```
type set is sequence
syntax expression: '{...}' ↔ set([...])

'coercion rules for sets'
sequence(S)=S     (S set)
set(S) =   if S is explicit-sequence
             then remove-duplicates(order(S))
           else S                          (S sequence)

'evaluation of sets'
eval(S) = set(eval(sequence(S)))     (S set)

'operations on sets'
(A,B) explicit-set
A∪B={(x | xεA),(x | xεB)}      'union'
A∩B={xεA | xεB}                'intersection'
A−B={xεA | x∉B}                'set difference'
A×B={[x,y] for xεA for xεB}    'cartesian product'
#A=Σ_{xεA} 1                   'count'
```

Figure 4. Set as an Extension to SCRATCHPAD
(assume extension: '[xεS | ...]' → [x **for** xεS | ...])
(note:'**for** xεS ...' causes iteration over 'sequence(μS)')

6. AN EXAMPLE.

The following example illustrates how a user might add a new special purpose package to SCRATCHPAD for the formal manipulation of power series (this facility is available in ALTRAN; see [4]). The need for this can be amply illustrated in applications, e.g. in perturbation theory where one seeks an approximate solution to an equation as a power series expansion in one or more small parameters [11]. Another application is in obtaining a power series expansion of a function around a point to a finite number of terms. Here, the approach of repeated differentiation is usually hopeless. For example, if one wants the power series expansion of $e^x/(sin(x)^3+cos(x)^3)$ by repeated differentiation, one finds, e.g. the third derivative already to contain 23 terms each a rational function involving up to the 4th power of $sin(x)^3+cos(x)^3$ in the denominator! Yet this example is strictly a numerical problem. One can represent the power series of e^x, $sin(x)$, and $cos(x)$ by arrays of rational numbers and do formal power series operations with these arrays to produce the desired expansions. This approach results in considerable savings in computing time and space required for solving many problems.

We represent truncated power series by explicit-Maps with domain {0,...,n} for some positive integer n, and whose components are rational functions. First, a new type *tps* will be introduced to represent truncated power series:

type tps **is** <i:raf **for** i ε [0,...,n]> (n ε [0,...])

Secondly, we introduce two special variables: *tpsvar*, to identify the power series variable; and, *tpsord*, to denote the order of the series approximation.

special tpsvar, tpsord
tpsvar identifier
tpsord ε [1,2,...]

Next, we give declarations to be used in forthcoming statements:

a,b,c tps; p,q,n ε [0,...]
i ε [0,...,p]; a_i raf; a **is** <i:a_i>
j ε [0,...,q]; b_j raf; b **is** <j:b_j>
k ε [0,...,n]; c_k raf; c **is** <k:c_k>

Here and below, we assume the range convention so that, e.g. <i:a_i> is an abbreviation for <i:a_i **for** i ε [0,...,p]>.

The only way to create an object of type tps is through the use of the coercion function *tps()*. Initially, we will describe the coercion function only on Identifiers, Integers, and Maps having the requisite syntax. The truncated power series will be "normalized" on coercion so that trailing 0's are dropped from the representation and so that the approximation has at most tpsord terms.

tps(x)=<0:0,1> **if** x=tpsvar **else** <0:x> (x identifier)
tps(n)=<0:n> (n integer)
tps(a)=<i:a_i **for** $0\le i\le m$>
(m=min[tpsord,max[(i | $a_i\neq 0$),0]])

We now describe evaluation for truncated power series, first for the algebraic operators +,−,×, and / (we take the liberty of using some of the various syntax extensions allowed in standard SCRATCHPAD):

$-a=c$ ($c_k=-a_k$, n=p)
$a+b=c$ ($c_k=a_k'+b_k'$, n=min[tpsord,max[p,q]])
$a-b=a+(-b)$
$a\times b=c$ ($c_k=\Sigma_{0\le i\le k}a_i'\times b_{k-i}'$, n=min[tpsord,p+q])
a/b= **cases**:
a=tps(0): tps(0)
b=tps(0): error("0 DENOMINATOR")
T: **let** k=low(b)
cases:
$k=0: 1/b_0\times a\times c$
($c_0=1$;
$c_j=-\Sigma_{0\le k\le j-1}c_k\times b_{j-k}$ **if** $0<j\le n$)
(n=q; c_k tps)
k>low(a): error("TPS DIVIDE ERROR")
T:a/b **where** ($a=<0:a_k,...,a_p>, b=<0:b_k,...,b_q>$)

where

$a_i'=a_i$ **if** $0\le i\le p$ **else** 0
$b_j'=b_j$ **if** $0\le j\le q$ **else** 0
low(a)=min[$(0\le i\le p \mid a_i\neq 0), 1+$tpsord] ,

and then exponentiation:

a^0=tps(1) ($a\neq$tps(0))
$a^1=a$
$a^n=a^{n/2}\times a^{n/2}$ (n=2,4,...)
$a^n=a\times a^{n-1}$ (n=3,5,...)
$a^n=1/a^{-n}$ (n integer, n<0)
$a^y=$ **cases**:
$a_0\neq 0: 1/a_0\times\Sigma_{0\le j\le n}c_j$ **where** ($c_0=1$;
$c_j=(y-j+1)/i\times c_{j-1}\times b$ **if** $0<j\le n$)
where ($b=<0:0,a_1,a_2,...,a_p>$; c_k tps)
$T: x^y\times b^y$ (k=low(a); $b=<0:a_k,...,a_p>$)
(x=tpsvar,n=tpsord)
($a\neq$tps(0), y **not** integer)

with special cases:

$tps(0)^y=0$ ($y\neq 0$)
$tps(1)^y=1$ (y arbitrary) ,

and, finally, differentiation and integration:

$\partial_x a=c$ ($c_k=(k+1)\times a_{k+1}$, n=p−1)
(n=max[0,p−1], x=tpsord)
$\int_x a=c$ ($c_0=0$; $c_k=a_{k-1}/k$ if k>0)
(n=min[tpsord,p+1], x=tpsord)

The semantics of all of the above commands is exemplified by that for a+b: "an expression of the form a+b, where a and b are of type *tps* and have the respective formats $<0:a_0,...,a_p>$ and $<0:b_0,...,b_q>$ for some p and q, transforms to $c=<0:c_0,...,c_n>$ where n is defined to be the minimum of tpsord and the maximum of p and q and where c_k is the sum of a_k and b_k; the final result is then obtained by coercing c to c' of type *tps* which has trailing 0's deleted."

Next, we coerce algebraic expressions by breaking them down recursively until special cases result:

(u,v) **not** tps
tps(u+v)=tps(u)+tps(v) (u,v≠0)
tps(u−v)=tps(u)−tps(v) (u,v≠0)
tps(u×v)=tps(u)×tps(v) (u,v∉{0,1})
tps(u/v)=tps(u)/tps(v) (v≠1)
$tps(u^m)=tps(u)^m$ (m > 0, m integer)

We now add coercion functions for common functions:

x=tpsvar; i ε [0,1,...,tpsord]
tps(e^x)=<i:1/i!> (e literal)
$tps(\cos(x))=<i:1/i!\times(-1)^{i//2}\times\delta_{(i\ even)}>$
$tps(\sin(x))=<i:1/i!\times(-1)^{i//2}\times\delta_{(i\ odd)}>$
...

Here δ_b is similar to Kronecker delta and defined as follows:

$$\delta_{TRUE}=1; \qquad \delta_{FALSE}=0$$

The suffix operators even and odd may be introduced by:

syntax suffix-op: ('even','odd') 900
i even=**if** i is 2n **then** TRUE **else** FALSE (i,n integer)
i odd=¬(i even) (i integer)

Finally, we create a function which will compute the Taylor series expansion of e around the point x=0 to n terms:

taylor(e,x,n)=($\Sigma_{0\le i\le m}$(a.i)×x^i **where** (tpsvar=x,
tpsord=n;a=tps(e); m=length(a))
(e expression,x identifier, n ε [1,2,...])

m,n ε [1,2,...]; i ε [1,...,m]; j ε [1,...,n]
$a_{i,j}$ **is** type-*f*
type *f*-matrix **is** <[i,j]: $a_{i,j}$>
f-matrix(x)=x (x **is** <[i,j]: $a_{i,j}$>)

'declarations'
p,q ε [1,2,...]; k ε [1,...,p]; l ε [1,...,q]
(A,B,C) *f*-matrix
A=<[i,j]:$a_{i,j}$> ($a_{i,j}$ type-*f*)
B=<[k,l]:$b_{k,l}$> ($b_{k,l}$ type-*f*)
C=<[i,l]:$c_{i,l}$> ($c_{i,l}$ type-*f*)

'operations with matrices'
A+B=C ($c_{i,l}=a_{i,l}+b_{i,l}$) (m=p, n=q)
A−B=A+(−B)
A×B=C ($c_{i,l}=\Sigma_k a_{i,k}\times b_{k,l}$) (n=p)
A/B=A×(1/B)
...
'operations with scalars'
x scalar
x+A=A+x=matrix(<[i,j]:$a_{i,j}+x\times\delta_{i=j}$>)
x−A=x+(−A)
x×A=A×x=matrix(<[i,j]:x×$a_{i,j}$>)
...
'matrix functions'
transpose(A)=matrix(<[j,i]:$a_{i,j}$>)
trace(A)=$\Sigma_i a_{i,i}$ (m=n)
I_n=A ($a_{i,j}=\delta_{i=j}$,m=n)
det(A)= $\Sigma_i(-1)^{1+i}\times a_{1,i}\times$minor(A,1,i) (m=n)
minor(A,p,q)=det(matrix(<[i,j]:$a_{i,j}$ | i≠p∧j≠q>))
(p,q ε [1,2,...,n]; n=m)
χ(A)=det(A−λ×I_n) (m=n)
submatrix(A,u,v)=matrix(<[i,j]:$a_{i,j}$ **for** i ε u **for** j ε v>)
if u⊂[1,...,m] ∧ v⊂[1,...,n]
...

Figure 5.
Matrices over *f* as a Type Extension
(*f* denotes a ground field, e.g.
integer, poly, raf, complex, etc.)

a) Initialization:

n,k integer; $n\ge k$; $k\ge 1$
a **is** [a_1,...,a_n]; $a_i\ne a_j$ ($i\ne j$)

b) $S_k(a)$=totality of distinct sequences of k distinct elements chosen from [a_1,...,a_n]:

$$S_0(a)=\{[]\}$$

$$S_k(a)=\cup_{1\le i\le n}\{[a_i]\ \textbf{join}\ S_{k-1}([a_j\ \textbf{for}\ 1\le j\le n \mid j\ne i])\}$$

c) $S_{k,n}$ = totality of sequences of k distinct integers chosen from 1 to n:

$$S_{k,n}=S_k([1,2,...,n])$$

d) perm(a) = set of all permutations of [a_1,...,a_n]

$$perm(a)=S_n(a)$$

e) $Q_k(a)$ = totality of sequences of k distinct elements chosen from [a_1,...,a_n] with strictly increasing subscripts:

$$Q_k(a)=\{x\in S_k(a) \mid \forall_{1\le i<k}(x.i<x.(i+1))\}$$

f) A more efficient definition of $Q_k(a)$, using $Q^i{}_k(a)$= subset of sequences in $Q_k(a)$ having a_i as first element:

$$Q_k(a)=\cup_{1\le i\le n-k+1} Q^i{}_k(a)$$

$$Q^i{}_1(a)=[a_i] \qquad (1\le i\le n)$$

$$Q^i{}_k(a)=[a_i]\ \textbf{join}\ \{Q^j{}_{k-1}(a)\ \textbf{for}\ i<j\le n-k+2\} \qquad (1\le i\le n-k+1, k\ge 2)$$

g) $Q_{k,n}$ = totality of strictly increasing sequences of k integers chosen from 1 to n:

$$Q_{k,n}=Q_k([1,2,...,n])$$

h) $E_k(a)$ = k^{th} elementary function of [a_1,...,a_n]:

$$E_k(a)=\Sigma_{S\in Q_k(a)}\Pi S$$

or, $$E_k(a)=\Sigma_{S\in Q_{k,n}}\Pi_{i\in S} a_i$$

Figure 6. Combinatorial Examples
(key: [a] **join** [b,c] → [a,b,c];
[a] **join** {b,c} → {[a] **join** b, [a] **join** c})

7. CURRENT IMPLEMENTATION

The present SCRATCHPAD system has evolved from earlier versions [3] and [8] and has been implemented in LISP primarily by J.H. Griesmer and the author. The system runs on the VM/370 and TSS time-sharing systems at the IBM Research Center, Yorktown Heights, New York.

SCRATCHPAD includes significant portions of other systems such as REDUCE2[12], MATHLAB[7], and SIN[17] which have been adapted for use in SCRATCHPAD by J.H. Griesmer and the author. Among the symbolic facilities currently available to the user are: manipulation of rational functions, polynomial greatest common divisor, user-controlled simplification, differentiation (all originally from REDUCE); polynomial factorization (from MATHLAB); symbolic integration (from SIN); and new facilities for manipulation of sequences(Maps) and sets, solution of equations, unlimited precision rational arithmetic (up to 9000 decimal digits), access to the FORTRAN subroutine library (due to H.F. Trotter), truncated power series, and symbolic matrix and APL array operations. Two dimensional output of expressions is provided by a modified version of CHARYBDIS[15] (due to F.W. Blair).

The following components of the SCRATCHPAD language have been implemented much as described here since early 1971: interpreter and evaluator with Markov model and L r E statement format; explicit, implicit and virtual Maps; predicate sets; iterators; and syntax extensions (implemented using META/LISP [13] and META/PLUS[14]). The "history file" concept allowing conversational backtracking was implemented in early 1972. The following represent some of the more recent ideas and/or work in progress on the user language: types and type extensions; handling of implicit and virtual sequences in patterns; implementation of Sequences using generators; modifications to existing input and output translators to handle more general forms of syntax extensions; generalized Statements; compiler design and implementation.

Communication to SCRATCHPAD is currently via 2741 or 3277 terminals with either the EBCDIC or APL character set. Algebraic expressions are expressed on input as in FORTRAN. Subscripts, superscripts, etc. are linearized according to a few simply stated rules. Various character substitutions are used, e.g. SUM for Σ, PROD for Π, .(). for {}, etc. Output is in 2-dimensional form(except when intended for subsequent input to the FORTRAN compiler).

8. ACKNOWLEDGEMENTS

The author would like to thank S. Bourne (Cambridge University), R. Loos (University of Utah), A.C. Norman (Cambridge University and IBM Research), P.C. Gilmore, J.H. Griesmer, D.Y.Y. Yun, and others at IBM Research, for many useful discussions and suggestions with regard to this paper. The choice of Sequence as the basic aggregate object was inspired by conversations with P.C. Gilmore. Section 5 and parts of Section 4 were done with the help of A.C. Norman. The provision for parameterized types was suggested by R. Loos. The procedural language programming style(with **let** and **cases**) was motivated by [2]. The design of set theoretic notations was influenced by the work of Earley [6] and others.

REFERENCES.

[1] Anderson, R.H., "Programming on a Tablet: A Proposal for a New Notation", in [16]

[2] Allen, C.D., Chapman, D.N., and Jones, C.B., "A Formal Definition of Algol 60", Technical Report 12-105, IBM United Kingdom Laboratories Limited, Hursley Park, Winchester Hampshire, August 1972

[3] 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

[4] Brown, W.S., ALTRAN User's Manual, Bell Telephone Laboratories, Murray Hill, New Jersey, 1971, Third Edition, November 1973

[5] Burge, W.H., "Even More Structured Programming", IBM Research Report RC 4604, October 31, 1973

[6] Earley, J., "Relational Level Data Structures in Programming Languages", University of California, Berkelely, California, 1973

[7] Engelman, C., "The Legacy of MATHLAB 68", in [18]

[8] Griesmer, J.H., and Jenks, R.D., "SCRATCHPAD/1 - An Interactive Facility for Symbolic Mathematics", in [18]. (Also available as IBM Research Report RC 3260)

[9] Griesmer, J.H., and Jenks, R. D., "The SCRATCHPAD System", IBM Research Report RC 3925, July 1972

[10] Griesmer, J.H., and Jenks, R.D., "SCRATCHPAD: A Capsule View", in [16]. (Also available as IBM Research Report RC 3972, August 1972)

[11] Hall, A.D., Solving a Problem in Eigenvalue Approximation with a Symbolic Algebra System, SIGSAM Bulletin No. 26, June 1973

[12] Hearn, A.C., "REDUCE2: A System and Language for Algebraic Manipulation", in [18]

[13] Jenks, R.D., "META/LISP: An Interactive Translator Writing System", IBM Research Report RC 2968, July 1970

[14] Jenks, R.D., "META/PLUS: The Syntax Extension Facility for SCRATCHPAD", Proceedings of the IFIP Congress 71, C.V. Freiman (Ed.), North-Holland, Amsterdam, 1972, pp. 382-384. (Also available as IBM Research Report RC 3529)

[15] Millen, J.K., "CHARYBDIS: A LISP Program to Display Mathematical Expressions on Typewriter-like Devices", in Interactive Systems for Experimental Applied Mathematics, M. Klerer and J. Reinfelds, eds., Academic Press, New York, 1968, pp. 79-90

[16] Morris, J.B. and Wells, M.B., ed., Proceedings of a Symposium on Two-Dimensional Man-Machine Communication, SIGPLAN Notices, Volume 7, Number 10, Association for Computing Machinery, New York, October, 1972

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

[18] Petrick, S.R., ed., Proceedings of the Second Symposium and Algebraic Manipulation, Association for Computing Machinery, New York, March 23-25, 1971