

Newsletter

Volume 1 Number 3

May 15, 1986

Scratchpad II

IBM Research

In this issue ...

This Issue	1
Power Series and Fast Numeric Evaluation of Algebraic Functions and Linear Differential Equations	1
A History of the SCRATCHPAD Project (1977-1986)	3
Some Questions and Answers About Scratchpad II	12
Scratchpad II System Status	14
Some Scratchpad II Constructor Name Abbreviations	18
Algebra Snapshot: Partially Derivative Work	18
Current and Recent Visitors to the Scratchpad II Group	20
Conference Announcements	21
The 1986 ACM-SIGSAM Symposium on Symbolic and Algebraic Computation	21
COMPUTERS & MATHEMATICS	22

The Scratchpad II Newsletter

An informal quarterly publication of the Computer Algebra Group, Knowledge Systems, Computing Technology Department, IBM Thomas J. Watson Research Center, Box 218, Yorktown Heights, New York 10598. © Copyright International Business Machines Corporation, 1986. All rights reserved.

Editor: Robert S. Sutor

Volume 1, Number 3

May 15, 1986

The Scratchpad II Computer Algebra Group

Manager

Richard D. Jenks
IBM VNET: JENKS at YKTVMZ
CSNET: jenks@ibm.com
914-945-1233

Algebra

Barry M. Trager
IBM VNET: BMT at YKTVMZ
CSNET: bmt@ibm.com
914-945-1868

Patrizia Gianni
IBM VNET: GIANNI at YKTVMZ

Compiler

Stephen Watt
IBM VNET: SMWATT at YKTVMZ
CSNET: smwatt@ibm.com
BITNET: smwatt@yktvmz
914-945-3405

Interface

William H. Burge
IBM VNET: BURGE at YKTVMT
914-945-1944

Rüdiger Gebauer
IBM VNET: GEBAUER at YKTVMZ
914-945-3882

Moss E. Sweedler

IBM VNET: SWEEDLE at YKTVMZ
914-945-2471

Interpreter

Robert S. Sutor
IBM VNET: SUTOR at YKTVMZ
CSNET: sutor@ibm.com
BITNET: sutor@yktvmz
914-945-2360

Michael Lucks

IBM VNET: LUCKS at YKTVMZ
CSNET: lucks@ibm.com
BITNET: lucks@yktvmz
914-945-3720

System

Martin L. Brock
IBM VNET: MBROCK at YKTVMZ
914-945-3124

Mohamed Mobarak

IBM VNET: MMOBARA at YKTVMZ
914-945-3204

Consultants

David and Gregory Chudnovsky
Professors of Mathematics,
Columbia University.
IBM VNET: SPAD at YKTVMZ

Victor Miller
IBM Research
IBM VNET: VICTOR at YKTVMZ

Cooperative Student

Vladimir A. Grinberg
Columbia University.
IBM VNET: VLAD at YKTVMZ

Secretary

Evelyn Zoernack
IBM VNET: EVELYNZ at YKTVMT
914-945-1187

This Issue

David and Gregory Chudnovsky continue their reports of work with **Scratchpad II** in their article "Power Series and Fast Numeric Evaluation of Algebraic Functions and Linear Differential Equations". This is their third contribution in as many newsletters and we look forward to their continued collaboration with the newsletter and the system.

Dick Jenks continues the account of the history of the **Scratchpad** project started by Jim Griesmer in the last newsletter. The new article discusses the period from 1977 to 1986.

The "Algebra Snapshot" this month is a bit different from the previous installments in that it demonstrates functions and results derived entirely from working within the interpreter. In "Partially Derivative Work", Moss Sweedler (visiting the Computer Algebra Group from the Department of Mathematics at Cornell University) shows how **Scratchpad II** was used to verify conjectures concerning differential forms in fields of positive characteristic.

Two announcements of conferences appear in this issue: "The 1986 ACM-SIGSAM Symposium on Symbolic and Algebraic Computation", which will be held in Waterloo, Ontario Canada, and the "COMPUTERS & MATHEMATICS" conference, which will take place in Stanford, California. Descriptions begin on page 21

Readers should note that the CSNET address for the **Scratchpad II** group members has been changed. The new address is IBM.COM and replaces the older IBM-SJ.ARPA. See the inside front cover for the addresses of individuals.

At the back of this newsletter, you will find a newly-developed reference card for **Scratchpad II** called "Getting Started." The card explains the basics of using the system and gives some examples of commands and operations to use in the interpreter. From its introduction: "Once you have learned a few commands and the operations relevant to the routines you wish to use, you can begin using **Scratchpad II** to solve real problems. A good way to learn is to experiment with the selection of routines and commands presented here."

Letters to the Editor about articles in the newsletter or about **Scratchpad II** are most welcome. Electronic mail is preferred and should be sent to the appropriate address as listed on the opposite page.

Robert S. Sutor

Power Series and Fast Numeric Evaluation of Algebraic Functions and Linear Differential Equations

by

David and Gregory Chudnovsky

We want to describe a facility for power series expansion of algebraic functions and solutions of linear differential equations with rational function coefficients, soon to be available in **Scratchpad II**. This facility is based on several algorithms that allow the computation of N terms in power series and Puiseux expansions of algebraic functions in $O(N)$ with $O(1)$ storage, and allow fast numeric evaluation of solutions of linear differential equations (including algebraic functions) everywhere on their Riemann surface. For the survey of earlier work on computations of power series expansions of algebraic functions and solutions of linear differential equations, that uses mainly the Newton method and requires $O(N \log N)$ operations, see Traub-Kung [1]. Our approach is based on linear (finite-term) recurrences that are satisfied by coefficients in power series expansions of solutions of linear differential equations.

We consider two groups of linear differential equations: (i) the first order matrix linear differential equations:

$$(1) \quad \frac{d}{dx} Y(x) = A(x) Y(x)$$

for an $n \times n$ matrix $A(x)$ with the coefficients from the rational function field $k(x)$ for a field k (of characteristic 0, see below), and (ii) the scalar n -th order linear differential equations

$$(2) \quad L[y] ::= \left(\sum_{i=0}^n a_i(x) y^{(i)}(x) = 0 \right)$$

for $a_i(x)$ from $k[x]$, $i = 0, \dots, n$.

Whenever $x = x_0$ is a regular singular point of (1) or (2) (i.e., all solutions of (1) or (2) are at worst algebraically singular at $x = x_0$, e.g., if $x = x_0$), is a regular point, the equations has a fundamental system of solutions with an expansion at $x = x_0$ of the form $(x - x_0)^\alpha (1 + c_1(x - x_0) + \dots)$ for a local exponent α determined by (1) or (2). In the case (2), the fundamental system of solutions at an arbitrary regular singularity $x = x_0$ is determined by the Frobenius method [2-3]. In this method, one looks at

$$L[(x - x_0)^\alpha] = (x - x_0)^\alpha f(x, \alpha),$$

where

$$f(x, \alpha) = f_0(\alpha) + \cdots + f_d(\alpha)(x - x_0)^d$$

Then the solutions of (2) forming the fundamental system correspond to roots α of an *indicial equation* $f_0(\alpha) = 0$, with the regular expansion

$$(3) y(x, \alpha) = (x - x_0)^\alpha \sum_{i=0}^{\infty} y_i(\alpha)(x - x_0)^i$$

and the coefficients $y_i(\alpha)$ determined by the recurrence

$$(4) \quad 0 = \sum_{j=0}^{\max(i,d)} y_{i-j} f_j(\alpha + i - j)$$

for $i = 1, 2, \dots$. An expansion (3) is slightly modified in the case of roots of the indicial equations differing by rational integers [2-3], when logarithmic terms might appear in (3).

In the matrix case (1), there are special fundamental systems of solutions, similar to (3), at regular singular points. The most important case of (1) is the so-called normal case, when all poles of elements of $A(x)$ (i.e., singularities of (1)), are of the first order:

$$A(x) = A_0 + \sum_{k=1}^l \frac{A_k}{x - s_k}$$

The canonical fundamental system of solutions of (1) at $x = s_k$ has the form

$$Y_k(x) = \left(\sum_{i=0}^{\infty} C_i (x - s_k)^i \right) (x - s_k)^{A_k}$$

where $C_0 = I$ and the matrices C_i determined through linear recurrences with coefficients polynomial in i .

All these expansions are well suited for symbolic generation by the **Scratchpad II** recurrence facility, and for their numeric evaluation in **G BF** (complex bigfloats) at complex x , within the convergence radius of the expansion (easily determined as the distance to the next singularity). To evaluate the power series expansion of algebraic functions using this method, one first derives the linear differential equation, with rational function coefficients, satisfied by an algebraic function. In **Scratchpad II** there exists already a facility that allows the generation

of a matrix equation (1) with $Y = (1, y, \dots, y^{d-1})$ for an algebraic function y , defined by an equation $Q(x, y) = 0$ of degree d in y [4].

The numeric part of the power series facility will allow one to evaluate a solution of a linear differential equation with given initial conditions at (a regular point) $x = x_0$, at another point $x = x_1$, whenever the path from x_0 to x_1 is specified (to obtain the values of a given branch of a solution). This is achieved by successive analytic continuation from x_0 to x_1 . The number of operations is estimated as follows:

Proposition: To analytically continue a solution of (1) or (2) from $x = x_0$ to $x = x_1$ along the path γ with a given precision M (of digits), one needs $O(M \log |\gamma|)$ operations, where $|\gamma|$ is the length of γ .

In characteristic p , power series solutions of (1) or (2) do not necessarily exist, and their existence is closely connected with algebraicity of all solutions. It is important, therefore, to look at mod p properties of denominators of coefficients in the expansions of solutions of (1) and (2) for $k = Q$ (see [5]).

More on this facility and its use will be reported. We want to thank B. Trager for constant help and discussions and R. Jenks for his interest and support.

References

- [1] Brent, R. P., and Traub, J. F., *SIAM J. Comput.* 9 (1980): 54 - 66.
- [2] Forsyth, A. R., *Theory of Differential Equations*, New York: Dover, 1959.
- [3] Chudnovsky, D. V., and Chudnovsky, G. V., *IBM Research Report* RC 11365 (Yorktown Heights, New York: September 13, 1985): 1-96.
- [4] *The Scratchpad II Newsletter*, Vol. 1, No. 2, Yorktown Heights, New York: IBM Corporation, January 15, 1986, p. 13.
- [5] Chudnovsky, D. V., and Chudnovsky, G. V., *IBM Research Report* RC 10645 (Yorktown Heights, New York: July 26, 1985): 1-96. Also *Lecture Notes in Mathematics*, Vol. 1135, New York: Springer-Verlag, 1985, pp. 52-100.

A History of the SCRATCHPAD Project (1977-1986)

by
Richard D. Jenks

1973: Beginnings

My writings relating to **Scratchpad II** go back to the fall of 1973 when I set out to write what I felt would be the definitive paper on the **Scratchpad** language [12]. The paper was a mixture of "what is," but mostly "what would be." In the introduction of the paper, I presented the language as an attempt to unify concepts from several languages into one single language. In re-reading this paper, I am struck by how well the current **Scratchpad II** language and system actually achieve the goals there set forth. On the other hand, back then I had little conception of what **Scratchpad II** would become nor any idea of what effort would be required to accomplish these goals. While I am very pleased to have played a role in the evolution of **Scratchpad II**, this article will make it clear that the current **Scratchpad II** language and system is the product of the ideas and efforts of many people.

One person who influenced my early work on **Scratchpad II** was Rüdiger Loos, then a member of the Utah Computational Physics Group. Both of us were preoccupied with the design and compilation of formal description languages for computer algebra systems. In particular, we were interested in the defining of new computational "domains" using rewrite rules (in this article, I will use the term *domain* to refer to algebraic structures for computation, e.g. "the integers," "polynomials over the integers," etc.). Rüdiger's paper [18] served as a precursor to his work in **ALDES**, while suggesting extensions beyond **ALDES** in the direction of **Scratchpad II**.

Rüdiger espoused the view that the *simplification* of an expression should mean the choice of an appropriate algebraic representation for the expression (e.g., as a matrix of polynomials, or a polynomial of matrices). This idea inspired the introduction of the **FORMAT** statement to **Scratchpad** [10] which caused output expressions to be converted to any of a wide range of algebraic structures. The syntax for describing these algebraic structures was later modified, extended, and called a *mode* in **Scratchpad II**.

My earliest development work on **Scratchpad II** goes back to my sabbatical leave to the Computational Physics Group at Utah during the academic year 1976-1977. I worked principally with Martin Griss and Michael Rothstein. Tony Hearn was himself away during that period.

My purpose in visiting Utah was to study the implementation of **MODE-REDUCE** [8], a system which provided for the definition and compilation of domains. In **MODE-REDUCE**, domains were created *statically*, that is, by compiling source code. My goal was to extend **MODE-REDUCE** so that domains would be created *dynamically* in response to user input (e.g., so that the domain "polynomials over the rational numbers" would be created in response to a user input such as $x + 1/2$).

It was also important that domains could be parameterized, e.g. *matrices of D*, where **D** could be the integers, the rational numbers, or any appropriate domain. Here "appropriate" means that **D** would belong to a specified class of domains. In particular, **D** should be a **Ring**. The programmer could write a matrix-addition function to add two compatible matrices and this function would execute properly regardless of the choice of ring **D**. Such a function is often called *polymorphic*. Polymorphism can lead to great economy as one need not have a different implementation for every possible type of matrix element: the same code is used over and over and will automatically work when a new ring is used as parameter.

In the first design [13], domains were represented by *transfer vectors*. These were vectors with entries labeled by generic operations and having functions as values. The matrix-addition function would be passed an extra argument, a transfer vector representing the coefficient domain. To do a coefficient addition, the function branched indirectly through the slot in the transfer vector labeled by "+".

At Utah, I did some experimental designs and finished a prototype compiler for a formal description language without parameterized types [14], (see Figure 1 on page 4).

At the first **MACSYMA** User's Conference, at Berkeley in the summer of 1977, I was introduced to David Barton and Richard Zippel, who were planning similar extensions to **MACSYMA**. We had many discussions about implementation issues and Barton pointed out the need for transfer vector slots to contain general *funargs* (function-environment pairs).


```

MODULE 'Matrix
IMPORT RationalFunction;
FOR (i,j,n,m,k,l) IN 1...INF DEF $(
  DEFMODE Matrix(n,m) =:
    (FOR i IN 1...n COLLECT
      (FOR j IN 1...m COLLECT RationalFunction));
  ..
  FOR A IN MATRIX(m,l)
    & B IN MATRIX(l,n)
    & C IN MATRIX(m,n) DEF
      A*B =: C WHERE C(i,j) =:
        +/(FOR k IN 1..l COLLECT A(i,k)*B(k,j));
  ..
  FOR (A,C) IN MATRIX(n,n) & p IN Integer DEF $(
    A+0 =: IDENT(n)
    A+1 =: A
    A+p =: C WHERE C =: IF p>1 THEN .. ELSE ..
  )$;
  ..
)$;

```

Figure 1.

1977: MODLISP

When I returned to Yorktown Heights in the fall of 1977, Jim Griesmer had left our group to become Manager of Education at IBM Research. David Yun later agreed to manage our small effort and he did this until 1983, when he left IBM Research to become Chairman of Computer Science at Southern Methodist University. David and I were keen on rebuilding *Scratchpad* from scratch. My initial goal was to build a system to do efficient computations with dynamically created domains. David's vision, on the other hand, was a system which would provide *knowledge modules*, that is, specially generated packages of compiled code that would be useful for specific applications, e.g. coding theory. We decided to implement the new system in *LISP/370* (which was extended over the years and became the IBM product *LISP/VM* in the summer of 1984). Our new implementation was dubbed *NEWSPAD* by David. "SPAD" was the common contraction of the old system name *Scratchpad* which was too long for CMS file names (and NY state license plates!).

The objective of David Barton and Rich Zippel at MIT in building a new algebra system was principally the performance of sophisticated mathematical calculations, with little or no regard to ease of use. My objective, while similar, was more ambitious. I wanted to develop a complete language and system with an easy-to-use user interface. I believed (and still believe) that an excellent interface is paramount for widespread use.

My experience at Utah led me to concentrate on formalizing some of the fundamental programming issues with which I was concerned. *MODLISP* [16], the result of this work, was designed to be a variation on *LISP*, plus a notion of "MODEs" which generalizes the normal notion of types in conven-

tional programming languages such as *PL/I* and *Pascal*. A *mode* is a syntax for designating a class of domains. If a mode contains no *pattern variables*, that is, special identifiers called, say, **1*, **2*, ..., the mode simply denotes a domain. For example, mode *P(I)* denotes the domain "polynomials over the integers." Modes containing pattern variables, however, describe classes of domains. Thus **1* denotes the class of all domains, *P(*1)* denotes polynomials over other domains, *M(P(*1))* denotes matrices of polynomials over other domains, and so on. Modes were thought to be of key importance for interactive ease-of-use.

Another central idea of *MODLISP* was that of a *modemap* to describe generic operators, e.g.,

determinant: **2* → **1* from **2*

Modemaps usually have associated predicates which qualify the pattern variables, e.g. for the above:

**2* is a square matrix of dimension **3* over **1*
 AND **3* is a positive integer
 AND **1* is a ring.

The "from"-clause describes the *domain of computation* where the domain-specific function is implemented.

As in *LISP*, the semantics of *MODLISP* are defined through evaluation. *LISP* evaluation can be described as mapping a pair (x, e) into a new pair (x', e') , where x denotes an S-expression to be evaluated, x' its value, and e and e' denote the initial and final environments. These environments record the values of variables before and after the evaluation. In *MODLISP*, evaluation sends triples (x, m, e) into triples (x', m', e') where m and m' denote the initial and final modes associated with x and x' , respectively.

One novelty of *MODLISP* was the provision for both rewrite assignment $a == b$ and normal assignment $a := b$ as primitives. Rewrite assignment means that the value of a is obtained by rewriting a by the value of b at *evaluation time*. Normal assignment, on the other hand, means that the value of a is obtained by rewriting a by the value of b at the *time of assignment*.

Another interesting property of *MODLISP* was that its evaluator served simultaneously as an interpreter and compiler. Given values for all arguments of a function, the evaluator would invoke the function to immediately produce a value. If, however, at least one argument value was not available, the evaluator would instead compile code to produce a value later.

Normal interpretation became a special case of compilation, done by a top-down tree walk using what is often called *symbolic execution*.

My plan was to build a prototype system based on **MODLISP** which emphasized semantic rather than syntactic issues (syntax design was thought to be a much easier problem). A *one-language* assumption was, however, fundamental: one language would serve both the interactive user and system implementer.

When James Davenport (then a graduate student at Cambridge) had his initial visit as a summer student in 1978, David Yun and I asked him to devise some method of checking for algebraic correctness (e.g., matrices of polynomials are allowed but matrices of strings are not). His initial design provided for an *attribute set*, a set of identifiers for describing algebraic properties of a domain, e.g., **PLUS-COMMUTATIVE**, **ONE** and **RING**. Certain attributes such as **RING** were to be *implied* (e.g., a domain would have the **RING** property if it had **PLUS-COMMUTATIVE**, **ONE**,...).

That summer was a period of major activity. An initial implementation of **MODLISP** was carried out with James Davenport, following discussions with David Yun, Arthur Norman (visiting from Cambridge University), and Josh Cohen, a summer student who would be with us for five consecutive years. This was the first of many intensive summers when we combined ideas and efforts to make considerable progress toward implementation. The next summer Josh Cohen and I implemented a complete **MODLISP** prototype system with domain towers of polynomials, rational functions, matrices, and algebraic numbers and functions, with an interactive language. This system provided a complete implementation of *resolve* for modes that could then be constructed. This would systematically determine appropriate algebraic representations for user input, e.g., polynomials over the rational numbers for $x+1/2$. This first prototype system, however, was to have a short life.

1979: Categories

In August, 1979, David Barton visited us as a summer student after a year at MIT where he implemented a new rational function package for **MACSYMA** (which, however, never became part of official **MACSYMA**). He had gained considerable experience in developing paradigms and techniques for implementing computer algebraic algorithms. The two of us implemented a substantial facsimile of his system in **LISP/370**. While his sys-

tem had no convenient user interface, it had notions of *alg-structs*, collections of operations representing groups, rings, and fields.

By the September of 1979, when James Davenport arrived as a Postdoctoral Fellow for a year at Yorktown, we were all convinced that *alg-structs* should be incorporated into our design. James suggested that we call the resulting structure a *category*, so named for its simplistic resemblance to the object of the same name in mathematics. Correspondingly, domain constructors (i.e., functions that create domains) were originally called *functors*.

Our categories included not only operators but also *attributes*, e.g., **commutative**("**") to assert that the "**" operator of the domain was commutative. Initially, attributes were used to designate axioms for domains, presumably for formal description or eventual program verification. Later (in 1981) we decided their practical uses were more important. An attribute would be given, for example, whenever the choice of an algorithm within a domain depended on it. In particular, **commutative**("**") would be given so that an algorithm could assume that multiplication was commutative. Of course, if the

Scratchpad II People: 1977-1986 (including > 1 month visitors)

Richard D. Jenks	6/68-
David Y. Y. Yun	9/73-2/83
David R. Barton	9/79, 7/80-8/80
Josh Cohen	S78-S82
James H. Davenport	S78, 9/79-8/80, S81-S85
Barry M. Trager	1/81-
Richard Anderson	6/82-9/82
David V. Chudnovsky	1/83- (consultant)
Gregory V. Chudnovsky	1/83- (consultant)
Patrizia Gianni	3/83-2/84, 7/84-10/84, 7/85-10/85
C. Andrew Neff	6/83-9/83
Victor S. Miller	8/83-7/84
Christine J. Sundaresan	9/83-10/84 (coop)
Scott C. Morrison	1/84-8/84, 6/85-9/85
Michael Rothstein	1/84-8/84
Robert S. Sutor	3/84-
Stephen Balzac	6/84-9/84
Albrecht Fortenbacher	9/84-8/85
Stephen M. Watt	11/84-
Michael Lucks	2/85-
Martin Brock	6/85-
Mohamed Mobarak	6/85-8/85, 1/86-
Vladimir A. Grinberg	7/85- (coop)
Julian A. Padget	1.5: 7/85
Jean Della Dora	9/85-10/85
Moss E. Sweedler	9/85-
Rüdiger Gebauer	12/85-
William H. Burge	1/86-

multiplication was not commutative, *commutative*("*") would not appear. Attributes are extremely desirable and useful for formal description and use of algebraic constructs.

In late 1979, James and I began implementation of a second prototype system based on MODLISP with categories and functors. This time, however, we designed a complete programming language including syntax for categories and functors, similar to, but semantically much different from, *Scratchpad II* today (see Figure 2 on page 7-(a)). James' contributions to system design, implementation, and performance, during that year and in summers to come were substantial. James has described his design of the representation and instantiation of categories and domains ([6]), and this is currently used in *Scratchpad II*. Domains are modeled by lists associating operations with implementations. Categories are represented by "shells": vectors whose elements are labeled by the operations of the category but are otherwise empty. Domains are copies of the shells where labeled slots are filled with funargs, and whose initial entries contain useful data. James' design deals with such problems as *operator subsumption*. For example, category *Group* has 3 operations of the form $x ** n$. One is for n an integer (from *Group*). The two others are inherited: one where n is a non-negative integer (from *Monoid*), the other where n is a positive integer (from *SemiGroup*). Questions dealt with include: Which definition is used in what context? Can the less-general operations be completely forgotten?

By the end of the summer of 1980, we had also implemented an interactive language with automatic type deduction similar to that in the present system and a compiler for functors and categories. Our system prototype provided rational functions, polynomials, algebraic numbers, matrices, gaussians, and various number domains, all on 5 pages of source code! It also used type resolution methods developed in the earlier system to allow convenient use of modes interactively.

During this year, David Barton was in his first year of graduate school at Berkeley and was developing yet another version of a computer algebra system, this time built on *Franz Lisp* and called *Andante* (apparently, *Allegro* and *Presto* were waiting in the wings!). David Barton returned to IBM in the July of 1980 for 6 weeks. Rather than have him write in code for the existing *NEWSPAD* system, I asked him instead to write "pseudo-*NEWSPAD*," that is, to use current *NEWSPAD* as much as possible, but to invent terminology needed to accomplish whatever generality and efficiency he required or desired.

David extended our 5 pages of *NEWSPAD* algebra code to 40 pages of pseudo-*NEWSPAD* code. David's non-working code, however, left little doubt that the implementation had to be extended to allow *multiple views*. For example, programs must be able to view domain *Integer* either as an ordered set, a ring, or both simultaneously.

1981: Programming Language Design

In the fall of 1980, I found the algebraic specification component of our language lacking in organization and clarity of concepts. (see Figure 2-(a)). When Barry Trager joined our group in January, 1981, we pored through the literature on programming languages and quickly realized that although our language required more general constructs than those offered by any existing language, the designs of the abstract datatype languages Russell [7] and CLU [20] most closely matched our needs. One afternoon, inspired by re-reading a Russell paper, it seemed obvious that categories and domains should be defined as functions with categories serving as the type of domains.

What followed was an intensive period of work with Barry. Together we worked to redefine all existing ideas about the computer algebra system and to look at it in a new light. We had many rewarding discussions with Jim Thatcher, who had started many years before us building a general foundation for semantics in computer programming languages, and later, for abstract data types. We also had fruitful discussions with Jerry Archibald (IBM Research), D. Ehrich (then at University of Dortmund), David B. Saunders (then of RPI), David Yun, and Richard Zippel. The culmination of this work was [17] which has served as the basis for the *Scratchpad II* programming language design (Figure 2-(b)).

The importance of this new design is the elevation of domains and categories to be computed objects which themselves have operations defined on them. The notion of multiple views is incorporated in the notion of "Join," an operation defined on categories. For example, *Integer* is declared to be of category *Join(OrderedSet, UniqueFactorizationDomain, EuclideanDomain, DifferentialRing)* which allowed it to be viewed in these four ways.

The name of a category is now fundamentally important. The category named *Ring*, for example, is understood to have all the axioms associated with *rings*, whether explicitly mentioned by attributes in the category definition or not. Unlike our previous design, it would not be possible to create category


```

functor GradedExt(Ring(r),OrdMonoid(s)):Ring(rr);
  extends
    FreeModule(r,s)
  declare
    p: rr
  operations
    degree(x): s
    content(x): r
  assert
    if r of IntegralDomain
    then rr of IntegralDomain
  specialcase
    s: NonNegativeInteger
  define
    1 -> (List Term(0,1)): rr
    if r of GcdDomain then
      content(p) ->
        (p=0 => 0; gcd/(u.c for u in p) )

```

(a) SPAD circa 1980

```

GradedExt(R: Ring, E: OrdMonoid): Ring with
  degree: $ -> E
  content: $ -> R
  if R has IntegralDomain then IntegralDomain
  == FreeModule(R,E) add
  Rep := List Record(k: E, c: R)
  p: $
  1 == [[0, 1]]
  if R has GcdDomain then
    content(p) ==
      (p=0 => 0; gcd/[u.c for u in p])

```

(b) current SPAD

Figure 2.

Ring by separately listing its component operations and attributes appearing in its definition.

The previous algebraic specification language had many keywords (David Barton's pseudo-NEWSPAD contributed 18!). Eventually every keyword proposed by Barton was either eliminated or replaced by an operator. The language has evolved over time toward an expression-based language using keywords only for control constructs (looping, exiting, branching) and otherwise having only generic operators which are user-definable.

During the summer of 1981 and those that followed, James Davenport furthered his work on the implementation of domains, extending them to provide mechanisms for multiple views of domains as runtime objects. James implemented the code for instantiation of domains used in current *Scratchpad II*. A domain is instantiated as a *principal view* in which all other views are imbedded. Some views and operations within particular views produced by a domain constructor are conditionally generated depending on values of parameters passed to the domain constructor. In our current implementation, functions which are passed domains as arguments have the responsibility of extracting the appropriate views required for their implementation.

We had long been interested in the design and implementation of *packages*, that is, collections of polymorphic functions which belong to no specific domain but rather are parameterized by domains

having certain algebraic properties. An example of such a function is *solve(eqs, vars)*, used to solve *eqs*, a set of polynomial equations over a field (call it *F*) for *vars*, a set of variables. This function is parameterized by the domain *F*. Our first implementation of packages in the 1980 system was quite limited and did not satisfy our inherent need for multiple views. Even after our new algebraic language was designed, it was almost a year later (1982) before we were able to identify the mechanisms in packages well enough to implement them. Indeed, packages, originally thought of as collections of polymorphic functions, have evolved to be nothing but a special kind of a domain. Our current view is that packages, like domains, can be passed to functions and that even views of packages would sometimes be needed. We now regard packages as domains without elements, or what is equivalent, domains with no operations on their elements. Stephen Watt has described packages as *boring domains*, that is, domains which are so dull that they don't have any operations to perform on their elements!

In the fall of 1981, I started to reimplement the compiler for the new abstract datatype design, our third and last fresh start. The reimplementation was centered solely on the development of a compiler for algebraic facilities and not on any aspects of an interpreter. This might be considered a step backward since the original implementation served both as an interpreter and compiler for the language. Our *one-language* requirement, however, was now modified with respect to evaluation semantics. The programming language (dubbed *SPAD*) was to be *strongly typed*, that is, the types of variables and expressions within a function were required to be uniquely determinable at compile time. The interactive language, on the other hand, while having the same syntax, would allow modes and automatic coercions.

I believe that good language design is difficult work and that any good initial design we might propose would require many iterations before it would stabilize toward one we would be happy with in the long run. To help ensure that we would eventually produce a good language, we decided to force ourselves to use it day-to-day. We chose to implement the *SPAD* compiler in *BOOT*, a typeless language we developed which translates directly into *LISP* but has the same control structures as *SPAD*. Exploratory work in writing new algebraic packages suggested many changes in *SPAD*. At the same time, writing the compiler in *BOOT* suggested several improvements in the expressive power of the language. Using this process, many improvements

were made to the language and these required several transformations of existing algebra and compiler code.

The language design contains some unconventional biases which I have accumulated over the years, e.g., that local variables assigned within a function do not require a type declaration if their type is defined by that of the right-hand-side, that variables assigned within a function are local by default, that indentation should be used to logically group program steps, and that "is" should be a primitive predicate for pattern-matching and structural assignment [12].

Several simplifications have been proposed recently by Stephen Watt. Perhaps surprisingly, most of these simplifications arise by making things more general rather than specific. For example, allowing the overloading of functions and record field names follows from the fact that now all named constants can be overloaded. Certain constructs which used to be viewed as language primitives are now to be handled as simple domains. For example, the original notation $a..b$ required a and b to be integers. We will soon regard $..$ as an infix operator for constructing an *interval* whose meaning is domain-dependent. Likewise, iteration (e.g., *for x in u repeat...*) and APL-reduction (e.g., $+/u$) used to require that u be a list or vector. Now it is defined categorically and may be used for any *aggregate*, that is whenever u belongs to a domain with appropriate operations. Except for some compatible extensions, our language has very substantially stabilized.

By early 1983, it was clear that the new compiler implementation was deficient. One fundamental mistake was that, instead of using pattern-based modemaps as in MODLISP, we decided to bring all instances of modemaps for domains in scope. This strategy led to a large number of modemaps to consider, resulting in unacceptable performance. The implementation was also incomplete with respect to facilities for error checking and uniqueness testing.

Fortunately, significant literature was mounting describing the type analysis for Ada, a language sharing some of our implementation difficulties. Among the interests of Victor Miller, who took an internal IBM sabbatical to work in our group in 1983, was to investigate how these new methods could be used as a basis for a new compiler for NEWSPAD. Victor later implemented a dynamic programming algorithm for type analysis of nested function calls. This algorithm was the starting point for Stephen Watt who joined us in late 1984 to implement a new compiler for the system.

Through the years, the size of the group had typically been 2 or 3. By March, 1984, the group had grown to have 7 members! Victor Miller had taken an internal IBM sabbatical to work in our group in 1983-1984; Patrizia Gianni was a one-year visitor from Pisa; Chris Sundaresan joined us as a coop graduate student from NYU; Scott Morrison, a Berkeley graduate student, and Michael Rothstein, a professor at Kent State, visited us for 8 months; and Bob Sutor, a graduate student on leave from Princeton, joined us after a brief stint in IBM Corporate.

By late 1982, I had finished a first draft of the *Programming Language Reference Manual* [3]. Now, early in 1984, we launched what might be called **The First Scratchpad II Group Project**: the completion of the reference manual. Chris Sundaresan was the coordinator of this project, and edited the final version of the document. Each group member was given a chapter to rewrite and was to lead the group discussion on the topics contained therein. Many weeks of long, spirited sessions followed where we discussed (and often argued about) the syntactic and semantic points in the manual. Despite the intrinsic difficulties with such a committee approach toward design and documentation, we not only accomplished a great deal but learned a lot in the process.

One result from our group meetings was the emergence of what we now believe to be the correct notion of *subdomain*. Previously, we had regarded a subdomain to be a domain in its own right, that is, with its own exported operations and function definitions. Now, we regard a subdomain to be a domain plus a predicate. This one seemingly minor redefinition has made a significant impact on the semantics of the language, and the simplicity and clarity of code. This finding led to the discovery in 1985 by Stephen Watt of a particularly simple definition of categories and packages in terms of domains and subdomains ([4], Chapter 3).

1983: The Interactive Language and Interface

David and Gregory Chudnovsky had become consultants to our group in early 1983 and quickly became leading proponents of the use of computer algebra in mathematical research. The Chudnovskys proposed a "get-together" of some dozen or so of their friends, leading mathematicians and physicists, for a NEWSPAD debut. The meeting won IBM Research support (thanks to Shmuel Winograd), and, through Jack Schwartz's interest, NYU was picked as the location. We planned for a

maximum of 75 people, but when NSF and others pledged support for it, we decided to make it a conference, calling it “.Computer Algebra as a Tool for Research in Mathematics and Physics,” and inviting other computer algebra systems to be represented. After a poster was distributed only 7 weeks before the April, 1984, date, registrations became so heavy we had to move the event from NYU to the Cooper Union. Over 500 attended the opening session.

One minor consequence of the New York meeting was the demise of the name **NEWSPAD**. Although we had wanted to use this name for the new system, we were told that it would have to be subjected to a copyright search. Since the previous name was already cleared, we opted to call the system **Scratchpad 84**, and, when it became clear that it would take much longer than one year to finish the system, we changed the name to **Scratchpad II**.

As the NYU meeting was initially planned to be the “coming-out” party for **NEWSPAD**, we busied ourselves with getting facilities together for the show. A major effort was made toward the implementation of the interactive language and interface for the system.

In early 1983, I had proposed a design for an interactive language for **Scratchpad II** similar in spirit to old **Scratchpad** but based on *maps* (similar to “projections” in **SMP** [1]), ideas from **MODLISP** (modes and rewrite/normal assignment), and with a new design for indexed streams. The rather lengthy design document was later condensed into [19] for presentation at **EUROSAM '84**.

Our overall language design is succinctly characterized by:

interactive language = compiled language + freedom

where freedom essentially means that

1. automatic conversion of an object of one datatype to another is allowed if it makes sense (that is, an integer can be converted to a rational number but a string cannot be automatically converted to a matrix), and
2. declarations are optional and allow general modes, not just domains.

These two differences make the language processors for the compiler and interpreter quite different. The differences can be exemplified by the assignment $u := f(x)$. In the compiled language, the principle of *strong typing* requires that the type S of x must identify a unique operation $f: S \rightarrow T$ and T must be the type of u . In the interpreter:

- Declarations are generally not needed to guide the selection of operations. For the assignment $u := f(x)$, the type of x and u , if not declared, have to be chosen with respect to a database of all possible operations f . Issue: should not declared information about u be used in mode-analyzing $f(x)$?
- Operation $f: S \rightarrow T$ may be selected even though it is not the unique operation which applies. Issue: what does *unique* mean?
- For the above assignment, a selected operation for f may require the conversion of the value of x from one datatype to another (e.g., x may be an integer whereas the argument of f is required to come from a field). Issue: what general mechanisms are required here?
- The above operation f may be selected if $\text{type}(x) \mapsto S$ and $T \mapsto \text{mode}(u)$ (\mapsto means “can be automatically converted to”). Issue: what conversions should be made automatically?
- The above operation f may be selected if $S \mapsto \text{type}(x)$ and data x can be “retracted” to type S (for example, a rational number 3 *retracts* to the integer 3). Issue: when should retraction be attempted?

The implementation of the interpreter has required a major effort over the past 3 years by several people and has had to deal with the above and many other issues. An initial mode-based top-down/bottom-up design which cached mode-analysis by compilation in **LISP** was initiated by me for our NYU debut, then taken over, modified, and substantially completed by Scott Morrison. The interpreter was then passed to Albrecht Fortenbacher, who essentially eliminated the top-down pass and substantially reorganized and extended the modemap selection process. Albrecht also completely rewrote the type resolution facility, extending it far beyond its previous capabilities. Bob Sutor wrote most of the coercion code and over the past year has made several extensions and general improvements to the interpreter.

The ultimate interpreter design generalizes the **Ada**-like bottom-up techniques to deal with modes and modemaps. Efficiency questions arise because of the quantity and complexity of modemaps (currently “*” has 9 modemaps, *factor* has 11). All modemaps associated with an operator are always considered whenever that operation appears in a user expression.

Polymorphic functions defined by packages have modemaps which identify the name and parameters of the package. As it turns out, this package must be instantiated with those parameters to create the appropriate environment to run the function.

When Bob Sutor first joined the group he set out to make **Scratchpad II** more of a programming environment than just a program. Group members were engulfed by a seemingly endless stream of information about new CMS EXECs (one group member complained that just reading Bob's mail left no time to get work done!). EXECs were engineered for on-line help, access to all system documentation, glossary and keyword search mechanisms, user queries and requests, forums for electronic interactions on design issues ("ideas are better written than said"), variant error and command levels from novice to expert, and bug reporting. He has also written numerous facilities for system management, automatic testing, and updating. Bob also has become our spokesman, presenting and organizing talks and demonstrations at many conferences.

In summer 1982, Richard Anderson, a graduate student from Stanford, tried the experiment of having each domain constructor describe how elements of that domain would display themselves. This seemed natural because of our modular design. The approach was abandoned however when we found that the length of code required for display always exceeded the rest of the code in the definition! In Fall, 1983, we decided to provide all domains with an operation *coerce* which converted a member of the domain to a tree structure representation for output. Our 2-dimensional output code was converted from that in old **Scratchpad**, which was in turn based on **CHARYBDIS** (originally due to Jonathan Millen of Mitre). Later improvements and modifications were made by Martin Brock.

One addition to our user interface was the on-line database allowing the interactive user to query the system, e.g., to list domain constructors, to describe their operations or attributes, and to display available operations. Mike Lucks endowed the database with integrity data, maintaining complete dependency and consistency information on disk to ease interactive debugging and minimize forced recompilations. A history file facility was originally implemented by Chris Sundaresan and later modified by Albrecht Fortenbacher. This provided a general workspace "undo" facility and kept track of user input lines and output expressions.

Shmuel Winograd, our department director, wanted to make the system available to excellent people in universities and research centers with hard prob-

lems. His approach toward users was novel for IBM Research: obtain a stand-alone computer (an IBM 4381), hook it up to a network (CSNET), and make it accessible to users (non-IBMers)! Although the computer was installed in 1984, considerable software, hardware, and configuration problems caused delays. Greg Puhak, an IBM systems programmer, has been outstanding in helping us overcome these problems and in his general support of the machine. Thanks to the work of Mohamed Mobarak, we have recently become accessible from Telenet and have full screen support for most terminal types. As of this writing, we have nearly 60 field-test agreements for use of the 4381.

One of the more interesting applications of **Scratchpad II** was the joint work in September, 1983 of Don Coppersmith and James Davenport on the breaking of an adaptation of the Diffie-Helman cryptographic scheme. To do the computations, James quickly built a sparse linear equation solver by extending existing facilities. The computation involved 2188 factorizations of polynomials over $GF(2)$ of degree approximately 40, then solving 1000 equations in 750 unknowns over the integers mod $2^{127} - 1$. The computation, which required 28 minutes of computing time on an IBM 3081, Model K, yielded a database from which codes could be broken in less than 1 minute of CPU time.

David and Gregory Chudnovsky have found interesting and novel applications for **Scratchpad II** for problems in mathematics that can be attacked constructively. Their elliptic curve calculations reported in Vol. 1, No. 1, of this newsletter have resulted in one of the best integer factorization algorithms in existence. More recently, they have been studying algorithms for power series expansions of algebraic functions (see page 1) and for complex root finding (to be reported in a future issue of this newsletter).

1985: The Universal Outlook

Most of the emphasis to date has been on the design and implementation of a computer algebra system. Today, nearly all the 40 pages of basic algebra code in the system has been derived from David Barton's pseudo-**NEWSPAD** code written in 1980. This includes: the category definitions, polynomial ring and univariate polynomial code (including subresultant gcds and resultants), and linear algebra code (including row echelon, determinant by minors and by gaussian elimination).

Whereas David Barton laid the groundwork, Barry Trager has organized and guided virtually all further

developments of algebra code which has now grown to nearly 300 pages [2]. Barry extended the compiler and run-time support facilities (with some help from James Davenport) to handle increasingly general and sophisticated algebraic structures and implementations. He enhanced the compiler, for example, to implement multivariate polynomials as nested towers of univariate polynomials, our first example of a recursively defined datatype. Barry is also responsible for the implementation of packages.

Patrizia Gianni wrote a complete implementation of multivariate factorization, including finite field factorization, univariate factorization, and Hensel lifting, also the multivariate square-free decomposition and multivariate gcds. This package was recently enhanced by Michael Lucks who implemented some heuristics similar to those used in *muMATH*. Andrew Neff, a summer student in 1983, wrote a real-root finding package and an initial Gröbner Basis package. The Gröbner facility has been recently enhanced by Rüdiger Gebauer and Michael Möller (see "Algebra Snapshot" in the next issue of this newsletter). Barry Trager has implemented integration of rational functions and added several domain constructors such as algebraic numbers, general algebraic extensions. His implementation of elementary functions provide for the manipulation of exponential, log, and trigonometric functions while leaving argument expressions in canonical form.

In May, 1985, we gave our first course at IBM Research in *Scratchpad II* to 30 invited people from inside and outside IBM. To prepare for this event, everyone in the group took part in getting the system ready for its first exposure to users. In particular, Stephen Watt took a break from his work on the new *Scratchpad II* compiler in *BOOT* to write some solve packages in *SPAD* to be used at the course. After the course, Stephen went back to work on the compiler, but this time with the decision to work only in *SPAD*. This shift in direction has turned out to be historic, for it signaled a fundamental change in the perspective of *Scratchpad II*. We now see it as a general purpose programming language as opposed to simply being a language for computer algebra.

In *Scratchpad II*, there has never been a distinction between the algebraic objects and the workhorse data structures used to represent them, e.g., the lists, vectors, unions, and record-structures which, like polynomials, have other objects as components. If you compute interactively, you will certainly want to mix these two kinds of objects, for example, to produce a list of integers.

It is therefore no fundamental change in perspective to view every other component of the *Scratchpad II* system as being organized as a collection of constructors, including the *Scratchpad II* compiler, interpreter, and user-interface. In the implementation of the compiler, for example, there is a need for domains such as symbol tables, hash-tables, and parse trees. The work has already added many useful constructors to the *Scratchpad II* library (we now have almost 250).

This universal point of view has been strongly adopted by our group as we move toward implementing *all Scratchpad II* in the *SPAD* language, that is, in its own language. This approach should lead to an implementation of the system enjoying the benefits of abstract datatype programming: modularity, reliability, and extensibility.

In a future issue of the *The Scratchpad II Newsletter*, Stephen Watt will take us from the past into the future of the *Scratchpad II* project. He will discuss the theoretical basis and design of the new *Scratchpad II* programming language.

Acknowledgments.

For the computer algebra effort at IBM, I especially thank Jim Griesmer, for providing an ideal work environment during his tenure as manager of the *Scratchpad* project, 1968-1976; David Yun, for his unswerving support as manager of the project from 1977-1983; and Richard Toupin, Se June Hong and Shmuel Winograd, for their support, encouragement, and enthusiasm for our project. I also thank Chris Sundaresan, Moss Sweedler and Stephen Watt for their comments on drafts of this paper.

References

- [1] Cole, Chris A.; Wolfram, S.; et al, *SMP, A Symbolic Manipulation Program*, Pasadena: California Institute of Technology, 1981.
- [2] Computer Algebra Group, *Basic Algebraic Facilities of the Scratchpad II Computer Algebra System*, Yorktown Heights, New York: IBM Corporation, March 1986.
- [3] Computer Algebra Group, *Scratchpad II Programming Language Reference*, Yorktown Heights, New York: IBM Corporation, December 1985.
- [4] Computer Algebra Group, *An Overview of the Scratchpad II Language and System*, Yorktown

Heights, New York: IBM Corporation, April 1986.

- [5] Davenport, J. H. and Jenks, R. D., "MODLISP," *Proceedings of LISP '80 Conference*, August, 1980. Also *IBM Research Report RC 8537* (Yorktown Heights, New York: October 29, 1980).
- [6] Davenport, J. H., *A New Algebra System*, Yorktown Heights, New York: IBM Corporation, 1981 (unpublished).
- [7] Demers, A., and Donahue, J., *Revised Report on Russell*, TR 79-389, Ithaca, New York: Department of Computer Science, Cornell University, September 1979.
- [8] Hearn, A. C., "A Mode Analyzing Algebraic Manipulation Program," *Proceedings of ACM 74*, San Diego, CA, 1974.
- [9] Griesmer, J. H.; Jenks, R. D.; and Yun, D. Y., *SCRATCHPAD User's Manual*, *IBM Research Report RC RA70* (Yorktown Heights, New York: June 1975).
- [10] Griesmer, J. H.; Jenks, R. D.; and Yun, D. Y., "The FORMAT statement in SCRATCHPAD," *SIGSAM Bulletin* 35, Vol. 9, No. 3, August 1975.
- [11] Griss, Martin L., "The Definition and Use of Data Structures in REDUCE," *Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation*, Yorktown Heights, NY, August 1976.
- [12] Jenks, R. D., "The SCRATCHPAD Language," *Proceedings of a SIGPLAN Symposium on Very High Level Languages*. Edited by B. Leavenworth, New York: Association for Computing Machinery, March 1974. Conference held at Santa Monica, California. Also *SIGSAM Bulletin*, Vol. 8, No. 2, May 1974.
- [13] "On the Design of a Mode-based Symbolic System," *Proceedings of the Tenth Hawaii International Conference on System Sciences*, January 1977, Honolulu, pp. 157-160.
- [14] Jenks, R. D., "Towards a Language for Implementing and Using Algebra Systems," Report 64, *Utah Computation Physics Group*, April 1978.
- [15] Jenks, R. D., "SCRATCHPAD/360: Reflections on a Language Design," *Proceedings of SEAS 1978 Anniversary Meeting*, Stressa, Italy, October, 1978. Also *IBM Research Report RC 7405* (Yorktown Heights, New York: November 11, 1978).
- [16] Jenks, R. D., "MODLISP: An Introduction," *Proceedings of EUROSAM '79*, Vol. 72: *Lecture Notes in Computer Science*. Edited by G. Goos and J. Hartmanis, New York: Springer-Verlag, June 1979. Also *IBM Research Report RC 8073* (Yorktown Heights, New York: January, 1980).
- [17] Jenks, R. D. and Trager, B. M., "A Language for Computational Algebra," *Proceedings of SYMSAC '81, 1981 Symposium on Symbolic and Algebraic Manipulation*, Snowbird, Utah, August, 1981. Also *SIGPLAN Notices*, New York: Association for Computing Machinery, November 1981, and *IBM Research Report RC 8930* (Yorktown Heights, New York).
- [18] Loos, Rüdiger G. K., "Toward a Formal Implementation of Computer Algebra," *Proceedings of EUROSAM '74*, Stockholm: Royal Institute of Technology, August, 1974. Also *SIGSAM Bulletin*, Vol. 8, No. 3, New York: Association for Computing Machinery, August, 1974.
- [19] Jenks, R. D., "A Primer: 11 Keys to New Scratchpad," *Proceedings of EUROSAM '84, 1984 International Symposium on Symbolic and Algebraic Computation*, Cambridge, England, July 1984.
- [20] Liskov, B., et al, *CLU Reference Manual*, TR-225, MIT/LCS, October, 1979.

Some Questions and Answers About Scratchpad II

The Computer Algebra Group welcomes your questions about Scratchpad II. Questions deemed to be most interesting to a wide audience will be answered in future columns, while more specific questions will be answered on an individual basis.

Q: What function should I call to find out the largest power of a prime number p dividing an integer q ?

A: Scratchpad II does not now have such a function in any of its domains or packages, but it is easy enough to write it in the interpreter. For example:

```
pow(p,q) ==
qr := q div p
if qr.remainder /= 0 then 0
else 1 + pow(p,qr.quotient)
```

The function *div* returns a record object with two fields: *quotient* and *remainder*. Dot notation (e.g., *qr.remainder*) is used to refer to the components of the record. By the way, if you only needed the quotient or remainder, the infix functions *quo* and *rem* are available.

Q: I'd like to produce a polynomial of the form

$$100x^{100} + 99x^{99} + \cdots + 2x^2 + x$$

What's the easiest way to do this in Scratchpad II?

A: The simplest way to do is by using a combination of a list former and "+" reduction over the list. Issuing

```
+/[i*x**i for i in 1..100]
```

will give you what you want. Since the list of terms is formed before the terms are added, a more space efficient solution would be the two step

```
p := 0
for i in 1..100 repeat p := p + i*x**i
```

Q: How do I tell whether two objects are equal? Every time I do something like $p = q$, I just get an equation and not *true* or *false*, as I had expected.

A: You are correct that the interpreter treats the equal sign as an operator for creating equations. To get *true* or *false*, the equation must be coerced to type **Boolean**.

```
(p = q) :: B
```

In situations where a **Boolean** object is expected (as in assignment to a declared variable or in an if statement), the coercion will be performed automatically for you.

Q: What's the easiest way to create a matrix? How do I find what operations are available?

A: Among the ways that matrices can be created are by assignment, by operations from a matrix domain (including **Matrix**, **RectangularMatrix** and **SquareMatrix**, which are abbreviated **M**, **RM** and **SM**, respectively) and by coercion from lists of lists.

For example, if m is a 2×2 square matrix over the integers, (say, by having been declared $m : SM(2,I)$), then

```
m := 1
```

creates an identity matrix. If an integer other than 1 is given, m becomes a diagonal matrix with the constant along the diagonal.

To explicitly specify the elements of m , the coercion from lists of lists may be used.

```
m := [[1, 2],[3, 4]]
```

creates the matrix

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

Once a matrix exists, you can change individual elements by referring to them using *dot* notation.

```
m.0.0 := 100
```

changes the top left corner element to 100. Matrix row and column indexing begins at 0.

The *show* command is used to display operation information. Issuing

```
)show M
)show RM
)show SM
```

will display the available operations for the 3 matrix domains mentioned above. You can fully specify the domain names if you want less general information. The command *show SM(2,I)* displays operation information about 2×2 matrices over the integers.

Q: Is there any way to save the contents of a workspace from session to session? It can be very time consuming to perform many computations just to get back to where I was last time.

A: The *history* system command has two options that can help you here. *history save monday* will save the relevant contents of the workspace to your A disk under the name MONDAY HISTORY A. The next time you enter **Scratchpad II**, issue *history restore monday* to restore the workspace to the state it was in when the original save was performed. Note that this restoration can be performed as many times as wish, as long MONDAY HISTORY remains on your A disk. Of course, you can also use filenames other than MONDAY.

Scratchpad II System Status

This section is used each issue to discuss the state of the components of the developing **Scratchpad II** system. The full names of constructors abbreviated in the following are contained in "Some Scratchpad II Constructor Name Abbreviations".

The Interpreter

As part of the preparation for rewriting the interpreter in the **Scratchpad II**, we have started using type **Void** for the return type of such statements as declaration, map definition, loop and "if-then" without the "else." Whereas these statements did not previously display a value and return type, they now show a value of "()" and a type of **Void**.

BigFloat is now the default domain of computation for floating point numbers (it was **Float**, which is standard IBM double float as implemented through **LISP/VM**). The number of digits to use is set using the function *precision*. For example,

```
precision 100
```

```
(1) 100
```

```
Type: NNI
```

causes **BF** to use 100 digits in calculation and output. By default, 28 digits are used. Note that the mantissa and exponent of a **BF** may be integers of arbitrary size.

Declared and defined user maps may now be passed to other such maps and to functions from the algebra library. Here is an example of a function that maps a given function across the exponents of a univariate polynomial. The first argument is the function to use on the exponents. We first declare and then define *exponMap*.

```
exponMap: (NNI -> NNI, P[x] I) -> P[x] I
```

```
(1) ()
```

```
Type: VOID
```

```
exponMap(exponFn, poly) ==
  newpoly := lc(poly) * x ** exponFn(degree poly)
  while (poly := red poly) <= 0 repeat
    newpoly := newpoly + lc(poly) *
      x ** exponFn(degree poly)
  newpoly
```

```
(2) ()
```

```
Type: VOID
```

The function *sq* just squares elements of **NonNegativeInteger**.

```
sq: NNI -> NNI
```

```
(3) ()
```

```
Type: VOID
```

```
sq n == n * n
```

```
(4) ()
```

```
Type: VOID
```

We'll define a polynomial and then map *sq* across its exponents using *exponMap*.

```
p := (2*x**3 - x + 6)**2
```

```
(5) 4x6 - 4x4 + 24x3 + x2 - 12x + 36
```

```
Type: P I
```

```
exponMap(sq,p)
  compiling exponMap with signature
  (NNI -> NNI,P[x]I) -> P[x]I
  compiling sq with signature NNI -> NNI
```

```
(6) 4x36 - 4x16 + 24x9 + x4 - 12x + 36
```

```
Type: P[x]I
```

Algebraic Facilities

New Polynomial Factorization and GCD Implementations

Faster routines for factoring polynomials and computing their greatest common divisors have been recently added to the system by Michael Lucks. The cost of multivariate Hensel lifting has been greatly reduced via the implementation of coefficient prediction and the dynamic correction of extraneous factors. Hensel lifting in the univariate case has also been significantly improved using a combined linear and quadratic scheme. Factorization of polynomials over finite fields is now facilitated by the new domain **SmallIntegerPolynomial**, which uses a dense representation for polynomials over the integers

mod p , $p < 2^{26}$. **SmallIntegerPolynomial** takes advantage of the LISP/VM macro facilities for small integers and supports a highly efficient version of distinct degree factorization (with probabilistic splitting) which consistently outperforms the **Scratchpad II** implementation of the Berlekamp algorithm, particularly for polynomials of high degree. GCD computations and related problems (e.g. as rational function operations and content factorization) have been made far more efficient via the new **PolynomialGcdPackage**, which uses a hybrid Hensel-PRS-HeuristicGCD implementation.

New Gröbner Basis Packages

New packages added to the **Scratchpad II** library by Rüdiger Gebauer and Michael Möller. allow the construction of Gröbner bases for multivariate polynomial ideals over fields and euclidean domains. Extended versions of these packages exist for solving systems of algebraic equations yielding all real solutions with a given accuracy, and for deciding whether a polynomial or an appropriate multiple belongs to a given ideal. Examples and theoretical background will be given in the next issue of *The Scratchpad II Newsletter* (Vol. 1, Num. 4) in the "Algebra Snapshot."

PartialFraction

The constructor **PartialFraction** has a single domain parameter **R** which must be a **EuclideanDomain**. A domain created with this constructor is a **Field** and **Algebra** over **R**.

Partial fractions are created by field operations on other partial fractions and two functions: **partialFraction**, which take a numerator from **R** and a denominator from **FactoredRing R**, and **coerce**, which takes an element of **QF FR R** to a element of **PFR R**.

As an example, we look at the decomposition for

$$\frac{1}{\prod_{i=1}^4 (x+i)^i}$$

We define the denominator as an element of **FactoredRing UnivariatePoly(x,RationalNumber)** (which is a **EuclideanDomain**), asserting that each term is a prime (by using **prfac**):

```
u : FR P[x] RN := */[prfac(x+i,i) for i in 1..4]
```

```
(1) (x + 1)(x + 2)^2(x + 3)^3(x + 4)^4
```

```
Type: FR P[x]RN
```

Note that one can also call **factor** on univariate polynomials over the rational number to get an element of **FR P[x] RN**. The partial fraction decomposition of u is

```
partialFraction(1,u)
```

$$(2) \quad \frac{1}{648(x+1)} + \frac{(-\frac{1}{4})x + \frac{7}{16}}{(x+2)^2} + \frac{(-\frac{17}{8})x^2 - 12x - \frac{139}{8}}{(x+3)^3} + \frac{(\frac{607}{324})x^3 + (\frac{10115}{432})x^2 + (\frac{391}{4})x + \frac{44179}{324}}{(x+4)^4}$$

```
Type: PFR P[x]RN
```

One can get an extended representation by padically expanding each numerator around the prime in the denominator.

```
padicFraction %
```

$$(3) \quad \frac{1}{648(x+1)} + \frac{1}{4(x+2)} + \frac{-1}{16(x+2)^2} + \frac{-17}{8(x+3)} + \frac{3}{4(x+3)^2} + \frac{-1}{2(x+3)^3} + \frac{607}{324(x+4)} + \frac{403}{432(x+4)^2} + \frac{13}{36(x+4)^3} + \frac{1}{12(x+4)^4}$$

```
Type: PFR P[x]RN
```

The two representations are algebraically equal. The compact form is used internally for faster calculations.

GaloisFieldQ

The domain **GaloisFieldQ** implements finite field extensions of **GaloisField**. It is implemented as a specialization of **SimpleAlgebraicExtension** where the irreducible modulus polynomial is computed automatically. The polynomial $x^2 + 1$ is irreducible over **GF 7**, but factors (as it must), in **GFQ(7,2)**.


```
u : P[x] GF 7 := x**2 + 1
```

$$(1) \quad x^2 + 1$$

```
Type: P[x]GF 7
```

```
factor u
```

$$(2) \quad x^2 + 1$$

```
Type: FR P[x]GF 7
```

```
u2 : P[x] GFQ(7,2) := u
```

$$(3) \quad x^2 + 1$$

```
Type: P[x]GFQ(7,2)
```

```
factor u2
```

$$(4) \quad (x + \beta)(x - \beta)$$

```
Type: FR P[x]GFQ(7,2)
```

The function *modulus* returns the irreducible polynomial used in the representation of GFQ over the appropriate GF.

```
modulus()$GFQ(7,2)
```

$$(5) \quad \beta^2 + 1$$

```
Type: SUP GF 7
```

The modulus polynomial is canonically chosen: it will be the same every time the particular finite field is used. For efficiency of computation, we intend to provide a library containing many such polynomials so that they need not be recomputed at every use.

One can get a random element of GFQ(7,2) as follows:

```
e := random()$GFQ(7,2)
```

$$(6) \quad \beta + 1$$

```
Type: GFQ(7,2)
```

The standard field and field extension operators are available.

```
-- absolute norm
norm e
```

$$(7) \quad 2$$

```
Type: GF 7
```

```
-- absolute trace
trace e
```

$$(8) \quad 2$$

```
Type: GF 7
```

The function *index* maps the positive integers between 1 and the size of the finite field into the field itself. The following code generates a list of all elements of GFQ(7,2).

```
[index(i)$GFQ(7,2) for i in 1..49]
```

$$(9) \quad [1, 2, 3, 4, 5, 6, \beta, \beta + 1, \beta + 2, \beta + 3, \beta + 4, \beta + 5, \beta + 6, 2\beta, 2\beta + 1, 2\beta + 2, 2\beta + 3, 2\beta + 4, 2\beta + 5, 2\beta + 6, 3\beta, 3\beta + 1, 3\beta + 2, 3\beta + 3, 3\beta + 4, 3\beta + 5, 3\beta + 6, 4\beta, 4\beta + 1, 4\beta + 2, 4\beta + 3, 4\beta + 4, 4\beta + 5, 4\beta + 6, 5\beta, 5\beta + 1, 5\beta + 2, 5\beta + 3, 5\beta + 4, 5\beta + 5, 5\beta + 6, -\beta, -\beta + 1, -\beta + 2, -\beta + 3, -\beta + 4, -\beta + 5, -\beta + 6, 0]$$

```
Type: L GFQ(7,2)
```

Number Theory and Combinatorics Package

A new package has been provided by Martin Brock that supplies some useful functions for number theory and combinatorics. Some of the functions provided are

<i>mu</i>	the Möbius μ function
<i>phi</i>	the Euler ϕ function or totient
<i>tau</i>	the number of divisors function
<i>sigma</i>	the sum of divisors function
<i>Legendre</i>	the Legendre symbol
<i>Jacobi</i>	the Jacobi symbol
<i>binomial</i>	the binomial coefficients
<i>multinomial</i>	the multinomial coefficients
<i>partition</i>	the partition function

In the following examples, recall (*The Scratchpad II Newsletter*, Vol. 1, Num. 2, page 11) that the *!* operator is used to distribute a function call across a list (e.g., $f! [a,b,c]$ is the same as $[f(a), f(b), f(c)]$).

The tau function on the first 20 integers is:

```
tau![1..20]
```

$$(1) \quad [1, 2, 2, 3, 2, 4, 2, 4, 3, 4, 2, 6, 2, 4, 4, 5, 2, 6, 2, 6]$$

```
Type: L I
```

The sigma function on the first 20 integers is:

```
sigma![1..20]
```

$$(2) \quad [1, 3, 4, 7, 6, 12, 8, 15, 13, 18, 12, 28, 14, 24, 24, 31, 18, 39, 20, 42]$$

```
Type: L RN
```

The *partition* function counts the number of ways a positive integer can be expressed as a sum of positive integers, without regard to order.

```
partition![1..10]
```

$$(3) \quad [1, 2, 3, 5, 7, 11, 15, 22, 30, 42]$$

```
Type: L I
```

The following matrix shows some values for the *Jacobi* function. Row i of the matrix represents the modulus i . Its entries are the values of the Jacobi

symbols as we go through the residue classes starting with 1. More formally, the (i,j) -entry of the matrix below is the value of the Jacobi symbol $(\frac{j}{i})$.

The coercion, ":: M I," is used to convert the initial expression from a list of lists of integers to a matrix of integers.

```
[[Jacobi(A,P) for A in [1..10]] _
  for P in [1..10]] :: M I
```

```
(4)
[ 1  1  1  1  1  1  1  1  1  1 ]
[ 1  0  1  0  1  0  1  0  1  0 ]
[ 1 -1  0  1 -1  0  1 -1  0  1 ]
[ 1  0  1  0  1  0  1  0  1  0 ]
[ 1 -1 -1  1  0  1 -1 -1  1  0 ]
[ 1  0  0  0 -1  0  1  0  0  0 ]
[ 1  1 -1  1 -1 -1  0  1  1 -1 ]
[ 1  0  1  0  1  0  1  0  1  0 ]
[ 1  1  0  1  1  0  1  1  0  1 ]
[ 1  0 -1  0  0  0 -1  0  1  0 ]
```

Type: M I

Other New Facilities

The domains **Binary** and **Hexadecimal** have been added. These will eventually be subsumed into the arbitrary radix facilities of **Scratchpad II**.

The ALGEBRA 4381

Until recently, remote users of **Scratchpad II** who have accessed the system via TELNET have been faced with a less than friendly line-at-a-time interface. This environment, while adequate for some applications, has proven unsuitable for efficient **Scratchpad II** use which may depend heavily on the convenience of the editing facilities. In addition, any calls to full-screen applications such as XEDIT, as are performed by the **Scratchpad II** help facility, have resulted in termination of the connection. Fortunately there appear to be solutions to these problems for a large segment of the current and potential **Scratchpad II** user community.

As of this writing, users of **Scratchpad II** based on UNIX 4.2 compatible systems will be able to emulate an IBM 327X display terminal over a TELNET connection to the ALGEBRA machine in Hawthorne. The package which performs the emulation, tn3270, supports a large number of the ter-

minals listed in the TERMCAP database and permits users to create definitions for currently unsupported terminals. tn3270 is to be a standard part of BSD UNIX release 4.3 and will therefore be supported in subsequent releases of the operating system.

In addition to providing full-screen support for those based at UNIX nodes, other **Scratchpad II** users will soon have the option of initiating full-screen **Scratchpad II** sessions using nothing more than any one of the many popular ASCII terminals capable of simple cursor movement, and a 300 or 1200 bps modem. It is expected that by late May the ALGEBRA machine will be a node on the GTE TELENET network. It will then be possible to reach ALGEBRA inexpensively by dialing a local TELENET access number issuing a simple connect request to the **Scratchpad II** 4381. Once connected to ALGEBRA a user may invoke SIM3278, a program which emulates an IBM 3278 display terminal for a wide variety of ASCII terminals. All emulation is performed at the **Scratchpad II** 4381 and does not require that any special software be installed by the remote user.

Further enhancements to the remote **Scratchpad II** user's environment include a front end for both TELNET and tn3270 which permits the substitution of locally developed input files for user input to a **Scratchpad II** session. The program is also capable of capturing **Scratchpad II** output to a local file. Also under development is a graphics display package which supports windowing and may incorporate a context sensitive help facility. Although the package is being developed on a PC/AT, it should be fairly easy to port to other systems as it is written in C and will communicate with output devices via the Virtual Device Interface. Those joint study participants interested in acquiring any of the above mentioned user support software should direct inquiries to the Computer Algebra Group at the address listed on the inside front cover of this newsletter.

Installations of Scratchpad II

As part of the joint study program, **Scratchpad II** has been installed in Pisa, Italy, and Grenoble, France. Installations in June are planned for IBM Research in Zurich, IBM Science Centers in Rome and Heidelberg, and universities in Linz, Austria, and Liege, Belgium.

Some Scratchpad II Constructor Name Abbreviations

The following are some of the abbreviations used for category, domain and package constructors in this newsletter.

Abbreviation	Constructor Name
BF	BigFloat
FR	FactoredRing
G	Gaussian
GF	GaloisField
GFQ	GaloisFieldQ
I	Integer
L	List
M	Matrix
NNI	NonNegativeInteger
P	Polynomial
PFR	PartialFraction
QF	QuotientField
RF	RationalFunction
RM	RectangularMatrix
RN	RationalNumber
S	String
SAE	SimpleAlgebraicExtension
SM	SquareMatrix
UP	UnivariatePoly

Algebra Snapshot: Partially Derivative Work

Say $\omega = \sum_{i=1}^n f_i(x_1, \dots, x_n) dx_i$ where each $f_i(x_1, \dots, x_n)$ is a polynomial in n variables. ω is called *exact* if there is a polynomial $g(x_1, \dots, x_n)$ with $\nabla g = \omega$, where

$$\nabla g = \sum_{i=1}^n \frac{\partial}{\partial x_i} g dx_i$$

ω is called *closed* if for each i and j :

$$\frac{\partial}{\partial x_i} f_j = \frac{\partial}{\partial x_j} f_i$$

A standard result we learn in calculus is that ω is exact if and only if ω is closed. This is true if we are working in a polynomial ring over any field of characteristic zero. It is false in positive characteristic, even in one variable. In a polynomial ring of one

variable over a field of characteristic p , let $\gamma = x^{p-1} dx$. This γ is closed (because $n = 1$) but is not exact. Assuming we are working with polynomials over a field of positive characteristic p , ω will be called *p-closed* if it is closed and for each i , f_i satisfies:

$$\left(\frac{\partial}{\partial x_i}\right)^{p-1} f_i = 0$$

Mitsuhiro Takeuchi and I have recently proven that in positive characteristic, ω is exact if and only if ω is *p-closed*. Previously we had developed machinery which proved that ω is exact if and only if ω satisfies a *more complicated condition* than the *p-closed* condition. Working by hand we found that the more complicated condition is equivalent to the *p-closed* condition in characteristics 2 and 3. Using **Scratchpad II** we looked at examples up to characteristic 17. In all the examples, *p-closed* was equivalent to the more complicated condition. Moreover, the way in which *p-closed* turned out to be equivalent to the more complicated condition, suggested one of the steps in our eventual proof of: **exact if and only if p-closed** for all positive characteristics.

Here are programs to generate three families of examples. In the all three families we are dealing with a ring R which has a derivative, $D: R \rightarrow R$. (Derivative meaning that $D(r+s) = D(r) + D(s)$ and $D(rs) = rD(s) + D(r)s$ for all r and s in R .) In the first two families R is commutative. The third family does not directly arise from the **exact if and only if p-closed** problem. In the third family R is not commutative. After looking at the first two families of examples and trying to figure out closed form formulas for the coefficients which appear, I became interested in looking at some non-commutative examples. The coefficients which appear in the third family are products of binomial coefficients.

First Family. Here we have a commutative ring R with elements f_0, f_1, f_2, \dots and x_1, x_2, x_3, \dots . A derivation $D: R \rightarrow R$ where $D(x_i) = x_{i+1}$ and $D(f_i) = f_{i+1} x_1$. Think of x_1 as dx and x_i as $d^i x$, where $D(d^i x) = d^{i+1} x$. The first few derivatives of f_0 are:

$$D(f_0) = f_1 x_1$$

$$D^2(f_0) = f_1 x_2 + f_2 x_1^2$$

$$D^3(f_0) = f_1 x_3 + 3 f_2 x_1 x_2 + f_3 x_1^3$$

$$D^4(f_0) = f_1 x_4 + 4 f_2 x_1 x_3 + 3 f_2 x_2^2 + 6 f_3 x_1^2 x_2 + f_4 x_1^4$$

The program to generate successive derivations of f_0 consists of three parts. First, the limiting value — lim — is selected. Next, the derivation is defined, as far as it will be needed. Finally, the derivation is applied repeatedly to f_0 . Here's the program:

```
lim := 17

d(z) ==
  ans : P I := 0
  for i in 0..lim-1 repeat
    ans := ans + f[i+1]*x[i]*pderiv(z,f[i])
  for i in 0..lim-1 repeat
    ans := ans + x[i+1]*pderiv(z,x[i])
  ans

value := 1*f[0]

for i in 1..lim repeat
  output "=====
  output i
  output "=====
  output (value := d value)
```

Second Family. We have a commutative ring R with elements $f_{i,j}$ for $i, j = 0, x_1, x_2, x_3, \dots, y_1, y_2, y_3, \dots$. A derivation $D: R \rightarrow R$ where $D(x_i) = x_{i+1}$, $D(y_i) = y_{i+1}$ and $D(f_{i,j}) = f_{i+1,j} x_1 + f_{i,j+1} y_1$. As before think of x_i as $d^i x$ (and y_j as $d^j y$). $f = f_{0,0}$ should be thought as a function of x and y . $f_{i,j}$ represents the i^{th} partial derivative with respect to x , j^{th} partial derivative with respect to y of f . The first two derivatives of $f_{0,0}$ are:

$$D(f_{0,0}) = f_{1,0} x_1 + f_{0,1} y_1$$

$$D^2(f_{0,0}) = f_{1,0} x_2 + f_{2,0} x_1^2 + 2f_{1,1} x_1 y_1 + f_{0,2} y_1^2 f_{0,1} y_2$$

The number of terms grows quickly. The program to generate successive derivations of $f_{0,0}$ is similar to the previous program to generate successive derivations of f_0 . It breaks into the same parts: select the limiting value, define the derivation, apply it.

```
lim := 17

dx(z) ==
  ans : P I := 0
  for i in 0..(lim-1) repeat
    ans := ans + x[i+1]*pderiv(z,x[i])

  for j in 0..(lim-1) repeat
    for i in 0..(lim-(1+j)) repeat
      ans := ans +
        f[i+1,j]*x[i]*pderiv(z,f[i,j])

  for j in 0..(lim-1) repeat
    for i in 0..(lim-(1+j)) repeat
      ans := ans +
        g[i+1,j]*x[i]*pderiv(z,g[i,j])

  ans
```

```
dy(z) ==
  ans := z - z
  for j in 0..(lim-1) repeat
    ans := ans + y[j+1]*pderiv(z,y[j])

  for i in 0..(lim-1) repeat
    for j in 0..(lim-(1+i)) repeat
      ans := ans +
        f[i,j+1]*y[j]*pderiv(z,f[i,j])

  for i in 0..(lim-1) repeat
    for j in 0..(lim-(1+i)) repeat
      ans := ans +
        g[i,j+1]*y[j]*pderiv(z,g[i,j])

  ans

d(z) == dx(z) + dy(z)

value := 1*f[0,0]

for i in 1..lim repeat
  output "=====
  output i
  output "=====
  output (value := d(value))
```

Third Family. Take the description of the first family but leave off the "commutative." Again we have a ring R with elements f_0, f_1, f_2, \dots and x_1, x_2, x_3, \dots . A derivation $D: R \rightarrow R$ where $D(x_i) = x_{i+1}$ and $D(f_i) = f_{i+1} x_1$. The first three derivatives of f_0 are:

$$D(f_0) = f_1 x_1$$

$$D^2(f_0) = f_1 x_2 + f_2 x_1^2$$

$$D^3(f_0) = f_1 x_3 + 2 \times f_2 x_1 x_2 + f_2 x_2 x_1 + f_3 x_1^3$$

Notice that $D^3(f_0)$ differs from $D^3(f_0)$ of the first family because of the non-commutativity of R here. This non-commutativity also seems to prevent us from setting up the program in a polynomial ring as we did in the previous two programs. Since Scratchpad II doesn't at the moment have non-commutative polynomial rings, how do we set up this program? Fake it. Use symbols g_{e_0, e_1, \dots, e_n} where such a symbol plays the role of $f_{e_0} x_{e_1} x_{e_2} \dots x_{e_n}$. With this correspondence:

$$D(g_{e_0, e_1, \dots, e_n}) = g_{e_0+1, 1, e_1, \dots, e_n} + g_{e_0, e_1+1, \dots, e_n} + g_{e_0, e_1, e_2+1, \dots, e_n} + \dots + g_{e_0, e_1, \dots, e_n+1}$$

The program cannot be set up as simply as the previous two because as we take further derivatives, the n of g_{e_0, e_1, \dots, e_n} grows. In other words the limiting value — lim — also determines the length of g_{e_0, e_1, \dots, e_n} , which must be considered in defining the derivation. A little thought about which terms are reachable as successive derivations of f_0 ($= g_0$), greatly reduces the number of iterations in the deri-

$$c(e_1, e_2, \dots, e_n) = \binom{\binom{n}{1} e_1 - 1}{e_n - 1} \binom{\binom{n-1}{1} e_1 - 1}{e_{n-1} - 1} \binom{\binom{n-2}{1} e_1 - 1}{e_{n-2} - 1} \dots \binom{e_1 - 1}{e_1 - 1}$$

Figure 3.

vation. (It also explains the confusing limits in the "repeats" below.) Thinking about which terms are reachable as successive derivations of f_0 leads to the consideration of which terms of the i^{th} derivative of f_0 contribute to which terms of the $(i+1)^{\text{st}}$ derivative of f_0 . This is the key to obtaining the closed form of the formula which gives the i^{th} derivative of f_0 . Here is the program:

```

lim := 6

d(z) ==
ans : P I := 0

ans := ans + g[1,1]*pderiv(z,g[0])

for i in 1..(lim-1) repeat
  ans := ans + (g[2,1,i] + g[1,i+1])
    * pderiv(z,g[1,i])

for i in 1..(lim-1) repeat
  for j in 1..(lim-(1+i)) repeat
    ans := ans + (g[3,1,i,j] + g[2,i+1,j]
      + g[2,i,j+1]) * pderiv(z,g[2,i,j])

for i in 1..(lim-2) repeat
  for j in 1..(lim-(1+i)) repeat
    for k in 1..(lim-(1+i+j)) repeat
      ans := ans
        + (g[4,1,i,j,k] + g[3,i+1,j,k]
          + g[3,i,j+1,k] + g[3,i,j,k+1])
          * pderiv(z,g[3,i,j,k])

for i in 1..(lim-3) repeat
  for j in 1..(lim-(2+i)) repeat
    for k in 1..(lim-(1+i+j)) repeat
      for m in 1..(lim-(1+i+j+k)) repeat
        ans := ans
          + (g[5,1,i,j,k,m]
            + g[4,i+1,j,k,m]
            + g[4,i,j+1,k,m]
            + g[4,i,j,k+1,m]
            + g[4,i,j,k,m+1])
            * pderiv(z,g[4,i,j,k,m])

for i in 1..(lim-4) repeat
  for j in 1..(lim-(3+i)) repeat
    for k in 1..(lim-(2+i+j)) repeat
      for m in 1..(lim-(1+i+j+k)) repeat
        for n in 1..(lim-(1+i+j+k+m)) repeat
          ans := ans
            + (g[6,1,i,j,k,m,n]
              + g[5,i+1,j,k,m,n]
              + g[5,i,j+1,k,m,n]
              + g[5,i,j,k+1,m,n]
              + g[5,i,j,k,m+1,n]
              + g[5,i,j,k,m,n+1])
              * pderiv(z,g[5,i,j,k,m,n])

ans

value := 1*g[0]

for i in 1..lim repeat
  output "====="
  output i
  output "====="
  output (value := d value)

```

As mentioned at the start, looking at the examples produced by **Scratchpad II** lead to a closed form for the iterated derivations of g_0 . Here it is for $N > 0$:

$$D^N g_0 = \sum_{n=1}^N \sum_{\substack{e_1, \dots, e_n \\ 1 \leq e_i \\ \sum e_i = N}} c(e_1, e_2, \dots, e_n) g_{n, e_1, e_2, \dots, e_n}$$

The coefficients $c(e_1, e_2, \dots, e_n)$ are products of binomial coefficients, as shown in Figure 3.

Moss E. Sweedler

Current and Recent Visitors to the Scratchpad II Group

Martin Brock

A recent M.S. graduate in Electrical Engineering from M.I.T.. System: 5/85-6/86.

Rüdiger Gebauer

Wiss. Ang., Inst. f. Angewandte Mathematik, Universität Heidelberg, Im Neuenheimer Feld 294, D 6900 Heidelberg. Interface: 12/85-11/86.

Professor Patrizia Gianni

Dipartimento di Matematica, Università Di Pisa, Pisa, Italy. Algebra: 1/86.

Vladimir A. Grinberg

Graduate Student, Columbia University, New York, New York. Interpreter: 12/85-7/86.

Bernhard Kutzler

Johannes Kepler Universität, Institut für Mathematik, A-4040 Linz, Austria. Algebra: 2/86.

Michael Lucks

An M.S. graduate in Computer Science from the University of Hawaii. Interpreter: 2/85-9/86.

Mohamed Mobarak

Undergraduate student, Department of Computer Science, Cornell University, Ithaca, New York. System: 1/86-8/86.

Dr. Michael Möller

Fernuniversität Hagen, Institut für Mathematik, 5800 Hagen, West Germany. Algebra: 3/86.

Professor Moss E. Sweedler

Department of Mathematics, Cornell University, Ithaca, New York. Interface: 9/85-8/86.

Conference Announcements

The 1986 ACM-SIGSAM Symposium on Symbolic and Algebraic Computation

July 21-23, 1986
University of Waterloo
Ontario, CANADA

This conference is the major forum for the presentation of research ideas, algorithms, systems, and applications of computers in Symbolic and Algebraic Manipulation. It is sponsored by the Association for Computing Machinery (ACM) - Special Interest Group on Symbolic and Algebraic Manipulation (SIGSAM).

Papers, Tutorials, System Demonstrations, and Proposals for Panel Discussions were invited on the following topics:

Algebraic and Analytic Algorithms

Simplification and Canonical Forms

- Polynomial and rational function manipulation
- Complexity of sequential and parallel algorithms
- Differential and other functional equations
- Integration, Summation, Series, Sequences

Problem Representation and Solution

- Rule systems; Knowledge-based systems
- Logic programming
- Special architectures for symbolic processing

Languages and Systems

- Symbolic processing languages and systems
- Workstations and user interfaces
- Symbolic/Numeric problem solving

Applications

- Mathematics: group theory, number theory, etc.
- Robotics, Geometry
- Theorem Proving; Correctness of programs; generation of programs
- Applications in Education, Science, and Engineering

Conference Information

An advance program brochure containing registration and housing information will be mailed to recipients of this announcement in May.

General Chairman:	Richard Fateman
Treasurer:	Guy Cherry
Local arrangements:	Keith Geddes
Program Committee	
Chairman:	David Yun
Proceedings Editor:	Bruce Char

Program Committee:

Dennis Arnon	Bruno Buchberger
Bruce Char	Gene Cooperman
James Davenport	Gaston Gonnet
Erich Kaltofen	Jed Marti
Tateaki Sasaki	David Stoutemyer
Charles Sims	Paul Wang
David Yun	

Program Committee Assistants:

Larry Leff	Franz Lichtenberger
------------	---------------------

for General Conference Information:

Prof. Richard Fateman
EECS Dept: 573 Evans Hall
University of California
Berkeley, CA 94720
(415) 642-1879
arpanet: fateman@berkeley

COMPUTERS & MATHEMATICS

July 30 - August 1, 1986
 Stanford University
 Stanford, California

This conference is devoted to the examination of the present and future relationship of computers with mathematics. The conference will also serve as a major forum for the interaction between developers of software tools and methods for research in mathematics and related areas, and those interested in their use.

The conference will consist of invited addresses, tutorials and hands-on demonstrations of mathematical software systems, and contributed poster sessions. Six invited speaker sessions are as follows:

Computers & Mathematics Research

B. Birch (Oxford)
 C. Sims (Rutgers)
 G. Andrews (Penn State)
 R. Askey (Wisconsin)

Computers & Symbolic Mathematics

D. Stoutemyer (Hawaii)
 R. Loos (Karlsruhe)
 B. Trager (IBM Research)

Computers & New Directions in Mathematics

W. Haken (Illinois)
 D.&G. Chudnovsky (Columbia)
 W. Gosper (Symbolics)

Computers & Physics

M. Feigenbaum (Cornell)
 K. Wilson (Cornell)
 T. Regge (Torino)

Computers & Number Theory

H. Lenstra Jr. (Amsterdam)
 R. Graham (AT&T Bell Labs)
 P. Erdos (Hungarian Academy)
 D. H. Lehmer (Berkeley)

Mathematics & Computer Science

R. Karp (Berkeley)
 W. Bledsoe (Texas)
 P. Henrici (E.T.H., Zurich)

System tutorials/hands-on demonstrations will be given for:

- ACRITH (IBM)
- CAYLEY (Univ. of Sydney)
- ELLPACK (Purdue)
- MACAULAY (Columbia)
- MACSYMA (Symbolics, Inc.)
- MAPLE (Univ. of Waterloo)
- MATLAB (The MathWorks, Inc.)
- muMATH (Soft WareHouse)
- REDUCE (RAND Corp.)
- SCRATCHPAD II (IBM Research)
- SMP (Inference Corporation)
- VIEWS (Tektronix)

Historical introductions will be given by J. Rice (Purdue) and J. Sammet (IBM).

Two short courses will be presented:

- Computational Algebraic Geometry
 Chm: D. Mumford (Harvard), Tuesday, July 29
- Symbolic and Algebraic Computation
 Chm: D. Yun (SMU), Saturday, August 2.

The conference is sponsored by AAAI (American Association for Artificial Intelligence) and ACM SIGSAM (Special Interest Group in Symbolic and Algebraic Manipulation). Contributors include the National Science Foundation, System Development Foundation, DARPA, AAAI, and IBM Research.

The number of registrations is limited. For information, registration, and campus housing forms, please contact:

Tiyo Asai
 Conference Secretary
 Computers & Mathematics
 c/o Thomas J. Watson Research Center
 P.O. Box 218
 Yorktown Heights NY 10598

For your information ...

The **Scratchpad II** Computer Algebra System currently runs only under the IBM VM/SP operating system on mainframe computers. A three megabyte segment is shared among all users on a given computer, and each user requires at least a four megabyte virtual machine.



Scratchpad II Getting Started

THE ADVENTURE BEGINS

This document contains information for getting started with **Scratchpad II** as it exists April 21, 1986 under the IBM VM/CMS operating system. Since **Scratchpad II** is in development, later editions of this card may vary.

Please note that **Scratchpad II** is not a product and therefore receives no official marketing support from IBM. It is a research project and requests for changes and enhancements should be made directly to the implementors at the address below.

It is possible that this material may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming or services in your country.

This card was produced by Ruediger Gebauer and Moss Sweedler. Questions and comments should be sent to:

Robert S. Sutor
Computer Algebra Group
IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598

Draft: 21 April 1986

© Copyright International Business Machines Corporation 1986. All rights reserved.

INTRODUCTION

Scratchpad II is a programming language for symbolic manipulation and exact computation. Its greatest strength lies in the library of routines to handle various types of problems in engineering, mathematics and science. This card is designed to help you get started. Once you have learned a few commands and the operations relevant to the routines you wish to use, you can begin using **Scratchpad II** to solve real problems. A good way to learn is to experiment with the selection of routines and commands presented here. If you are interested in an operation which is not mentioned here, it still may be in the system. Look at more complete printed documentation or browse through the **Scratchpad II** documentation online.

Scratchpad II can be used from the keyboard or you can feed in files which contain what you would have typed. For getting started and for short experiments we recommend using **Scratchpad II** from the keyboard. For longer experiments and for developing programs it is convenient to write files which are used as input for the **Scratchpad II** interpreter.

Typing:

<i>spad2</i>	starts your Scratchpad II session.
<i>)quit</i>	ends your Scratchpad II session.

Contents:

NUMBERS	2-3
COMPARISONS and BOOLEAN VALUES	3
ELEMENTARY FUNCTIONS	3
POLYNOMIALS	4
MATRICES	5
LISTS	6
COMMANDS	7
RESERVED WORDS	7
TIPS	8
THE ADVENTURE CONTINUES	9

On later pages this hollow font is used where a value must be filled in.

NUMBERS

Integers can be arbitrarily long:

23445674653432789021736748919

Rational numbers are entered as quotients of integers:

56875498387465 / 576476287467674

Floating point numbers are entered as a mantissa with optional exponent. No spaces on either side of the E.

.012 34.56 789.

1.E2 .3E+4 5.6E-6

Floating point numbers have a default precision of 28.

precision \square

changes the precision to \square .

Operation	Use Example	Result
absolute value	<i>abs</i> (<i>a</i>) <i>abs</i> (-165.57)	165.57
add	<i>a</i> + <i>b</i> 3 + 7/5	22/5
negate	-(<i>b</i>) -(-8)	8
subtract	<i>a</i> - <i>b</i> 3/4 - 2/7	13/28
multiply	<i>a</i> * <i>b</i> 3 * 4	12
divide	<i>a</i> / <i>b</i> (3/2) / .5	3.0
exponential	<i>a</i> ** <i>b</i> -7 ** 3	-343
factor	<i>factor</i> (<i>a</i>) <i>factor</i> (165)	3*5*11
multiply out factored integer	<i>expand</i> (<i>a</i>) <i>expand</i> (3*5*11)	165
minimum	<i>min</i> (<i>a</i> , <i>b</i>) <i>min</i> (16,16.5)	16.
maximum	<i>max</i> (<i>a</i> , <i>b</i>) <i>max</i> (17,58)	58

NUMBERS continued

Operation	Use Example	Result
least common multiple	<i>lcm</i> (<i>a</i> , <i>b</i>) <i>lcm</i> (9,21)	63
greatest common divisor	<i>gcd</i> (<i>a</i> , <i>b</i>) <i>gcd</i> (9,21)	3

COMPARISONS and BOOLEAN VALUES

The Boolean values are: *true* and *false*. Comparisons return Boolean values. To negate a Boolean use \neg . On some keyboards \neg is gotten by typing \wedge .

Operation	Use	Example
less than	<	1 < 1
less than or equal	< =	2 <= 2
equal	=	3 = 4
not equal	\neg =	3 \neg = 3
greater than or equal	> =	5 >= 6
greater than	>	8 > 7

ELEMENTARY FUNCTIONS

Evaluating an elementary function at a floating point argument, produces a floating point value. When evaluating an elementary function at other arguments, the answer is left in symbolic form.

e() gives the constant e.
pi() gives the constant pi.

Operation	Use	Example
exponential base e	<i>exp</i>	<i>exp</i> (-1.2E-3)
log base e	<i>log</i>	<i>log</i> 4.
trigonometric functions	<i>sin</i> , <i>cos</i> , <i>tan</i> <i>cosec</i> , <i>sec</i> , <i>cotan</i>	<i>sin</i> (.5*pi())
hyperbolic functions	<i>sinh</i> , <i>cosh</i> , <i>tanh</i> <i>cosech</i> , <i>sech</i> , <i>cotanh</i>	<i>tanh</i> (-2.E-1)

POLYNOMIALS

How to type in polynomials:

```
x**6 + 3/4*x**4 + .9*x + 1
73*x**3 - 15*y**2*x**5 + 11
( (bob + BOB)**12 + b06 )**3
```

Variable names consist of letters and digits starting with a letter. Case makes a difference.

To	Use
add two polynomials	$e + f$
subtract two polynomials	$e - f$
multiply two polynomials	$e * f$
exponentiate, where n is an integer	$e ** n$
divide without remainder	$e \text{ quo } f$
get remainder	$e \text{ rem } f$
divide, to get a rational function	e / f
factor	$\text{factor}(e)$
multiply out factored polynomial	$\text{expand}(d * e * f \text{ etc })$
least common multiple	$\text{lcm}(e, f)$
greatest common divisor	$\text{gcd}(e, f)$
get list of variables	$\text{varlist}(f)$
get main variable	$\text{mainvar}(f)$
get leading coefficient	$\text{leadingCoef}(f)$
get trailing coefficient	$\text{trailingCoef}(f)$
derivative of f with respect to x	$\text{pderiv}(f, x)$

MATRICES

How to type in matrices:

```
[ [1,2,3], [4,5,6], [7,8,9] ] a M
```

gives the matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

```
[[2*D,J+3,-S**4],[C/5,-E,K-C]] a M
```

gives the matrix:

$$\begin{bmatrix} 2*D & J+3 & -S**4 \\ C/5 & -E & K-C \end{bmatrix}$$

Matrix positions are numbered starting at 0 rather than 1. The (0,2) entry in the previous matrix is: $-S**4$.

Operation	Use
add two matrices of the same shape	$m1 + m2$
subtract two matrices	$m1 - m2$
multiply two matrices of appropriate shape	$m1 * m2$
exponentiate, where n is a non-negative integer	$m ** n$
determinant of square matrix	$\text{determinant}(m)$
inverse of non-singular square matrix	$\text{inverse}(m)$
transpose	$\text{transpose}(m)$
number of rows	$\text{nrows}(m)$
number of columns	$\text{ncols}(m)$
element in row r , column c of matrix m . Numbering starts at 0,0.	$m(r, c)$
get row r	$\text{row}(m, r)$
get column c	$\text{column}(m, c)$
put in row echelon form	$\text{rowEchelon}(m)$
rank	$\text{rank}(m)$

LISTS

How to type in lists:

```
[1,2,3,4,5,6]
[3*x**2 + x - 7, y**3 - 4, xyz]
[[2*D,J+3,-S**4],[C/5,-E,K-C]]
```

List positions are numbered starting with 0 rather than 1. The entry in position 1 of the previous list is: [C/5,-E,K-C]. The empty list is denoted by: [].

Operation	Use Example	Result
length of list	<i>size</i> (\mathcal{U}) or <i>#</i> (\mathcal{U}) #[a,b,c,d]	4
first element	<i>first</i> (\mathcal{U}) first([a,b,c])	a
n -th element of list \mathcal{U} .	\mathcal{U} (n) [a,b,c](1)	b
Numbering starts at 0.	[a,b,c](3)	not defined
n -th element from the end	\mathcal{U} (- n) [a,b,c](-1)	c
last element	<i>last</i> (\mathcal{U}) last[a,b,c,d,e]	e
list without first element	<i>rest</i> (\mathcal{U}) rest[a,b,c]	[b,c]
drop n elements	<i>drop</i> (\mathcal{U} , n) drop([a,b,c,d],1) drop([a,b,c,d],-2)	[b,c,d] [a,b]
delete all occurrences of \mathcal{E}	<i>delete</i> (\mathcal{E} , \mathcal{U}) delete(x,[x,y,x])	[y]
\mathcal{E} as new first element	<i>cons</i> (\mathcal{E} , \mathcal{U}) or [\mathcal{E} , : \mathcal{U}] cons(0,[1,2]) [0,:[1,2]]	[0,1,2] [0,1,2]
concatenate two lists	<i>append</i> ($\mathcal{U}1$, $\mathcal{U}2$) or [: $\mathcal{U}1$,: $\mathcal{U}2$] append([a,b],[c,d]) [:[a,b],:[c,d]]	[a,b,c,d] [a,b,c,d]
reverse list	<i>reverse</i> (\mathcal{U}) reverse [a,b,c]	[c,b,a]

COMMANDS

To	Use
end Scratchpad II session)quit
get help)help
undo last step)undo
undo all steps)clear all
clear $x1, x2, \dots, xn$ (The "p" abbreviates "properties".))clear p $x1\ x2 \dots xn$
restore workspace of previous session)history)restore
read from file rather than keyboard. File-type should be: INPUT.)read FileName
turn off time message)time off
turn on time message)time on
turn off type message)type off
turn on type message)type on
turn off prompt)prompt none
turn on prompt)prompt plain)prompt step)prompt verbose

RESERVED WORDS

Certain Scratchpad II words are considered reserved. They may not be used as names of variables or functions. Here is a list of these words.

<i>add</i>	<i>exquo</i>	<i>leave</i>	<i>return</i>
<i>and</i>	<i>from</i>	<i>not</i>	<i>then</i>
<i>by</i>	<i>has</i>	<i>or</i>	<i>until</i>
<i>case</i>	<i>if</i>	<i>otherwise</i>	<i>when</i>
<i>constant</i>	<i>in</i>	<i>pretend</i>	<i>where</i>
<i>div</i>	<i>is</i>	<i>quo</i>	<i>while</i>
<i>else</i>	<i>isnt</i>	<i>rem</i>	<i>with</i>
<i>exit</i>	<i>iterate</i>	<i>repeat</i>	<i>yield</i>

TIPS

To	Use	Example
assign a value to a variable	<code>:=</code>	<code>vA3r := [1,2]</code>
create a function	<code>==</code>	<code>f(x,y) == x*y+3</code>
use last output	<code>%</code>	<code>vV := 3**% + 2</code>
use output at step \bar{n}	<code>%%(\bar{n})</code>	<code>x1 := %(1)</code>
use output \bar{n} steps back	<code>%%(-\bar{n})</code>	<code>F := % + %(-2)</code>
use consecutive integers	<code>..</code>	<code>[-99..99]</code>
apply an operation to individual elements of a list	<code>!</code>	<code>min(![1..7],4)</code> <code>![1..3]+![1..7]</code>
use \mathbb{X} with subscripts	<code>\mathbb{X}[y]</code>	<code>var[iable]</code>
APL reduction	<code>/</code>	<code>+/[1,2,3]</code>
suppress output	<code>;</code>	<code>x:=[1..999];</code>
put several inputs on one line	<code>;</code>	<code>x:=3;y:=4;z:=x+y</code>
convert to a different representation	<code>::</code> or <code>@</code>	<code>.75 a RN</code> <code>3/4 :: F</code> <code>[[1,2],[3,4]]aM</code>

The RN, F and M, just above, are the Scratchpad II abbreviations for: **R**ationalNumber, **F**loat and **M**atrix. These are examples of common data types which in Scratchpad II are called *domains* (domains of computation). To see a list of domains, type:

)what domains

End a line with an underscore if you want the line below it to be considered an extension.

THE ADVENTURE CONTINUES

Try entering

```
234**567
```

Try entering

```
Q[5;1;11;7]
Q[;1;11;7]
Q[;1;;7]
```

Try entering

```
s == sin(x)
x := 3
s
x := 3.
s
x := pi()
s
precision 1000
x := pi()
s
```

Try entering

```
factorial(n | 0 < n) == */[1..n]
factorial(-1)
factorial 0
factorial 1
factorial 567
```

Try entering

```
[factor(z**4 - n**2) for n in 1..5]
```

Try entering

```
1/(sqrt(sqrt(n) + 25)) - 2
```

Try entering

```
(a,b) : FactoredRing Integer
c : FR I
a:= 10**11
b:= 111000
a*b
expand %
a := a + 3
b := 211
b := b - 1
c := 322111
d : FR I := 4332221111
```

Try entering

```
3*z**2 + 2*z + 1
% + 0.0
```