Newsletter
Volume 2, Number 1
November 1, 1987

Scratchpad H

IBM Research

In this issue ...

The Scratchpad II Newsletter

An informal publication of the Computer Algebra Group, Mathematical Sciences Department, IBM Research Division, Thomas J. Watson Research Center, Box 218, Yorktown Heights, New York 10598.

Editor: Robert S. Sutor

Volume 2, Number 1 November 1, 1987

The Scratchpad II Computer Algebra Group

Manager

Richard D. Jenks
IBM VNET: JENKS at YKTVMX
BITNET: jenks@yktvmz
914-945-1233

Algebra

Barry M. Trager
IBM VNET: BMT at YKTVMX
BITNET: bmt@yktvmz
914-945-1868

Patrizia Gianni
IBM VNET: GIANNI at YKTVMX
914-945-1868

Manuel Bronstein
IBM VNET: MANUEL at YKTVMX
BITNET: manuel@yktvmz
914-945-2065

Michael B. Monagan
IBM VNET: KIWI at YKTVMX
BITNET: kiwi@yktvmz
914-945-2065

Compiler

Stephen Watt
IBM VNET: SMWATT at YKTVMX
BITNET: smwatt@yktvmz
914-945-3405

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

Marc Gaetano
IBM VNET: MGAETAN at YKTVMX
914-945-3720

Interpreter

Robert S. Sutor
IBM VNET: SUTOR at YKTVMX
BITNET: sutor@yktvmz
914-945-2360

Consultants

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

Secretary

Sandra Wityak
IBM VNET: WITYAK at YKTVMT
914-945-1187

Scratchpad II System Availability

The IBM Research Division has made Scratchpad II available to a limited number of universities running the VM/SP operating system on IBM mainframe computers. The minimum recommended machine configuration is a 4381 (or IBM mainframe of comparable processor speed) with 16 megabytes of real memory. For best performance, Scratchpad II is usually installed with a 3 megabyte DCSS (shared segment). Each mainframe user is required to have at least an 8 megabyte virtual machine. If you or a colleague would like to avail yourself of Scratchpad II, kindly send a letter describing the nature of the use you would make of the system to:

Richard D. Jenks,
Computer Algebra Group,
IBM Research Division,
T. J. Watson Research Center,
P. O. Box 218,
Yorktown Heights, NY 10598.

In turn, the Computer Algebra Group will initiate a joint study between IBM Research Division and your department. A one-time tape fee of $1000 will be charged.

The system will also soon be available on a limited basis for beta-test on IBM RT PC workstations running the AIX operating system. For good performance we recommend that workstations have at least 8MB real memory and 140MB disk space. Beta-test sites are not limited to universities. All applications for beta-test site agreements must be directed in writing to Vicky Markstein, IBM Corporation, 11400 Burnet Road, Austin TX 78759, Dept D46, Bldg 803, phone 512/823-4072. Please forward a copy of your letter to Richard Jenks at the above address. There may be a one-time software charge (yet to be determined) for the RT PC system.

Current mainframe installations of Scratchpad II include:

Site	*Investigator*
Brown Univeristy	L. Sirovich
Carnegie Mellon University	D. Scott
Chalmers Univiversity, Sweden	P. Dahlberg
Cornell University	K. Dennis
CERN, Geneva	E. van Herwijnen
ETH, Zurich	E. Engeler
CNUSC, Montpellier, France	J. Della Dora
Princeton University	I. Fuchs
Scuola Normale, Pisa	G. Gentili
SMU	D. Y. Y. Yun
Univ. of Bath	J. Davenport
Univ. of Geneva	P. Bartholdi
Univ. of Liege	H. Caprasse
Univ. of Linz	B. Buchberger
Univ. of Liverpool	Hodgkinson
Univ. of Pisa	P. Gianni
Univ. of Rome	A. Miola
Univ. of Singapore	D. Flath
Univ. of Washington	B. Eichinger
Univ. of Waterloo	B. Char
Weizmann Institute, Jerusalem	S. Druck

IBM site installations of Scratchpad II include:

IBM Brazil, Rio De Janiero
IBM England, Hursely
IBM Germany, Boeblingen
IBM Heidelberg Scientific Center
IBM Los Angeles Scientific Center
IBM Madrid Scientific Center
IBM Norway, Bergen
IBM Research, Almaden
IBM Research, Yorktown
IBM Rome Scientific Center
IBM Burlington Laboratory

Pending joint study contracts include:

Site	*Investigator*
GMD, Bonn	F. Schwarz
IMA, Minnesota	W. Miller
Penn State	H. Knoble
Riso, Denmark	D. Koenig
Tulane University	J. Diem
Univ. of Chicago	R. Larson
Univ. of Illinois, Chicago	N. Sabelli
Univ. of Karlsruhe	T. Beth
Univ. of Missouri	M. Jacobs
Univ. of Washington	S. Hallstrom

The German "Gesellschaft für Informatik" has established a special interest group for Computer Algebra. Dr. Fritz Schwarz has been elected as its speaker. There will be a newsletter distributed about 2 to 3 times a year on an irregular basis. At present it has about 400 members. For further information please contact Dr. F. Schwarz, GMD, Institut F1, Postfach 1240, 5205 St. Augustin, West Germany, or over the EARN net GF1002 at DBNGMD21.

New Analytic Methods of Polynomial Root Finding

by
David and Gregory Chudnovsky

One often needs to determine (complex) roots of a univariate polynomial within a given (high) precision. This is necessary, for example, to determine the singularities and local multiplicities of linear differential equations. Particularly important are methods that can be vectorized or parallelized. As it turns out, it is often advantageous to compute algebraic functions instead and then evaluate them to obtain good approximations to roots of polynomials. Along these lines we have discovered a class of new algorithms that compute in parallel all complex roots of sparse polynomials. The theoretical part of our methods is based on the study of Fuchsian linear ordinary differential equations satisfied by all branches of a given algebraic functions (the so-called differential resolvents associated with algebraic equations) [1]. Classical expressions of differential resolvents are known for trinomials

$$x\,q\,y^p + y^q - 1 = 0$$

where expressions of roots $y = y(x)$ are given by hypergeometric power series expansions at $x = 0$, according to Lambert, Lagrange, Ramanujan, Mellin:

$$x = \frac{n}{q} \sum_{N=0}^{\infty} (-qx)^N \frac{\Gamma\left(\frac{1+pN}{q}\right)}{\left(N!\,\Gamma\left(\frac{1+pN}{q} - N + 1\right)\right)}$$

This series converges when

$$|x| \leq p^{\frac{-p}{q}} \times (p-q)^{\frac{p-q}{q}}$$

For a general k-nomial

$$y^n + x[1]y^m + \cdots + x[k-2]y^l - 1 = 0$$

the Barnes-type integrals and systems of Fuchsian linear ordinary differential equations in $x[1], \ldots, x[k-2]$ for y (as a function of $x[1], \ldots$) were given by Mellin [2]. Using our methods [3] of analytic continuation of solutions of Fuchsian linear ordinary differential equations, we construct all roots of an arbitrary k-nomial by analytic continuation of each branch of y separately, starting from initial conditions of y at $x[i] = 0$ as roots of unity, through the singularities of differential resolvents:.

THEOREM 1. *One can compute all n roots of a k-nomial of degree n with the precision M (of leading digits) in $O(k^2 \log^2 M)$ parallel steps on n processors. If $k << n$, then one can compute m ($\leq n$) roots of a k-nomial of degree n with the precision M (digits) in $O(k \log^2 M)$ parallel steps on m processors.*

These algorithms are well suited for practical implementation on vector machines, and were studied by us on an IBM 3090 VF (vector) and a CRAY-II.

Similar methods of analytic continuations of algebraic functions provide efficient algorithms of polynomial root finding for algebraic polynomials of large degrees (in the thousands) without any special structure. One of the problems that we considered is the analysis of the distribution of complex roots of a real polynomial whose random coefficients are independent and normally distributed (the distribution of real roots is described in [4]).

References

[1] Chudnovsky, D.V., and Chudnovsky, G.V., *IBM Research Report* RC 11365 (Yorktown Heights, New York, September 13, 1985), 1-96. Also, *J. of Complexity* v.2, No.4 (1986), v.3, (1987) 1-25.

[2] Mellin, Hj., *Ann. Acad. Sci. Fenn.*, t. 7 (1915).

[3] Chudnovsky, D.V., and Chudnovsky, G.V., *IBM Research Report* RC 12036 (Yorktown Heights, New York, July 23, 1986), 1-68. *Lecture Notes in Mathematics,* v. 1240, New Yprk: Springer-Verlag, 1987, 1-68.

[4] Kac, M., *Probability and related topics in physical sciences,* New York: Interscience, 1961, Ch. I.

Domains and Subdomains in Scratchpad II

by
Stephen M. Watt

The implementation of the Scratchpad II computer algebra system is based upon a compiled, strongly typed language. This language provides packages of polymorphic functions and parameterized, abstract datatypes with operator overloading and multiple inheritance. To express the intricate interrelationships between the datatypes necessary for the description of mathematical objects, several techniques based on the notion of *category* have been used.

Categories have been used to enforce relationships between type parameters and to provide the mechanism for multiple inheritance. They also allow the language to be statically type checked and the generation of efficient code. This article describes the role of categories in Scratchpad II. (Scratchpad II uses the word "category" consistently with the computer science terminology of algebraic specification, which is somewhat inconsistent with the mathematical terminology of category theory.) Please note that this article refers to some language features that are not yet available in publicly distributed versions of the system. The compiler supporting these deatures is scheduled for release in early 1988.

Domains

In an environment where each operation may have many meanings, it is often useful to give a "domain of computation" to specify the operations of interest. A *domain* is a Scratchpad II object which provides a set of operations and attributes. Domains are used to implement datatypes and packages.

In Scratchpad II every object belongs to a unique domain. For example, 2 belongs to the domain Integer. Domains are first class run-time values which belong to the domain Domain. Objects belonging to a domain may be manipulated by operations it exports. For example, two of the operations Integer provides are

```
"+": (Integer, Integer) -> Integer
"=": (Integer, Integer) -> Boolean
```

The operations exported by a domain need not combine only members of the domain itself. The signatures of the exported operations may involve any types whatsoever. For example, the domain RationalNumber exports the operations

```
"/": (Integer, Integer) -> RationalNumber
characteristic: () -> NonNegativeInteger
```

Note that RationalNumber does not appear in the signature for ***characteristic***. A domain which does not export any operations for creating or manipulating members is a ***package***.

Domains may be created by functions, providing parameterized types and packages of polymorphic functions. As an example of a parameterized type, when Integer is passed to the function Stack the domain which we denote Stack(Integer) is returned.

The need for polymorphic functions stems from the desire to implement a given algorithm only once, and to be able to use the program for any values for which it makes sense. For example, the Euclidean algorithm for computing greatest common divisors can be used for values belonging to any type which is a Euclidean domain. The following package takes a Euclidean domain as a type parameter and exports the operations ***gcd*** and ***lcm*** on that type.

```
GCDpackage(R: EuclideanDomain): with
        gcd: (R, R) -> R
        lcm: (R, R) -> R
    == add

        -- Euclidean algorithm
        gcd(x,y) ==
           while y ¬= 0 repeat (x,y) := (y,x rem y)
           normalize x

        lcm(x, y) ==
           (x exquo gcd(x,y))::R * y
```

The exported operations can equally well be used for many types, the integers or polynomials over GaloisField(7) being two examples. Although the same ***gcd*** program is used in both cases, the operations it uses (***rem***, ***normalize***, etc.) come from the type parameter R.

Subdomains

An object may belong to any number of subdomains. The number 2, for example, belongs to many subdomains of Integer, including PositiveInteger, EvenInteger and SmallInteger. The domain Integer belongs to many subdomains of Domain, in-

cluding Monoid, AbelianGroup, Ring and Algebra(Integer).

A ***subdomain*** consists of

- a domain
- a boolean function that characterizes which members of the domain belong to the subdomain
- additional operations defined on the subdomain.

A variable may be declared to belong to a domain or to a subdomain. When a variable is declared to belong to a subdomain, its domain is that which the subdomain restricts. That is, values lying in a subdomain also lie in the domain and may be used in all the appropriate domain operations.

The subdomain may provide operations which supplement or supersede those of the domain but which are restricted to values lying in the subdomain. This restriction is for two reasons. First, operations which are closed on the domain may not be closed on the subdomain. Second, if a value has been determined to lie in a subdomain or is the result of a closed subdomain operation, then the subdomain may provide a more efficient implementation than the domain operation.

Often the subdomain predicate can be determined at compile-time. In places where this cannot be done, a run-time check may be inserted when a value must belong to a subdomain. Certain subdomains which are known to the compiler can be more highly optimized than others.

Categories

While polymorphic packages allow the implementation of algorithms in a general way, it is necessary to ensure that these algorithms may *only* be used in meaningful contexts. It would *not* be meaningful to try to use GCDpackage above with Stack(Integer) as the parameter. To restrict the use to cases where it makes sense, Scratchpad II has the notion of categories.

A ***category*** in Scratchpad II is a restriction on the class of all domains. It specifies what operations a domain must support and certain properties the operations must satisfy. A category may be created using a category constructor such as the one below.

```
OrderedSet(): Category == Set with
   -- operations
      "<": ($,$) -> Boolean
      max: ($,$) -> $
      min: ($,$) -> $
   -- attributes
      irreflexive "<"    -- not (x < x)
      transitive  "<"    -- x < y and y < z
                         --   implies  x < z
      total "<"          -- not(x < y) and not(y < x)
                         --  implies x=y
```

OrderedSet gives a category which extends the category Set by requiring three additional operations and three properties, or *attributes*.

A declaration is necessary in order for a domain to belong to a given category: having the necessary operations and attributes does not suffice. This is because the attributes are not intended to give a complete set of axioms, but merely to make explicit certain facts that may be queried later. It is usually the case that belonging to a category implies that a domain must satisfy conditions that are not mentioned as attributes. For example, in OrderedSet there is no attribute relating ***min*** and " < ", although such relations are implied.

A type parameter to a domain or package constructor is usually required to belong to an appropriate category. For example, in the previous section the parameter R to GCDpackage was declared to belong to the category EuclideanDomain.

Separation of Contract and Implementation

The use of categories in restricting the type parameters to a domain or package constructor allows algorithms to be specified in very general contexts. For example, since all table types belong to a common category, algorithms can be written that do not need to know the actual implementation (for example, whether it is a hash table in memory or a file on disk). As an example of algebraic generality, consider the domain of linear ordinary differential operators, which is declared as follows

```
LinearOrdinaryDifferentialOperator(A,M): T == B where
    A:  DifferentialRing
    M:  Module(A) with deriv: $ -> $

    T == GeneralPolynomialWithoutCommutativity(A,
      NonNegativeInteger) with
        D:       () -> $
        ".":     ($, M) -> M
    ...
    ...
```

This domain defines a ring of differential operators which act on an A-module, where A is a differential ring. The type of the coefficients, A, is declared to belong to the category DifferentialRing and type of the operands, M, is declared to belong to the category Module(A) with a derivative operation. The constructed domain of operators is declared to belong to a category of general polynomials with coefficients A and two additional operations. The operation D creates a differential operator and "." provides the method of applying operators to elements of M.

Multiple Views and Multiple Inheritance

It is often necessary to view a given domain as belonging to different categories at different times. Sometimes we want to think of the domain Integer as a belonging to Ring, sometimes as belonging to OrderedSet, and at other times as belonging to other categories. For a domain to have multiple views, it should be declared to belong to the Join of the appropriate categories. For example, the following keyed access file datatype may be viewed either as a table or as a file:

```
KeyedAccessFile(Entry: Set): T == B where
    FileRec  ==> Record(key: String, entry: Entry)
    ErrorMsg ==> String

    T == Join(FileCategory(LibraryName, FileRec),
          TableCategory(String, Entry, ErrorMsg)) _
        with
             pack: $ -> $

    B == add ...
```

An important use of categories is to supply default implementations of operations. So long as certain primitive operations are provided by a domain, others can be implemented categorically. For example, supplying only "<" and "=" allows definitions of ">", "<=" and ">=". Thus a domain may inherit operations from a category. The use of Join provides multiple inheritance.

Compile-Time Binding

A domain object contains several vectors of function/environment pairs. When operation bindings are not known at compile-time (as for the operations exported by a type parameter), the operations are performed by calling the function in the appropriate slot of a vector. When operation bindings are known at compile time, a code fragment for the operation may be placed in line.

The scope rules in Scratchpad II and the operations applicable to domain values have been designed so that when a domain is known to belong to a particular category, it is also known exactly what operations it exports. From this, the precise location of each function is determined. Thus when a function is called using the general mechanism, a hard-coded offset is used.

Scratchpad II is implemented on top of LISP/VM. Since the exact number and type of arguments are known at compile time, much of the usual function can be omitted. This, in conjunction with compile-time knowledge of function offsets, makes function calling in Scratchpad II faster than that of the underlying Lisp system.

Construction of Algebraic Error Control Codes (ECC) on the Elliptic Riemann Surface

by
Martin Hassner
William H. Burge
Stephen M. Watt

In this paper we make use of the power series facility of Scratchpad II to construct algebraic ECC on the elliptic Riemann surface of genus one. We present the construction of a specific example that highlights the method.

Linear algebraic ECC are formally defined as ideals in a function field. A message word which consists of k symbols in a ground field is mapped by a matrix encoder over the ground field into a code word which consists of n symbols, $n > k$. The encoder matrix is the null space of a check matrix that constrains the symbols of the code word to be the residues at n distinct singular points of first order of a function whose poles are a fixed set of pointers that locate the symbols inside each code word. An additional constraint imposed on each code function is its divisibility by a fixed function or linear system of functions whose zero set is disjoint from the pole set. A code consists of all the n-vectors of residues associated with such a system of functions which form an ideal. Within this algebraic framework it is possible to classify and determine (lower) bounds on the performance, i.e, efficiency vs. correction power, of algebraic ECC.

	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8
X_0	1	0	α	1	0	β	0	α	1
X_1	0	1	0	1	α	0	β	1	α
X_2	1	1	1	0	1	1	1	0	0

Figure 1. 9 rational points

The bulk of the existing theory of algebraic ECC is centered on ideals of rational functions in the rational function field obtained as a simple transcendental extension of the ground field $GF(2^m)$. The choice of this ground field was dictated by applications of the theory to computer data. The theory was extended recently by the Russian coding theorist Goppa, who formulated the unifying algebraic framework presented above, to fields of algebraic functions in one variable obtained by algebraic extensions of the rational function field [2]. Further interest in this theoretical development was produced by a recent paper in which it was shown that codes defined on elliptic modular curves, i.e, in elliptic modular function fields, have asymptotic parameters that exceed the Gilbert bound,a lower bound on the asymptotic performance of linear codes [4]. This bound was believed to be tight until the appearance of this result.

The construction described in [2] uses a geometric model for the field of algebraic functions defined on an algebraic curve. As a specific example a code constructed on the nonsingular elliptic cubic over the ground field $GF(2^2) = \{0, 1, \alpha, \beta\}$ is presented. The equation of the nonsingular cubic over this ground field in the projective coordinates $\{X_0, X_1, X_2\}$ is

$$X_0^3 + X_1^3 + X_2^3 = 0 \qquad (1)$$

On this curve there are 9 rational points with coordinates in this ground field (see Figure 1).

These 9 points are the inflection points of the cubic, each has a triple tangent. They constitute the pole divisor $D = \{P_i\}, i = 0, \ldots, 8$, whose degree is the block length of each code word, $n = 9$. The linear system of functions chosen for the zero divisor G is $\{X_0^2, X_1^2, X_2^2, X_0X_1, X_1X_2, X_0X_2\} = \{\phi_i\}, i = 1, \ldots, 6$. This linear system generates all the conics, in particular the conic $\phi_0 = X_0X_1 + X_1X_2 + X_0X_2$ which assumes nonzero values at all the nine points $\{P_i\}, i = 0, \ldots, 8$. The check matrix is obtained as

$$\left\{ \frac{\phi_i}{\phi_0} (P_j) \right\}$$

and the generator matrix given in [2] is the following

$$\begin{pmatrix} 1 & 0 & 0 & \alpha & \alpha & \beta & 0 & \beta & 1 \\ 0 & 1 & 0 & \alpha & \beta & 0 & 1 & \beta & \alpha \\ 0 & 0 & 1 & 0 & 1 & \alpha & 1 & 1 & \alpha \end{pmatrix}$$

The number of points at which a conic intersects a cubic is 6, by Bezout's theorem, hence the degree of the zero divisor G is 6. The genus of the elliptic curve is $g = 1$. These two parameters determine the code separation d and the number of error control symbols r estimated through the Riemann-Roch theorem

$$H = \begin{pmatrix} 1 & 1 & 1 & q^{2/3}+1 & \alpha q^{2/3}+\alpha & \beta q^{2/3}+\beta & \alpha q^{2/3}+\alpha & q^{2/3}+1 & \beta q^{2/3}+\beta \\ q^{2/3}+1 & \alpha q^{2/3}+\alpha & \beta q^{2/3}+\beta & q^{2/3}+1 & \beta q^{2/3}+\beta & \alpha q^{2/3}+\alpha & 1 & 1 & 1 \\ q^{2/3}+1 & \beta q^{2/3}+\beta & \alpha q^{2/3}+\alpha & 1 & 1 & 1 & \beta q^{2/3}+\beta & q^{2/3}+1 & \alpha q^{2/3}+\alpha \\ q^{1/3}+1 & \beta q^{1/3}+\beta & \alpha q^{1/3}+\alpha & q^{2/3}+1 & q^{2/3}+1 & q^{2/3}+1 & \beta q^{1/3}+\beta & q^{1/3}+1 & \alpha q^{1/3}+\alpha \\ q^{2/3}+1 & q^{2/3}+1 & q^{2/3}+1 & q^{1/3}+1 & \alpha q^{1/3}+\alpha & \beta q^{1/3}+\beta & \alpha q^{1/3}+\alpha & q^{1/3}+1 & \beta q^{1/3}+\beta \\ q^{1/3}+1 & \alpha q^{1/3}+\alpha & \beta q^{1/3}+\beta & q^{1/3}+1 & \beta q^{1/3}+\beta & \alpha q^{1/3}+\alpha & \beta+1 & \beta+1 & \beta+1 \end{pmatrix}$$

Figure 2. Parity check matrix of rank 6

$$\begin{aligned} d &\geq \deg G - 2g + 2 \\ r &\leq d + g - 1 \end{aligned} \qquad (2)$$

The actual parameters for the example considered are $d = 6, r = 6$.

We will now introduce an analytic model for the nonsingular elliptic cubic based on the Riemann surface for the elliptic normal curve $w = \sqrt{z(1-z)(1-k^2z)}$. This surface is a torus which becomes simply connected by introducing two cuts. It has two periods, the complete elliptic integrals of the first kind

$$K = \int_0^1 \frac{dz}{2w} \quad \text{and} \quad iK' = \int_1^{k^2} \frac{dz}{2w}$$

The elliptic integral of the first kind

$$u = \int_0^z \frac{dz}{2w}$$

maps the cut torus conformally onto a rectangle whose sizes are $2K$ and $2iK'$. The inverse map from this fundamental domain onto the torus determines the coordinates $\{z, w\}$ of a point u in the complex plane. This map is described by quotients of power series in the variable $q = e^{\pi i K'/K}$ and named Theta-series in honor of Jacobi who used this notation. In particular the coordinates on the torus of points that lie on a regular grid inside the fundamental domain can be expressed in terms of quotients of Theta- series with broken characteristic [3]. For our particular example these are the 9 trisection points and the corresponding 9 characteristics are obtained by allowing g and h to assume values in the set $\{-1,0,1\}$; the argument v below is $u/2K$.

$$\vartheta\begin{bmatrix} g \\ h \end{bmatrix}(v) = \sum_{m=-\infty}^{m=\infty} q^{\left(m+\frac{g}{3}\right)^2} e^{2i\left(m+\frac{g}{3}\right)\left(v+\frac{h\pi}{3}\right)} \qquad (3)$$

The 9 trisection points are imbedded into the projective plane by triple products of these series such that the characteristic sum in each product is congruent to zero *mod* 3. The projective coordinates chosen are

$$\begin{aligned} X_0 &= \vartheta\begin{bmatrix} 0 \\ 0 \end{bmatrix}(v)\vartheta\begin{bmatrix} 1 \\ 0 \end{bmatrix}(v)\vartheta\begin{bmatrix} -1 \\ 0 \end{bmatrix}(v) \\ X_1 &= \vartheta\begin{bmatrix} 0 \\ 1 \end{bmatrix}(v)\vartheta\begin{bmatrix} 1 \\ 1 \end{bmatrix}(v)\vartheta\begin{bmatrix} -1 \\ 1 \end{bmatrix}(v) \\ X_2 &= \vartheta\begin{bmatrix} -1 \\ -1 \end{bmatrix}(v)\vartheta\begin{bmatrix} 0 \\ -1 \end{bmatrix}\vartheta\begin{bmatrix} 1 \\ -1 \end{bmatrix}(v) \end{aligned} \qquad (4)$$

The Sylvester form of the nonsingular cubic in the projective plane is a power series identity satisfied by these three power series

$$X_0^3 + X_1^3 + X_2^3 + 6m X_0 X_1 X_2 = 0 \qquad (5)$$

The parameter m is a modular invariant that parametrizes the family of cubics. Its value is $m = -(1 + 2c^3)/6c^2$ where c is given in terms of theta-nulls [5]

$$c = \frac{\vartheta\begin{bmatrix} 0 \\ 1 \end{bmatrix}(0)\vartheta\begin{bmatrix} 1 \\ 1 \end{bmatrix}(0)\vartheta\begin{bmatrix} -1 \\ 1 \end{bmatrix}(0)}{\vartheta\begin{bmatrix} 0 \\ 0 \end{bmatrix}(0)\vartheta\begin{bmatrix} 1 \\ 0 \end{bmatrix}(0)\vartheta\begin{bmatrix} -1 \\ 0 \end{bmatrix}(0)} \qquad (6)$$

The reason that equations (1) and (5) have different forms is that the coordinate origin in the projective plane in [2] is picked such that the term $6m X_0 X_1 X_2$ is zero. The coefficients of the power series in X_0, X_1, and X_2 at the 9 trisection points are algebraic integers obtained from the cubic root of unity. In Scratchpad II by a simple coercion we force them to lie in $GF(2^2)$. The projective coordinates of all 9 trisection points are obtained by using the transformation formula for Theta-series[3]; the parameter τ in equation (7) is iK'/K.

$$\begin{aligned} &\vartheta\begin{bmatrix} g \\ h \end{bmatrix}\left(v + g'\frac{\pi\tau}{3} + h'\frac{\pi}{3}\right) = \\ &\vartheta\begin{bmatrix} g+g' \\ h+h' \end{bmatrix}(v)\, e^{-\frac{ig'}{3}\left(\frac{2\pi(h+h')}{3} + \frac{g'\pi\tau}{3}\right)} \end{aligned} \qquad (7)$$

We have implemented in Scratchpad II the Jacobi Theta series in a package named THETA3 and with it we have checked equation (5). In the computation of the check matrix we make use of a Pade approximation algorithm [1] that was implemented in a Scratchpad II package named PADE which approximates $\{X_1/X_0, X_2/X_1\}$ by quotients of polynomials in $q^{1/3}$. We chose the lowest degree approximants with numerator of degree 1 and denominator of degree 2. The approximants satisfy equation (5) exactly. The parity check matrix is computed as $\{\phi_i(P_j)\}$ where we use the same linear system as in [2]. All the elements in this matrix are nonzero, it has rank 6, and it is given in Figure 2 on page 6.

The generator matrix obtained as the null space of this matrix is as follows

$$\begin{pmatrix} \alpha & 1 & \beta & \alpha & \beta & 1 & 0 & 0 & 0 \\ 0 & \alpha & \alpha & 1 & 1 & 0 & 1 & 1 & 0 \\ \alpha & 0 & \beta & \alpha & \alpha & 0 & 1 & 0 & 1 \end{pmatrix}$$

The code over $GF(2^2)$ generated by this matrix has distance $d = 6$ and 6 control symbols, i.e., the same parameters as the code produced in [2]. Furthermore the weight enumerators of the two codes are identical, both have 63 nonzero code words of block length 9 out of which 36 have Hamming weight 6 and 27 have Hamming weight 8.

To conclude, we have shown an alternative method for the derivation of algebraic ECC on algebraic curves that seems to give equivalent codes. In comparison with the geometric model used by Goppa, the Riemann surface provides a local analytic structure which should be useful in the reconstruction of code functions, i.e, in decoding. So far this problem has remained unsolved within the framework of the geometric method.

The first author acknowledges the warm support of the Computer Algebra Group during his visit in the winter of 1987 as well as several stimulating discussions with Professors Chudnovsky from Columbia University.

References

[1] Baker, G.A., and Graves-Morris, P.R., "Pade approximants: Basic Theory Part I", *Encyclopedia of Mathematics*, vol. 13, Reading, Massachusetts: Addison-Wesley, 1981.

[2] Goppa, V.D., "Codes on algebraic curves," *Soviet Math.* 24 (1981): 170-172.

[3] Krazer, A., *Lehrbuch der Thetafunktionen*, New York: Chelsea, 1970.

[4] Katsman, G.L., Tsfasman,M.A. and Vladut, S.G., "Modular curves and codes with a polynomial construction," *IEEE Tr. on IT*, 30 (1984): 353-355.

[5] Sievert, H., *Die Parameterdarstellung der Kurven 3. Ordnung durch Thetafunktionen*, Pr. Bayreuth, 1905.

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: My terminal does not have brackets ("[" and "]") or braces ("{" and "}"). Is there another way to enter these characters so that I can create lists and sets? prime number p dividing an integer q?

A: The 2 character combinations "(|" and "|)" and "(<" and ">)" may be substituted for "[" and "]" and "{" and "}", respectively.

```
(|1,2,3|)

   (1)  [1,2,3]

   Type: L I

(<1,2,3>)

   (2)  {1,2,3}

   Type: FSET I
```

Another way of accomplishing the same thing is to use the n-ary function ***construct*** instead of "[" and "]". The function ***brace*** may be applied to a list to create a set.

```
construct(1,2,3)

   (3)  [1,2,3]

   Type: L I

brace construct(1,2,3)

   (4)  {1,2,3}

   Type: FSET I
```

Q: How do I evaluate a polynomial with a particular value for a variable?

A: Throughout Scratchpad II the function ***eval*** is used to do what you want. For example, given the polynomial

```
p := x**3 - (x*y**2 - 2*z)**2

                2       2     2 4    3
   (1)  - 4z  + 4x y z - x y  + x

   Type: P I
```

you can substitute 10 for x by issuing

```
eval(p,x,10)

                2       2         4
   (2)  - 4z  + 40y z - 100y  + 1000

   Type: P I
```

The value need not be a constant. You can substitute another polynomial for the variable.

```
eval(p,x,z+1)

   (3)
      3      4     2  .    2      4     2
     z  + (- y  + 4y  - 1)z  + (- 2y  + 4y  + 3)z
   +
       4
     - y  + 1

   Type: P I
```

There also a form of *eval* that does *simultaneous* substitution for several variables.

```
eval(p,[x,y,z],[y,z,x])

          2 4          2    3     2
   (4)  - y z  + 4x y z  + y  - 4x

   Type: P I
```

As one would expect, this is different from sequential substitution.

```
eval(eval(eval(p,x,y),y,z),z,x)

          6     4    3     2
   (5)  - x  + 4x  + x  - 4x

   Type: P I
```

Q: What is the difference between *f* and *g* as defined by

```
f == 3
g() == 3
```

A: The difference is that *g* is a nullary function that may be called to return the value 3 while *f* is a rule that evaluates to 3. You can see this by defining and *g* as above and and then asking for their values.

```
f
   Compiling body of rule f to compute value of type I

   (3)  3

   Type: I
```

So when you mention *f* you get 3.

```
g

   (4)  g () == 3

   Type: □
```

That is, *g* is a function. You must call the function to get 3.

```
g()
   Compiling function g with signature () -> I

   (5)  3

   Type: I
```

Rules are convenient to use when the definitions have dependencies on values that are changing. Functions may also be used in this case and must be used when you have one or more arguments or when you want to create a nullary function object. For example, the function ***apply*** will take a nullary function object and then call it, returning the result.

```
apply nullaryFun == nullaryFun()

h : () -> I

h() == 3


apply h
   Compiling function apply with signature
     (() -> I) -> I
   Compiling function h with signature () -> I

   (8)  3
```

Streams and Power Series

Streams have been in Scratchpad II for some time. They are now implemented by a domain constructor Stream and may be infinite, unlike lists. A stream is represented by a list whose last element is a function that contains the wherewithal to create the rest of the list from that point, should it ever be required.

The stream functions provided are ***take***, ***drop***, ***elt***, ***null***, ***cons***, ***first*** and ***rest*** (similar to the list functions), together with functions for creating finite and infinite streams. There are also packages that supply several general purpose functions from streams to streams. Since some of these functions operate on functions, another package called MappingPackage has been provided to simplify the expression of functional arguments. Some examples follow:

```
)set streams calculate 5
```

All evaluated elements are printed, but at least the first 5 will be evaluated.

```
a := [1..]
```

```
   (1)  [1,2,3,4,5,...]
```

```
b := [i+1 for i in a]
```

```
   (2)  [2,3,4,5,6,...]
```

Select the 20^{th} element:

```
b.20
```

```
   (3)  22
```

The first 21 elements of b are evaluated:

```
b
```

```
   (4)
   [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
    16, 17, 18, 19, 20, 21, 22, ...]
```

The stream of odd integers

```
[i for i in a | oddp i]
```

```
   (5)  [1,3,5,7,9,...]
```

```
[[i,j] for i in a for j in b]
```

```
   (6)  [[1,2],[2,3],[3,4],[4,5],[5,6],...]
```

```
)set streams calculate 10
```

append(a,b) concatenates streams a and b

```
append([i for i in a while i<7],a)
```

```
   (7)
   [1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
    11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, ...]
```

The sum of a finite stream of integers:

```
reduce(0,_+$I,take(a,10))
```

```
   (8)  55
```

A stream of partial sums:

```
scan(0,_+$I,a)
```

```
   (9)  [1,3,6,10,15,21,28,36,45,55,...]
```

To save space infinite streams of the same element have a loop at the end.

```
a:ST I := repeating([8])
```

```
   (10)  [8]
```

```
[i+1 for i in a]
```

```
   (11)  [9]
```

The functions in the Stream domain and stream packages are particularly suitable for the implementation of algorithms on power series. The domain PowerSeries is provided as a field, and the domain UnivariatePowerSeries and an elementary function package adds to it the functions ***exp, log, sin, cos, tan,***, composition, lagrange inversion, reversion together with the solution of linear differential equations in power series.

A general method of producing programs which solve recursion or differential equations in power series by the method of undetermined coefficients has been developed in which the program can be written down almost immediately from the defining relation. In the method of undetermined coefficients a trial series together with an initial value or two is substituted into the recursion or differential equation, and then coefficients of equal powers are equated.

In these programs the trial series is made up of the initial values followed by the as yet unevaluated stream. The tail of the stream is then defined in terms of the whole stream and when elements are required the trial series becomes the resulting stream. The program, because it uses functions that operate on whole streams, rather than stream elements has the same structure as the defining relation.

For example e raised to the power series power $A(x)$, has defining relation

$$(e^{A(x)})' \equiv A'(x)e^{A(x)}$$

The corresponding program for generating the power series exp A, in Scratchpad II, where A is a power series is

```
exp A == integrate(1,deriv A*exp A))
```

in which ***integrate*** and ***deriv*** integrate and differentiate power series. Some examples follow. The command

```
)set streams calculate n
```

will cause the series up to the n^{th} coefficient to be printed.

The following declares and assigns x to be a UPS(x,RN), in other words a Univariate-PowerSeries with variable x and with rational number coefficients.

```
x := ps x

   (12)  x
```

We can now compute easily with power series involving x.

```
exp x

   (13)
                1  2    1  3    1  4     1  5
     1 + x + (-)x  + (-)x  + (--)x  + (---)x
                2       6       24       120
   +
      1  6     1  7     1  8      1  9
     (---)x  + (----)x  + (-----)x  + (------)x
      720      5040      40320      362880
   +
         1     10       11
     (-------)x   + 0(x   )
      3628800
```

```
cos x ** cos x

   (14)
          1  2    7  4    19  6    1597  8
     1 - (-)x  + (--)x  - (---)x  + (-----)x
          2       24       180      40320
   +
        373  10       11
     - (-----)x   + 0(x   )
        32400
```

```
x/(exp x-1)

   (15)
          1      1  2     1  4      1  6
     1 - (-)x + (--)x  - (---)x  + (-----)x
          2      12       720      30240
   +
           1      8         1      10       11
     - (-------)x  + (--------)x   + 0(x   )
       1209600       47900160
```

```
exp(exp x-1)

   (16)
                2    5  3    5  4    13  5    203  6
     1 + x + x  + (-)x  + (-)x  + (--)x  + (---)x
                     6       8       30       720
   +
      877  7    23  8    1007  9     4639  10
     (----)x  + (---)x  + (-----)x  + (------)x
      5040      224       17280      145152
   +
        11
     0(x   )
```

```
atan (x)

                1  3    1  5    1  7    1  9       11
   (18)  x - (-)x  + (-)x  - (-)x  + (-)x  + 0(x   )
                3       5       7       9
```

Power series provide a method of solving differential equations when all else fails. The function ***lde*** solves the n^{th} order linear differential equation, its argument is a list of power series coefficients. The two solutions of

$$y'' + (\cos x)y' + (\sin x)y = 0$$

are

```
lde([sin x,cos x])
   (19)
   [
          1  2    1  4    31  6     379  8
      1 - (-)x  + (-)x  - (---)x  + (-----)x
          2       6       720      40320
     +
          1639  10       11
       - (------)x   + 0(x   )
          907200
     ,
        1  3    1  5     59  7     31  9       11
     x - (-)x  + (--)x  - (----)x  + (----)x  + 0(x   )
        3       10       2520      6480
     ]
```

Power series are also used as enumerating generating functions. For example, the function ***lambert*** will transform one series into another in which the coefficient A_n of x^n is the sum of the coefficients of the original a_n for all i that divide n, including 1 and n. In other words, if $f(x)$ is a power series, then *lambert(f)* is the power series

$$f(x) + f(x^2) + f(x^3) + f(x^4) + \ldots$$

The series for the number of divisors of n is

```
lambert(x/(1-x))

   (20)
          2     3     4     5     6     7     8
     x + 2x  + 2x  + 3x  + 2x  + 4x  + 2x  + 4x
   +
        9     10       11
     3x  + 4x   + 0(x   )
```

Using this function it is possible to expand certain infinite products as power series. For example the enumerating generating function for partitions is

$$\prod_{n=1}^{\infty} \frac{1}{(1-q^n)}$$

```
partitions := exp(lambert(log(1/(1-x))))

   (21)
                2     3     4     5     6     7
     1 + x + 2x  + 3x  + 5x  + 7x  + 11x  + 15x
   +
         8     9      10      11
     22x  + 30x  + 42x   + O(x  )

euler := 1/partitions

                  2   5   7     11
   (22)  1 - x - x  + x  + x  + O(x  )
```

The generating function for partitions into distinct parts is:

$$\prod_{n=1}^{\infty}(1+q^n)$$

```
exp(lambert(log(1+x)))

   (23)
              2     3     4     5     6     7     8
     1 + x + x  + 2x  + 2x  + 3x  + 4x  + 5x  + 6x
   +
        9      10      11
     8x  + 10x   + O(x  )
```

The generating function for the Legendre polynomials is

$$\frac{1}{(1-2xt+t^2)^{1/2}}$$

and with suitable declarations for x and t may be expanded directly, as follows.

```
(1-2*x*t+t**2)**(-1/2)

   (24)
                      2           2
     1 + x*t + ((3/2)x  - 1/2)t
   +
             3           3
     ((5/2)x  - (3/2)x)t
   +
             4           2           4
     ((35/8)x  - (15/4)x  + 3/8)t
   +
             5           3           5
     ((63/8)x  - (35/4)x  + (15/8)x)t
   +
             6           4           2           6
     ((231/16)x  - (315/16)x  + (105/16)x  - 5/16)t
   +
                 7           5           3
       (429/16)x  - (693/16)x  + (315/16)x
     +
       - (35/16)x
   *
      7
     t       .
   +
       8
     O(t )
```

These examples show the present capability of writing expressions that denote power series. It should be possible in the future to enter differential or recursion equations that define new power series in terms of existing ones as suggested in the example for exp above. Other plans are to make multivariate power series more usable and to add Puiseux series.

William H. Burge
Stephen M. Watt
Scott C. Morrison

Primary Decomposition of Ideals

Scratchpad II now provides a facility for the primary decomposition of polynomial ideals over fields of characteristic zero. The algorithm is discussed in [1] and works in essentially two steps:

1. the problem is solved for 0-dimensional ideals by "generic" projection on the last coordinate
2. a "reduction process" uses localization and ideal quotients to reduce the general case to the 0-dimensional one.

The Scratchpad II constructor IdealDomain represents ideals with coefficients in any field and supports the basic ideal operations, including intersection, sum and quotient. IdealDecompositionPackage contains the specific functions for the primary decomposition and the computation of the radical of an ideal with polynomial coefficients in a field of characteristic 0 with an effective algorithm for factoring polynomials.

The follow examples illustrate the capabilities of this facility. First consider the ideal generated by x^2+y^2-1 (which defines a circle in the (x,y)-plane) and the ideal generated by x^2-y^2 (corresponding to the straight lines $x = y$ and $x = -y$.

```
f,g : DMP([x,y],RN)
n,m : L DMP([x,y],RN)

m := [x**2+y**2-1]

          2   2
   (3)  [x  + y  - 1]

   Type: L DMP([x,y],RN)

n := [x**2-y**2]

          2   2
   (4)  [x  - y ]

   Type: L DMP([x,y],RN)
```

We find the equations defining the intersection of the two loci. This correspond to the sum of the associated ideals.

```
id := ideal m  + ideal n

          2   1  2   1
   (5)  [x  - -,y  - -]
               2      2

   Type: DIDEAL(RN,DP(2,NNI),[x,y],DMP([x,y],RN))
```

We can check if the locus contains only a finite number of points, that is, if the ideal is zero-dimensional.

```
iszerodim id

   (6)  true

   Type: B

iszerodim(ideal m)

   (7)  false

   Type: B
```

We can find polynomial relations among the generators (f and g are the parametric equations of the knot).

```
f := x**2-1

         2
   (8)  x  - 1

   Type: DMP([x,y],RN)

g := x*(x**2-1)

         3
   (9)  x  - x

   Type: DMP([x,y],RN)

relationsIdeal [f,g]

              2     3     2
   (10)  [- %D  + %C  + %C ]

   Type: L P RN
```

We can compute the primary decomposition of an ideal.

```
l: L DMP([x,y,z],RF I)

   Type: VOID

l:=[x**2+2*y**2,x*z**2-y*z,z**2-4]

          2     2   2      2
   (12)  [x  + 2y ,x z  - y z,z  - 4]

   Type: L DMP([x,y,z],RF I)
```

```
ld:=primaryDecomp ideal l

                 1    2              1    2
   (13)  [[x + - y,y ,z + 2],[x - - y,y ,z - 2]]
                 2                   2

   Type: L DIDEAL(RF I,DP(3,NNI),[x,y,z],DMP([x,y,z],RF I))
```

We can intersect back:

```
"intersect"/ld

                1       2  2
   (14)  [x - - y z,y ,z  - 4]
                4

   Type: DIDEAL(RF I,DP(3,NNI),[x,y,z],DMP([x,y,z],RF I))
```

We can compute the radical of every primary component. Their intersection is equal to the radical of the ideal of l.

```
rr:="intersect"/[radical ld.i for i in 0..1]

              2
   (15)  [x,y,z  - 4]

   Type: DIDEAL(RF I,DP(3,NNI),[x,y,z],DMP([x,y,z],RF I))

ss:=radical ideal l

              2
   (16)  [x,y,z  - 4]

   Type: DIDEAL(RF I,DP(3,NNI),[x,y,z],DMP([x,y,z],RF I))
```

References

[1] Gianni, P., Trager, B., and Zacharias, G., "Gröbner Bases and Primary Decomposition of Polynomial Ideals," to appear in *Journal of Symbolic Computation.*

Patrizia Gianni

Mappings as First Class Objects

Defining and Applying Mappings

Mappings can be as important as the values on which they act. In Scratchpad II functions are treated as first class objects; function-valued variables can be used in any way that variables of other types may be used.

Mappings may be defined interactively in the interpreter or they may be defined in a library of compiled code, as are the operations provided by types.

The simplest thing that can be done with a function object is to apply it to arguments to obtain a value.

```
5 + 6

   (1)  11

Type: I
```

If there are several functions with the same name, the interpreter will choose one of them. An attempt is made to choose the function according to certain generality criteria.

When a particular function is wanted, the plus on GF(7) for example, it can be specified by a *package call* using "$".

```
5 +$GF(7) 6

   (2)  4

Type: GF 7
```

Manipulating Mapping Values

Probably the next simplest thing is to assign a function value to a variable.

```
-- assigning + from GF(7) to a variable
plusMod7  := _+$GF(7);  plusMod7(5, 6)

   (3)  4

Type: GF 7
```

To access the value of the function object for a top level map it must be declared first.

```
double: I -> I
double n == 2*n

f := double;  f 13

   (6)  26

Type: I
```

Mappings can be accepted as parameters or returned as values. Here we have an example of a function as a parameter

```
-- apply takes a function as 1st parameter
-- and invokes it on the 2nd parameter

apply: (I -> I, I) -> I

apply(f, n) == f n


apply(double, 32)

   (9)  64

Type: RN
```

and as a return value

```
-- trig returns a function as its value
trig: I -> (BF -> BF)

trig n ==
  if oddp n then sin$BF else cos$BF


t := trig 1;  t 0.1

   (12)  0.099 83341 66468 28152 30681 4198

Type: BF
```

Several operations are provided to construct new functions from old. The most common method of combining functions is to compose them.

"*" is used for functional composition.

```
quadruple := double * double;  quadruple 3

   (13)  12

Type: I
```

"**" is used to iterate composition.

```
octuple := double**3;  octuple 3

   (14)  24

Type: I
```

diag gives the diagonal of a function. That is, if g is `diag f` then `g(a)` is equal to `f(a,a)`.

```
square := diag _*$I;  square 3

   (15)  9

Type: I
```

twist transposes the arguments of a function. If g is defined as `twist f` then `g(a,b)` has the value `f(b,a)`.

```
power := _**$RN;
rewop := twist power;  rewop(3, 2)

   (17)  8

Type: RN
```

Mappings of lower arity can be defined by restricting arguments to constant values. The operations *cur* and *cul* fix a constant argument on the right and on the left, respectively. For unary functions, *cu* is used.

```
square := cur(power, 2);
square 4    -- square(a) = power(a,2)

   (18)  16

Type: RN
```

It is also possible to increase the arity of a function by providing additional arguments. For example, ***vur*** makes a unary function trivially binary; the second argument is ignored.

```
binarySquare := vur(square);
binarySquare(1/2, 1/3)

         1
   (19)  -
         4

Type: RN
```

The primitive combinator for recursion is ***recur***. If `g` is `recur(f)` then `g(n,x)` is given by `f(n,f(n-1,..f(1,x)..))`.

```
fTimes := recur _*$NNI;
factorial := cur(fTimes, 1::NNI);
factorial 4

   (20)  24

Type: NNI
```

Mappings can be members of aggregate data objects. Here we collect some in a list. The unary function `incfn.i` takes the `i`-th successor of its argument.

```
incfn := [(succ$SUCCPKG)**i for i in 0..5];
incfn.4 9

   (21)  13

Type: I
```

Mappings as Program-Environment Pairs

In practice, a mapping consists of two parts: a piece of program and an environment in which that program is executed. The display of mapping values appear as `theMap(s, n)`, where `s` is a hideous internal symbol by which the program part of the mapping is known, and `n` is a numeric code to succinctly distinguish the environmental part of the mapping.

```
recipMod5 := recip$GF(5)

   (22)  theMap(MGF;recip;$U;17,642)

Type: GF 5 -> Union(GF 5,failed)
```

```
plusMod5  := _+$GF(5)

   (23)  theMap(MGF;+;3$;12,642)

Type: (GF 5,GF 5) -> GF 5
```

```
plusMod7  := _+$GF(7)

   (24)  theMap(MGF;+;3$;12,997)

Type: (GF 7,GF 7) -> GF 7
```

Notice above that the program part of `plusMod5` is the same as for `plusMod7` but that the environment parts are different. In this case the environment contains, among other things, the value of the modulus. The environment parts of `recipMod5` and `plusMod5` are the same.

Creating "Own" Variables

When a given mapping is restricted to a constant argument, the value of the constant becomes part of the environment. In particular when the argument is a mutable object, closing over it yields a program with an *own* variable. For example, define `shiftfib` as a unary mapping which modifies its argument.

```
FibVals := Record(a0: I, a1: I)

   (25)  Record(a0: I,a1: I)

Type: DOMAIN
```

```
shiftfib: FibVals -> I

shiftfib r ==
  t := r.a0
  r.a0 := r.a1
  r.a1 := r.a1 + t
  t
```

Now `fibs` will be a nullary program with state. Since the parameter `[0,1]` has not been assigned to a variable it is only accessible by `fibs`.

```
fibs := cu(shiftfib, [0,1]$FibVals)

   (29)  theMap(%G12274,721)

Type: () -> I
```

```
[fibs() for i in 0..30]

   (30)
   [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,
    233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946,
    17711, 28657, 46368, 75025, 121393, 196418, 317811,
    514229, 832040]

Type: L I
```

Fixed Point Operations

A fixed point x of a map f is a point in the domain of f such that $x = f(x)$. Given a function which operates on a recursively defined data type, it is often possible to compute a useful fixed point. A powerful method for manipulating infinite structures is to compute the fixed point of structure transforming functions. As well as providing a functional mechanism for constructing self-referential structures, a combination of lazy evaluation and self-reference may be achieved.

Consider a recursively defined data type T and the class of functions mapping T → T. Certain functions in this class have trivial fixed points: the identity and constant valued functions. Some functions in the class may have no fixed point. (The fact that negation has no fixed point leads to the Russell paradox.) Other functions may have a fixed point which it is impossible to compute effectively.

If we restrict our attention to functions which do not perform operations on their argument but rather just include it in a new structure which is returned as the value then we may always compute a fixed point.

As an example, an infinite repeating list of values can be obtained as follows:

```
cons1234(l) == cons(1,cons(2,cons(3,cons(4,l))))

repeating1234 := fixedPoint cons1234
```

A fixed point finding operation is provided which operates on a stream transforming function and finds its fixed point, a stream.

```
a:=integers 1

   (2)  [1,2,3,4,5,6,7,8,9,10,...]
```

The function below prefixes a 1 to an integer stream.

```
f1(x: ST I): ST I == cons(1,x)

f1 a

   (4)  [1,1,2,3,4,5,6,7,8,9,10,...]
```

and the fixed point of *f1* is an infinite stream of 1's

```
b := fixedPoint f1

   (5)  [1]
```

Similarly

```
f2(x: ST I): ST I == append([1,2,3,4,5,6], x)

fixedPoint f2

          ----------
   (8)  [1,2,3,4,5,6,1]
```

Here is another way to define the Fibonacci number stream. The plus operation takes two streams and adds them pair-wise.

```
f3(fib: ST I): ST I == cons(1,fib+cons(0,fib))

f3 b

   (10)  [1,1,2,2,2,2,2,2,2,2,...]

fixedPoint f3

   (11)  [1,1,2,3,5,8,13,21,34,55,...]
```

The stream of Catalan numbers:

```
f4(cat: ST I): ST I == cons(1,cat*cat)

fixedPoint f4

   (14)  [1,1,2,5,14,42,132,429,1430,4862,...]
```

The function *integ* integrates a stream viewed as the coefficients of a power series.

```
integ b

                1 1 1 1 1 1 1 1 1
   (15)  [1,-,-,-,-,-,-,-,-,-,...]
          2 3 4 5 6 7 8 9 10
```

Here we compute the fixed point of the function g that integrates a stream, and adds the constant term 1.

```
g(e: ST RN -> ST RN) == cons(1,integ e)

fixedPoint g

                1 1 1   1   1   1     1     1
   (18)  [1,1,-,-,--,---,---,----,-----,------,...]
          2 6 24 120 720 5040 40320 362880
```

It is also possible to find the fixed point of a function that transforms a pair of streams to a pair of streams.

```
k(tr: L ST I): L ST I == [cons(0,tr.1),1/(1-tr.0)]

k([cons(0,b),b])

   (20)  [[0,1],[1,1,2,4,8,16,32,64,128,256,...]]
```

The fixed point of k is two mutually recursive streams. Computing this provides another way to obtain the stream of Catalan numbers.

```
fixedPoint(k, 2)

   (21)
   [[0,1,1,2,5,14,42,132,429,1430,...],

    [1,1,2,5,14,42,132,429,1430,4862,...]]
```

Stephen M. Watt
William H. Burge

Work in Progress

This section describes some work on the Scratchpad II system that is being undertaken but is not yet complete.

Support for Data Structures in Scratchpad II

An effort to "categorize" the data structures available in Scratchpad II is now under way. Until recently, the algebraic facilities in Scratchpad II have made use of only a few data structures, in particular, records, lists and vectors. However, now that the new compiler is being written in Scratchpad II, the need for other traditional data structures has arisen. For example, various parts of the new compiler make use of stacks, sets, tables and doubly linked lists. New applications of Scratchpad II will make use of dictionaries, graphs, priority queues, etc.. Furthermore, as the system evolves, the need for specialized and highly efficient data structures will arise. One such example is Stream which provides a mechanism for lazy evaluation. To systematically describe the relationships between data structures and the operations available for them, we are building a category hierarchy for data structures.

The following partial definition for the categories Collection, IndexedAggregate and FiniteLinearAggregate should illustrate our point. Note that Type is the category in which all domains in the system belong.

```
Collection(S:Type): Category with
   -- number of items in the collection
   #: $ -> NonNegativeInteger

   -- top level copy of the collection
   copy: $ -> $

   map: ((S,S)->S,$) -> $

   if $ has shallowlyMutable then
     mapInPlace: ((S,S)->S,$) -> $
```

```
IndexedAggregate(Index:Type,S:Type): Category ==
  Collection S with
    elt: ($,Index) -> S

    if $ has shallowlyMutable then
      setelt: ($,Index,S) -> S

FiniteLinearAggregate(S:Type): Category ==
  IndexedAggregate(Integer,S) with
    concatenate: ($,$) -> $

    if S has OrderedSet then
      sort: $ -> $     -- sort using the "<" from S

      if $ has shallowlyMutable then
        sortInPlace: $ -> $
```

The category Collection describes homogeneous aggregates of objects with the operations *#*, ***copy***, ***map*** and the operation ***mapInPlace*** (if a collection has the attribute shallowlyMutable). A collection must have the attribute shallowlyMutable if any of its operations update (mutate) it. The prefix "shallowly" indicates that we are referring to the replacement of one component with another, rather than updating the component itself. For example, the ***mapInPlace*** operation updates a collection "in place" whereas the ***map*** operation creates a new structure. Notice that this implies that for shallowlyMutable collections, the ***map*** operation may be defined by applying ***mapInPlace*** to a copy of the collection. Indexed aggregates are collections which are indexed by objects of some type. A table of key/entry pairs is an example of an indexed aggregate. A table is a collection of entries indexed by keys. A finite linear aggregate is an indexed aggregate where the index is an Integer. For example, a string is a finite linear aggregate of characters.

This categorization of the data structures provides two main benefits. The first and primary benefit is the inheritance and sharing of code. For example, a generic sorting routine can be written for shallowlyMutable finite linear aggregates whose components belong to an ordered set (one could implement quick-sort in terms of the operations *#*, ***elt*** and ***setelt***). Secondly, hierarchies give a useful and systematic method for defining consistent and comprehensive sets of operations on types. They explain certain similarities and differences between types. For example, lists and strings are both finite linear aggregates and hence can be concatenated, but lists are also recursive aggregates.

There is an additional benefit from having a hierarchical organization for the data structures. We intend to provide as part of the user interface a way in which a user can explore category hierarchies. The

user will be able to display the various categories in the system, see how they are related to one another, find out which domains belong to which categories and what operations are available. This will make it easier for a user to find out if there already exists a data structure in the system suitable for his purposes, or maybe one of the existing data structures will suffice with minor modifications.

Michael B. Monagan

Integration of Algebraic and Mixed Functions

We are now implementing an integration algorithm for algebraic and mixed functions, as well as unifying the existing integration packages into one implementation. The transcendental algorithm is being extended to handle arbitrary primitive extensions, which allows the system to integrate an element of $K(t)$, whenever t' is in K.

The following examples contain integrals that the development Scratchpad II system can now compute.

```
x : SPF RF RN

   Type: VOID

x*sqrt(a+b*x)

                ________
   (2)  x\|b x + a

   Type: SPF RF RN

integrate(%,'x)

        2 2 2   2          4  2
        - b x + -- a b x - -- a
        5       15         15       ________
   (3)  ------------------------- \|b x + a
                   2
                  b

   Type: IR SPF RF RN

x**3*sqrt(x**2+a)

           3 | 2
   (4)  x \|x  + a

   Type: SPF RF RN

integrate(%,'x)

                                           ______
         1  4   1     2     2   2    |  2
   (5)  (- x  + -- a x  - -- a )\|x  + a
         5      15        15

   Type: IR SPF RF RN

x/sqrt(a+b*x)

              x
   (6)  ----------
          ________
        \|b x + a

   Type: SPF RF RN

integrate(%,'x)

        2       4
        - b x - - a
        3       3       ________
   (7)  ----------- \|b x + a
             2
            b

   Type: IR SPF RF RN

1/(x**2*sqrt(x**2+a))

            1
            --
             2
            x
   (8)  ---------
          ______
         | 2
        \|x  + a

   Type: SPF RF RN

integrate(%,'x)

                 ______
          1     | 2
   (9)  - --- \|x  + a
          a x

   Type: IR SPF RF RN

erf(x)*exp(-x**2) / (erf(x)**3-erf(x)**2-erf(x) + 1)

                2
             - x
           %e     erf(x)
   (10)  -----------------------------
               3         2
         erf(x)  - erf(x)  - erf(x) + 1

   Type: SPF RF RN

integrate(%,'x)

   (11)
         1   ____
         - \|%pi
         4           ---                       1   ____
       - ---------- + >        %A log(erf(x)%A - - \|%pi )
         erf(x) - 1  ---                       8

                        2   1    ____2
                      %A  - -- \|%pi   = 0
                            64

   Type: IR SPF RF RN
```

Manuel Bronstein

Scratchpad II System Release Notes

This column lists changes made to Scratchpad II public systems over the last 7 months and replaces the "Status" column in previous newsletters. It describes new developments, modifications and bug fixes. Compiled by Robert S. Sutor. Unless otherwise specified, all changes refer to the interpreter and interactive interface.

Release: *SPAD2T: Early March, 1987*

1. You can now turn the history facility on and off by issuing

```
)set history on
```

and

```
)set history off
```

respectively. If it is off when the system starts, a message will tell how to turn it on. There will also be a warning of the workspace being cleared if it is nonempty and an attempt is made to toggle the history facility from on to off. off.

2. If your **SPADPROF INPUT** file does not define a command prompt, one will not be displayed. Issue

```
)set message prompt
```

to see the possible choices.

3. You now cannot use constructor abbreviations as identifiers on the left hand side of assignments or function definitions. This means the following is now illegal:

```
P(x) == x**2 + 1
```

Until types are completely handled by the interpreter analysis and not partly in the parser, this restriction will remain.

4. The "|" facility to create elements of SimpleAlgebraicExtension now uses a new facility called *resolveTCat*. In short, *resolveTCat*(T, C) tries to return a type of category C to which all objects of T can be coerced. Although not yet complete, it tries to be intelligent about the conditional categories of the top level constructor of T. Thus

resolveTCat(Gaussian(Integer)),Field)

is Gaussian(RationalNumber) and not just QuotientField(Gaussian (Integer)) . This facility will be used in several other places, including category-directed modemap selection.

5. Modemap selection may appear slightly faster for functions of several homogeneous arguments.

6. General modemap selection now partitions the set of possibly applicable modemaps into 2 groups:

   - those packages containing one or more of the top level constructor names of the arguments in the package names
   - everything else

   The first set is searched before the second, and this can often reduce the modemap selection process by 80% or more.

   Why it makes sense: There are some operations that are not in domains for organizational reasons or logical reasons. For example, some vector functions that have been recently added are in VectorPackage1. These should go in Vector but we don't want to recompile everything that uses Vector right now. Also, a function

```
map: ( A -> B, Vector A) -> Vector B
```

cannot go in Vector because 2 under-domains are needed (A and B). So this function is in VectorPackage2.

We have been trying to organize such auxiliary functions in packages using the naming conventions shown above. For example, Stream has 3 packages

StreamPackage1 (takes 1 argument)
StreamPackage2 (takes 2 arguments)
StreamPackage3 (takes 3 arguments)

Using this convention, it makes sense to look in these packages after checking the domains of the arguments for applicable functions.

Why it helps: Many (most?) of the functions that operate on a domain happen to be in domains with names such as the above. It is logical to look at these before the seemingly unrelated other constructors.

Coercions of the form D(T1) → D(T2) use the *map* function contained in such auxiliary packages. Thus coercions will be faster. The functions implementing "!" also lie in such packages.

What this means to you: You should follow the naming conventions shown when creating extra packages for domains.

7. `factor(n)` where `n` is an Integer will return FactoredRing Integer instead of FactoredForm Integer. The second domain is a Ring and so is generally much more useful.

8. To display LISP code generated for functions, issue

```
)set system functioncode on
```

instead of the now obsolete

```
)set system display on
```

To display optimized LISP code, issue

```
)set system optimization on
```

9. The operation of `%%` somewhat more robust, but we believe that the behavior of `)undo` is still unreliable.

10. This workspace introduces "frames" to a Scratchpad II interpreter session. A frame can be thought of a logical session within a physical session that you get when you start the system. You can have as many frames as you have room for. Each frame has its own step number, environment and history.

The frame that you get when you start is called "initial". Since every frame has its own independent history file, the basic history file is now called **INITIAL SPADHIST** (it was called **CURRNT HISTORY**). The history file of the last session you were in is now called **LASTSESS SPADHIST** (it was **ANCIENT HISTORY**).

You can create a new frame "zippy" by issuing `)frame new zippy`. If the history facility is on, the file **ZIPPY SPADHIST A** will be created as you enter commands. If you wish to go back to what you were doing in the "initial" frame, use `)frame next` or `)frame last` to cycle through the ring of available frames. If you want to throw away a frame (say "zippy"), issue `)frame drop zippy` If you omit the name, the current frame is dropped. To find out the names of all frames, issue `)frame names`.

If you do use frames with the history facility on, you may want to clean your A disk of **SPADHIST** files from time to time. The **INITIAL SPADHIST** file is renamed to **LASTSESS SPADHIST** when you start a new physical Scratchpad II session.

A couple of `)set` flags have been added to make it easier to tell where you are among multiple frames.

```
)set message frame on|off
```

will print more messages about frames when it is set on. By default it is off.

```
)set message prompt frame
```

will give a prompt that looks like

```
initial (1) ->
```

when you start. That is, the frame name and step make up the prompt.

Release: *SPAD2W: Late March, 1987*

1. We have been improving the output routines to make the radical and exponential forms from ElementaryFunction more readable. In that domain, something like `sqrt(x + 1)` is stored as `exp(log(x+1)/2)` and `x**x` is `exp(x*log(x))`. While this is true and useful, seeing the output as such is not that enlightening. Try `e : EF P I := x**(x**(x**(x**x)))` to see that it now looks "like it should." Also: output for roots has been improved.

2. On the request of Fritz Schwarz', we have added SymbolPackage to the system to allow the manipulation of scripted symbols under program control. The functions ***name, scripted?*** and ***scripts*** examine symbols, and the functions ***script, subscript*** and ***superscript*** build symbols after evaluating both the base name and the scripts. Run **SYMBOL INPUT** for a demonstration.

3. The continued fraction code has been revamped so that continued fractions are no longer restricted to be simple and finite. The output for continued fractions is now more compact and

more readily admits line-breaking. Run **ECF INPUT** for a demonstration.

4. On the request of Fritz Schwarz', we have added IndexedVariable to the system. The domain IV(u) provides an ordered set of variables with name u with integer subscripts. The intent of this domain is similar to OrderedVarlist, but when you want OV [u1, u2, u3...]. Like OV, the representation is very space and time efficient; each member of IV(u) is represented by the integer value of its subscript.

5. SquareMatrix now belongs to SquareMatrixCategory (seemed reasonable) and the latter has been moved from **LODO SPAD** to **MATRIX SPAD**. The definition of SMDR in **LODO SPAD** has been commented out.

6. There is now a distinction between the following

```
a == 2
b() == 2
```

In the first case, if you ask for a you will get the value of the rule **a**, which is 2.

In the second case, if you ask for b you will get the nullary function that b defines. In particular, you must ask for `b()` to get the computed value. Nullary functions are thus now first class objects.

7. The interpreter in SPAD2W can now use Scratchpad II library functions for retraction. A retraction is a special kind of coercion that coerces a "degenerate" object of type t to a underlying (usually) type t1. For example, rational numbers with denominators equal 1 can be retracted to integers, or constant polynomials can be retracted to the ring of coefficients. The new interpreter function is ***retractByFunction*** and will work for types that belong to the category RetractableTo(t1). These now include DMP, NDMP, G, Q, FFX, SM but will shortly include others. It works by first calling the function ***retractable?:*** from the domain. If the result is true, the function ***retract.*** is called and the answer returned.

   What this means to you as a Scratchpad II programmer is that you can can control when the interpreter pulls back values to underlying domains. By making your domains belong to the above category (if appropriate), you can allow the interpreter to do retractions. Moreover, it allows us to remove special code from the interpreter that knows the data representation of domains. Please keep this in mind when you update older domains or create new ones. See **GAUSS SPAD** for an example.

8. Please read the top of **SPAD SPADLIBS** for an explanation of the prefixes. Essentially, "*" is a comment, "?" means to check for consistency but not to put constructor information in **MODEMAP DATABASE**, "+" means to expose the constructor, "-" means to not expose it. Both of the last two forms have the information placed in **MODEMAP DATABASE**. Do not worry what "expose" means yet; just put a "+" next to everything except those things that obviously shouldn't have their ops available to the world (like MAPHACKS).

9. A processing change for **SPAD SPADLIBS**: instead of just reading the first available **SPAD SPADLIBS**, it will read every one on accessed disks. The union of all **LISPLIBs** in all such files will be used.

10. Rüdiger Gebauer's complete Gröbner Basis Package is now part of the public system.

Release: *SPAD2X: May 5, 1987*

This release has many changes from the previous system and is NOT compatible with SPAD2W because it has about 170 changed LISPLIBs. Here is a summary of the changes, in no particular order.

1. `)fc, )fec,` etc. now look in the last file you edited if it cannot locate a function.

2. When a failure occurs in interpret-code mode for a function, a better cleanup is performed.

3. Many categories now belong to RetractableTo. This will allow the interpreter to do automatic retraction of, say, constant polynomials to the coefficient ring. All but one interpreter retraction function that knows specific datatypes have been removed.

4. We have augmented ExtendedPolynomial and MPolyCat with several more functions, and these both now belong to CommonPolynomialCategory and RetractableTo.

5. We have added QuotientFieldCategory. QuotientField, RationalNumber and RationalFunction belong to it. This has RetractableTo plus PartialDifferentialRing SortedExpressions (conditionally). It also has some conditional functions to compute ***floor, ceil, whole*** and ***fract.***

6. RationalNumber is now just QuotientField Integer plus ***random*** and ***abs***.
7. RationalFunction R is now just QuotientField Polynomial R plus a couple of ***eval*** functions.
8. The HenselGCDPackage now will recall itself if the computed gcd does not pass an ***exquo*** test. QuotientField now checks that the division in cancelgcd is successful. Warning: we suspect there might be a bug in ***exquo*** somewhere.
9. We augmented BasicMatrixCategory with some more functions, including a ***pderiv***. Some of the specific matrix domains had a few functions added.
10. Most of the xxxPKG1 packages for the above domains were absorbed into the constructors. This will speed up modemap selection.
11. The output from the `)display dependents` and `)display ancestors` commands now list the Scratchpad II source file of the constructors. We hope you find this more useful than the old display.
12. Henselfactorize now has a check for a polynomial being irreducible by Eisenstein's criterion.
13. ComplexFloat has been improved with more added functions.
14. ElementaryFunctionPowerSeries has been changed. It now contains only the logarithmic, exponential and circular functions as power series to power series functions. The functions ***exp, log, sin, cos, tan, csc, sec, cot, asin, acos, atan, acsc, asec*** and ***acot*** are generally applicable to a field which has a TranscendentalFunctionCategory. Some alternative forms are generally applicable to power series in a field with restrictions on the constant term. The hyperbolic functions have been put into a separate package called HYPS for HyperbolicFunctionPowerSeries.
15. A bug was fixed in the coercions from SparseUnivariatePolynomial to and from UnivariatePoly in nontrivial cases. For example, P[x] RN → SUP QF P[x] RN was broken.
16. Packages SparseUnivariatePolynomialIntegration and UnivariatePolyIntegration have been added for integration of univariate polynomials over a field. An error break is taken when in characteristic $p > 0$ and p divides an exponent.
17. The interpreter can now handle $ forms with 1,0, true and false. E.g., `1$P(I)` or `true$Boolean`.
18. ArithmeticFunctions has been renamed to IntegerNumberTheoryFunctions and expanded. It now contains ***euler*** and ***bernouilli*** number functions, plus a blazingly fast ***fibonacci***. The ***classNumber*** function was removed as it was not implemented. Several other functions have been renamed.
19. The PolynomialNumberTheoryFunctions package contains routines to compute the bernoulli, cyclotomic, euler, hermite, laguere, legendre, chebyshev polynomials of the first and second kinds and also a function for computing the fixed divisor of a polynomial. The fixed divisor f of a polynomial $a(x)$ in $\mathbb{Z}[x]$ is the largest integer ≥ 1 that divides $a(x=k)$ for all k in $\mathbb{Z}$.
20. Functions ***odd?, even?*** and ***evenp*** were added to IntegerFunctions1.
21. A bug was fixed whereby COLLECTs with nontrivial steps did not work. That is, `[i for i in 10..1 by -1]` now works.

Release: *SPAD2Y: June 5, 1987*

1. The handling of `return` has been fixed in the interpreter. Run **TESTRET INPUT** for examples.
2. Simultaneous assignments like `(a,b) := (1,2)` or `(a,b) := (b,a)` now work. However, this only works for direct assignments and not things like

```
(a,b) := if a > b then (a,b) else (b,a)
```

3. An anomaly in output script lists for subscripted symbols was fixed. We added support for all 4 types of scripts to the SCRIPT Formula Formatter output code. The same was done for TeX™ (TeX is a trademark of the American Mathematical Association) output and the TeX™ output will also now produce root signs.
4. Some functions were moved from ComplexFloat to BigFloat and from BigFloat to ConversionPackage. BigFloat now has a nullary function ***precision*** that returns the current precision. The one argument ***precision*** returns the previous value as the result. New

functions in BigFloat include ***numeric, atan2*** and exponentiation by a RationalNumber.

5. We enhanced the FORTRAN output routines. Subscripts and roots are now handled. Some Scratchpad II elementary function names are translated into their FORTRAN equivalents. For example, `tan(log(x))` is now translated into `TAN(ALOG(X))`.

Release: *SPAD2Z: June 19, 1987*

1. We fixed a problem with the display of objects as parameters to constructors. Also fixed a bug in the constructor display routines so that things like #1 do not appear. Issue `)show SAE` to see the current proper display with both fixes.
2. We fixed an infinite loop that occurred when something like

```
u : P[x] I := z + 1   or
u : P[x] I := z + z + 1
```

was entered (illegal types in coercions).

3. We fixed a bug in Henselfactorize (Eisenstein criterion) that was incorrectly determining that some polynomials were irreducible.
4. We added some more messages so that things like

```
)history save
```

that have the wrong syntax will tell the user what to do.

5. We enhanced the modemap selection to better handle subdomains in ordering feasible functions. This fixes the strange behavior seen when raising elementary functions to powers.
6. We added a message about current unavailability of forms like `l.first := 8`..
7. We added functionality of forms like `l.first`, `m.last` and `k.last`.
8. The IntegerPrimesPackage has two new functions

```
prevPrime(n) -- compute the first prime < n
nextPrime(n) -- compute the first prime > n
```

The primality testing routine ***prime?*** has been rewritten as follows. The Solovay/Strassen algorithm was replaced by the Miller/Rabin algorithm. Both algorithms are probabilistic algorithms. Given a number n, the Solovay/Strassen algorithm tests whether

$$J(\frac{a}{n}) = a^{\frac{n-1}{2}} \bmod n$$

where a is a random integer. If n is composite, this test fails with probability > 1/2. The Miller/Rabin test is similar, but has the advantage of failing with probability > 3/4 . Furthermore, it is easier to compute: it does not require computation of the jacobi symbol $J(\frac{a}{n})$. We also included a test for divisibility of small primes.

Release: *SPAD2ZA: July 17, 1987*

1. The selection of the operation ***factor*** for multivariate polynomials has been improved. The problem was seen when the user attempted to factor objects belonging to DistributedMultivariatePolynomial with integer or rational number coefficients.
2. We greatly improved the printing of function definitions in the interpreter.
3. We changed the printing of "failed" so that it looks like a string (which it is).
4. We changed the representation of the value of `factor(0)` in the IntegerFactorizationPackage . This now agrees with the standard representation of 0 in FactoredRing Integer. This fixes a problem seen when doing something like:

```
f := factor ! gauss(1,1)
f**2
```

5. The definition of ***permutation*** in **COMBINAT SPAD** has been corrected to be $\frac{n!}{(n-k)!}$.
6. We recompiled **POLCOEFF SPAD** and fixed what appears to be a consistency bug when calling the ***coeff*** function.
7. We fixed a bug in showing subscripted variables in constructor expressions (e.g., `MP([x[1;2],y],l)`).
8. Coercions now work from QuotientField D1 D2 → D1 QuotientField D2 if D1 QuotientField D2 is a field. In particular, when D1 = Gaussian.
9. We fixed a bug in the commuting coercion of multivariate polynomials. This was seen when a message about MPolyCatPackage1 was displayed.

Release: *SPAD3A: August 16, 1987*

1. We have made some changes to the way the system responds when an external interrupt (PF12) occurs when)set break nobreak or)set break query hold.)set break nobreak is the default for most Scratchpad II users.

   Previously, the user was simply returned to top level if an interrupt occurred and)set break nobreak held. That was inconvenient if he/she just wanted to see if anything was happening. The following console shows the new behavior:

```
1+2
  loading I LISPLIB K for domain Integer

CP EXT

  >> User interrupt:
  You have chosen to interrupt Scratchpad II
  processing.

  You have three options. Enter:
    continue  to continue processing,
    top       to return to top level, or
    break     to enter a LISP break loop.

  Please enter your choice now:
break

  Enter (FIN) when you are ready to continue
  processing where you interrupted the system,
  enter (UNWIND 50) when you wish to return
  to top level.

  Error: forced break,
  Break taken,
  FIN with any value, otherwise UNWIND to exit
  break loop.

(UNWIND 50)

  Scratchpad II

1+2
  loading I LISPLIB K for domain Integer

CP EXT

  >> User interrupt:
  You have chosen to interrupt Scratchpad II
  processing.

  You have three options. Enter:
    continue  to continue processing,
    top       to return to top level, or
    break     to enter a LISP break loop.

  Please enter your choice now:
top

  You are being returned to the top level of the
  interpreter.

  Scratchpad II

1+2
  loading REPSQ LISPLIB K for package
    RepeatedSquaring
  loading INTFACT LISPLIB K for package
    IntegerFactorizationPackage

CP EXT

  >> User interrupt:
  You have chosen to interrupt Scratchpad II
  processing.

  You have three options. Enter:
    continue  to continue processing,
    top       to return to top level, or
    break     to enter a LISP break loop.

  Please enter your choice now:
continue

  Processing will continue where it was
  interrupted.

  loading NTHROOT LISPLIB K for package NthRoot
  loading FF LISPLIB K for domain FactoredForm

  (1) 3

  Type: I
```

2. An implementation of multisets is now available in the system library (see **MSET SPAD**). They are represented as tables with the multiset members acting as keys and the number of times they occur being the associated entries. So 4 is only stored once in the multiset s below.

```
s: Multiset I

Type: VOID

s := {1,2,3,4,5,4,3,2,3,4,5,6,7,4,10}

  (2) {5,5,2,2,3,3,3,6,4,4,4,4,1,7,10}

  Type: MSET I
```

3. Functions calls for the Vector domain are now being optimized as if it were a primitive domain. The time for 50,000 vector elts has gone down from .680 seconds to .068. If vector exported a ***qelt*** function with no bounds or run-time type checking, then we would see another factor of 8 improvement. However for almost all Scratchpad II code, the cost of elting is now negligible.

4. Strings no longer use lisp ELTs to extract characters, eliminating the previous inefficient implementation of characters from Scratchpad II

5. *elt* and *setelt* from List are faster now, but it is now an error to call them with a negative or out of bounds index.

6. Both explicit and implicit segments are supported in REPEATs and COLLECTs, i.e., the following all work

```
s := 10..19
for i in 10..19 repeat output(i**2)
for i in s repeat output(i**2)

c1 := [i**2 for i in 10..19]
c2 := [i**2 for i in s]
c1 =$L(I) c2   -- true
```

7. In the ongoing struggle to improve the on-line documentation, we have started to completely rewrite the system commands and their options. For example, `)help display` will show the latest information about the `)display` command. We added a new flag to allow you to see this information in fullscreen mode.

```
)set help fullscreen on
```

will place you in the standard CMS HELP facility looking at the **DISPLAY HELPSPAD** file.

8. In cleaning up the system files aspect of things, we have removed calls to NORMEDIT and EDITANY from the source code. There is now only one exec that is used to edit files and it is called SPADEDIT. For most people it just calls XEDIT. You can copy this file to your A disk and customize it all you want, for example, if you want to call LEX, P-EDIT, STONEAGEEDIT or whatever.

Release: *SPAD3B: September 10, 1987*

---

1. The `)stats` command is dead. The functionality has been moved to an option on `)trace`.

   To remind you, the options `)count`, `)time` and `)storage` on `)trace` will cause statistics to be kept on execution counts, total execution time and storage usage within a function.

```
)trace )stats       -- will display these stats
)trace )stats reset -- will set them all to 0
```

2. The `)compile` system command has had 2 options renamed. The `)noscan` option is now called `)break` and the `)scan` option is called `)nobreak`.

3. The system commands `)read`, `)edit` and `)compile` are newly documented and can viewed with the new on-line help procedures we are phasing in. There are changes to the way they remember file names and what file types they accept.

4. The "Quick Guide to Programming in the Scratchpad II Interpreter" is now available on-line via the **SPADHELP** facility.

5. We have removed the `)equiv` system command and implemented symbolic input macro substitution via "= = >". The function that was intended for output with `)equiv` will be later implemented using a scheme that is more in keeping with abstract datatypes and code that actually works.

   Example:

```
%j ==> quatern(0,0,1,0)
```

   will cause the right hand side to be substituted for the left hand side in input parse forms. The result type of this form is Void. To remove a macro, redefine it to be itself:

```
%j ==> %j
```

   To view the macros that are defined, issue `)what macros`. As with `)equiv`, there are 3 predefined macros: `%i`, `%e` and `%pi`.

6. We put new versions of NSQFREE, MLIFT and MFACTOR on the system disk. These versions have some bugs fixed and a few improvements. Also availbale now: updated versions of PRODUCT, DIDEAL and IDECOMP.

7. We added a function ***irreducible?*** in DDFACT that tests if a given polynomial (with coefficients in a finite field) is irreducible. With this function a result for someone at the IBM Scientific Center in Rome that took something > 7 hours (because we had to factor the polynomial) now takes 0.395 seconds.

Release: *SPAD3D: October 7, 1987*

---

1. The system now retracts unions before modemap selection when data is available. This helps avoid long searches before a final retraction, as in

```
m : M I := [[1,2],[4,3]]
m * inverse m
```

When data is not available, the union cannot and is not removed. Not that if a final result is a union, it is not retracted. Basically, you should almost always get the same results, but much faster. In cases where you get different results, they will probably be what you expected anyway.

2. We have added a feature to the interpreter that allows it to try to choose default values for declared but unassigned variables. One problem with complicated towers of polynomial types is the correct placement of variables in the tower. For example,

```
p : P[x] RF I
p := x
```

the *x* should go into the P[x] part, not the RF I part. While the interpreter is generally pretty good at this (and would do the right thing above) there are times when it misses because the target information could not be reasonably pushed down far enough. By declaring your variables first, you help the interpreter and yourself by giving more information at a low level. Moreover, the interpreter will give the variable the "obvious" value when used. *N.B.:* if declare and then assign the variable, all bets are off on your subsequently being able to use the variable in a type declaration. This is not yet supported for subscripted variables. Here are some examples:

```
x : P[x] I

  Type: VOID

-- give x the obvious value and type

x

  (2) x

  Type: UP(x,I)

x + 1

  (3) x + 1

  Type: UP(x,I)

-- allow reuse of x in a type declaration

y : P[x] I

  Type: VOID

y  -- can't do anything here

  y is declared as being in UP(x,I) but has
  not been given a value.

)clear all

  All user variables and function definitions
  have been cleared.

z : P RN

  Type: VOID

-- should be Polynomial RationalNumber,
-- that is, the declared type is helping
-- out here.

z + 1

  (2) z + 1

  Type: P RN

)clear all

  All user variables and function definitions
  have been cleared.

-- an easy way to work with power series
x : UPS(x,RN)

  Type: VOID

x

  (2) x

  Type: UPS(x,RN)

1/(1 - x)

                  2   3   4   5   6
  (3) 1 + x + x + x + x + x + x

          7   8   9   10     11
       + x + x + x + x  + O(x  )

  Type: UPS(x,RN)
```

3. The definition of "-" in RadixExpansion has been corrected.

4. We fixed a problem whereby the history facility was confusing nullary functions and rewrite rules when saving and restoring workspaces. (October 19, 1987)

5. We fixed the display of function names when they appear inside type expressions. (October 20, 1987)

6. A message is displayed and the history facility is not enabled if the system is started with the user's A disk not being writable. (October 23, 1987)

Release: *SPAD3F: November, 1987*

1. We fixed a bug in the version of ***eval*** for multivariate polynomials that took a polynomial, a list of variables and a list of values as arguments.
2. In the interpreter, the expression { } is now parsed as `brace()` instead of `brace []`.
3. We enhanced the coercion facility to handle coercions among more types of linear aggregates. For example, it is now possible to coerce from Vector Integer to HashSet Polynomial RationalNumber.
4. In the compiler, explicit coercions from types to subtypes will be supported with run time checking, i.e.

```
x: I := foo()
x :: NNI
```

This fix should allow us to remove most uses of ":" as "pretend" from Scratchpad II code.

5. The NthRoot package has been replaced with the IntegerRoots package. The old ***root*** function has been renamed ***approxRoot*** to make it clear that it is computing an approximation. We have also included the following:

```
square?: I -> Boolean
power?: (I,NNI) -> Boolean
```

which return true iff the argument is a perfect square, and an nth power, respectively,

```
sqrt: I -> Union(I,"failed")
root: (I,NNI) -> Union(I,"failed")
```

which return the square root if the argument is a perfect square, and the nth root if the argument is an nth power, respectively,

```
approxSqrt: I -> I
approxRoot: (I,NNI) -> I
```

which returns 0 for a negative argument x, otherwise s such that $-1 < s - \sqrt{x} < 1$ and $-1 < s - \sqrt[n]{x} < 1$, respectively. If n is the length of the number, ***approxRoot*** has a running time of $\log(n)\, n^2$. ***approxSqrt*** uses a variable precision Newton iteration, hence it's running time is n^2. Also included is

```
root: I -> Record(base:I,exponent:NNI)
```

which computes the largest root of its argument, If n is the length of the input, the running time of ***root*** is $\log(n)^2\, n^2$.

6. After dividing out small factors (< 10000) integer factorization now tests to see the number is a perfect power before trying the Pollard method. This is a cheap heuristic and can be very effective if it succeeds. Integer factorization of numbers with lots of small factors (factorials for example) has been improved. For example, the time to factor 200! was reduced from 6.2 to .08 seconds on an IBM 3081.
7.)set break query is now the default.
8. With the above setting for break, you are displayed a menu that asks if you want to continue, return to top level or enter a lisp break loop. This menu now accepts 1 letter abbreviations for the choices and will only accept a valid choice (rather than just continuing after and invalid choice).
9. At top level, a semicolon ending an expression to suppress output will not figure into the evaluation time. Depending on the machine on which you are running the system, this time was .003 to .006 seconds.

Current and Recent Visitors to the Computer Algebra Group

Manuel Bronstein
Ph. D. recipient (May, 1987) from the Department of Mathematics, University of California, Berkeley, California. Algebra: 9/87-8/88.

Claire Di Crescenzo
Lab. TIM3 / INPG, 46 Av. Felix VIALLET, F-38031 Grenoble CEDEX, France. Algebra: 10/87-11/87.

Dominique Duval
Institut Fourier, Universite de Grenoble 1, BP74, 38402 Saint-Martin-D'Heres CEDEX, France. Algebra: 10/87-11/87.

Marc Gaetano
Laboratoire d'Informatique, Universite de Nice, Parc Valrose, 06034 Nice, France. Compiler: 11/86-11/87.

Rüdiger Gebauer
Springer-Verlag, 175 Fifth Avenue, New York, New York 10010. Interface: 12/85-1/87.

Patrizia Gianni
Dipartimento di Matematica, Universita Di Pisa, Pisa, Italy. Algebra: 1/87, 7/87-9/87, 11/87-11/88.

Klaus Kusche
Johannes Kepler Universität, Institut für Mathematik, A-4040 Linz, Austria. Algebra: 4/87.

Bernhard Kutzler
Johannes Kepler Universität, Institut für Mathematik, A-4040 Linz, Austria. Algebra: 4/87.

Martin Hassner
Signal Processing and Coding Group, Storage Systems and Technology Department, IBM Almaden Research Center, San Jose, CA. Algebra: 1/87-3/87.

Mohamed Mobarak
Undergraduate student, Department of Computer Science, Cornell University, Ithaca, New York. System: 6/87-8/87.

Michael B. Monagan
Formerly a member of the Symbolic Computation Group, University of Waterloo Waterloo, Ontario Canada. Algebra: 4/87-4/88.

Gerhard Schneider
Fachbereich Mathematik, Universität Essen, Universitätsstr. 3, 4300 Essen 1, West Germany. Algebra: 1/87-2/87.

Fritz Schwarz
GMD, Institut F1, Postfach 1240, West Germany. in 1987. Algebra: 1/87-6/87.

Moss E. Sweedler
Department of Mathematics, Cornell University, Ithaca, New York. Interface: 6/87-8/87.

Recent Publications by Computer Algebra Group Members

Gonnet, G. H., and Monagan, M. B., "Solving systems of Algebraic Equations or the Interface between Software and Mathematics," Proceedings of the conference *Computers and Mathematics,* Stanford, California, 1986.

Jenks, R. D., Sutor, R. S., and Watt, S. M., "Scratchpad II: An Abstract Datatype System for Mathematical Computation," *IBM Research Report* RC 12327 (Yorktown Heights, New York: November 17, 1986).

Della Dora, J., and Watt, S. M., *Factorisation d'Opérateurs Différentiels du Second Ordre,* Institut National Polytechnique de Grenoble TIM3, research report RR 634 -M-, November 1986.

Watt, S. M., and Jenks, R. D., "Abstract Datatypes, Multiple Views and Multiple Inheritance in Scratchpad II," *Proceedings of the Programming Language Technology ITL Conference,* Februrary 10-12, 1987, Tucson Arizona, IBM Corporation.

Sutor, R. S., and Jenks, R. D., "The Type Inference and Coercion Facilities in the Scratchpad II Interpreter," Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques, SIGPLAN Notices 22, 7, pp. 56-63, New York: Association for Computing Machinery, July 1987, and *IBM Research Report* RC 12595 (Yorktown Heights, New York: March 19, 1987).

Schwarz, F., "Programming with Abstract Data Types: The Symmetry Package SPDE in Scratchpad," *Proceedings of the Bad Neuenahr Conference on Trends in Computer Algebra,* May 1987, to appear (Springer-Verlag).

Bronstein, M., "An Algorithm for the Integration of Elementary Functions," *Proceedings of the 1987 European Conference on Computer Algebra (EUROCAL 87),,* June 2-5, 1987, Leipzig German Democratic Republic, to appear (Springer-Verlag).

Burge, W., and Watt, S. M., "Infinite Structures in Scratchpad II," *Proceedings of the 1987 European Conference on Computer Algebra (EUROCAL 87),,* June 2-5, 1987, Leipzig German Democratic Republic, to appear (Springer Verlag).

Jenks, R. D., and Sutor, R. S., "On the Design of The Scratchpad II Interpreter," *Proceedings of the 1987 European Conference on Computer Algebra (EUROCAL 87),,* June 2-5, 1987, Leipzig German Democratic Republic, to appear (Springer Verlag).

Gianni, P., "Properties of Gröbner Bases under Specializations," *Proceedings of the 1987*

European Conference on Computer Algebra (EUROCAL 87),, June 2-5, 1987, Leipzig German Democratic Republic, to appear (Springer Verlag).

Sutor, R. S., "Scratchpad II: A Language and System for Computer Algebra," *Proceedings of IBM Hochschulkongress '87,* July 7 - 9. Munich, West Germany: IBM Deutschland Gmbh.

Moore, R. A., Scott, T. C., and Monagan, M. B., "A Classical many-particle relativistic Lagrangian," *Physical Review Letters,* 1987.

Granville, A. J., and Monagan, M. B., "The First case of Fermat's Last Theorem for all Prime Exponents up to 714,591,416,091,389," *Transactions of the American Mathematical Society,* 1987.

Gianni, P., Trager, B., and Zacharias, G., "Gröbner Bases and Primary Decomposition of Polynomial Ideals," to appear in *Journal of Symbolic Computation.*

Conference Announcements

Symposium on Computer Algebraic Integration & Solution of Differential Equations

This conference will be held at the T.J. Watson Research Center, Yorktown Heights, New York, on November 19-20, 1987. The co-chairmen for the conference are P. Dean Gerber and Richard Jenks, Mathematical Sciences Department, IBM Research Division. Proceedings of papers on the symbolic solution of differential equations will be submitted to Springer-Verlag for publication. Invited speakers and the titles of their talks are:

Carl Andersen (William & Mary), "Investigating a Hybrid Perturbation Galerkin Solution Technique"

Dieter Armbruster (Cornell), "Averaging for non-autonomous ODEs and Bifurcations in PDEs"

Manuel Bronstein (IBM Research), "Symbolic Integration: Algebraic and Mixed Functions Cases"

David V. Chudnovsky and Gregory V. Chudnovsky (Columbia), "Computer Experiments in Geometry and Arithmetic of Differential Equations"

Dominique Duval (Grenoble), "Local Solutions of ODEs and Algebraic Numbers"

P. Dean Gerber (IBM Research), "Symbolic Solution of Quadratic Ordinary Differential Equations"

Niky Kamran (Institute for Advanced Study), "ODEs, Cartan's Method of Equivalence, and Rat. Curves in Complex Surfaces"

Micheline Musette (Brussels), "Use of MACSYMA to Check Existence of Multisoliton Solutions of Evolution Equations"

Ken Myer and Dieter Schmidt (Cincinnati), "The Motion of the Moon"

Richard H. Rand (Cornell), "Perturbation and Bifurcation Calculations in MACSYMA"

Fritz Schwarz (GMD, Bonn), "A Factorization Algorithm for Linear Ordinary Differential Equations"

Michael F. Singer (North Carolina State), "Closed Form Solution of Linear Differential Equations"

Barry M. Trager (IBM Research), "Symbolic Integration of Elementary Functions: History and Perspective"

Richard Zippel (Symbolics, Inc.), "Qualitative Approaches to Ordinary Differential Equations"

For further information on the symposium, please contact Sandra Wityak, Conference Secretary, IBM Research, P.O. Box 218, Yorktown Heights, NY 10598, 914/945-1187

Computers & Mathematics II

Preliminary Announcement
June 12-16, 1989
Kresge Auditorium, MIT

Organizing Committee:
D. Chudnovsky (Columbia), R. Jenks (IBM), co-chairmen; A. Nerode (Cornell), P. Wang (Kent State), J. McCarthy (Stanford), G. Chudnovsky (Columbia), J. Moses (MIT)
In cooperation with: ACM SIGSAM, SAME, ACM/GBC

Local Arrangements:
Ellen Golden (Symbolics, Inc.)

Contributed Program Chairman:
Erich Kaltofen (RPI)

Format:
28 invited speakers + contributed papers

Exhibits:
Publishers
Computer hardware/software, by invitation

Publisher:
Springer-Verlag (Proceedings on sale at meeting)

Housing:
MIT dormitories/local hotels

Tentative Program

Monday, June 12
Morning: Invited Session 1
Ramanujan: The 2nd 100 Years
Afternoon: Contributed Papers
Evening: Math Software Presentations

Tuesday, June 13
Morning: Invited Session 2
Computers and Mathematical Physics
Afternoon: Invited Session 3
Computers as a Res. Tool in Math
Evening: Computer Graphics Program

Wednesday, June 14
Morning: Invited Session 4
Mathematics & Computer Graphics
Afternoon: Contributed Papers
Evening: Wine and Demo Buffet

Thursday, June 15
Morning: Invited Session 5
Computer Algebra & Mathematics
Afternoon: Contributed Papers
Evening: Computers & Music Program

Friday, June 16
Morning: Invited Session 6
Mathematics & Supercomputing
Afternoon: Invited Session 7
Mathematics & Combinatorics

Some Scratchpad II Constructor Name Abbreviations

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

Abbreviation	*Constructor Name*
BF	BigFloat
FLOAT	ComplexFloat
DDFACT	DistinctDegreeFactorize
DIDEAL	IdealDomain
DMP	DistributedMultivariatePolynomial
EF	ElementaryFunctions
FF	FactoredForm
FFX	FiniteFieldExtension
FR	FactoredRing
G	Gaussian
GF	GaloisField
I	Integer
IDECOMP	IdealDecompositionPackage
IV	IndexedVarlist
L	List
M	Matrix
MLIFT	Mlifting
MP	MultivariatePolynomial
MSET	Multiset
NNI	NonNegativeInteger
NSQFREE	NSquareFree
OV	OrderedVarlist
P	Polynomial
PRODUCT	Product
Q	Quaternion
QF	QuotientField
RF	RationalFunction
RM	RectangularMatrix
RN	RationalNumber
S	String
SAE	SimpleAlgebraicExtension
SE	SortedExpressions
SM	SquareMatrix
SPF	SpecialFunctions
SUP	SparseUnivariatePolynomial
UP	UnivariatePoly
UPS	UnivariatePowerSeries
VOID	Void

At last count, the Scratchpad II library consisted of 69 category constructors, 170 domain constructors and 217 package constructors.

IBM

Scratchpad II Interactive Interface and Interpreter Reference Summary

Draft Edition 0.12 (September 22, 1987)

This reference summary contains information about using the **Scratchpad II** Interactive Interface and Interpreter as it existed on November 13th, 1987 under the IBM VM/CMS operating system. Since **Scratchpad II** is a system in development, it is possible that later editions of this card exist and reflect changes made to the system.

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 implementers.

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 form has been produced by the

Computer Algebra Group, Mathematical Sciences Department
IBM Research Division
Thomas J. Watson Research Center
Box 218
Yorktown Heights, NY 10598 USA

Questions should be sent to the above address, care of Robert S. Sutor.

Contents

Syntax Notation for this Summary

...	indicates that the item preceding this symbol may be repeated an arbitrary number of times.
< >	encloses items which are optional
\	separates alternatives
`Example`	in this summary, text in a monospace font represents examples of possible input to the interactive interface.

Preliminary Notes

1. Enter the **Scratchpad II** interactive interface by issuing SPAD2 to CMS.
2. **Scratchpad II** identifiers are case-sensitive: "firstvar" and "firstVar" are different identifiers. To avoid superseding constructor names and abbreviations, it is a good policy to begin identifier names with lowercase letters.
3. To suppress output from a statement, end the line with a semicolon (";").
4. **Scratchpad II** comments start with "--" (two hyphens) and continue until the end of the line.
5. Statements may be collected together into files with filetype INPUT and read sequentially through the interactive interface by using the system command **)read** (see page 9).
6. If typed directly to the interactive interface, statements can only be one line long unless they are terminated by the continuation character "_". The underscore character can also be used to *escape* that might otherwise not be parsed correctly.
7. To terminate an on-going computation, either press PF12 or press PA1, type *EXT* and then press the enter (return) key. If nothing seems to happen, press the enter key several more times. When you are prompted with a question about what you want to do, enter *top* to terminate your calculation and return to top level, or enter *continue* to resume your calculation.
8. In this summary, the reader should consider the terms "type" and "mode" equivalent to the term "datatype".
9. There is a noticeable delay the first time an object of a given type is used (for internal system reasons). Subsequent applications of the type will not exhibit this delay.

Common Constructor Abbreviations

Within the interactive interface, all constructor abbreviations and expansions may be displayed by using the **)what** system command (see page 10). A constructor abbreviation may be used anywhere in the interactive interface that a constructor name is expected (except in the **)abbreviate** system command).

A	**Any**	**NNI**	**NonNegativeInteger**
AR	**Array**	**P**	**Polynomial**
B	**Boolean**	**PI**	**PositiveInteger**
BF	**BigFloat**	**QF**	**QuotientField**
E	**Expression**	**RF**	**RationalFunction**
EF	**ElementaryFunction**	**RM**	**RectangularMatrix**
F	**Float**	**RN**	**RationalNumber**
G	**Gaussian**	**S**	**String**
GF	**GaloisField**	**SAE**	**SimpleAlgebraicExtension**
I	**Integer**	**SEG**	**Segment**
L	**List**	**SM**	**SquareMatrix**
LIB	**Library**	**TBL**	**Table**
M	**Matrix**	**UP**	**UnivariatePoly**
MP	**MultivariatePolynomial**	**V**	**Vector**

Some Alternate Constructor Forms

The **Scratchpad II** interactive interface allows the user to use any of several alternate forms for specifying matrix, polynomial and rational function domains. The following are examples:

Form	*Is Equivalent to*
M I	Matrix Integer
M[2] I	SM(2,I), SquareMatrix(2, Integer)
M[3,5] I	RM(3,5,I), RectangularMatrix(2, 3, Integer)
P I	Polynomial Integer
P[x] I	UP(x,I), UnivariatePoly(x, Integer)
P[x,y] I	MP([x,y],I), MultivariatePolynomial([x,y], Integer)
RF I	RationalFunction Integer
RF[x] I	QF P[x] I, QF UP(x,I), QuotientField UnivariatePoly(x, Integer)
RF[x,y,z] I	QF P[x,y,z] I, QF MP([x,y,z],I), QuotientField MultivariatePolynomial([x,y,z], Integer)

Common Constructions

integers These are written as consecutive digits (0 to 9), possibly preceded by a minus (-) sign. There is no limitation to the number of digits.

```
0
737373737
-84848735783674993769000023
```

bigfloats These must contain a decimal point, may have an optional prefix minus sign (-), may be followed by an "E" (or "e") and an integer, and have the number of digits set by the function ***precision***.

```
0.11111111111111111111
-1.2E78
3.4e
```

symbols Unless previously declared, an identifier typed to the interactive interface will be given the default type **Symbol**. This can, however, be easily coerced to a polynomial type having the identifier among its potential variables.

```
x
x :: P I
```

polynomials If an undeclared symbol is included in a sum or product with other symbols and coefficients, the resulting type of the expression will be **Polynomial Integer**. This can be made into another type of polynomial by coercion or assignment to a declared variables.

```
x + 1
(x+1)**2 :: P[x] I
p : P[y,w,z,x] RN := x + 3*y**3
```

lists Lists are enclosed in square brackets. If the types of the elements cannot be resolved to a single type, the resulting object has type **List Any**

```
[1,4,3,5]        -- type is L I
[2,5/6,"hello"] -- type is L Any
```

segments usually represent ranges of values. For example, a range of consecutive integers may be represented as a segment. It would be written as the lowest integer, followed by "..", followed by the highest integer.

```
1..100
f n == n*f(n-1) when n in 2..100
```

Unbounded segments are formed if the value after the ".." is omitted. As a user convenience, lists containing segments are automatically expanded to contain the elements in the segment ranges.

```
3..
[1..4,10..30]
[1,5,10,100..105,1000..]
```

records are produced by the domain constructor **Record** and have elements identified by *tags*. A record can be produced by a function or by defining the entire structure at time. The tags may be used to access or assign the individual parts of a record.

```
r : Record(quotient: I, remainder: I)
r := 63638778 div 45258
r.quotient
r.remainder
r := [8,5]  -- note use of brackets
r.quotient := 76
```

unions are created by the domain constructor **Union**. These are *discriminated* unions: an element of union is associated with one of the typed branches. (see the case statement on page 4).

```
u : Union(I,F,S)
u := "hi, there"
u case S
u := 5.6
u case I
-- next returns Union(I,"failed")
74748 exquo 57
```

These are useful if a computation may return objects of several different types.

Functions and Operators

Use the **)show** system command (details on page 10) to display all functions and operators for a given domain, category or package. Use the *attributes* (or abbreviation thereof) option to see the attributes.

```
)show I
)sh RN
)sh P[x]
)sh L M RF FN )att
```

and, or and **not** are **Boolean** operators supplied by the interactive interface These will soon be provided by **Boolean** itself.

Almost all **Scratchpad II** functions are prefix functions. The few exceptions include the following infix operators

+	*	**	-	/
div	exquo	in	quo	rem

Parentheses may be used to control order of evaluation. In addition, they are required for prefix functions of more than one argument and optional for single argument functions (though they may be needed to override the default operator precedence rules).

```
6 div 3
factor 98939
gcd(77373, 45242)
1 + x**2
(1 + x)**2
```

Statements

assignment :=
assigns a value to a variable

```
variable := expression
```

case
this **Boolean**-valued infix operator is used to determine the branch in which a value of a union lies.

```
i := 6773637 exquo 77266
if i case "failed" then output "does not divide"
```

coercion ::
is used to change and object of one datatype into an object of another.

```
8 :: P[x,y] RN
[[1,2],[3,4]] :: M[2] P I
```

declaration :
declares one or more variables to have a given datatype. If multiple variables are declared in a single statement, they must be separated by commas and enclosed in parentheses.

```
i : Integer
(r,s): RN
```

exit =>
terminate the block with the value following the keyword **exit** or the symbols **=>**.

```
fib n ==
  n = 0 => 1
  n = 1 => 1
  fib(n-1) + fib(n-2)
```

for
is a loop statement employed to iterate over the elements of an aggregate or segment.

```
for i in 1..10 repeat sum := sum + i
for x in l repeat factors := factors * factor x
```

Also see: **leave, repeat, until, while.**

function signature declaration
is used to inform the interactive interface of the argument and result datatypes for a function. Only the datatypes are listed, and not the formal argument names.

```
fac: Integer -> Integer
expt: (Float, Integer) -> Integer
maxmin: L I -> Record(max: I, min: I)
```

function definition ==
is used to define a function.

```
sq(x) == x*x
fac100() == */[2..100]
addEm(x,y,z) == x + y + z
```

Definition may be combined with signature declaration.

```
maxmin(l : L I): Record(max : I, min: I) ==
  null l => [0,0]
  ["max"/l, "min"/l]
```

Also see: **function signature declaration.**

has
this **Boolean**-valued infix operator is used to test whether a domain has a specific operation, category or attribute.

```
I has factor: I -> I
Integer has EuclideanDomain
I has commutative("*")
```

is
performs pattern matching on lists, with possible assignment side-effects. Returns a **Boolean**.

```
l := [1..6]
l is [a,b]     -- false, l has 6 elements
l is [a,:b]    -- true
a              -- is 6
b              -- is [2,3,4,5,6]
l is [a,.,:b] -- true
b              -- is [3,4,5,6]
```

Also see: **isnt.**

isnt
performs pattern matching on lists, with possible assignment side-effects. Returns a **Boolean**.

```
l := [1..6]
l isnt [a,b]     -- true
l isnt [a,:b]    -- false
a                -- is 6
b                -- is [2,3,4,5,6]
l isnt [a,.,:b] -- false
b                -- is [3,4,5,6]
```

If *p* is a pattern as above, *l* **isnt** *p* is exactly the same as **not** (*l* **is** *p*).

Also see: **is.**

iterate
*** not yet implemented
transfers processing control to the iteration section of the immediately containing **repeat** loop.

Also see: **for, leave, repeat, until, while.**

leave
exits the immediately containing **repeat** loop.

```
sum := 0
for i in 1.. repeat
  sum := sum + i
  if sum > 1000 then leave
```

Also see: **for, iterate, repeat, until, while.**

repeat
general loop statement.

```
x := 1  -- initial value
repeat
  x := x + x**2
  if x > 1000 then return x
```

Also see: **for, leave, until, while.**

return
exits a function with the value following the **return** keyword.

```
return (x+1)**n
```

suchthat |
can be used as a filter in iteration or in the definition of an object in an algebraic extension.

```
for j in 1..10 | oddp j repeat output j
i | i**2 + 1
```

Also see: **for, repeat.**

until
conditional test for termination of a loop *after* the loop body has been executed.

```
x := 1  -- initial value
until x > 1000 repeat x := x + x ** 2
```

Also see: **for, leave, repeat, while.**

while
conditional test for (possible continued) execution of a loop *before* the loop body has been executed.

```
x := 1  -- initial value
while x <= 1000 repeat x := x + x ** 2
```

Also see: **for, leave, repeat, while.**

Reserved Words

The following words should be considered reserved and not used as variable or function names. Some of the words represent infix operators or domain constructors, but most are terms used in the **Scratchpad II** programming language.

add	exit	leave	then
and	exquo	not	Union
by	has	or	until
capsule	if	otherwise	when
case	in	quo	while
Category	is	Record	with
constant	isnt	rem	
div	iterate	repeat	
else	Join	return	

Special Functions

% Is the value returned by the previous computation.

```
determinant m
factor %   -- factors the determinant of m
```

%%(*n*) Is the value returned by step *n* or, if *n* is negative, the value returned |*n*| steps ago. Thus %%(-1) = %.

```
2**10
3**737
%%(-2)  -- returns result of 2**10
%%(-2)  -- now returns result of 3**737
```


Commonly Used Commands

The following lists some commonly used system commands and synonyms for system commands. The syntax for some commands is abridged.

)? display all system commands.

)clear all clear all variable and rule definitions and values (produces a completely empty workspace).

)cms *command* passes *command* to CMS for execution.

)disc disconnect from the CMS session, but remain in **Scratchpad II**. When you reconnect to CMS, you will still be in the interactive interface.

)edit *fn* <*ft* <*fm*>> Enters the system editor on the file listed. *ft* defaults to INPUT and *fm* defaults to A.

)fortran on\off turn FORTRAN-style output display on and off.

)logoff terminate both the **Scratchpad II** and CMS session.

)quit exit the **Scratchpad II** interactive interface and return to CMS.

)read *fn* read the file *fn* INPUT A through the interactive interface a line at a time.

)show *constr* show the operations present in the constructor *constr*.

)storage on\off turn display of storage use statistics on and off.

)time on\off turn computation time display on and off.

)type on\off turn computation result type display on and off.

)wd display all domain constructors and their abbreviations. This is a synonym for **)what domains**.

System Commands: Complete List

The following is an abbreviated description of all system commands. Some system commands can only be used at certain user levels (which can be assigned and queried by **)set userlevel**). The command **)?** displays all commands available at the currently set level.

System command names may be abbreviated by typing enough letters to make the name unique. Thus **)l** is ambiguous, while **)lo** is uniquely expanded to be **)load** (as opposed to, say, **)lisp**).

)abbreviations category\domain\package\remove *abbrev fullname*
Assigns or removes abbreviations for constructors. For example, to assign an abbreviation of **MD** to the new domain **MyDomain**, issue
)abbrev domain *MD MyDomain*

)boot <*bootmsg*>
hands *bootmsg* to the BOOT parser for translation into and execution by **LISP/VM**. If *bootmsg* is not given on the line with the command, it must be given on subsequent input lines.

)clear all\property\mode\value\rule *ident* **...**
If *all* is specified, the entire workspace is cleared. Otherwise, for each specified identifier *ident*, the specified associated property is cleared. *property* indicates that everything should be cleared for each *ident*.

```
)cl p x y -- clear all info about x and y
)cl v all -- clear all values
)cl m foo -- clear declared mode for foo
```

)cms <*command*>
If present, *command* is passed to the CMS operating system for execution. If no argument is given, CMS subset mode is entered. You can return to the interactive interface by issuing RETURN while in CMS subset mode.

)compile <*fn* <**SPAD** <*fm*>>> <**)constructor c**> <**)functions** *fun* **...**> <**)new**>
invokes the **Scratchpad II** language compiler on one or more constructors in a file with filename *fn*, filetype SPAD and filemode *fm*. If the **)new** option is given, the new **Scratchpad II** compiler is used. If the **)constructor** option is given, only that constructor in the file is compiled. If the **)functions** option is present, only those functions listed that are in the given file or constructor are compiled. The *fn* is remembered from the last **)edit** or *compiler* command.

)cp <*command*>
If present, *command* is passed to the CP command processor for execution. If no argument is given, you are placed in CP. You can return to the interactive interface by issuing B while in CP.

)display property\mode\value\rule *ident* **...**
displays the given information about each *ident* listed. If no *ident* is present, the information about all user-defined identifiers in the workspace is displayed. Use *property* to see value, rule and mode information.

```
)d p      -- see info of all identifiers
)d v x y  -- see values of x and y
)d m      -- see modes of everything
)d r bar  -- see bar rule info
```

)display ancestors\dependents\operation &Z.ident.
shows constructor ancestors or dependents or categorical modemap data about an operation.

```
)d a I    -- see immediate constructors
          -- on which Integer depends
)d d I    -- see immediate constructors
          -- which depend on Integer
)d op sin -- see general information
          -- about all sin operations
```

)edit <*fn* <*ft* <*fm*>>>
where *fn ft fm* is the name of a CMS file to be edited. If not given, *ft* defaults to INPUT and *fm* defaults to A. If *fn* is not given, the last file read or edited is read. If a new file is requested, you must respond to the prompt regarding whether you want to create the file, quit or specify a different file.

)fin will exit **Scratchpad II** and place you in the **LISP/VM** interpreter. To return to **Scratchpad II**, issue the function call (spad).

)frame names \ next \ last \ add *name* **\ drop** <*name*>
is used to manipulate interpreter frames, which can be viewed as distinct logical **Scratchpad II** sessions within one physical session.

)help *<command>*

displays information about the command *command*, if specified. Otherwise, the fullscreen **Scratchpad II** documentation and news facility is entered.

)history)on\off\restore\save\show\write *<opts>*

controls the session history facility. By default, it is on, though **)history)off** terminates the saving of definitions and computations. To return to the state of the previous session, issue **)history)restore**. To write an INPUT file on your A disk containing all user input since the beginning of the session or since the last **)clear all**, issue **)history)write**.

To display previous inputs and outputs, use the **)history)show** command. **)history)show** by itself will display up to 20 of the last input lines (fewer if you haven't typed in 20 lines). **)history)show** *n* will display up to *n* of the last input lines. **)history)show both** will display up to 5 of the last input and output lines. **)history)show** *n* **both** will display up to *n* of the last input and output lines.

)lisp *<lispmsg>*

hands *lispmsg* to the **LISP/VM** interpreter to be processed. If *lispmsg* is not given on the line with the command, it must be given on subsequent input lines.

)load fn ... <)cond> <)update>

This is used to manually load compiled **Scratchpad II** code into the interactive interface. This is usually necessary only for new constructors that were not previously known to the system or old constructors that were preloaded and then changed. *fn* is the abbreviation for a constructor name. The **)cond** option implies that the constructor *fn* should only be loaded if it is already loaded. The **)update** option forces updating of the internal database of constructor information. This is not necessary if the code is being loaded from a non-system disk.

)pquit

This is used to exit the **Scratchpad II** interactive interface and return to CMS. You will be prompted to confirm your intention of exiting the workspace (if you do, all data in the workspace will be lost).

)quit

This is used to exit the **Scratchpad II** interactive interface and return to CMS. Depending on the value of an environment variable (queried and assigned by **)set quit**) you may be prompted to confirm your intention of exiting the workspace (if you do, all data in the workspace will be lost).

)read *<fn <ft <fm>>>*

where *fn* is the CMS filename of a file containing **Scratchpad II** statements to be executed by the interactive interface. If not given, *ft* defaults to INPUT and *fm* defaults to A. If *fn* is not given, the last file read or edited is read.

)set *<var ... <value>>*

is used to control the workspace environment. Issue **)set ?** for more information.

```
)set          -- display all top-level variables
)set message -- display all message variables
-- display current message-time setting
)set message time
-- turn on message-time printing
)set message time on
```


)show *constr <)attr>*

shows operations or attributes for a category, domain or package constructor. By default, operations are shown, but attributes will be displayed if the **)attr** option is given.

```
)sh List
)sh List I
)sh RF I )att
)sh Set
)sh LISTPKG2
```

)suspend

This is used to temporarily suspend **Scratchpad II** and return to CMS. You may be able to return to **Scratchpad II** by issuing SPAD2 to CMS.

)synonym *<syn fullexpression>*

If no arguments are given, this command will print a list of all defined synonyms for system commands. Otherwise, this defines *syn* as a synonym for *fullexpression*.

```
)synonym                -- displays all synonyms
)syn ws what synonyms -- defines )ws to be = to
                        -- )what synonyms
```

)trace *function \ constructor ... <)option ...>*

traces the execution of one or more given functions or all exported functions from a constructor. The following is an abridged listing of **)trace** options.

)count	counts the number of times a function is called. Use the **)stats** option to **)trace** to list or reset the counts.
)math	displays function arguments and output in a mathematical format, where appropriate.
)off	turn off tracing
)stats	display or reset count, timer or storage statistics. Add the argument *reset* to reset them, omit it to list them.
)storage	shows storage use during exection of function. Use the **)stats** option to **)trace** to list or reset the counts.
)timer	times the total amount of time spent in a function. Use the **)stats** option to **)trace** to list or reset the counts. **)stats** system command.
)vars	display the values of variables in the function as they are assigned.

)undo *n*

This will undo the last *n* computations. If not given, *n* defaults to 1. Certain destructive operations cannot be undone, nor can the action of a **)clear all**.

See also: **)history**

)what *things <pattern ...>*

where *things* is one of *categories*, *domains*, *packages*, *commands*, *synonyms* and *pattern* is a string of characters. This command lists all *things* containing any of the *pattern*.

```
)what domains
)what commands ed read
```

See also: **)display,)show**

)workfiles <)boot *fn ...> <)lisp fn ...>*

is used to identify the filenames of LISP and BOOT files in which to find functions to be edited or compiled. This is generally only useful (or needed) for newly written functions.

User Notes

The following space is reserved for your notes when you use the **Scratchpad II** interactive interface. Please send copies of notes of general interest to the person and address listed on the first page of this summary.