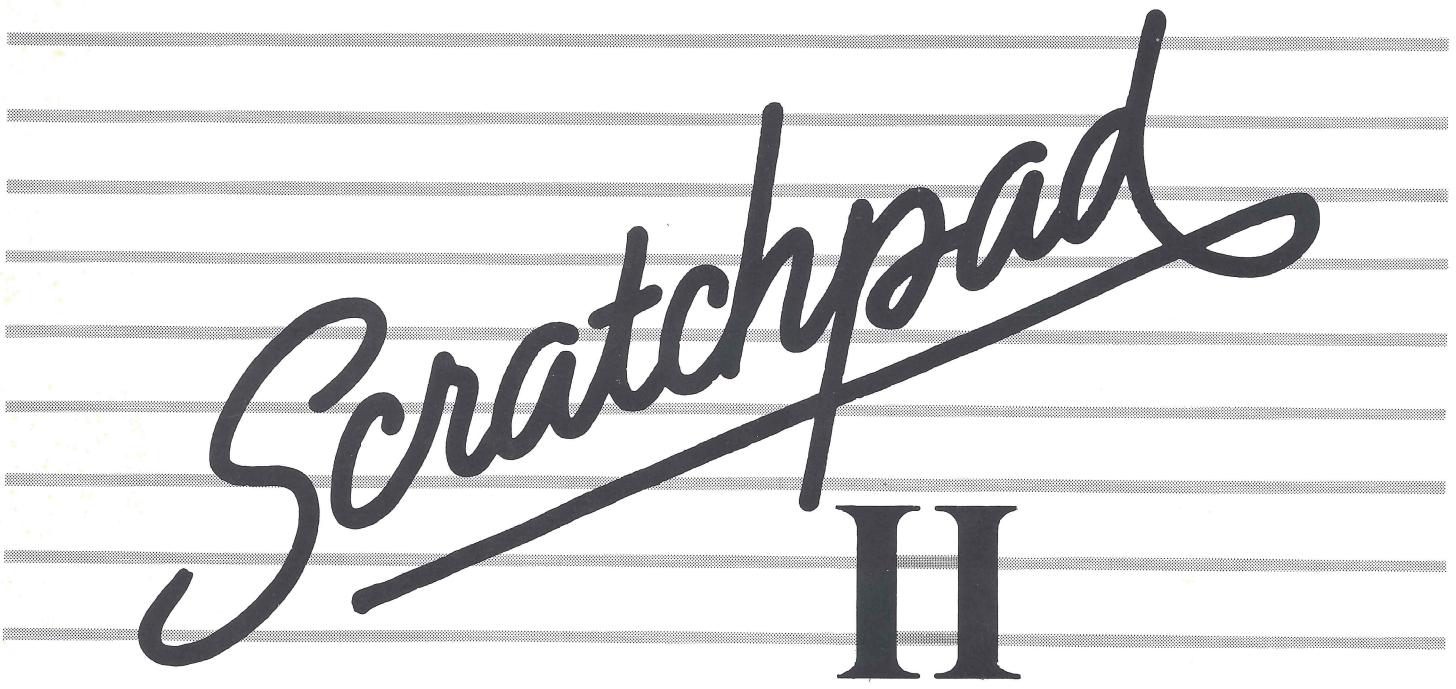

IBM Research

Scratchpad II
An Experimental Computer Algebra System

Stephen R. Balzac
James H. Davenport
Patrizia Gianni
Richard D. Jenks
Victor S. Miller
Scott C. Morrison
Michael Rothstein
Christine J. Sundaresan
Robert S. Sutor
Barry M. Trager

Mathematical Sciences Department
IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598

Scratchpad II Abbreviated Primer and Examples

Edition date: 06/28/84 Print date: 07/18/84

For more information, please contact:

Richard D. Jenks
IBM Thomas J. Watson Research Center
Box 218
Yorktown Heights, NY 10598

CONTENTS

Welcome to Scratchpad II 1

11 Keys to Scratchpad II 3
Key #1: Workspace 4
Key #2: Expressions 5
Key #3: Forms 6
Key #4: Rules 6
Key #5: Maps 7
Key #6: Quote Marks 8
Key #7: Assignments 8
Key #8: Modes 9
Key #9: Categories 10
Key #10: Domains 12
Key #11: Packages 16

Appendix 1: Programming Language Syntax 18

Appendix 2: Domain Table 19

Appendix 3: Algebraic Categories 20

Appendix 4: Scratchpad II Operators 21

Appendix 5: Scratchpad II System Commands 23

Appendix 6: The Scratchpad II Database:)what and has 26

References 29

Examples. 30
Example 1: Big Floats 30
Example 2: Matrix Computations 32
Example 3: Integer Factorization 37
Example 4: Polynomial Factorization 38
Example 5: Factored Form Rings 40
Example 6: Simple Algebraic Extensions 43
Example 7: Grobner Basis Computations 45
Example 8: Complex Root Package 48
Example 9: Rational Function Integration 50

Welcome to Scratchpad II

This document contains an abbreviated primer for and examples of the language of **Scratchpad II**, a new implementation of **SCRATCHPAD** which has been under design and development by the Computer Algebra Group at the IBM Research Center during the past 6 years. In addition, appendices contain details of various parts of the language including syntax, operators, system commands and lists of basic *domains* and *categories*.

The basic design goals of the **Scratchpad II** language and interface to the user are:

- to provide a "type-less" interactive language suitable for on-line solution of mathematical problems by novice users with little or no programming required;
- a programming language suitable for the formal description of algorithms and algebraic structures which can be compiled into run-time efficient object code.

Scratchpad II is a language with parameterized types and generic operators particularly suited to computational algebra. A flexible framework is given for building algebraic structures and defining algorithms which work in as general a setting as possible.

The **Scratchpad II** language for computational algebra is able to express algorithms for dealing with algebraic objects in their most natural level of abstraction. We can illustrate this concept with two simple algorithms. First, we wish to write a function "max" which computes the maximum of two elements of any set on which an ordering predicate is defined. One approach to this problem is to make the ordering predicate an additional parameter to max. Thus max might be defined by:

```
max(x,y,lessThan)== if lessThan(x,y) then y else x
```

In more complicated algorithms, the number of additional parameters required gets out of hand. Our approach is instead to require the parameters x and y of max to be elements of some specific algebraic structure which has a "less than" operation '<' implemented by some function. We will call such an algebraic structure a *domain*. An inspection of this algorithm reveals that it really only depends on certain abstract properties of "lessThan": that it is a total order. In order not to repeat the same definition for every domain which has a total order, we introduce the concept of *category* which provides an abstraction of properties of domains. Any domain which satisfies the abstract properties given by a category is said to *belong* to the category. In this context '<' is a *generic* operation which may have different function definitions for different domains. Our definition of max, with suitable declarations, becomes:

```
max(x,y) == if x < y then y else x
```

The requirement that a generic operation have a particular name does not characterize its algebraic properties. In order to assert that a domain is a totally ordered set with respect to '<', it may have *attributes* which permit a description of the algebraic properties of its operations, e.g.

```
total("<")
```

Attributes may, for example, be used to distinguish between a totally ordered and a partially ordered set.

In a second example, we examine the classical algorithm for computing the "gcd," the greatest common divisor, of two integers:

```
gcd(x,y) == if x=0 then y else gcd(y, remainder(x,y))
```

Although this algorithm was originally intended to be used only on integers, a cursory examination shows that only a few properties of the integers are actually required. In fact, the same algorithm can be used on gaussian integers, univariate polynomials over fields, or any other domain which has an appropriate remainder function. We may specify the minimum requirements of an algebraic structure that the gcd algorithm to be applicable. As above, we may introduce a category. In this case it is "the category of Euclidean domains." Any domain of this category will be an integral domain with a generic function "remainder" satisfying two requirements. The first is that

$$\text{remainder}(x,y) = x - q*y$$

for some q in the domain. The second is that remainder sequence generated by any two elements of the domain always reaches 0 in a finite number of steps. These two requirements are sufficient to guarantee the correctness of our gcd algorithm.

Categories provide a set of required generic operations together with a set of attributes describing the required algebraic properties of these operations. Domains provide specific functions which implement the operations and satisfy the attributes. Thus we may speak of "the category of totally-ordered sets" as the class of all domains which have the above '<' operation with specific algebraic properties, and the "the domain of the integers" as an example of a member of that category since it has a function implementing '<' which provides that operation and satisfies those properties.

A domain or a category is a computed object that can be assigned to a variable, passed as a parameter, and returned from functions. A category may be produced by explicitly listing operations and attributes, evaluating a variable, or by invoking a function which returns a category. Categories may be augmented, or *joined* with other categories to produce a new category containing all of the operations and attributes of the individual categories.

To summarize, the **Scratchpad II** language design provides the useful notions of *domains* and *categories* for the abstract description of algorithms for computational algebra. The facility for categories is unique to our language and its use seems to be invaluable in describing algorithms for computational algebra.

11 Keys to Scratchpad II

In the following primer, the **Scratchpad II** language is introduced by 11 keys with each successive key introducing a additional capability of the language. The language is thus described as a "concentric" language with each of the 11 levels corresponding to a language subset. These levels are more than just a pedagogic device, since they correspond to levels at which the system can be effectively used. Level 1 is sufficient for naive interactive use; levels 2-8 progressively introduce interactive users to capabilities of the language; levels 9-11 are for system programmers and advanced users. Levels 2, 4, 6, and 7 give users the full power of **LISP** with a high-level language; level 8 introduces "type declarations;" level 9 allows polymorphic functions to be defined and compiled; levels 10-11 give users an Ada-like facility for defining types and packages (those of **Scratchpad II** are dynamically constructible, however). One language is used for both interactive and system programming language use, although several freedoms such as abbreviation and optional type-declarations allowed at top-level are not permitted in system code. The interactive language (levels 1-8) is a blend of original **SCRATCHPAD** [GRJY75], some proposed extensions [JENK74], work by Loos [LOOS74], SETL [DEWA79], SMP [COWO81], and new ideas; the system programming language (levels 1-11) superficially resembles **Ada** but is more similar to CLU [LISK74] in its semantic design.

The presentation of the language in this primer omits many details to be covered in the **Scratchpad II** System Programming Manual [SCRA84] and an expanded version of this primer which will serve as a primer for **Scratchpad II** users [JESU84]. In addition, the description of the 11 keys describes the interactive language somewhat ahead of its current implementation. On most terminals the characters → and ← will be typed as "==" and ":=" respectively.

Notation and Vocabulary. The basic vocabulary of **Scratchpad II** consists of identifiers, reserved words, numbers, strings, and operator symbols. Identifiers are used to name constants, variables, and functions. They must begin with a letter but may be followed by letters (upper/lower case is distinguished) or digits, e.g. "a3b4" and "a3B4" are distinct identifiers. Two reserved words "true" and "false" are boolean constants, others (e.g. if and then) are operators ("Appendix 4: Scratchpad II Operators" on page 21), still others are names of functions provided in the **Scratchpad II** library. Numbers are of two basic kinds: integers and floats. Integers are written as a sequence of digits. Floats contain a decimal point and/or a scale factor, e.g. 12.34E-12. Many other kinds of numbers are available but must be referenced by parameterized forms (e.g. sqrt(2)), special symbols (e.g. %i), or coercion expressions (e.g. 2@BF to create a BigFloat (floats of arbitrary precision)). Strings are sequences of any printable characters enclosed with double quotes, e.g. "a string". Operation symbols (e.g. +,*) are groups of special characters used as operator symbols in the language ("Appendix 4: Scratchpad II Operators" on page 21) When operations are adjacent, grouping is governed by the left- and right-precedence numbers as described in [SCRA84], e.g. "2↑3+4*5" means "((2↑3)+(4*5)". Parentheses are used solely for grouping and have no other significance. Curly brackets {} can be used anywhere in a **Scratchpad II** program to write comments.

Key #1: Workspace

Scratchpad II can be used as a symbolic desk calculator. You simply enter expressions at your terminal as commands to **Scratchpad II**. **Scratchpad II** will interpret your expressions, do the computation, and output the result. Every expression you enter and every result you create is saved in a **workspace** for later review, editing, and retrieval. You may work on several problems at once, pursue side problems to make experimental definitions and trial substitutions and later undo their effect to return to a previous line of pursuit. The workspace saves all your results for later recall or for a future session.

In the example conversations which follow, user input begins in column 1, system responses are indented to column 4. Anything enclosed in curly brackets is a comment.

```
11+11  {what is 11 plus 11?}

   (1)  22

111↑111 {integer computations remain fully accurate}

   (2)  1073620128884742258012145650466955019598507239942248048047759_
        1117562507619578334702249122617009363462146610374309298696 7_
        7777863300673101594633035586669100910260177855872955396221 4_
        2057315437069730229375357546494103400699864397711

(x+1)↑6  {what is the result of expanding x+1 to the 6th power?}

          6     5      4      3      2
   (3)  x  + 6x  + 15x  + 20x  + 15x  + 6x + 1

factor(deriv(%,x)) {The symbol % refers to the last expression displayed}

              5
   (4)  6(x+1)
```

A number of **system commands** perform various services for the user. These include: **)edit**, **)read**, and **)write**, which allow you to edit, read, or write to output files; **)on** and **)off** which turn on/off various system flags such as **FORTRAN** output; **)what**, which provides an interactive query of algebraic facilities; **)compile**, and **)trace** which cause functions to be compiled and traced. Also, command **)undo** un-does all computations back to some point; **)redo** is similar except that commands may be edited and then reexecuted.

```
)undo 3  {undo every command back until after display of (3)}

% {what is the last expression displayed}

          6     5      4      3      2
   (5)  x  + 6x  + 15x  + 20x  + 15x  + 6x
```

Key #2: Expressions

With the exception of the system commands which begin with ")", every input to, and output from, the system is an expression. In addition to the simple arithmetic expressions illustrated thus far, there are numerous infix and prefix operations available in the language. This primer principally focuses on the notion of "sequence", the only aggregate in top-level **Scratchpad II**. A **sequence** consists of zero or more expressions separated by commas and enclosed in square brackets [ ].

```
[1,"hoho",[x,true]]  {sequences can have members of any kind}
   (1)  [1,"hoho",[x,true]]
[11,12,13,14,15]  {integer segments are generally abbreviated}
   (2)  [11..15]
[1,1..3,5,8,13,21] {the first 8 Fibonacci numbers}
   (3)  [1,1..3,5,8,13,21]
[2↑i-1 for i in 1,1..3,5,13,21] {sequences can also be created by for-constructs}
   (4)  [1,1,3,7,31,255,8191,2097151]
[2↑i-1 for i in 1..20 | isPrime i] {for-constructs may have filters}
   (5)  [3,7,31,127,2047,8191,131071,524287]
[0..9]-10 {sequences may be added to or multiplied by a scalar}
   (6)  [-10,-9,-8,-7,-6,-5,-4,-3,-2,-1]
[1..50 → 0, 51..100 → 1]  {a seq. whose first 50 elements are 0, next 50 are 1}
   (7)  [1..50 → 0, 51..100 → 1]
[0..] {sequences may be infinite, here the sequence of integers from 0}
   (8)  [0..]
-[0..]+[0..]  {inf. seq. may be added, e.g. here to form an inf. seq. of 0's}
   (9)  [0.. → 0]
[1..50 → 0,51.. → 1] {a generalization of an above example as an infinite sequence}
   (10) [1..50 → 0,51.. → 1]
```

Key #3: Forms

Forms are used to name objects and functions. Forms are symbols which can have 5 kinds of parameters: subscripts, superscripts, pre-superscripts, pre-subscripts or functional arguments. Scripts are enclosed in [ ] brackets which have no preceding blank.

```
f[i;j](u,v) {f with 1 subscript, 1 superscript, and two functional arguments}
```

$$(1)\quad {}^{j}_{i}f(u,v)$$

Two forms with the same leading name and the same number and kinds of scripts/arguments refer to the same function, e.g. the function x with 1 subscript, denoted by x[□]. Examples of use of this function are: x[1], x[k], and x[i+j]. Juxtaposition of a form f to an expression x always means "f applied to x" and may be written "f(x)" or simply "f x". Successive juxtapositions nest to the right, e.g. "f g x" and "f(g(x))" are equivalent. Application is also indicated by infix "." for which successive applications nest to the left, e.g. "x.i.j" and "(x.i).j" are equivalent.

If a function takes two or more functional arguments, its arguments are written as "tuples": expressions enclosed in parentheses and separated by commas. Tuples do not denote data objects but rather are syntactic denotations for groups of two or more values passed to, or returned from, a function. Inner parentheses are superfluous, e.g. if "g u" returns two values x and y, all the following are equivalent: "f(g u,z)", "f((x,y),z)", and "f(x,y,z)". When applications are mixed with infix and prefix operations, applications are done first. Thus "f x + g (x,y)" and "(f x)+g(x,y)" are equivalent.

Key #4: Rules

Rules have the format: <form> → <expression>. Rules perform no computation; they simply describe how to compute something you later want to ask for. Once the notions of rules and expressions have been mastered, **Scratchpad II** programs can be written to perform any computable function.

To define the Legendre polynomials p(0),p(1),..., one can simply write a recurrence relation using three rules.

```
p(0) → 1
p(1) → x
p(n) → (2*n-1)*x*p(n-1)/n-(n-1)*p(n-2)/n when n in 2..
```

These rules cause no computation but simply tell **Scratchpad II** how to compute p(i) for each i=0,... . To obtain values for the nth polynomial, one simply types the expression "p(n)". The expression p(n) will then be continually rewritten using the above rules and simplified until no further change is possible. The result is called the "value" of p(n).

```
[p(n) for n in 5..6] {what are p(5) and p(6)?}
```

$$(1)\quad \left[\frac{63x^5 - 70x^3 + 15x}{8},\ \frac{231x^6 - 315x^4 + 105x^2 - 5}{16}\right]$$

Key #5: Maps

Maps provide a general mechanism for representing mappings of source expressions to target expressions. Maps thus represent many things: sequences of rules, finite and infinite sequences, and, indeed, general functions.

```
{The simplest example of a map is a sequence as described under key #2}
[1,1..3,5,8,13,21]  {the mapping 0 → 1, 1 → 1, 2 → 2,...,7 → 21}

   (1)  [1,1..3,5,8,13,21]

% [1,3,5,7,9] {apply map to each element; "□" denotes an undefined value.}

   (2)  [1,3,8,21,□]

)clear properties p
p(0) → 1   {again define Legendre polynomials}
p(1) → x
p          {the value of p is now the mapping 0 → 1, 1 → x}

   (3)  [1,x]

p(n) → (2*n-1)*x*p(n-1)/n-(n-1)*p(n-2)/n when n in 2..
p         {p's value is a map which tells how to generate values of p}
                                 (2n - 1)x*p(n - 1) - (n - 1)p(n - 2)
   (4)  [1,x,(n | n in 2..) → ------------------------------------]
                                              n

p [0..5] {what are the first 6 Legendre polynomials?}

                    2          3          4       2          5       3
                  3x  - 1    5x  - 3x   35x  - 30x  + 3   63x  - 70x  + 15x
   (5)  [1,x,---------,----------,-----------------,-------------------]
                  2          2             8                 8

{define a function to generate the Fibonacci series}
fib 0 → 1; fib 1 → 1; fib n → fib n-1 + fib n-2 when n in 2..
fib  {the value of function fib is a map i → ith Fibonacci number, i=0,..}

   (6)  [1,1,(n | n in 2..) → fib(n-1)+fib(n-2)]

{Infinite sequences are obtained by composition with infinite sequences}
fs → fib [1..] {the infinite sequence of Fibonacci numbers}
oddFn → [n for n in fs | isOdd n] {any sequence can be iterated over}
(3*oddFn-1) [2*i for i in 1..3] {or used in arithmetic expressions}

   (7)  [8,38,101]

{Maps may be defined explicitly}
cover(n) → [0..n → 1, (n+1).. → 0]
cover(-1) = [0.. → 0]  {two seq. are equal if they have the same generators}

   (8)  true

primes → sieve [2..] where sieve [h,:t] → [h,:sieve [y for y in t | h ¬|| y]]
n s → s [0..n-1] when n in 0.. {new notation for 1st n elements of a sequence}
10 primes  {what are the first 10 primes computed using sieve method?}

   (9)  [2,3,5,7,11,13,17,19,23,29]

primesLessThan n → [x for x in primes while x < n]
primeDivisorsOf n → [p for p in primesLessThan (n//2) | p || n]
divisors → primeDivisorsOf [1..]  {a sequence of sequences}
```

Key #6: Quote Marks

Up until now, if you re-evaluate the result of any computation, you will always get the same result back again. Welcome now to a world with quote-marks! The value of an expression preceded by a quote-mark is the expression itself. Using quote-marks, it is now possible to delay evaluation. Quote-marks are also used to name specific symbolic arguments for rules.

The value of a quoted-expression is the expression itself; extra evaluations are performed by the function "ev"; each level of ev will remove exactly one quote-mark.

```
a → ''b; b → 7; [a,ev a,ev ev a]

   (3)  ['b,b,7]
```

If an argument to a function appearing on the left-side of a rule is a quoted expression, the rule is understood to give the value of the function applied to that symbolic constant.

```
f(x) → 0      {define f(x) = 0 for all x}
f('y) → a     {except let f have the value a at point "y"}
f               {what is f?}

   (1)  ['y → 1, x → 0]

f [0,y] {what are the values of f at 0 and y?}

   (2)  [0,a]
```

Key #7: Assignments

Assignments have the syntax: <form> ← <expression>. Assignments always cause <expression> to be immediately evaluated with the result "assigned" to <form>. The next time the value of <form> is requested, that assigned value is immediately returned without re-evaluation. Using assignments, you can permanently capture a computed result and be assured that it never will change.

```
u → [x,y,z]
u where (x → y+1; y → z+1; z → 7) {the ordering or rules is generally unimportant}

   (1)  [9,8,7]

u where (x ← y+1; y ← z+1; z ← 7) {assignments compute values immediately}

   (2)  [y+1,z+1,7]

u where (z ← 7; y ← z+1; x ← y+1) {the ordering of assignments is usually critical}

   (3)  [9,8,7]

u where (x → y+1; y ← z+1; z → 7) {Right and left assignments can always be mixed}

   (4)  [z+2,z+1,7]

{Define fib so that all previously computed values are saved}
fib 0 → fib 1 → 1
fib n → fib n ← fib (n-1) + fib (n-2) when n in 2..4
fib

   (5)  [1,1,(n | n in 2..) → fib n ← fib (n-1) + fib (n-2)]
```

```
fib 11

   (6)   144

fib

   (7)   [1,1,2,3,5,8,13,21,34,55,89,144,
           (n | n in 2..) → fib n ← fib (n-1) + fib (n-2)]

{fib can also be defined so as to save only the last two values}
fib 0 → fib 1 → 1
fib n → (fib n ← fib (n-1) + fib (n-2) {compute and store new value};
         fib (n-2) ←  {delete previously computed value};
         fib n {return new value}) when n in 2..
fib 11

   (8)   144

fib

   (9)   [10 → 89,144,(n | n in 2..) →
           (u ← fib n ← fib n-1 + fib n-2; fib (n-3) ← □; u)]

{Assignments could also have been used to create the Legendre polynomials}
p(0) ← 1
p(1) ← 1
p(n) ← (2*n-1)*x*p(n-1)/n-(n-1)*p(n-2)/n for n in 2..4
p       {what are the first 5 Legendre polynomials?}

                    2            3              4       2
                  3x  - 1      5x  - 3x      35x  - 30x  + 3
   (10)  [1,'x,'(---------),'(----------),'(-----------------)]
                    2            2                 8
```

Key #8: Modes

A *mode* is an expression denoting a set of representations. Using modes, you can preselect output formats, re-format something already computed, or specify an algebraic structure in which a computation will take place. Expression "e @ M" causes expression e to be converted to a representation indicated by mode M.

The example which follows is from [LOOS74].

```
u → z↑3*(y↑2-1)/(x+.i)+3*z*(x-.i)/(y+.i) -1
u@P[z]□ {display u as a polynomial in z}

                  2
             3  y  - 1          - 3.i + 3x
   (1)   u: z (--------) + z(------------) - 1
                .i + x           .i + y
```

The symbol □ stands for a "don't care" mode to be determined by the system. The □ can generally be omitted (e.g. "u@P[z]" in above example). Similarly, "u@P[z]G" will display u as a polynomial in z with Gaussian coefficients; "u@P[z]G(FP)", as a polynomial in z with Gaussian coefficients with real and imaginary parts factored (over the integers); and, "u@P[z]FP", as a polynomial in z with coefficients factored (over the Gaussian integers). A mode which contains no explicit or implicit □ designates a specific algebraic structure called a *domain*. In general, modes only partially specify a domain into which an expression is to be converted with the details of that structure left to be determined by the system. When used as a prefix-operator, □ denotes an arbitrary gap in the domain specification, e.g. "u@□(G I)" means "convert u to any domain over the Gaussian integers". Similarly, "u@P(□I)" means "convert u to a polynomial over any domain whose ground domain is the integers".

A *declaration* "x: M" is used to force any value assigned to x to be first converted to a representation given by M.

```
v: G P[z]  {gaussian expressions with coefficients as polynomials in z}
v ← u      {what is u expressed as a G(P[z])?}
```

```
   (2)

         2                                                  2
  3  x*y  - x          3x*y - 3                3  - y  + 1          - 3x - 3y
 z (----------) + z(----------) - 1  +  .i(z (----------) + z(-----------))
       2                 2                       2                 2
      x  + 1            y  + 1                  x  + 1            y  + 1
```

Declarations may also be used to declare the modes of arguments and return values of functions. For example, in the following definition of Legendre polynomials, the function p is declared to map the non-negative-integers into the domain of polynomials in x over the rational numbers.

```
p: NNI -> P[x]RN
p(0) → 1
p(1) → 1
p(n) → (2*n-1)*x*p(n-1)/n-(n-1)*p(n-2)/n otherwise
)on compile
```

The ")on" system command causes the rules for p to be compiled when first invoked; compiled rules generally run 10-100 times faster than if interpreted. Here again the declaration for p is optional, in the above case, unnecessary. When rules are to be invoked with the compiler option turned on, rules are mode-analyzed. In the above case, the declared mode would be chosen by default.

Declarations are used to declare the *type* of an argument or return value. One example of a type is a mode. Another is a "category".

Key #9: Categories

In the last section, it was shown how a declaration can be used to declare that an argument or return value of a function is to come from some domain. The concept of *category* allows you to be less specific. Using categories, it is possible to declare arguments and return values of functions to come from any domain with specific algebraic properties. Categories thereby permit the definition and compilation of functions which can be used in the widest possible algebraic context.

The system provides a number of built-in categories. Among these is OrderedSet.

```
D: OrderedSet  {let domain D be a member of category OrderedSet}
x,y: D         {let x and y be members of D}

max(x,y): D → if x < y then y else x
```

Category OrderedSet denotes the class of all domains which have a total-ordering operation "<". This definition of max can be used to compute the maximum of two elements from any domain which is an ordered set.

```
max(3,-3) {what is the maximum of two integers?}

   (1)  3

max(5.3,3.5) {what is the maximum of two floats?}
```

```
(2)   5.3
```

A category designates a class of domains having certain specific operations and algebraic properties as described below. The simplest, most basic algebraic category is Set.

```
)what Set  {what is a Set?}
   3 operations:
        □=□: ($,$) -> Boolean
        coerce: $ -> Expression
        coerce: Expression -> Union($, "failed")
```

The above description of Set may be interpreted as follows: "A domain D is a set (a member of Set, the category of all sets) if it has an operation □=□: (D,D) -> Boolean and the above two coerce operations." The expression appearing to the right of the ":" is called the signature of the operation. In the notation □=□, the □ shows where the arguments go and therefore this means "the infix operation =". Categories are created by functions called "category-constructors", that is, functions which return a category. The category-constructor for Set is defined as follows:

```
Set: Category → with
   {operations}
      □=□: ($,$) -> Boolean
      coerce: $ -> Expression
      coerce: Expression -> Union($, "failed")
```

The above syntax defines Set to be a category-constructor with no arguments which returns an object of type Category (the class of all categories). In order to assert that the operations of a domain satisfy certain algebraic properties, *attributes* may be included in the category definition. Attributes designate mathematical facts such as axioms and theorems that domains are asserted to have. For example, the category OrderedSet extends Set to include one new operation □<□ and one attribute: total(□<□) (which asserts that the infix operation '<' is total).

```
OrderedSet: Category → Set with
   {operations}
      □<□: ($,$) -> Boolean
   {attributes}
      total(□<□)    {not(x < y) and not(y < x)  implies x=y}
```

Another way of forming a category is by performing the **join** of two other categories, that is, the category formed by directly combining the operations and attributes of one category with those of the other, e.g.:

```
FiniteField: Category → Join(Field,Finite)
```

Categories may also be parameterized, e.g:

```
Algebra(R:SimpleRing): Category → Join(SimpleRing,Module(R)) with
   {operations}
      coerce: R -> $
      ...
```

The category-constuctor Algebra creates the category of all R-algebras, that is, algebras over a given ring R. For example, the function Algebra applied to Integer produces the category of all domains D which are both simple rings and algebras over the integers and which have the additional operation "coerce: Integer -> D".

Categories are organized into a hierarchy as shown in Appendix 3. The definitions of all system categories are interactively accessible. Using the system editor, users can inspect all system categories and define new ones needed for their work. In the current implementation of **Scratchpad II**, however, it is not possible to change an existing category without recompiling functions which reference them.

Key #10: Domains

The objects you compute with (e.g. integers, polynomials, matrices) are members of algebraic structures called domains. In **Scratchpad II**, domains are objects too! Domains are created by functions called "domain-constructors." All domain-constructors are interactively accessible for user inspection and modification. By use of the system editor, users can create new domains of their own and modify those provided by the system.

A **domain** consists of two parts:

1. a category — a set of operations and a set of attributes, and
2. a set of functions which implement these operations so as to satisfy the attributes, that is:

Domain = Category + Functions

All domains are created by functions called *domain-constructors*. The return type of a domain-constructor is a category, called the "target category", which describes the operations and attributes of the domain it produces. A "capsule" part of the domain-constructor definition defines the set of functions which implement the operations described in the target category. Domains provide an abstract datatype component to the language. Whereas the target category provides "public" information about the domain, the capsule part describes a representation for objects of the domain and is "private." Programs which manipulate objects of the domain are permitted only to use the exported operations of the domain and, in particular, are not permitted to take advantage of any knowledge of how objects of the domain are implemented. The format of a domain-constructor definition closely parallels the above equation for domains.

```
Gaussian(R:Ring): TargetCategory → Definition where
  TargetCategory →                  {public part}
    Join(Ring,Algebra(R)) with
      real: $ -> R
      imag: $ -> R
      [□,□]: (R,R) -> $
  Definition → add                  {private part}
    {representation}
      Rep ← Record(real:R,imag:R)
    {declarations}
      x,y: $
      r,i: R
      n: Integer
    {define}
      0 → [0,0]
      1 → [1,0]
      gauss(r,i) → [r,i]
      real(x) → x.real
      imag(x) → x.imag
      x + y → [x.real+y.real,x.imag+y.imag]
      - x   → [-x.real,-x.imag]
      r * x → [r*x.real,r*x.imag]
      n * x → [n*x.real,n*x.imag]
      x * y → [x.real*y.real-x.imag*y.imag,x.imag*y.real+y.imag*x.real]
      characteristic → characteristic$R
                ...
```

Figure 1. Gaussian Domain-Constructor

<Domain-constructor Form>: <Target Category> → add <Capsule>

public part private part

Example 1: Gaussian.

A domain-constructor Gaussian (slightly simplified) takes a ring R as an argument and produces the domain "Gaussian over R" (Figure 1 on page 12). Three special cases are the Gaussian integers (R=Integer), the Gaussian rationals (R=RationalNumber), and the complex numbers (R=Float). Its target category lists the operations which can be performed on Gaussian domains: the ring operations — such as 0,1,+,-,*,↑; scalar multiplication (from Algebra(R)); two functions, real and imag, to select real and imaginary parts; and [a,b], to construct a Gaussian object from 2 members a and b of R.

Inside the domain-constructor definition, the symbol "$" stands for the domain "Gaussians-over-R" which the constructor creates. The capsule (everything following "add") consists of one or more statements to be processed in order. An essential component of a capsule is Rep which describes the representation of the objects of the domain. Rep is always assigned a value which is a predefined domain, here, Record(real:R,imag:R). Records are basic constructors which implement general Cartesian products of domains [SCRA84]. Here, the record type provides operations "x.real" and "x.imag" to select the real and imaginary parts of a record object x, and "[a,b]" to construct a record object from two members a,b of R. Within a capsule — and nowhere else — the set of operations on $ are extended to include the operations of the representation.

The capsule contains a set of declarations followed by a list of rules defining a set of functions, one function for each operation mentioned in TargetCategory. The declarations are useful, for example, to disambiguate the 3 definitions of "*" in the capsule. As with categories, the statements of a capsule are all left-aligned.

Remarks. Definitions of domain-constructors, as well as those of categories and package-constructors (covered later), represent system code which is always compiled, never interpreted. Several freedoms allowed in top-level **Scratchpad II** are not allowed in system code where code legibility is considered of paramount importance. For example, within each function definition in a capsule, the following *strong-typing rules* are required:

1. the type of every identifier must be constant and unambiguous and
2. the type of each simple expression must be determinable by its context.

It is never permissible to omit the type of an argument of a function unless uniquely implied by its signature in the target category. Also, unlike conversational **Scratchpad II** where it is possible to add a real r to an integer i and get a real result, the conversion of an object from one type to another must be made explicit by use of one of several "conversion" expressions.

Example 2. Localization.

Domain-constructor Localization produces a domain of localizations, fractions for which the numerator domain N and denominator domain D may be different. For example, if N is "polynomials over R", D may be "factored polynomials over R" or "factored integers". Domain N is required to be a module over a ring R in order to allow scalar multiplication. Addition of localizations requires the ability to multiply elements of D into elements of N; D is thus required to be a subset of R. Finally, in order to add localizations, D must be closed under multiplication, that is, D must be a monoid.

```
Localization(N,R,D): TargetCategory → Definition
 where
  R: Ring
  N: Module(R)
  D: SubsetCategory(Monoid,R)
  TargetCategory →                {public part}
    Module(R) with
      if N has OrderedSet then OrderedSet
      if N has Algebra(R) then Algebra(R)
      □/□:($,D) -> $
      numer: $ -> N
      denom: $ -> D
  Definition → add                {private part}
    {representation}
      Rep ← Record(num:N,den:D)
    {declarations}
      x,y: $
      n: Integer
      r: R
      d: D
    {definitions}
      0 → [0,1]
      -x → [-x.num,x.den]
      x=y → y.den*x.num = x.den*y.num
      numer x → x.num
      denom x → x.den
      if N has OrderedSet then
        x < y → y.den*x.num < x.den*y.num
      x+y → [y.den*x.num+x.den*y.num, x.den*y.den]
      n*x → [n*x.num,x.den]
      r*x → if r=x.den then [x.num,1] else [r*x.num,x.den]
      x/d → [x.num,d*x.den]
      if N has Algebra(R) then
        1 → [1,1]
        x*y → [x.num*y.num,x.den*y.den]
        characteristic → characteristic$N
               ...
```

Figure 2. Localization Domain-Constructor

The definition of Localization is given in Figure 2 on page 14. The statements in the where-clause are processed in order. Since the mode of N depends on R, R must be introduced before N. Also, the TargetCategory must occur before the Definition since it tells the compiler what functions must be provided in the Definition.

The meaning of the first if-then clause in target category is "if N has all the operations and attributes of OrderedSet, the domain Localization(N,R,D) has them as well."This if-clause in the target category requires a corresponding one in the capsule in order that the function for '<' be conditionally introduced into the domain. The use of infix $ in the definition of characteristic is needed for type disambiguation. The need for $ is avoidable in other situations. For example, the rule for 0 could have otherwise been defined by "0 → [0$N,1$D]"; but the need for $ in this case is obviated by the uniqueness of the [ ] construct in context.

Example 3. Quotient Field.

This domain-constructor produces the domains of localizations in which numerator and denominator elements come from the the same integral domain. Its target category has two conditional categories: OrderedRing and DifferentialRing. The latter conditional has the interpretation: "localizations are differentiable if elements of D are differentiable". The definition of QuotientField has a new form of definition:

```
QuotientField(D: IntegralDomain): TargetCategory -> Definition where
  TargetCategory ->                  {public part}
    Join(Field,Algebra(D)) with
      if D has OrderedRing then OrderedRing
      if D has DifferentialRing then DifferentialRing
      □/□: (D,D) -> $
      numer: $ -> D
      denom: $ -> D
  Definition ->                      {private part}
    Localization(D,D,D) add
      {representation}
        Rep <- Record(num:D,den:D)
      {declarations}
        x,y: $
        nn,dd: D
        n: Integer
      {definitions}
        recip(x) ->
          x.num = 0 => "failed"
          [x.den,x.num]
        nn / dd -> [nn,dd]
        if D has DifferentialRing then
          deriv(x) -> [deriv(x.num)*x.den - x.num*deriv(x.den), x.den↑2]
        if D has UniqueFactorizationDomain then
          {local declarations}
            cancelGcd: $ -> D
            normalize: $ -> $
          {local definitions}
            normalize(x) ->
              uca:=unitNormal(x.den)
              x.den:=uca.coef
              x.num:=x.num*uca.associate
              x
            cancelGcd(x) ->
              d:=gcd(x.num,x.den)
              x.num:=(x.num exquo d)@D
              x.den:=(x.den exquo d)@D
              d
          {redefinitions}
            recip(x) ->
              x.num = 0 => "failed"
              normalize [x.den,x.num]
            nn / dd -> (cancelGcd (z:=[nn,dd]);z)
            x + y  ->
               z:=[x.den,y.den]
               d:=cancelGcd(z)
               g:=[z.den*x.num + z.num*y.num, d]
               cancelGcd(g)
               g.den:= g.den * z.num * z.den
               normalize g
            x - y  ->
               z:=[x.den,y.den]
               d:=cancelGcd(z)
               g:=[z.den*x.num - z.num*y.num, d]
               cancelGcd(g)
               g.den:= g.den * z.num * z.den
               normalize g
            x * y  ->
               (x,y):=([x.num,y.den],[y.num,x.den])
               cancelGcd x; cancelGcd y;
               normalize [x.num*y.num,x.den*y.den]
            n * x ->
               y:=[n@D,x.den]
               cancelGcd y
               normalize [x.num*y.num,y.den]      ...

  RationalNumber: TargetCategory -> QuotientField Integer where
    TargetCategory -> Join(OrderedRing,Field,DifferentialRing) with
                        □/□: (Integer,Integer) -> $
                        numer: $ -> Integer
                        denom: $ -> PositiveInteger
```

Figure 3. Quotient Field Domain-Constructor

<New Constructor>: <Target Category> → <Old Constructor> add <Capsule>

public part — private part

The domain is initially created by instantiating Localization with A=D=R, then "add"ing in newly required functions and redefinitions. The capsules for Localization and QuotientField are independent and self-contained. On the other hand, because one extends the other, the representations given by Rep in each must be isomorphic. The system checks for such isomorphisms on instantiation.

Key #11: Packages

All functions which are not defined by domains are are organized into clusters called *packages*. Packages are created by functions called "package-constructors" which generally take domains as well as other types of values as arguments. All system package-constructors are interactively accessible for user inspection or modification. Using the system editor, users can create packages of their own or augment those provided by the system.

In **Scratchpad II**, functions (e.g. "factorize") are usually defined in their most general context and are therefore parameterized by one or more domains (e.g. "polynomials over a field F"). Packages are thus simply parameterized collections of functions where the parameters are often domains of some category (e.g. some field F).

All packages are defined by package-constructors which have the same syntax as domain-constructors. There is one difference, however. Whereas a domain-constructor creates a domain $ of computational objects, packages are simply collections of functions. As a consequence, all operations in the target category of a package-constructor are free of $.

Example 4. GrobnerPackage

The skeletal package-constructor for Grobner basis is shown in Figure 4. The package exports two functions: *grobner* which, given a list l of general polynomials over a domain R, produces a

```
GrobnerPackage(nvars,R,P): TargetCategory → Definition where
  nvars: Integer
  R: Field
  P: GeneralPolynomial(R,DirectProduct(nvars,NonNegativeInteger))
  TargetCategory → with
    reduce: (P,List(P)) -> P
    grobner: List(P) -> List(P)
  Definition → add
    reduce(S:P,Basis:List(P)) →
      j: Integer ← 0
      n ← #Basis-1
      t: Union(Expon, "failed")
      while j <= n repeat
        if not ((t ← degree(S) - degree(Basis.j)) case "failed")
           then
             S ← lc(Basis.j) * S - monom(t, lc(S)) * Basis.j
             j ← 0
           else j ← j+1
       S
         ...
```

Figure 4. Grobner Basis Package-Constructor

Grobner basis for l; and *reduce*, which given an polynomial p and a basis B, will reduce p with respect to B. The definition *grobner*, which requires locally defined data structures and numerous locally defined subfunctions, is omitted.

Appendix 1: Programming Language Syntax

```
Expression          ::= Conditional | Loop | OperatorExpression
Conditional         ::= if Expression then Expression  else Expression>
Loop                ::= <Iterator...> repeat Expression
    Iterator        ::= While | Until | For
    While           ::= while Expression
    Until           ::= until Expression
    For             ::= for Primary in Expression <Suchthat>
        Suchthat    ::= | Expression
OperatorExpression  ::= Prefix <InfixOp Expression | SuffixOp>
    Prefix          ::= PrefixOp Prefix | Reduction | Application
    PrefixOp        ::= {see Appendix 4}
    InfixOp         ::= {see Appendix 4}
    SuffixOp        ::= {see Appendix 4}
    Reduction       ::= ReductionOp Application
        ReductionOp ::= InfixOp /
        Application ::= Composition <Application>
            Composition ::= Primary. ...
Primary             ::= Literal | QuadPhrase | Parameter | Sequence | Enclosure
Literal             ::= Integer | Float | Boolean | String | Symbol
    Integer         ::= Digit...
    Float           ::= Mantissa <Exponent>
        Mantissa    ::= Digit... <Fraction> | Fraction
        Fraction    ::= . Digit...
        Exponent    ::= E <-> Digit...
    Boolean         ::= true | false
    String          ::= " <Any...> "
    Symbol          ::= First <Alphanumeric...>
        First       ::= $ | Letter
        Alphanumeric ::= Letter | Digit
QuadPhrase          ::= <AnyNonBlank...> QuadString...
    QuadString      ::= □ <AnyNonBlank...>
Parameter           ::= # Digit
Sequence            ::= [ <SequenceSpec> ]
    SequenceSpec    ::= Expression <IteratorTail>
    IteratorTail    ::= <repeat> Iterator...
Enclosure           ::= ( Expression )
```

Appendix 2: Domain Table

Basic Domains

E	Expression	F	Float
I	Integer	N	Name
S	String	SI	SmallInteger

SubDomains

EI	EvenInteger	ESI	EvenSmallInteger
NI	NegativeInteger	NNI	NonNegativeInteger
NNSI	NonNegativeSmallInteger	NPI	NonPositiveInteger
NPSI	NonPositiveSmallInteger	NSI	NegativeSmallInteger
OI	OddInteger	OSI	OddSmallInteger
PI	PositiveInteger	PSI	PositiveSmallInteger

Basic Constructors

L	List(D)	R	Record(s:D,...)
U	Union(D,...)	V	Vector(n,D)

Number Domains

FI	FactoredInteger	GF	GaloisField(p)
IM	IntegerMod(p,b)	BF	BigFloat(n)
RN	RationalNumber		

Algebraic Domains

A	Algebraic(e,D)	BBT	BalancedBinaryTree(D)
B	Boolean	CF	ContinuedFraction(e,D)
CP	ContentPrimitivepart(D)	DP	DirectProduct(n,D)
DMP	DistributedMultivariatePolynomial(vl,D)	F	Fraction(D,D)
FM	FreeModule(R,S)	FP	FactoredPolynomial(vl,D)
G	Gaussian(D)	IB	IntegralBasis(D)
LA	LocalAlgebra(R,S,D:=S)	M	Matrix(D)
MM	ModMonic(D,D)	P	Polynomial(D,E:=Integer)
PS	PowerSeries(x,D)	PR	PolynomialRing(R,E)
Q	Quaternion	QF	QuotientField(D)
RF	RationalFunction(vl,D)	SM	SquareMatrix(n,D)
SMP	SparseMultivariatePolynomial(f,D,vl)	SUP	SparseUnivariatePolynomial(D)
UP	UnivariatePolynomial(x,D)	VP	VectorPolynomial(D)

Appendix 3: Algebraic Categories

Category	Extends	Operations
Set		=, coerce
AbelianGroup	Set	0, +, -
OrderedSet	Set	<
SemiGroup	Set	*
Rng	AG, SG	
Monoid	SemiGroup	1
Group	Monoid	inv
SimpleRing	Rng, Monoid	characteristic, recip, coerce
Module(R:SimpleRing)	AbelianGroup	scalar multiplication
Algebra(R:SimpleRing)	SR, Module(R)	coerce
Ring	Algebra($)	
DifferentialRing	Ring	deriv
IntegralDomain	Ring	isAssociate, ← exquo.
SkewField	Ring	/
UniqueFactorizationDomain	IntegralDomain	gcd, factor, isPrime
EuclideanDomain	UFD	sizelp, div, quo, rem
Field	ED, SkewField	
Finite	Set	size, random
GaloisField	Field, Finite	
VectorSpace(S:Field)	Module(S)	
QuotientObject(S:Set)	Set	reduce, lift, coerce

Appendix 4: Scratchpad II Operators

Key: Each operator with a meaningful relative precedence is accompanied by an *operator precedence code* in the format (XnnY,mm) where:

X = prefix(P), suffix(S) or infix(default) operator
nn = precedence
Y = right(R)-, non(N)- or left(default)-associative
mm = optional right-precedence

		Separators	
,	(40)	tuple/sequence separator	f(x,y) [a,b]
;	(30)	block separator	(x ← 1; y ← 2; x+y)
where	(70,10)	qualifier	a where b ← 1
		Assignment	
→	(90R)	delayed	f(x) → x+1; a → b
←	(85R)	immediate	f(x) ← x+1; a ← b
		Conditional	
if	(P100)	conditional expression	if a then b else c
then	(P40)		
else	(P40)		
=>	(80N)	exit from surrounding block	a => b ≡ if a then exit b
		Logical	
and	(150)	evaluate while true	a and b and c
or	(150)	evaluate until true	a or b or c
not	(P160)	logical negation	not a
		Compare	
<	(180N)	less than	a < b
<=	(180N)	less than or equal	a <= b
=	(180N)	equal	a = b
¬=	(180N)	not equal	a ¬= b
>	(180N)	greater than	a > b
>=	(180N)	greater than or equal	a >= b
		Arithmetic	
+	(200)	plus	4 + 3 ≡ 7
-	(200) (P200)	difference, minus	4 - 3 ≡ 1; -3
*	(210)	times	4 * 3 ≡ 12
/	(210)	quotient	24/18 ≡ 4/3
||	(210)	does divide exactly	3 || 5 ≡ false
¬||	(210)	does not divide exactly	3 ¬|| 5 ≡ true
div	(210)	quotient and remainder	11 rem 3 ≡ [quotient: 3,remainder: 2]
rem	(210)	remainder	11 rem 3 ≡ 2
quo	(210)	quotient	11 quo 3 ≡ 3
exquo	(210)	exact quotient or "failed"	4 exquo 2 ≡ 2; 4 exquo 3 ≡ "failed"
**	(220)	exponentiation	4 ** 3 ≡ 256
		Application	
(blank) or .		apply	(f g) h ≡ f.g.h
!		apply each	[f,g]!x ≡ [f x,g x]
		Iteration	
for	(P100)	do for each element	for x in y repeat ...
until	(P100)	do iterate then test	until x < 0 repeat ...
while	(P100)	test then do iterate	while x > 0 repeat ...
repeat		do forever	repeat ...

		Control Exit	
leave		to leave a loop	while x repeat (.. leave u ..)
return		to return from a function	f(x) → (.. p => return u ..)
case	(180N)	union case branch	if x case A then ...
		Connectors	
with	(260N) (P260)	category operator	SomeCategory → with ...
add	(180N) (P180)	capsule operator	SomeConstructor → add ...
		Mode Operators	
:	(230R)	declare	n: Integer
::	(230R)	coerce	n:: Integer
@	(230)	operation fixing domain	f(n @ Integer)@Integer
pretend		force to mode N	n @ Integer pretend N
->		source/target separator	f: Integer -> Integer
has	(180N)	category predicate	if N has Ring then ...
		Quote Mark	
'	(P240)	quoted expression	x:= '(a+b)
		Constructor/Destructor	
:	(P120)	segment operator	[a,:b]
is	(180N)	destructuring predicate	if x is ["COND",:pl] then ...
isnt	(180N)	destructuring predicate	if x isnt ["COND",:pl] then ...
=	(P200)	tests equality to variable	pal [a,:b,=a] → pal b
		Rule Operators	
where	(70)	expression qualifier	a where b ← 3
otherwise	(S50)	gives rule least preference	f(0) → 1; f(n) → 0 otherwise
when	(60N)	rule qualifier	f(x) → 0 when x > 0
		Miscellaneous	
/		APL-reduction	+/[f(x) for x in S]
#	(P240)	size	#S
$		domain operator qualifier	x:= a +$R b

Appendix 5: Scratchpad II System Commands

Key:	Capitalized words are keywords and uncapitalized words are user-parameters. Angle-brackets indicate optional parts of a message.

SYNTAX

SystemCommand:	BasicCommand Option...
BasicCommand:	) KeyWord BasicParameter...
KeyWord:	Identifier
BasicParameter:	Primary
Option:	) OptionKeyWord OptionParameter...
OptionKeyWord:	Identifier
OptionParameter:	Primary

COMMANDS

)EDIT <fn <ft <fm>>>

- means: Edit file of filename=fn, filetype=ft, and filemode=fm If fn is missing, the last file read/edited is used; Default ft=INPUT,fm=A

)READ <fn <ft <fm>>>

- means: Read file of filename=fn, filetype=ft, and filemode=fm If fn is missing, the last file read/edited is used; Default ft=INPUT, fm=A

)CLEAR option ...

- option is: MODES OPERATIONS PROPERTIES RULES VALUES

)CMS message

- message denotes the rest of the line after "CMS "
- means: Pass message to CMS for execution

)CP message

- message denotes the rest of the line after "CP "
- means: Pass message to CP for execution

)LISP Sexpr

- Sexpr may be given over several lines
- means: Evaluate Sexpr using LISP interpreter. Atomic S-expressions must end with a blank.

)DISPLAY option ...

- option is one of: MODES OPERATIONS PROPERTIES RULES VALUES
- Format 1: **)DISPLAY** key ...
  - key is one of: MODES PROPERTIES RULES VALUES
  - GENERAL FORMAT IS: **)D** key <var1 var2 ... varLast>
  - Display the "key" of variables var1, var2, ..., varLast. If no vars are given, the "key" of all active variables are given.

- Format 2: **)DISPLAY OPERATIONS** ...
  - GENERAL FORMAT IS: **)D O** <op1 op2 ... oplast> <) **SEARCH>**
  - where: op1, op2, ..., oplast are either quoted strings (e.g. "+") or operator names (e.g. gcd)
  - Display all active modemaps for operations op1, op2, ... oplast. If no ops are given, active modemaps for all operations will be displayed
  - If **)SEARCH** is given, inactive modemaps are listed as well. (Note: a modemap is inactive if it requires at least one operand from an uninstantiated domain)

)TRACE ...

- Format 1: **)TRACE ?**
  - means: list the current functions which are traced
- Format 2: **)TRACE** f1 ...
  - where: f1,... are function names
  - GENERAL FORMAT IS: **)T** <fn1 fn2 ... fnLast> <option1 ...>
  - Trace units fn1, fn2, ..., fnLast with options option1,...
  - For BEFORE,AFTER, and COND, symbols #1, #2, ... may be used to indicate arguments 1, 2,...; #0, for value returned, and, #9, for caller.
  - Option key-words can be given in mixed case.
  - Options (for tracing function *fn*):
    - **)AFTER Sexpr**
      means: Evaluate Sexpr on exit from *fn*
    - **)BEFORE Sexpr**
      means: Evaluate Sexpr before entering *fn*
    - **)BREAK items**
      means: Break during execution of *fn* (before=break on entry, after=break on exit,id=break after id:=something
    - **)CALLER**
      means: give name of function which directly calls *fn*
    - **)COND Sexpr**
      means: Print only Sexpr has non-NIL value
    - **)COUNT**
      means: Set *fn*/DEPTH to 0, increment on each call
    - **)COUNT n**
      means: same, except trace only first n times then untrace
    - **)DEPTH**
      means: Set *fn*/DEPTH to 0, increment on call, decrement on return
    - **)DEPTH n**
      means: same but trace only when <= n
    - **)FROM fn1**
      means: trace when *fn* is directly called from fn1
    - **)ONLY items**
      means: Only print items mentioned (a=args,v=value,c=caller, 1=first arg,2=second arg,...) e.g.)only 1 v
    - **)RESTORE**
      means: restore previous trace-- ")t)r" means retrace variables last untraced; new options are added to previous

- **)VARS ALL**
  means: trace assignments to all variables within *fn*.
- **)VARS var1 ...**
  means: trace assignments to var1,... within *fn*.
- **)WITHIN fn1**
  means: Print only during times when fn1 is in execution

- Format 3: **)TRACE** d1 ...
  - where: d1,... are names or functor forms which evaluate to domains
  - GENERAL FORMAT IS:)T <d1 d2 ... dLast> <option1 ...>
  - Trace d1, d2, ..., dLast with options option1,...
  - Options:
    - **)OF** dom1 ...
      means: only trace functions OF domain dom1,...
    - **)VARS** var1 ...
      means: trace assignments to var1,... in traced fns.
    - **)OPS** op1 ...
      means: trace only op1,... within traced units (functors)

)UNTRACE i1 ...

- means: Untrace function or domain i1,...

)WHAT i1 ... <)a>

- means: What operations do i1,... provide where i1 denotes a category, domain, or package
- For domains and packages, unimplemented operations are enclosed in {}.

Appendix 6: The Scratchpad II Database:)what and has

```
I has UniqueFactorizationDomain  {is Integer a UFD?}

   (1)  true

I has Field                      {is Integer a field?}

   (2)  false

I has OrderedSet

   (3)  true

F has Field                      {are the reals (Float) a field?}

   (4)  true

SquareMatrix(2,RN) has Ring

   (5)  true

SquareMatrix(2,RN) has IntegralDomain

   (6)  false

SquareMatrix(2,RN) has commutative("*")

   (7)  false

SquareMatrix(2,RN) has commutative("+")

   (8)  true

)w Ring
Operations of category Ring :
   □+□ : ($,$) -> $                    □*□ : ($,$) -> $
   □*□ : ($,$) -> $                    □*□ : (I,$) -> $
   □**□ : ($,NNI) -> $                 -□  : $ -> $
   □-□ : ($,$) -> $                    □=□ : ($,$) -> B
   characteristic : () -> NNI          coerce : $ -> $
   coerce : I -> $                     coerce : $ -> E
   coerce : E -> Union($,failed)       recip : $ -> Union($,failed)
   One : () -> $                       Zero : () -> $

)what I
Operations of domain Integer :
   □<□ : ($,$) -> B                    □+□ : ($,$) -> $
   □*□ : ($,$) -> $                    □*□ : ($,$) -> $
   □*□ : (I,$) -> $                    □**□ : ($,NNI) -> $
   -□  : $ -> $                        □-□ : ($,$) -> $
   □=□ : ($,$) -> B                    abs : $ -> $
   associates? : ($,$) -> B            characteristic : () -> NNI
   coerce : $ -> $                     coerce : I -> $
 { coerce : $ -> E}                  { coerce : E -> Union($,failed)}
   deriv : $ -> $                      □exquo□ : ($,$) -> Union($,failed)
 { factorise : $ -> FF $}              gcd : ($,$) -> $
   max : ($,$) -> $                    min : ($,$) -> $
 { numberOfDigits : ($,$) -> $}        oddp : $ -> B
 { prime? : $ -> B}                    □quo□ : ($,$) -> $
   random : () -> $                    recip : $ -> Union($,failed)
   □rem□ : ($,$) -> $                  sizelp : ($,$) -> B
   unit? : $ -> B                      One : () -> $
 { SqFr : $ -> FF $}                   Zero : () -> $
   □div□ : ($,$) -> [quotient:$,remainder:$]
   expressIdealElt : ($,$,$) -> Union(notInIdeal,[coef1:$,coef2:$])
   princIdeal : ($,$) -> [coef1:$,coef2:$,generator:$]
   unitNormal : $ -> [unit:$,coef:$,associate:$]

)what F
Operations of domain Float :
   □<□ : ($,$) -> B                    □+□ : ($,$) -> $
   □*□ : ($,$) -> $                    □*□ : ($,$) -> $
```

```
   □*□ : (I,$) -> $                        □**□ : ($,I) -> $
   -□  : $ -> $                            □-□ : ($,$) -> $
   □/□ : ($,$) -> $                        □=□ : ($,$) -> B
   associates? : ($,$) -> B                characteristic : () -> NNI
 { coerce : $ -> $}                        coerce : I -> $
 { coerce : $ -> E}                      { coerce : E -> Union($,failed)}
   cos : $ -> $                          { deriv : $ -> $}
   exp : $ -> $                            □exquo□ : ($,$) -> Union($,failed)
 { factorise : $ -> FF $}                  gcd : ($,$) -> $
   inv : $ -> $                            ln : $ -> $
   log : $ -> $                            log2 : $ -> $
   max : ($,$) -> $                        min : ($,$) -> $
 { prime? : $ -> B}                        □quo□ : ($,$) -> $
   recip : $ -> Union($,failed)            □rem□ : ($,$) -> $
   sin : $ -> $                            sizelp : ($,$) -> B
   unit? : $ -> B                          One : () -> $
 { SqFr : $ -> FF $}                       Zero : () -> $
   □div□ : ($,$) -> [quotient:$,remainder:$]
   expressIdealElt : ($,$,$) -> Union(notInIdeal,[coef1:$,coef2:$])
   princIdeal : ($,$) -> [coef1:$,coef2:$,generator:$]
   unitNormal : $ -> [unit:$,coef:$,associate:$]
 { CharthRoot : $ -> Union($,NotInField)}

)w P[x]RN
Operations of UnivariatePoly[x]RN :
   □+□ : ($,$) -> $                        □*□ : ($,$) -> $
   □*□ : ($,$) -> $                        □*□ : (I,$) -> $
   □*□ : (RN,$) -> $                       □**□ : ($,NNI) -> $
   -□  : $ -> $                            □-□ : ($,$) -> $
   □=□ : ($,$) -> B                        associates? : ($,$) -> B
   characteristic : () -> NNI              coef : ($,NNI) -> RN
 { coerce : $ -> $}                        coerce : I -> $
   coerce : RN -> $                      { coerce : $ -> E}
 { coerce : E -> Union($,failed)}          content : $ -> RN
   degree : $ -> NNI                       discriminant : $ -> RN
   □exquo□ : ($,$) -> Union($,failed)    { factorise : $ -> FF $}
   gcd : ($,$) -> $                        lc : $ -> RN
   map : (RN -> RN,$) -> $                 mindeg : $ -> NNI
   monom : (NNI,RN) -> $                   pderiv : $ -> $
 { prime? : $ -> B}                        □quo□ : ($,$) -> $
   recip : $ -> Union($,failed)            red : $ -> $
   □rem□ : ($,$) -> $                      resultant : ($,$) -> RN
   sizelp : ($,$) -> B                     unit? : $ -> B
   var : () -> E                           varPol : () -> $
   HornerEval : ($,RN) -> RN               MonicDivide : ($,$) -> [q:$,r:$]
   MonicRemainder : ($,$) -> $             MultExpts : ($,NNI) -> $
   One : () -> $                           PrimPart : $ -> $
   PseudoRemainder : ($,$) -> $            SqFr : $ -> FF $
 { SubresGcd : ($,$) -> $}                 Zero : () -> $
 {  exquo  : ($,RN) -> Union($,failed)}
   □div□ : ($,$) -> [quotient:$,remainder:$]
   expressIdealElt : ($,$,$) -> Union(notInIdeal,[coef1:$,coef2:$])
   □exquo□ : ($,RN) -> Union($,failed)
   princIdeal : ($,$) -> [coef1:$,coef2:$,generator:$]
   unitNormal : $ -> [unit:$,coef:$,associate:$]
   DivExpts : ($,NNI) -> Union($,failed)

)what G F
Operations of domain Gaussian F :
   □+□ : ($,$) -> $                        □*□ : ($,$) -> $
   □*□ : ($,$) -> $                        □*□ : (F,$) -> $
   □*□ : (I,$) -> $                        □**□ : ($,NNI) -> $
   -□  : $ -> $                            □-□ : ($,$) -> $
   □=□ : ($,$) -> B                        associates? : ($,$) -> B
   characteristic : () -> NNI            { coerce : $ -> $}
 { coerce : F -> $}                        coerce : I -> $
 { coerce : $ -> E}                      { coerce : E -> Union($,failed)}
   □exquo□ : ($,$) -> Union($,failed)      □exquo□ : ($,F) -> Union($,failed)
 { factorise : $ -> FF $}                  gauss : (F,F) -> $
   gcd : ($,$) -> $                        imag : $ -> F
 { prime? : $ -> B}                        □quo□ : ($,$) -> $
   real : $ -> F                         { recip : $ -> Union($,failed)}
   □rem□ : ($,$) -> $                      sizelp : ($,$) -> B
   unit? : $ -> B                          One : () -> $
```

```
 { SqFr : $ -> FF $}                          Zero : () -> $
 { □div□ : ($,$) -> [quotient:$,remainder:$]}
   expressIdealElt : ($,$,$) -> Union(notInIdeal,[coef1:$,coef2:$])
   princIdeal : ($,$) -> [coef1:$,coef2:$,generator:$]
   unitNormal : $ -> [unit:$,coef:$,associate:$]


)w M
Matrix(R:Ring) is a domain constructor
   Operations:
 □+□ : ($,$) -> Union($,failed)          □*□ : (R,$) -> $
 □*□ : (I,$) -> $                        □*□ : ($,$) -> Union($,failed)
 □*□ : ($,V R) -> Union(V R,failed)      -□  : $ -> $
 □-□ : ($,$) -> Union($,failed)          □=□ : ($,$) -> B
 coerce : $ -> E                         coerce : E -> Union($,failed)
 col : ($,I) -> V R                      copyMatrix : $ -> $
 elt : ($,I,I) -> R                      elt : ($,I) -> V R
 map : (R -> R,$) -> $                   minor : ($,I,I) -> $
 ncols : $ -> NNI                        nrows : $ -> NNI
 row : ($,I) -> V R                      rowEchelon : $ -> $ if R has ED
 setcol : ($,I,V R) -> $                 setelt : ($,I,I,R) -> $
 setelt : ($,I,V R) -> $                 setrow : ($,I,V R) -> $
 transpose : $ -> $                      zero : (I,I) -> $
 HConcat : ($,$) -> Union($,failed)      SquareTop : $ -> Union($,failed)
 VConcat : ($,$) -> Union($,failed)
 □exquo□ : ($,R) -> Union($,failed) if R has ID
 nullSpace : $ -> Li V R if R has Field
 EchelonLastRow : $ -> $ if R has Field

)w List
List(S:Set) is a domain constructor
   Operations:
 □#□ : $ -> NNI                          □=□ : ($,$) -> B
 append : ($,$) -> $                     atom : $ -> B
 coerce : $ -> E                         coerce : E -> Union($,failed)
 cons : (S,$) -> $                       copy : $ -> $
 delete : (S,$) -> $                     drop : ($,I) -> $
 elt : ($,rest) -> $                     elt : ($,first) -> S
 elt : ($,last) -> S                     elt : ($,I) -> S
 first : $ -> S                          □in□ : (S,$) -> B
 insert : (S,$) -> $                     last : $ -> S
 lastTail : $ -> $                       list : S -> $
 maxIndex : $ -> I                       member : (S,$) -> B
 nconc : ($,$) -> $                      nil : () -> $
 nreconc : ($,$) -> $                    nreverse : $ -> $
 null : $ -> B                           position : (S,$) -> I
 removeDuplicates : $ -> $               rest : $ -> $
 rest : ($,NNI) -> $                     reverse : $ -> $
 setelt : ($,rest,$) -> $                setelt : ($,first,S) -> S
 setelt : ($,I,S) -> S                   setDifference : ($,$) -> $
 setIntersection : ($,$) -> $            setUnion : ($,$) -> $
 size : $ -> NNI                         sort : ($,(S,S) -> B) -> $
 take : ($,I) -> $

)w packages
 CRA CRApackage                CTP  CycloTomicPackage
 DDF DistinctDegreeFactorize   FF   FiniteFactorize
 GP  GrobnerPackage            HF   HenselFactorize
 HP  HenselPackage             IFP  IntegerFactorizationPackage
 MC  MiscellaneousCoerces      MF   MFactorize
 NR  NthRoot                   Ra   Ratint
 RZP RealZeroPackage           RZPQ RealZeroPackageQ
 SP  SolvePackage              UPSF UnivPolySquareFree

)what "*"
 * is an operation with 9 modemaps:
    (M *,M *) -> Union(M *,failed) from M *
    (*,M *) -> M * if * has Ring from M *
    (I,M *) -> M * from M *
    (M *,V *) -> Union(V *,failed) if * has Ring from M *
    (*,FactoredFormRing(*,*1)) -> * if * has A *1 from FactoredFormRing(*,*1)
    (*1,*) -> * if * has Mod *1 and *1 has SR
    (I,*) -> * if * has AG
    (NNI,*) -> * if * has AM
    (*,*) -> * if * has SG
```

References

[COWO81] Cole, C. A., Wolfram, S., et al, "SMP: A Symbolic Manipulation Program", California Institute of Technology, July, 1981.

[DAVJE80] Davenport, J. H. and Jenks, R. D., "MODLISP", Proceedings of LISP '80 Conference, August, 1980 (also available as IBM Research Report RC 8537, October 29, 1980).

[DEWA79] Dewar, Robert B. K., "The SETL Programming Language,", Computer Science Department, NYU Courant Institute of Mathematical Sciences, August, 1979.

[GRJY75] Griesmer, J. H., Jenks, R. D., and Yun, D. Y. Y., "SCRATCHPAD User's Manual", IBM Research Report RA 70, June 1975.

[JENK74] Jenks, R. D., "The SCRATCHPAD Language," Proceedings of a Symposium on Very High Level Languages, SIGPLAN Notices, Vol. 9, No. 4, April 1974 (Reprinted in SIGSAM Bulletin, Vol. 8, No. 2, May 1974).

[JENK79] Jenks, R. D., "MODLISP: An Introduction", Proc. EUROSAM 79 (Springer-Verlag Lecture Notes in Computer Science 72) pp. 466 - 480 (also available is slightly modified form as: "MODLISP: A Preliminary Design", IBM Research Report RC 8073, January 18, 1980).

[JESU84] Jenks, R.D., and Sundaresan, C.J., "The 11 Keys to Scratchpad II: A Primer", in preparation.

[JETR81] Jenks, R.D., and Trager, B.M., Proceedings of SYMSAC '81, 1981 Symposium on Symbolic and Algebraic Manipulation, Snowbird, Utah, August, 1981 (also published in SIGPLAN Notices, November, 1981 and available as an IBM Research Report RC 8930).

[LISK74] Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C., "Abstraction Mechanisms in CLU", Communications of the ACM, Vol. 20, No. 8, August, 1977.

[LOOS74] Loos, R., "Towards a Formal Implementation of Computer Algebra", Proceedings of Eurosam '74, SIGSAM Bulletin, Vol. 8, No. 3, August 1974, pp. 9-16.

[SCRA84] Davenport, J.H., Gianni, P., Jenks, R.D., Miller, V., Morrison, S., Rothstein, M., Sundaresan, C.J., Sutor, R.S., Trager, B.M., "Scratchpad II System Programming Language Manual", in preparation.

Examples.

Example 1: Big Floats

BigFloat numbers give extra precision when integer arithmetic is not indicated, yet single or double precision is not enough.

```
(a,b,c,d,e,p): BigFloat

precision 20     {Let's start with a relatively low accuracy}

   (1)  20
                      Time: 0.019(I) + 0.(E) + 0.01(P) = 0.029 SEC.

p := pi()

   (2)  3.141 59265 35897 93238 5
                      Time: 0.028(I) + 0.005(E) + 0.017(P) = 0.05 SEC.
```

{ A Fallacy:
set save
e sup < pi sqrt 163 >
set restore
is an integer. }

```
d:= sqrt 163

   (3)  12.76 71453 34803 70466 2
                      Time: 0.029(I) + 0.017(E) + 0.016(P) = 0.062 SEC.

exp(p*d)

   (4)  26253 74126 40768 744.0
                      Time: 0.108(I) + 0.049(E) + 0.017(P) = 0.174 SEC.

precision 60     { Sometimes, increasing accuracy may show our foibles... }

   (5)  60
                      Time: 0.016(I) + 0.(E) + 0.009(P) = 0.025 SEC.

p:=pi()

   (6)
3.141 59265 35897 93238 46264 33832 79502 88419 71693 99375 10582 09749 4
                      Time: 0.028(I) + 0.005(E) + 0.043(P) = 0.076 SEC.

d:= sqrt 163

   (7)
12.76 71453 34803 70466 17109 52009 78089 23473 82363 78030 12588 51212 6
                      Time: 0.029(I) + 0.021(E) + 0.04(P) = 0.09 SEC.
```

{Notice how, this time, the value is not an integer at all!}

```
exp(p*d)

   (8)
26253 74126 40768 743.9 99999 99999 92500 72597 19818 56888 79353 85633 7
                      Time: 0.107(I) + 0.15(E) + 0.046(P) = 0.303 SEC.

a:=sin(p/4)

   (9)
0.707 10678 11865 47524 40084 43621 04849 03928 48359 37688 47403 65883 4
                      Time: 0.24(I) + 0.15(E) + 0.042(P) = 0.432 SEC.

a*a

  (10)  0.5
                      Time: 0.091(I) + 0.007(E) + 0.018(P) = 0.116 SEC.
```

```
b:=cos(p/4)

  (11)
0.707 10678 11865 47524 40084 43621 04849 03928 48359 37688 47403 65883 4
                    Time: 0.241(I) + 0.144(E) + 0.042(P) = 0.427 SEC.

c:=cos(p/12)          { An example of an algebraic number }

  (12)
0.965 92582 62890 68286 74974 31997 28897 36763 39048 39008 40455 04023 43
                    Time: 0.241(I) + 0.101(E) + 0.042(P) = 0.384 SEC.

16*c**4-16*c**2+1

  (13)   0
                    Time: 0.402(I) + 0.02(E) + 0.004(P) = 0.426 SEC.
```

{A final example: π to 1000 places: }

```
precision 1000

  (14)  1000
                    Time: 0.737(I) + 0.003(E) + 0.076(P) = 0.816 SEC.

pi()


  (15)
3.141 59265 35897 93238 46264 33832 79502 88419 71693 99375 10582 09749 44592 30
781 64062 86208 99862 80348 25342 11706 79821 48086 51328 23066 47093 84460 9550
5 82231 72535 94081 28481 11745 02841 02701 93852 11055 59644 62294 89549 30381
96442 88109 75665 93344 61284 75648 23378 67831 65271 20190 91456 48566 92346 03
486 10454 32664 82133 93607 26024 91412 73724 58700 66063 15588 17488 15209 2096
2 82925 40917 15364 36789 25903 60011 33053 05488 20466 52138 41469 51941 51160
94330 57270 36575 95919 53092 18611 73819 32611 79310 51185 48074 46237 99627 49
567 35188 57527 24891 22793 81830 11949 12983 36733 62440 65664 30860 21394 9463
9 52247 37190 70217 98609 43702 77053 92171 76293 17675 23846 74818 46766 94051
32000 56812 71452 63560 82778 57713 42757 78960 91736 37178 72146 84409 01224 95
343 01465 49585 37105 07922 79689 25892 35420 19956 11212 90219 60864 03441 8159
8 13629 77477 13099 60518 70721 13499 99998 37297 80499 51059 73173 28160 96318
59502 44594 55346 90830 26425 22308 25334 46850 35261 93118 81710 10003 13783 87
528 86587 53320 83814 20617 17766 91473 03598 25349 04287 55468 73115 95628 6388
2 35378 75937 51957 78185 77805 32171 22680 66130 01927 87661 11959 09216 48413
9
                    Time: 24.354(I) + 0.006(E) + 0.551(P) = 24.911 SEC.
```

Example 2: Matrix Computations

The following is a simple demonstration of matrix handling in the system. We first of all declare a matrix whose entries are polynomials in a..d. This is small, and prints neatly. As we play with it, it grows until it can no longer be printed 'naturally', whereupon the system switches to a 'item-at-a-time' printing mode.

```
(m1,m2,m3): SM(2,MP([a,b,c,d],I))
m1 := [[a,b],[c,d]]

        |a b|
   (1)  |   |
        |c d|
                    Time: 0.095(I) + 0.005(E) + 0.041(P) = 0.141 SEC.

m2 := (m1+1)**2

        | 2                                 |
        |a  + 2a + b*c + 1  a*b + b*d + 2b  |
   (2)  |                                   |
        |                          2        |
        | a*c + c*d + 2c   b*c + d  + 2d + 1|
                    Time: 0.28(I) + 0.021(E) + 0.104(P) = 0.405 SEC.

m3 := (m2+1)**2

   (3)  Matrix1
Where
      Matrix1(1,1)
  :=
          4     3     2         2                                  2 2          2
        a  + 4a  + 3a b*c + 8a  + 2a*b*c*d + 8a*b*c + 8a + b c  + b*c*d
             4b*c*d + 8b*c + 4

      Matrix1(1,2)
  :=
          3     2         2         2          2                        2
        a b + a b*d + 4a b + 2a*b c + a*b*d  + 4a*b*d + 8a*b + 2b c*d
                2        3        2
        4b c + b*d  + 4b*d  + 8b*d + 8b

      Matrix1(2,1)
  :=
          3     2         2          2         2                          2
        a c + a c*d + 4a c + 2a*b*c  + a*c*d  + 4a*c*d + 8a*c + 2b*c d
                2        3        2
        4b*c  + c*d  + 4c*d  + 8c*d + 8c

      Matrix1(2,2)
  :=
          2                              2 2          2                       4
        a b*c + 2a*b*c*d + 4a*b*c + b c  + 3b*c*d  + 8b*c*d + 8b*c + d
                3     2
        4d  + 8d  + 8d + 4

                    Time: 0.271(I) + 0.091(E) + 0.652(P) = 1.014 SEC.
```

{We can also construct polynomials with matrix coefficients. Here again the printing is either 'whole matrix' or 'item-at-a-time', depending on the size of each individual matrix.}

```
(p1,p2,p3) :UP(x,SM(2,MP([a,b,c,d],I)))
p1 :=x*m1+m2

                | 2                                 |
        |a b|   |a  + 2a + b*c + 1  a*b + b*d + 2b  |
   (4)  |   |x + |                                   |
        |c d|   |                          2        |
                | a*c + c*d + 2c   b*c + d  + 2d + 1|
                  Time: 0.499(I) + 0.012(E) + 0.134(P) = 0.645 SEC.
```

```
p2 :=p1*m1

        | 2                  |
        |a  + b*c  a*b + b*d|
   (5)  |                    |x + Matrix1
        |                  2 |
        |a*c + c*d b*c + d   |
Where
                      3     2
   Matrix1(1,1) := a  + 2a  + 2a*b*c + a + b*c*d + 2b*c

                     2                        2       2
   Matrix1(1,2) := a b + a*b*d + 2a*b + b c + b*d  + 2b*d + b

                     2                       2      2
   Matrix1(2,1) := a c + a*c*d + 2a*c + b*c  + c*d  + 2c*d + c

                                            3     2
   Matrix1(2,2) := a*b*c + 2b*c*d + 2b*c + d  + 2d  + d

                  Time: 0.384(I) + 0.044(E) + 0.314(P) = 0.742 SEC.

p3 :=x*m1+m1

        |a b|    |a b|
   (6)  |   |x + |   |
        |c d|    |c d|
                  Time: 0.496(I) + 0.011(E) + 0.069(P) = 0.576 SEC.

p3**2

   (7)
      | 2                  |     |  2                        |
      |a  + b*c  a*b + b*d| 2   |2a  + 2b*c  2a*b + 2b*d|
      |                   |x  + |                          |x
      |                 2 |     |                        2 |
      |a*c + c*d b*c + d  |     |2a*c + 2c*d 2b*c + 2d   |
        | 2                |
      |a  + b*c  a*b + b*d|
      |                   |
      |                 2 |
      |a*c + c*d b*c + d  |
                  Time: 0.25(I) + 0.042(E) + 0.269(P) = 0.561 SEC.

p3**3

   (8)
      |    3                       2              2       2|
      |   a  + 2a*b*c + b*c*d    a b + a*b*d + b c + b*d | 3
      |                                                  |x
      | 2                2      2                   3   |
      |a c + a*c*d + b*c  + c*d    a*b*c + 2b*c*d + d    |
        |      3                        2                2         2|
        |   3a  + 6a*b*c + 3b*c*d     3a b + 3a*b*d + 3b c + 3b*d | 2
        |                                                        |x
        | 2                2       2                     3    |
        |3a c + 3a*c*d + 3b*c  + 3c*d     3a*b*c + 6b*c*d + 3d    |
        |      3                        2                2         2|
        |   3a  + 6a*b*c + 3b*c*d     3a b + 3a*b*d + 3b c + 3b*d |
        |                                                        |x
        | 2                2       2                     3    |
        |3a c + 3a*c*d + 3b*c  + 3c*d     3a*b*c + 6b*c*d + 3d    |
        |    3                          2                2       2|
        |   a  + 2a*b*c + b*c*d    a b + a*b*d + b c + b*d |
        |                                                  |
        | 2                2      2                   3   |
        |a c + a*c*d + b*c  + c*d    a*b*c + 2b*c*d + d    |
                  Time: 0.254(I) + 0.532(E) + 0.731(P) = 1.517 SEC.
```

```
p3**4
```

{In this display, the system notices that the x4 and x**0 coefficients are the same, so only gives them one name.}**

```
                 4             3             2
   (9)  Matrix1*x  + Matrix2*x  + Matrix3*x  + Matrix2*x + Matrix1
Where
                    4     2                   2 2         2
   Matrix1(1,1) := a  + 3a b*c + 2a*b*c*d + b c  + b*c*d

                    3     2          2         2     2          3
   Matrix1(1,2) := a b + a b*d + 2a*b c + a*b*d  + 2b c*d + b*d

                    3     2             2         2       2        3
   Matrix1(2,1) := a c + a c*d + 2a*b*c  + a*c*d  + 2b*c d + c*d

                    2                   2 2          2    4
   Matrix1(2,2) := a b*c + 2a*b*c*d + b c  + 3b*c*d  + d

                    4     2                    2 2          2
   Matrix2(1,1) := 4a  + 12a b*c + 8a*b*c*d + 4b c  + 4b*c*d

                    3     2          2          2     2          3
   Matrix2(1,2) := 4a b + 4a b*d + 8a*b c + 4a*b*d  + 8b c*d + 4b*d

                    3     2             2         2       2        3
   Matrix2(2,1) := 4a c + 4a c*d + 8a*b*c  + 4a*c*d  + 8b*c d + 4c*d

                    2                   2 2          2     4
   Matrix2(2,2) := 4a b*c + 8a*b*c*d + 4b c  + 12b*c*d  + 4d

                    4     2                   2 2          2
   Matrix3(1,1) := 6a  + 18a b*c + 12a*b*c*d + 6b c  + 6b*c*d

                    3     2          2          2     2          3
   Matrix3(1,2) := 6a b + 6a b*d + 12a*b c + 6a*b*d  + 12b c*d + 6b*d

                    3     2             2         2       2        3
   Matrix3(2,1) := 6a c + 6a c*d + 12a*b*c  + 6a*c*d  + 12b*c d + 6c*d

                    2                   2 2          2     4
   Matrix3(2,2) := 6a b*c + 12a*b*c*d + 6b c  + 18b*c*d  + 6d

                  Time: 0.347(I) + 0.51(E) + 1.326(P) = 2.183 SEC.
```

```
p3 :=x*m1 + transpose m1

       |a b|    |a c|
   (10) |   |x + |   |
       |c d|    |b d|
                 Time: 0.479(I) + 0.01(E) + 0.068(P) = 0.557 SEC.
```

```
p3**2
```

{Notice that the coefficient of x is not divisible by 2. There is no reason why it should be, since polynomials over a matrix domain are not commutative, and the binomial theorem does not apply. The system in fact applies the binomial theorem to exponentialte when the coefficients are commutative, but not otherwise.}

```
   (11)
      | 2                    |
      |a  + b*c  a*b + b*d| 2
      |                   |x
      |                 2 |
      |a*c + c*d b*c + d  |
      |       2    2    2                                  |    | 2                  |
|     2a  + b  + c       a*b + a*c + b*d + c*d|     |a  + b*c  a*c + c*d|
|                                             |x + |                   |
|                              2    2     2   |    |                  2 |
|a*b + a*c + b*d + c*d      b  + c  + 2d      |    |a*b + b*d b*c + d  |
                  Time: 0.242(I) + 0.038(E) + 0.291(P) = 0.571 SEC.
```

{We can also have matrices of matrices. Note how output may have to be nested.}

```
(mm1,mm2): SM(2,SM(2,MP([a,b,c,d],I)))
mm1 := [[m1,m1],[m1,m1]]

          ||a b| |a b||
          ||   | |   ||
          ||c d| |c d||
   (12)   |           |
          ||a b| |a b||
          ||   | |   ||
          ||c d| |c d||
                  Time: 0.063(I) + 0.007(E) + 0.137(P) = 0.207 SEC.

mm2 := [[m1,transpose m1],[m1,-m1]]

          ||a b|   |a c|  |
          ||   |   |   |  |
          ||c d|   |b d|  |
   (13)   |                |
          ||a b| |- a - b||
          ||   | |       ||
          ||c d| |- c - d||
                   Time: 0.058(I) + 0.005(E) + 0.157(P) = 0.22 SEC.

mm2**2

   (14)  Matrix1
Where
                    |   2          2                    |
                    | 2a  + b*c + c   2a*b + b*d + c*d  |
   Matrix1(1,1) := |                                   |
                    |                    2              2 |
                    |a*b + a*c + 2c*d  b  + b*c + 2d    |

                    |          2    2                                  |
                    |         b  - c          - a*b + a*c + b*d - c*d  |
   Matrix1(1,2) := |                                                  |
                    |                                  2    2          |
                    |- a*b + a*c + b*d - c*d        - b  + c           |

                    |0 0|
   Matrix1(2,1) := |   |
                    |0 0|

                    |   2    2                          |
                    | 2a  + b  + b*c  a*b + a*c + 2b*d  |
   Matrix1(2,2) := |                                   |
                    |                          2     2  |
                    |2a*c + b*d + c*d  b*c + c  + 2d    |

                    Time: 0.279(I) + 0.087(E) + 0.4(P) = 0.766 SEC.
```

```
mm2**3

   (15)  Matrix1
Where
   Matrix1(1,1) := Matrix2

   Matrix1(1,2) := Matrix3

   Matrix1(2,1) := Matrix4

   Matrix1(2,2) := Matrix5

                     3      2                 2
   Matrix2(1,1) := 2a  + a*b  + 2a*b*c + a*c  + 2b*c*d

                     2                       3    2        2
   Matrix2(1,2) := 2a b + a*b*d + a*c*d + b  + b c + 2b*d

                     2                        2    3       2
   Matrix2(2,1) := 2a c + a*b*d + a*c*d + b*c  + c  + 2c*d

                          2               2      3
   Matrix2(2,2) := 2a*b*c + b d + 2b*c*d + c d + 2d

                     3      2                 2    2     2
   Matrix3(1,1) := 2a  + a*b  + 2a*b*c + a*c  + b d + c d

                     2                       3       2    3        2
   Matrix3(1,2) := 2a c + 3a*b*d - a*c*d - b  + 2b*c  + c  + 2c*d

                     2                       3     2        2    3
   Matrix3(2,1) := 2a b - a*b*d + 3a*c*d + b  + 2b c + 2b*d  - c

                      2      2    2               2      3
   Matrix3(2,2) := a*b  + a*c  + b d + 2b*c*d + c d + 2d

                     3      2                 2
   Matrix4(1,1) := 2a  + a*b  + 2a*b*c + a*c  + 2b*c*d

                     2                       3    2        2
   Matrix4(1,2) := 2a b + a*b*d + a*c*d + b  + b c + 2b*d

                     2                        2    3       2
   Matrix4(2,1) := 2a c + a*b*d + a*c*d + b*c  + c  + 2c*d

                          2               2      3
   Matrix4(2,2) := 2a*b*c + b d + 2b*c*d + c d + 2d

                     3      2                 2
   Matrix5(1,1) := - 2a  - a*b  - 2a*b*c - a*c  - 2b*c*d

                     2                       3    2        2
   Matrix5(1,2) := - 2a b - a*b*d - a*c*d - b  - b c - 2b*d

                     2                        2    3       2
   Matrix5(2,1) := - 2a c - a*b*d - a*c*d - b*c  - c  - 2c*d

                          2               2      3
   Matrix5(2,2) := - 2a*b*c - b d - 2b*c*d - c d - 2d

                  Time: 0.29(I) + 0.622(E) + 1.239(P) = 2.151 SEC.
```

Example 3: Integer Factorization

This is a demonstration of the factoring of values of cyclotomic polynomials which are irreducible and therefore have values which are relatively hard to factor}

```
factor(3**17-1)     {notice dynamic loading of needed packages.}

   LOADING IFACT
   LOADING FMOD
   LOADING NTHROOT

   (2)  2*1871*34511
                    Time: 0.775(I) + 0.003(E) + 0.039(P) = 0.817 SEC.

factor(3**23-1)

   (3)  2*47*1001523179
                    Time: 0.411(I) + 0.002(E) + 0.031(P) = 0.444 SEC.

factor(3**31-1)

   (4)  2*683*102673*4404047
                    Time: 0.786(I) + 0.002(E) + 0.037(P) = 0.825 SEC.

factor(3**41-1)

   (5)  2*83*2526913*86950696619
                    Time: 1.003(I) + 0.002(E) + 0.035(P) = 1.04 SEC.

factor(3**53-1)

   (6)  2*107*24169*3747607031112307667
                    Time: 1.468(I) + 0.002(E) + 0.034(P) = 1.504 SEC.

factor(3**67-1)

   (7)  2*221101*441019876741*475384700124973
                    Time: 123.165(I) + 0.002(E) + 0.035(P) = 123.202 SEC.

factor(111111111111111111111111)

   (8)  3*7*11*13*37*73*101*137*9901*99990001
                    Time: 0.357(I) + 0.002(E) + 0.04(P) = 0.399 SEC.

factor(1111111111111111111111111111111111111111111111111)

   (9)  35121409*3163629087634585250014061540387263822 79
                    Time: 10.73(I) + 0.002(E) + 0.033(P) = 10.765 SEC.
```

Example 4: Polynomial Factorization

Scratchpad II offers extensive polynomial factoring facilities. The following examples display some factorizations of univariate polynomials over finite fields and multivariate polynomials over the Integers.

```
u2:P[x]GF(2)     {UnivariatePolynomial in x over GaloisField 2}

u2:=x**51+1

         51
   (4)  x   + 1
                    Time: 0.186(I) + 0.005(E) + 0.017(P) = 0.208 SEC.

factor u2

   (5)
        8    7    6    5    4           8    5    4    3
      (x  + x  + x  + x  + x  + x + 1)(x  + x  + x  + x  + 1)(x + 1)
  *
        8    4    3          2          8    7    5    4
      (x  + x  + x  + x + 1)(x  + x + 1)(x  + x  + x  + x  + 1)
  *
        8    7    4    3    2          8    7    6    4    2
      (x  + x  + x  + x  + x  + x + 1)(x  + x  + x  + x  + x  + x + 1)
                    Time: 0.097(I) + 1.148(E) + 0.3(P) = 1.545 SEC.

u2:=x**73+1

         73
   (6)  x   + 1
                    Time: 0.186(I) + 0.007(E) + 0.017(P) = 0.21 SEC.

factor u2

   (7)
        9    7    4    3       9    4    2          9    8    6    3
      (x  + x  + x  + x  + 1)(x  + x  + x  + x + 1)(x  + x  + x  + x  + 1)
  *
        9          9    8    7    5       9    8
      (x  + x + 1)(x  + x  + x  + x  + 1)(x  + x  + 1)(x + 1)
  *
        9    6    5    2       9    6    3
      (x  + x  + x  + x  + 1)(x  + x  + x  + x + 1)
                    Time: 0.097(I) + 2.35(E) + 0.293(P) = 2.74 SEC.

u5:P[x]GF(5)     {UnivariatePolynomial in x over GaloisField 5}

u5:=x**51+1

         51
   (8)  x   + 1
                    Time: 0.406(I) + 0.006(E) + 0.018(P) = 0.43 SEC.

factor u5

   (9)
        2
      (x  + 4x + 1)
  *
        16     15     13     10     9     8    5     3     2
      (x   + 2x   + 2x   + 3x   + 4x  + 3x  + x  + 3x  + 2x  + 4x + 1)
  *
        16     15     14     13     11     8     7     6     3
      (x   + 4x   + 2x   + 3x   + x   + 3x  + 4x  + 3x  + 2x  + 2x + 1)
  *
      (x + 1)
  *
          16     15     14     13     12     11     10     9     8     7    6
        x   + 4x   + x   + 4x   + x   + 4x   + x   + 4x  + x  + 4x  + x
                5    4     3    2
        4x  + x  + 4x  + x  + 4x + 1
                    Time: 0.205(I) + 2.314(E) + 0.466(P) = 2.985 SEC.

f:P[x,y,z]I      {MultivariatePolynomial in (x,y,z) over Integer}
```

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

          4        4     5      4      3 3     2 3      5     5
   (10)  x y*z + 3x z - y z - 3y z - 2y z  - 6y z  - y*z  - 3z
                    Time: 3.312(I) + 0.036(E) + 0.122(P) = 3.47 SEC.

factor f

                      2    2    2   2    2    2
   (11)  z(y + 3)(x  + y  + z )(x  - y  - z )
                    Time: 1.388(I) + 2.087(E) + 0.183(P) = 3.658 SEC.

f:=(7*y*x**2-3*y**2-4*z**2)*(y**2*x**2+y**2+z**2)*(x*z*y+3*z)

   (12)
        5 4       4 3       3 5       3 4       3 3 3      3 2 3      2 4
     7x y z + 21x y z - 3x y z + 7x y z - 4x y z  + 7x y z  - 9x y z
           2 3       2 2 3       2   3         5          3 3          5     4
     21x y z - 12x y z  + 21x y*z  - 3x*y z - 7x*y z  - 4x*y*z  - 9y z
             2 3       5
     - 21y z  - 12z
                    Time: 3.397(I) + 0.068(E) + 0.367(P) = 3.832 SEC.

factor f

             2 2    2    2    2      2     2
   (13)  z(x y  + y  + z )(7x y - 3y  - 4z )(x*y + 3)
                    Time: 0.098(I) + 4.687(E) + 0.207(P) = 4.992 SEC.

f:=(x**2+y**4+z)*(x+y+z**5)*(x**2+y**4+z**3)*(x**2-y**2)*(y**2-z**2)

   (14)
       7 2    7 2    6 3    6 2 5    6   2    6 7     5 6     5 4 2    5 4
     x y  - x z  + x y  + x y z  - x y*z  - x z  + 2x y  - 2x y z  - x y
        5 2 3    5 2 2    5 2     5 5    5 3     4 7     4 6 5     4 5 2
     x y z  + x y z  + x y z - x z  - x z  + 2x y  + 2x y z  - 2x y z
           4 5     4 4 7    4 4 5    4 3 3    4 3 2    4 3     4 2 8    4 2 7
     - x y  - 2x y z  - x y z  + x y z  + x y z  + x y z + x y z  + x y z
        4 2 6    4   5    4   3    4 10    4 8    3 10    3 8 2     3 8
     x y z  - x y*z  - x y*z  - x z  - x z  + x y  - x y z  - 2x y
        3 6 3     3 6 2    3 6     3 4 5    3 4 3    3 4     3 2 5    3 2 4
     x y z  + 2x y z  + x y z - x y z  - 2x y z  - x y z + x y z  + x y z
        3 2 3    3 6    2 11    2 10 5    2 9 2    2 9    2 8 7    2 8 5
     x y z  - x z  + x y  + x y  z  - x y z  - 2x y  - x y z  - 2x y z
        2 7 3    2 7 2    2 7     2 6 8    2 6 7    2 6 6    2 5 5
     x y z  + 2x y z  + x y z + x y z  + 2x y z  + x y z  - x y z
           2 5 3    2 5     2 4 10    2 4 8    2 4 6    2 3 5    2 3 4
     - 2x y z  - x y z - x y z  - 2x y z  - x y z  + x y z  + x y z
        2 3 3    2 2 10    2 2 9    2 2 8    2   6    2 11     12      10 2
     x y z  + x y z  + x y z  + x y z  - x y*z  - x z  - x*y  + x*y  z
           8 3      8      6 5      6 3     4 4     2 6    13    12 5
     - x*y z  - x*y z + x*y z  + x*y z  - x*y z  + x*y z  - y  - y  z
          11 2    10 7    9 3    9     8 8    8 6    7 5    7 3    6 10    6 8
     y  z  + y  z  - y z  - y z - y z  - y z  + y z  + y z  + y z  + y z
           5 4    4 9    3 6    2 11
     - y z  - y z  + y z  + y z
                    Time: 2.999(I) + 0.137(E) + 2.319(P) = 5.455 SEC.

factor f
   (15)
                     2    4      2    4    3                   5
   (y - z)(y + z)(x  + y  + z)(x  + y  + z )(x - y)(x + y + z )(x + y)
                    Time: 0.144(I) + 7.829(E) + 0.198(P) = 8.171 SEC.
```

Example 5: Factored Form Rings

Given an Algebra A over a Ring R, it is frequently more convenient to deal with factored expressions rather than ones multiplied-out. In particular, denominators of fractions should usually be left factored. By defining the domain constructor FactoredFormRing A, one can take advantage of factored forms in calculation. Let's look at two examples, one with factored integers and another with factored P[x]I.

```
i := 2**8 * 78**7 * 111**3 * 74534

   (1)  458379883137934911201804288
                    Time: 1.1(I) + 0.009(E) + 0.077(P) = 1.186 SEC.

j := 2**4 * 45**3 * 162**6 * 774325

   (2)  20406611309976203126400000
                    Time: 0.501(I) + 0.005(E) + 0.009(P) = 0.515 SEC.

(x,y,z): FFR I

x := factor i

          16 10  7  3
   (3)  2  3  13 37 83*449
                    Time: 0.297(I) + 0.01(E) + 0.048(P) = 0.355 SEC.

y := factor j

          10 30 5
   (4)  2  3  5 47*659
                    Time: 0.156(I) + 0.005(E) + 0.032(P) = 0.193 SEC.

x*y     {note the answer stays factored}

          26 40 5  7  3
   (5)  2  3  5 13 37 47*83*449*659
                    Time: 0.284(I) + 0.013(E) + 0.046(P) = 0.343 SEC.

y*x     {note the answer is sorted by factor}

          26 40 5  7  3
   (6)  2  3  5 13 37 47*83*449*659
                    Time: 0.135(I) + 0.01(E) + 0.047(P) = 0.192 SEC.

x+y     {once again, we will get a factored answer}

          10 10
   (7)  2  3  1109*3557*2007307818601
                    Time: 0.565(I) + 0.002(E) + 0.029(P) = 0.596 SEC.

(r,s): QF FFR I {perhaps you would like a quotient field?}

r := x/y          {note cancellation because I is a UFD}

           6  7  3
          2 13 37 83*449
   (8)  ----------------
              20 5
             3  5 47*659
                    Time: 0.724(I) + 0.007(E) + 0.096(P) = 0.827 SEC.

s := (x ** 9) / y

           134 60  63  27  9   9
          2   3  13  37  83 449
   (9)  ------------------------
                   5
                  5 47*659
                    Time: 0.5(I) + 0.007(E) + 0.106(P) = 0.613 SEC.
```

```
FFactorial 250

   (10)
       244 123 62 40  24  20  14  13  10  8  8  6  6  5  5  4  4  4  3  3
      2   3   5  7  11  13  17  19  23  29 31 37 41 43 47 53 59 61 67 71
  *
        3  3  3  2  2   2   2   2   2   2
      73 79 83 89 97 101 103 107 109 113 127*131*137*139*149*151*157*163
  *
      167*173*179*181*191*193*197*199*211*223*227*229*233*239*241
                    Time: 0(I) + 0(E) + 1.385(P) = 1.385 SEC.
```

{Now for some polynomials.}

```
(u,v): P[x]I

(f,g): FFR P[x]I

u := (x-1)**2 * (x+1)**2

           4     2
   (11)  x  - 2x  + 1
                    Time: 1.7(I) + 0.011(E) + 0.044(P) = 1.755 SEC.

v := x**5 + 5*x**3 - x

           5     3
   (12)  x  + 5x  - x
                    Time: 0.453(I) + 0.009(E) + 0.034(P) = 0.496 SEC.

HenselFactorize P[x]I
```

{Let's instantiate a package. Note the domains being dynamically brought into scope.}

```
   (13)  HenselFactorize(P[x]I)
                    Time: 0(I) + 0(E) + 0.054(P) = 0.054 SEC.

f := factor u
   LOADING UNIFACT
   LOADING DDFACT
   LOADING MODMON
   LOADING HENSEL

                 2       2
   (14)  (x + 1)  (x - 1)
                    Time: 1.754(I) + 0.018(E) + 0.043(P) = 1.815 SEC.

g := factor v

           4     2
   (15)  (x  + 5x  - 1)x
                    Time: 0.089(I) + 0.142(E) + 0.054(P) = 0.285 SEC.

f ** 67     {Note that the result remains factored.}

                  134       134
   (16)  (x + 1)    (x - 1)
                    Time: 0.203(I) + 0.006(E) + 0.043(P) = 0.252 SEC.

f * g

                 2       2  4     2
   (17)  (x + 1)  (x - 1) (x  + 5x  - 1)x
                    Time: 0.175(I) + 0.009(E) + 0.076(P) = 0.26 SEC.

(f + g) * f

            5    4     3     2                 2       2
   (18)  (x  + x  + 5x  - 2x  - x + 1)(x + 1)  (x - 1)
                    Time: 0.234(I) + 0.008(E) + 0.099(P) = 0.341 SEC.

(n,m): QF FFR P[x]I
```

```
n := f / g
```

$$(19)\quad \frac{(x+1)^2(x-1)^2}{(x^4+5x^2-1)x}$$

```
                    Time: 0.516(I) + 0.006(E) + 0.107(P) = 0.629 SEC.
```

```
m := g / f
```

$$(20)\quad \frac{(x^4+5x^2-1)x}{(x+1)^2(x-1)^2}$$

```
                    Time: 0.346(I) + 0.007(E) + 0.104(P) = 0.457 SEC.
```

```
n * m
```

{Once again, we have cancellation because P[x]I is a UFD}

```
   (21)  1
                    Time: 0.195(I) + 0.014(E) + 0.175(P) = 0.384 SEC.
```

Example 6: Simple Algebraic Extensions

The following example illustrates the calculation of the integral closure of a Simple algebraic EXtensIon (=SEXI) of a polynomial ring over the Rational Numbers. This is done by means of the Zassenhaus algorithm for integral closure as adopted by Barry Trager to polynomial rings. In all cases we deal with the affine coordinate ring of a plane algebraic curve. The examples come from the book Algebraic Curves by Fulton.

First we define the original ring, display some of its properties, calculate the integral closure, and display the same properties. (Note, the actual calculation of the integral closure is not made until the first property of the integral closure is asked for. That's why there are relatively large timings for rank)

For each curve we display the rank (degree of the curve), the DiscMatrix, which is (trace(x[i]x[j]) where x[i] is a basis of the ring over P[x]RN. We then display the discriminant of the curve (which is the determinant of DiscMatrix)

{This curve has a node of multiplicity 3 with 3 tangents. But is normal (i.e. its coordinate ring is integrally closed).}

```
f:=(x**2+y**2)**2 + 3*x**2*y-y**3

            4    3     2 2     2      4
   (1)     y  - y  + 2x y  + 3x y + x
                    Time: 5.253(I) + 0.034(E) + 0.152(P) = 5.439 SEC.

SS := SEXI(RN,x,y,f)

                          4    3     2 2     2      4
   (2)    SEXI(RN,x,y,y  - y  + 2x y  + 3x y + x )
                    Time: 0.018(I) + 0.002(E) + 0.095(P) = 0.115 SEC.

rank$SS

   (3)   4
                    Time: 1.002(I) + 0.001(E) + 0.008(P) = 1.011 SEC.

DiscMatrix$SS

   (4)
   |                                    2                     2            |
   |    4           1              - 4x  + 1            - 15x  + 1         |
   |                                                                       |
   |                  2                 2                  4      2        |
   |    1          - 4x  + 1       - 15x  + 1            4x  - 20x  + 1    |
   |                                                                       |
   |     2             2              4      2               4      2      |
   |- 4x  + 1      - 15x  + 1      4x  - 20x  + 1         45x  - 25x  + 1  |
   |                                                                       |
   |      2          4      2         4      2          6       4      2   |
   |- 15x  + 1   4x  - 20x  + 1  45x  - 25x  + 1  - 4x  + 129x  - 30x  + 1|
                    Time: 0.386(I) + 0.002(E) + 0.488(P) = 0.876 SEC.

discriminant$SS

                10        8       6
   (5)    4096x   - 3312x  + 108x
                    Time: 0.096(I) + 0.003(E) + 0.05(P) = 0.149 SEC.
```

{Now for the integral closure}

```
TT:-IntegralClosure(P[x]RN,SEXI(RN,x,y,f))

                                       4    3     2 2     2      4
   (6)     IC(P[x],RN,SEXI(RN,x,y,y  - y  + 2x y  + 3x y + x ))
                    Time: 0.035(I) + 0.002(E) + 0.102(P) = 0.139 SEC.

rank$TT

   (7)   4
                    Time: 2.336(I) + 0.(E) + 0.008(P) = 2.344 SEC.
```

```
DiscMatrix$TT
```

(8)

$$\begin{bmatrix} 4 & 1 & -4x^2+1 & -15x^2+1 \\ 1 & -4x^2+1 & -15x^2+1 & 4x^4-20x^2+1 \\ -4x^2+1 & -15x^2+1 & 4x^4-20x^2+1 & 45x^4-25x^2+1 \\ -15x^2+1 & 4x^4-20x^2+1 & 45x^4-25x^2+1 & -4x^6+129x^4-30x^2+1 \end{bmatrix}$$

Time: 0.076(I) + 0.002(E) + 0.495(P) = 0.573 SEC.

```
discriminant$TT
```

(9) $4096x^{10} - 3312x^8 + 108x^6$

Time: 0.097(I) + 0.003(E) + 0.049(P) = 0.149 SEC.

Example 7: Grobner Basis Computations

Finding a Grobner basis for a set of polynomials is a powerful tool for solving many problems, such as finding the common roots of the polynomials or proving that two sets of polynomials generate the same ideal.

The system can find Grobner bases over any field. Here we illustrate this for the case of rational functions in a and b. The inputs have to be distributed polynomials, of which the system provides two types - DMPs and NDMPs. The only difference is that NDMPs order the monomials by total degree and then in reverse lexicographic order, whereas DMPs use plain lexicographic order. (It is interesting to note that this difference corresponds to one line of code in the implementation of the two types:

```
DistributedMultivariatePolynomial(vl,R): T == C
 where
  vl: List Expression
  R: Ring
  E == DirectProduct(#vl,NonNegativeInteger)
  T == GeneralPolynomial(R,E) with
         vmonom: Expression -> $
  C == PolRing(R,E) add
    Vec == Vector(NonNegativeInteger)
    vmonom(v:Expression):$ ==
       monom(Vec$[if z=v then 1 else 0 for z in vl]::E,1)

NewDistributedMultivariatePolynomial(vl,R): T == C
 where
  vl: List Expression
  R: Ring
  E == NewDirectProduct(#vl,NonNegativeInteger)
  T == GeneralPolynomial(R,E) with
         vmonom: Expression -> $
  C == PolRing(R,E) add
    Vec == Vector(NonNegativeInteger)
    vmonom(v:Expression):$ ==
       monom(Vec$[if z=v then 1 else 0 for z in vl]::E,1)
```

and that NewDirectProduct and DirectProduct only differ in the definition of the order:

```
x < y ==
  (nx, ny) :S :=0
  for i in 0..n repeat
    nx:=nx+x.i
    ny:=ny+y.i
  nx < ny => true
  ny < nx => false
  for i in 0..n repeat
    if x.i > y.i then return true
    if x.i < y.i then return false
  false
```

versus

```
x < y ==
  for i in 0..n repeat
    if x.i < y.i then return true
    if x.i > y.i then return false
    false
```

{Example computations: }

```
(p,q): QF P[a,b]I
```

{Let p and q be rational functions in a and b}

```
p:=(a+1)/b
```

$$(1)\quad \frac{a+1}{b}$$

Time: 0.52(I) + 0.022(E) + 0.041(P) = 0.583 SEC.

```
q:=(b+1)/a
```

$$(2)\quad \frac{b+1}{a}$$

Time: 0.399(I) + 0.015(E) + 0.035(P) = 0.449 SEC.

```
(u,v,w): DMP([x,y,z,t],QF P[a,b]I)

u:=p*y*x**2-1

          a + 1   2
   (3)   (-------)x y - 1
             b
                    Time: 0.869(I) + 0.031(E) + 0.068(P) = 0.968 SEC.

v:=y**3-q*z*x**2

          - b - 1   2     3
   (4)   (---------)x z + y
              a
                    Time: 1.064(I) + 0.031(E) + 0.073(P) = 1.168 SEC.

w:=z**3-12*t*x**2

              2     3
   (5)  - 12x t + z
                    Time: 1.032(I) + 0.024(E) + 0.052(P) = 1.108 SEC.

(uu,vv,ww): NDMP([x,y,z,t],QF P[a,b]I)

uu:=u

          a + 1   2
   (6)   (-------)x y - 1
             b
                    Time: 0.033(I) + 0.008(E) + 0.058(P) = 0.099 SEC.
```

{Notice that v and vv print differently since terms are ordered differently.}

```
vv:=v

         3     - b - 1   2
   (7)  y  + (---------)x z
                   a
                    Time: 0.028(I) + 0.005(E) + 0.067(P) = 0.1 SEC.

ww:=w

         3      2
   (8)  z  - 12x t
                    Time: 0.028(I) + 0.005(E) + 0.044(P) = 0.077 SEC.

grobner [u,v,w]

   (9)
                                                                    2
  2      - b     2       - a    3   2      - 1   3   4     - b  - b
 [x y + -------, x z + (-------)y , x t + (-----)z , y  + (----------)z,
        a + 1           b + 1              12                  2
                                                              a  + a

  3      - b - 1   4   2 2     - a*b - a - b - 1   7
 y t + (---------)z , y t  + (------------------)z ,
           12a                      144a*b

                                    2     2
    3     - 12b        3     - a b - a  - 2a*b - 2a - b - 1   10
 y*z  + (-------)t, y*t  + (-------------------------------)z  ,
          a + 1                                 2
                                          1728a*b

                                  3
  13                   - 20736a*b                    4
 z   + (-----------------------------------------)t ]
          3     3     2      2
        a b + a  + 3a b + 3a  + 3a*b + 3a + b + 1

                    Time: 0.097(I) + 2.173(E) + 0.995(P) = 3.265 SEC.
```

```
grobner [uu,vv,ww]
```

```
   (10)
     6                - a*b              2 2   4             - a*b         2
   [x t + (------------------------)y z , x z + (------------------)y ,
            12a*b + 12a + 12b + 12                 a*b + a + b + 1

     3       2     3      - b - 1   2     2       - b
    z  - 12x t, y  + (---------)x z, x y + -------]
                          a                 a + 1
```

```
                  Time: 0.181(I) + 0.636(E) + 0.435(P) = 1.252 SEC.
```

{As is normally the case, the NDMP ordering is substantially faster, and produces a more legible result, than the DMP ordering.}

Example 8: Complex Root Package

We use the Grobner Basis construction to show how one can reduce the task of calculating complex roots (to sufficiently high accuracy) to that of calculating real roots. We let z=x+iy in a polynomial in z, and then find f(z) = a(x,y) + i b(x,y), for some polynomials a and b. A root of f corresponds to the polynomials a and b vanishing simultaneously.

Thus we construct the ideal generated by a and b. The Grobner basis construction guarantees that its last element is a polynomial in x alone. One may solve this, and substitute into the other generators. Finer algebraic analysis shows that the basis must always be in the form [y**2 + r(x),y*s(x),t(x)] where degree t = (n*(n+1))/2, degree s = (n*(n-1))/2 , and t is a multiple of s. We use various cyclotomic polynomials as examples.

```
X := CycloTomicPackage(z)

   (1)  CTP z
                        Time: 0(I) + 0(E) + 0.016(P) = 0.016 SEC.

Y := ComplexRootPackage(z,[y,x])

   (2)  CRP(z,[y,x])
                        Time: 0(I) + 0(E) + 0.036(P) = 0.036 SEC.

(p1,p2,p3): P[z]I

p1 := Cyclotomic$X  4      {Example 1}

         2
   (3)  z  + 1
                        Time: 1.678(I) + 0.014(E) + 0.034(P) = 1.726 SEC.
```

{complex roots are obtained by solving for the real roots of the following:}

```
separate$Y p1

          2    2          3
   (4)  [y  - x  - 1,y*x,x  + x]
                        Time: 3.104(I) + 0.044(E) + 0.1(P) = 3.248 SEC.

p2 := Cyclotomic$X  7    {Example 2}

         6    5    4    3    2
   (5)  z  + z  + z  + z  + z  + z + 1
                        Time: 0.126(I) + 0.006(E) + 0.047(P) = 0.179 SEC.

separate$Y p2

   (6)
         [
                2      267386880    20      802160640    19      1443889152    18
               y  + (-----------)x   + (-----------)x   + (------------)x
                       61103                61103                61103
                        2005401600    17      2364162048    16      2483355648    15
                     (------------)x   + (------------)x   + (------------)x
                         61103                61103                61103
                        2393112576    14      2009579520    13      1407959040    12
                     (------------)x   + (------------)x   + (------------)x
                         61103                61103                61103
                        747429888    11      187379712    10      - 169123072    9
                     (-----------)x   + (-----------)x   + (-------------)x
                         61103                61103                61103
                        - 289451520    8      - 199260672    7      - 83124288    6
                     (-------------)x  + (-------------)x  + (------------)x
                          61103                61103                61103
                        - 16277568    5      2523072    4      1060800    3
                     (------------)x  + (---------)x  + (---------)x
                         61103                61103                61103
                        - 1196864    2      - 62016        - 9792
                     (-----------)x  + (---------)x + --------
                         61103                61103         61103
```

```
   15     5     14     7     13     7     12     21     11
y*x   + (---)y*x   + (---)y*x   + (---)y*x   + (----)y*x
          2            2            2            8
           21     10      - 37     8      - 33     7      - 63     6
(----)y*x   + (------)y*x  + (------)y*x  + (------)y*x
  16           64            64           256
          - 7     5      7      4      7      3      - 3
(-----)y*x  + (------)y*x  + (------)y*x  + (-------)y*x
  128          1024          2048          16384
         - 1
(-------)y
  32768
```

```
 21     7   20      19     21    18     105    17     231    16
x   + (---)x   + 7x   + (----)x   + (-----)x   + (-----)x
        2             2             8             16
          231    15     823    14     315    13     1561   12     651    11
(-----)x   + (-----)x   + (-----)x   + (------)x   + (-----)x
  16           64            32           256           256
         - 77    10     - 2835   9     - 2835   8     - 13211   7
(------)x   + (--------)x  + (--------)x  + (---------)x
  1024          2048          2048          16384
         - 9527   6     - 1463   5      329    4      105    3
(--------)x  + (-------)x  + (-------)x  + (-------)x
  32768         32768         32768         32768
          - 7     2      - 7          - 1
(-------)x  + (-------)x + -------
  32768         32768        32768
]
```

Time: 0.074(I) + 48.065(E) + 2.173(P) = 50.312 SEC.

p3 := Cyclotomic$X 9 **{Example 3}**

```
       6    3
(7)  z  + z  + 1
```

Time: 0.14(I) + 0.017(E) + 0.033(P) = 0.19 SEC.

separate$Y p3

```
(8)
      [
         2     12390760448   20     1728348160   17
        y  + (-------------)x   + (------------)x
                2367153              263017
                  10554818560   14     - 393204736   11     - 1474341440   8
        (-------------)x   + (-------------)x   + (--------------)x
            2367153              789051                789051
                   577290560   5     - 79037120   2
        (-----------)x  + (------------)x
           2367153             2367153
   ,

           15     1     12     - 13     9      7      6     - 25     3
        y*x   + (---)y*x   + (------)y*x  + (-----)y*x  + (------)y*x
                  4            32           128           4096
                    1
        (-------)y
          32768
   ,

           21     5   18     27   15     - 13   12     - 1465   9
        x   + (---)x   + (----)x   + (------)x   + (--------)x
                4            32           128           4096
                   1593    6     - 199   3      1
        (-------)x  + (-------)x  + -------
          32768         32768        32768
   ]
```

Time: 0.056(I) + 7.806(E) + 0.875(P) = 8.737 SEC.

Example 9: Rational Function Integration

Integration in finite terms has been one of the big applications of Computer Algebra: recent discoveries allow the closed form solution of many of the integrals which appear in practice. Presently, Scratchpad II has algebra code available for the integration of rational functions. Facilities allowing integration of expressions involving logarithms, exponentials, trigonometrics and square roots of quadratics are planned for early Fall, 1984, availability.

```
u: QF P[x]RN :=(x+1)**2/((x+1)**6+1)
```

$$(0)\quad \frac{x^2 + 2x + 1}{x^6 + 6x^5 + 15x^4 + 20x^3 + 15x^2 + 6x + 2}$$

Time: 0(I) + 0(E) + 16.305(P) = 16.305 SEC.

```
integrate(u,x)
```

$$(1)\quad \sum_{\alpha^2 + \frac{1}{36} = 0} \alpha * \log(6\alpha + x^3 + 3x^2 + 3x + 1)$$

Time: 0(I) + 0(E) + 7.82(P) = 7.82 SEC.

```
u:=(2*x**2+4)**4/(x**2-2)**5
```

$$(2)\quad \frac{16x^8 + 128x^6 + 384x^4 + 512x^2 + 256}{x^{10} - 10x^8 + 40x^6 - 80x^4 + 80x^2 - 32}$$

Time: 0(I) + 0(E) + 1.495(P) = 1.495 SEC.

```
integrate(u,x)
```

$$(3)\quad \frac{-10x^7 - 12x^5 - 24x^3 - 80x}{x^8 - 8x^6 + 24x^4 - 32x^2 + 16} + \sum_{\alpha^2 + \frac{-9}{2} = 0} \alpha * \log(x*\alpha - 3)$$

Time: 0(I) + 0(E) + 1.399(P) = 1.399 SEC.

```
u:=x**5/(x**4+x**2+1)**2
```

$$(4)\quad \frac{x^5}{x^8 + 2x^6 + 3x^4 + 2x^2 + 1}$$

Time: 0(I) + 0(E) + 1.177(P) = 1.177 SEC.

```
integrate(u,x)
```

$$(5)\quad \frac{-x^2 + 1}{6x^4 + 6x^2 + 6} + \sum_{\alpha^2 + \frac{1}{27} = 0} \alpha * \log((x^2 + 2)\alpha + \frac{x^2}{3})$$

Time: 0(I) + 0(E) + 1.016(P) = 1.016 SEC.