MACSYMA REFERENCE MANUAL

MACSYMA Reference Manual

THE MATHLAB GROUP
LABORATORY FOR COMPUTER SCIENCE
MIT

VERSION NINE
Second Printing
December 1977

VERSION NINE

MACSYMA

Reference Manual

THE MATHLAB GROUP

LABORATORY FOR COMPUTER SCIENCE

MIT

VERSION NINE
Second Printing
December 1977

Preface

This manual maintains the format of Version Eight but contains corrections and updates. It corresponds to Version 267 of MACSYMA which has approximately 221,000 words of compiled code. Any comments, suggestions, or criticisms are welcome and should be addressed to:

Mathlab Group - Room 828
Laboratory for Computer Science
MIT
545 Technology Square
Cambridge, Mass. 02139

Acknowledgment

This manual was written by Richard Bogen with contributions by Jeffrey Golden (editor, storage management, debugging, and batch chapters), Michael Genesereth (relational data base, appendix on the grammar), and Alexander Doohovskoy (explicit tensor manipulation). We are extremely grateful to Pitts Jarvis for the many hours he spent in formatting it for the PUB program to use the Xerox Graphics Printer and to Ellen Lewis for her helpful assistance.

Note

A report, thesis, etc. which contains results that were obtained by using MACSYMA should acknowledge that the work of the Mathlab group is currently supported, in part, by the United States Energy Research and Development Administration under Contract Number E(11-1)-3070 and by the National Aeronautics and Space Administration under Grant NSG 1323. A copy should also be sent to Prof. J. Moses at the address given above in the Preface.

1 Introduction

MACSYMA (Project *MAC*'s *SY*mbolic *MA*nipulation System) is a large computer programming system written in LISP [Mn1] used for performing symbolic as well as numerical mathematical manipulations. It is being developed by the Mathlab Group of the MIT Laboratory for Computer Science (formerly Project MAC).

With MACSYMA the user can differentiate, integrate, take limits, solve systems of linear or polynomial equations, factor polynomials, expand functions in Laurent or Taylor series, solve differential equations (using direct or transform methods), compute Poisson series, plot curves, and manipulate matrices and tensors. MACSYMA has a language similar to ALGOL-60 to permit the user to write his own programs for transforming symbolic expressions.

This manual is intended to be a complete reference for the principal features of MACSYMA as of the date shown on the cover. It is not meant to be a tutorial nor does it discuss all of the issues involved in the efficient manipulation of algebraic expressions. New features under development are, for the most part, not mentioned.

The user who is unacquainted with certain concepts of computer programming may find this document difficult on first reading. The novice will benefit by first reading the *MACSYMA Primer* [Mo5] and *An Introduction to ITS for the MACSYMA User* [Lew1]. This document serves as both a reference manual and as a user's manual. When it is used as a user's manual, any sections not of interest should be skipped. Sections which may be passed over on first reading are indicated by the symbol [] around the section number.

It is recommended that this manual be re-read from time to time after the user has worked with MACSYMA so that certain parts which were unclear on prior readings will be better understood in the context of increased familiarity with the system.

In a programming system such as MACSYMA there are often many ways to go about solving a given problem as well as many constraints and frustrations which must be dealt with. Some ways will not succeed due to space or time constraints and others may work but may be unnecessarily slow. Frequently a better understanding of the computer facilities will lead to a reformulation of the problem lending itself to a much improved solution. For some insight into the subject see [A1, Mo1, Mo2].

1.1 Logging in and out

MACSYMA is implemented on DEC PDP-10 computers which use the ITS operating system and on Honeywell 6180 computers using the Multics operating system. The following section concerns the protocol for usage on the PDP-10 at MIT known as the MC (MACSYMA Consortium) machine. Appendix I of this manual concerns the use of MACSYMA on Multics.

A user may gain access to the PDP-10 computer at MIT by (1) finding a terminal connected to it or (2) finding one on which he can dial to it over public telephone lines or (3) logging into it over the ARPA network. Once this is done the user should be communicating with the ITS time sharing system. To login he should type a control-Z (depressing the control key while typing the letter Z) which loads in DDT [Lew1] (the top level system program). Then he should type :LOGIN followed by a space and his user name. (All system commands, i.e. those which begin with a colon, are terminated by a carriage return). At this point the user can start up any of several system programs available (PEEK, TECO, etc.) by typing a : followed by the name of the program. In particular :MACSYM loads in and begins execution of MACSYMA. After printing some descriptive information, including the version number, MACSYMA prints (C1) which means that it is ready to accept input from the user. The entire sequence is shown below with the computer's output indented. *^char* [1] means the control key is to be held down while the next character *char* is typed.

```
     MC ITS nnnn CONSOLE ii FREE [2]
^Z [3]
     MC ITS nnnn DDT mmm
:LOGIN SMITH
:MACSYM
     This is MACSYMA vvv
```

1. The user should note that there is a separate character, ^, which is often used for exponentiation (as in line (C1) below). Although the two characters print in the same way the user should have no difficulty distinguishing between them. In the example, the "^5" could not be "control-5" since the context strongly indicates exponentiation.

2. nnnn represents a version number which changes from time to time; ii identifies the console.

3. Not printed on console

```
     (C1) NROOTS(2*X^5-X+5,-4,8);

     (D1)                 1

^Z
44130)    .CALL 44154 (IOT)
:LOGOUT
     MC ITS nnnn CONSOLE ii FREE
```

In line (C1) the user has typed a command which asks for the number of realroots between -4 and 8 of a quintic polynomial. In line (D1) MACSYMA has printed the answer. The ^Z causes an exit from MACSYMA to DDT after which the system typed ".CALL 44154 (IOT)" (meaning that MACSYMA was waiting for input when interrupted). Typing :LOGOUT causes the system to delete all the user's jobs (in this case MACSYMA was the only job) and to log him out of ITS. When the user finishes he should always log the console out before he leaves.

1.2 General Information

Command lines to MACSYMA are strings of characters representing mathematical expressions involving equations, arrays, functions, and programs. Extra spaces, tabs, and all carriage returns are ignored (except when these occur in quoted strings).

Command lines are terminated by ";" or "$" (dollar sign). A ";" causes the command line to be evaluated and the result displayed. The terminator "$" causes the command line to be evaluated but the result is not displayed.

When typing command lines, depressing the "rubout" or "delete" key deletes the previous character (on hardcopy devices and displays which cannot backspace, the deleted character is echoed). By typing "control-K" , the user obtains a copy of the current command line free of any echoed erasures. The two characters ?? delete the whole command line, and cause the line number to be redisplayed.

The command (input) lines are indexed by labels of the form "(Ci)" where i is incremented with each new command line typed by the user. Similarly, the results of computations are also indexed. There are two types of output lines. The ordinary output line is indexed by a label of the form "(Di)" ; thus, usually the ith input-output pair will be (Ci)-(Di). Sometimes,however, a computation produces several intermediate results (for example, several solutions to an equation); it is convenient to be able to reference these intermediate lines of output. They are indexed by labels of the form "(Ej)" where j is incremented by one for each intermediate line. For example,

```
(C1)  SOLVE(X^2 + B*X + C, X);
                                        2
                               SQRT(B    - 4 C) + B
(E1)                      X = - -----------------
                                        2
                                         2
                               B - SQRT(B   - 4 C)
(E2)                      X = - -----------------
                                        2
(D2)                              [E1, E2]
```

Note that there is no line D1 since intermediate results were produced and thus the line index was incremented. The general pattern of indexing is of the form

Ci, Ei,Ei+1,...,Ej, Dj .

A command line may refer to the results of any previously indexed line (even if it was not displayed) through the use of the line labels. For example, the user might say SUBSTITUTE(7,B,E2); which would substitute 7 for B in the expression E2 above. The immediately preceding D-line is conveniently referenced by the symbol "%".

If the input line contains a syntax error, it will be reprinted and the location of the error will be indicated as closely as possible by a special string, ***$*** .

From a C-line, the user in need of assistance can type:

SEND*(message)* which will send your *message* to some MACSYMA system programmer who is logged in at that time. If no approriate recipient is available, the message will be sent to MACSYMA's mail file where it will be seen and will be answered. If the *message* consists of more than one word, then *message* must begin and end with double quotes.

BUG*(message)* sends a bug note to MACSYMA. If more than one word, *message* must begin and end with double quotes.

MAIL*(message)* sends mail to MACSYMA using the syntax above.

DESCRIBE*(command)* takes as argument any MACSYMA *command* and prints out the relevant portion of the MACSYMA manual.

EXAMPLE*(command)* gives examples (in DEMO mode) of the use of selected functions in MACSYMA.

PRIMER*()* provides an on-line primer for the novice including an introduction to MACSYMA syntax, assignment and function definition, and the simplification commands.

1.3 Levels of Control

When :MACSYM is typed, a LISP system extended with MACSYMA programs is loaded into main memory from auxiliary storage. Special top level programs read in, evaluate, simplify, and display the user's expressions. All of the functions to be presented in subsequent sections are actually LISP programs which, when they are called, may invoke many other LISP programs in a process that is invisible to the user.

Switching between DDT, MACSYMA, and the LISP system in which it is embedded is accomplished by typing the following characters:

Control-^ typed while in MACSYMA causes LISP to be entered. The user can now type any LISP S-expression and have it evaluated. Typing (CONTINUE) or ^G (control-G) causes MACSYMA to be re-entered.

^Z causes an immediate exit to DDT. If one is already in DDT then ?? is printed. At this point the user can run some other program like PEEK or TECO [Lew1]. When in DDT, typing :CONTINUE causes the current job to be resumed. (:JOB MACSYM should be typed first if the user wishes to re-enter a MACSYMA which is not the current job).

1.4 Miscellaneous Information

Files in the ITS system have two names each of at most 6 characters. They are referenced by giving the two names as well as the device (default is DSK) and the directory name where the file resides (default is the same as the user's login name). A > sign may be used for the second file name and stands for the name which is the largest numerically if there exists a file with the given first name and a numeric second name; otherwise, the > sign represents the "greatest" name in an alphanumeric sense. A < sign may be similarly used for the "least" name.

Any of the four descriptors (1st name, 2nd name, device, user) may be omitted and either the default or the value given in a previous command (if there was one given) will be used.

For those without a disk directory of their own, the one named USERS is available. When placing a file on this directory the user should indicate in some manner (such as by the first file name) the name of the user who created it.

There is a special mail file on the ITS system for holding comments from users which are of general interest and for listing changes to MACSYMA which occur from time to time. The DDT command :MAIL MACSYM followed by a carriage return and text terminated by a control-C is used to place comments in this mail file. The user's login name and time of message are added automatically. (Control-D may be used to cancel this or any other DDT command prematurely). If the user encounters any bugs in MACSYMA then he should report these in MACSYMA mail. Mail may also be sent to other users by using the :MAIL command followed by the user's login name. If the user has received mail the message --MAIL-- will appear on his console after he logs in. Typing a space will cause the mail to be printed. The DDT command :PRMAIL MACSYM is used to print out the MACSYMA mail file. The user should do this occasionally to be informed of changes to the system and of other users' comments. :PRMAIL may also be used to print out any user's mail by following it with the user's login name. Control-S may be used to silence the printout. In addition, the command :PRINT MACSYM;UPDATE > may be used to print a file describing updates to MACSYMA since the last version of the manual. The update file should be checked regularly so the user can be informed of changes to MACSYMA.

For further information on DDT commands see [Lew1]. Typing :? will list the commands with a brief description. In particular, one command worth noting is :KILL which kills the current job.

An Introduction to ITS for the MACSYMA User [Lew1] and the *MACSYMA Primer* [Mo5] are very useful to the novice.

2 MACSYMA's Data Types and Statement Types

This section describes the kinds of expressions MACSYMA permits and their meanings. Chapters 5 and 6 should be referred to where necessary in order to clarify the examples presented. Default values of options are indicated in square brackets, [...].

2.1 Numbers

Numbers are integers, rational numbers, floating point numbers, or "bigfloats". Integers consist of a string of digits not containing a period; rational numbers are the quotient of two integers and are written as numerator/denominator; floating point numbers are written as in FORTRAN, i.e. strings of digits containing a period and optionally followed by an integer exponent beginning with the letter E; and bigfloats are written exactly like floating point numbers except using the letter B rather than E (the B must be included to indicate a bigfloat). Negative numbers begin with a minus sign. There is no limit on the number of digits in an integer or rational number but non-zero floating point numbers must have absolute value between .14E-38 and 1.7E38 and are limited to approximately 8 digits precision. This is the hardware limitation of the computer. Bigfloats may have any number of digits. The default precision is 16 but the user can change this by setting the FPPREC[16] to an integer representing the desired precision.

```
        -17253733574534     6.023E23  -1.6E-19

    37.5678349872508325688-98         3.14159      227

        -3354665557331/66724255465544    -.7B0
```

2.2 Names

Names are used to designate variables, functions, and arrays. A name consists of a string of letters (which may include %) and digits. It may also include other characters but these must be preceded with a \ when typed in. Names can be of any length and must begin with a letter (unless the leading character is a \). Lower case letters may be typed, but they are normally converted into the corresponding upper case letters.

```
%PI   EPSILON   X10Y30ISASTRANGENAME   \*SPECIAL
```

2.3 Quoted Strings

A string of characters of any length may be constructed by enclosing the string in quotation marks. To include a quotation mark, semicolon, or dollar sign in the string it is necessary to precede it with a \ when typed in. Quoted strings are useful as messages (such as those giving instructions for entering data) or in as descriptive titles for printed data.

Certain names that are reserved because of their function as keywords (operators or delimiters) are listed in Appendix II. If these are used out of their normal context they must be quoted.

```
"INPUT AMOUNT IN \$"    "RIEMANN'S \"ZETA\" FUNCTION"
```

2.4 Atomic Variables and Assignment

A name which may be assigned an arbitrary value is a variable. A variable might or might not be subscripted (see 2.6.2). A non-subscripted variable is designated as an "atomic variable". Atomic variables are assigned values by writing the name of the variable followed by a : followed by an expression representing the value to be assigned to the variable. A variable can be assigned a new value at any time. The value of a variable can be a number, a matrix, a list, or any MACSYMA expression. If a variable is not assigned a value then it just represents itself. There are many variables which have already been assigned values. These are called "MACSYMA" variables or options. They are provided in order to give the user some control over the way in which MACSYMA performs its operations. *The user should choose names other than these for his variables.* The index to this manual distinguishes these "MACSYMA" variables and options with a colon (":") between the variable and its default value. In the text of the manual, the default value is indicated with square brackets as in the next sentence. MYOPTIONS[[]] is a MACSYMA variable---an "infolist" (see 8.1)---which is a list of all the MACSYMA options ever reset by the user.

Some simple examples of assignment follow. (The comments in parentheses are only for the reader's benefit and are not actually typed to or by MACSYMA.)

Note that MACSYMA automatically assigns labels Ci to the user's input lines and Di to the output lines. These labels behave as assigned variables and can be referenced by the user.

```
line label     expression          comment

(C1)           A:16$               (integer)
(C2)           LAMBDA: -3/37$      (rational number)
(C3)           X:D1;               (X is assigned the value of D1)
(D3)                16
(C4)           RHO:SIGMA;           (since SIGMA has no value at this
                                   time RHO is assigned the symbol SIGMA)
(D4)                SIGMA
(C5)           SIGMA: .005$        (floating point)
(C6)           RHO;                (RHO still has its old value since
(D6)                SIGMA           it hasn't been reassigned a new one)
```

Since the value assigned may be any expression it may in particular be another assignment and therefore multiple assignments are permitted. Thus A:B:C:X+1 assigns X+1 to A, B, and C.

It is important to note that the expression assigned to the variable is not re-copied. Only a pointer to the expression is assigned. Thus in the above example, only one copy of X+1 is created.

The MACSYMA variable **VALUES** (see 8.1) gives a list (see 2.7) of all the user's atomic variables which have been bound (i.e. have been assigned values).

The assignment operator :: assigns the value of the expression on its right to the *value* of the quantity on its left, which must evaluate to an atomic variable or subscripted variable (sec. 2.6.2). Thus continuing with the above examples:

```
(C7)   RHO::LAMBDA$              (Note that the :: causes the value of
(C8)   SIGMA;                    LAMBDA, i.e. -3/37, to be assigned to
(D8)        -3/37                the value of RHO, i.e. SIGMA.)
(C9)   VALUES;
(D9)      [A, LAMBDA, X, RHO, SIGMA]
```

2.5 Mathematical Operators

Mathematical expressions are constructed by using the following operators and also functions (see 2.6). The usage and priorities from highest to lowest are:

Operator Name	Symbol	Usage
factorials	!! !	postfix
exponentiation	** or ^	infix
non-commutative exponentiation	^^	infix
non-commutative multiplication	.	infix
div′sn mult′pn	/ *	infix
negation	-	prefix
add′n subt′n	+ -	infix

If an operator is referred to out of context it must be enclosed in quotation marks.

EXPT is used to display exponentiation when the base or exponent become unwieldy.

! is the factorial which is the product of all the integers from 1 up to its argument. Thus 5! = 1*2*3*4*5 = 120. The value of the option FACTLIM[-1] gives the highest factorial which is automatically expanded. If it is -1 then all integers are expanded.

!! stands for double factorial which is defined as the product of all the consecutive odd (or even) integers from 1 (or 2) to the odd (or even) argument. Thus 8!! is 2*4*6*8 = 384.

Period is used for non-commutative product. It must be preceded and followed by a space when any ambiguity can arise with respect to floating point numbers. Non-commutative exponentiation is used in the usual sense that M^^2 means M . M.

Operators of equal priority are performed left to right. Parentheses can be used to change the order of evaluation. Also functional application has the highest priority. Thus SIN(A*X^Y/Z!)^2 means (SIN(A*(X^Y)/(Z!)))^2

The operands may be any MACSYMA expressions whose values are the correct types of data.. Note that every statement in MACSYMA yields a value even if the value is only a trivial one.

MACSYMA has no restriction on the mixing of modes of operands. Integers, rationals, floating point numbers, and bigfloats may be freely intermixed in an expression; when conversions are necessary, the priority of conversion is in the order of the types just mentioned. If floating point numbers or bigfloats of differing precision are combined in a operation, they will be converted to floating point or bigfloat numbers of the current precision by padding with zeroes or by dropping off low order digits and rounding.

Floating point underflow will return 0.0 unless the MACSYMA variable ZUNDERFLOW[TRUE] [1] is FALSE in which case an error will be signaled.

2.6 Functions and Arrays

2.6.1 Functions

A function is written as a name followed by the arguments to the function separated by commas and enclosed in parentheses. The arguments may be any MACSYMA expressions.

A function of a fixed number of arguments can be defined in MACSYMA by using the := operator. The left side of a function definition consists of the name of the function followed by the list of formal parameters enclosed in parentheses. The right side consists of the function body. When a function is called, the formal parameters will be bound to the actual arguments, any free variables in the function body will take on the values which they have at the time of the call, and the function body will be evaluated. It is permissible to define functions which are recursive to an arbitrary depth. Care should be taken when passing an expression which contains a variable with the same name as a formal parameter to a function defined with that formal parameter as circularity could result when it is evaluated (see 3.2).

The MACSYMA variable FUNCTIONS is a list of all user defined non-subscripted functions.

1. The square brackets enclose default options

The MACSYMA function **DISPFUN** may be used to display the definition of a function (see 10).

```
(C1) F(X):=X^2+Y$

(C2) F(2);

(D2)                Y + 4

(C3) Y:7$

(C4) F(2);
(D4)                  11

(C5) G(Y,Z):=F(Z)+3*Y;

(D5)         G(Y, Z) := F(Z) + 3 Y

(C6) G(2*Y+Z,-.5);

(D6)               3 (Z + 14) + Z + 14.25

(C7) FUNCTIONS;
(D7)            [F(X), G(Y, Z)]
```

The example involving the function G above requires some explanation. In going from C6 to D6 the following occurs:

(1) The arguments to G are evaluated giving Z+14 and -.5 (Y has the value 7).

(2) G is then invoked and has its formal parameters bound. Y to Z+14 (the first argument) and Z to -.5 (the second argument). The evaluation of G then causes F to be invoked on the argument -.5

(3) F has its formal parameter X bound to -.5 and returns the result of the evaluation X^2+Y with the current bindings which gives Z+14.25

(4) The evaluation of G continues with 3*Y which yields 3*(Z+14). This is added to the result from (3) and returned.

2.6.2 Arrays

Arrays enable one to refer to a collection of elements by using a single name. An element of an array is referred to by a subscripted variable which is a name followed by a list of subscripts enclosed in square brackets. Arrays in MACSYMA are of two types[1] - declared or undeclared. Declared arrays are similar to FORTRAN arrays. The user declares the number of dimensions and indicates the maximum value of each subscript. The system then allocates space for the entire array. To declare an array the user types:

ARRAY*(name, dim1, dim2, ..., dimk)*

This sets up a k-dimensional array. A maximum of five dimensions may be used. The subscripts for the *ith* dimension are the integers running from 0 to *dimi*. If the user assigns to a subscripted variable without declaring the corresponding array, an undeclared array is set up.

Undeclared arrays, otherwise known as hashed arrays (because hash coding is done on the subscripts), are more general than declared arrays. The user does not declare their maximum size, and they grow dynamically by hashing as more elements are assigned values. The subscripts of undeclared arrays need not even be numbers. However, unless an array is rather sparse, it is probably more efficient to declare it when possible than to leave it undeclared. The ARRAY function can be used to transform an undeclared array into a declared array.

Array elements can be assigned values explicitly with the : operator or implicitly by means of an associated function, and the values assigned may be any MACSYMA expression. To understand implicit assignment we must understand what MACSYMA does when asked to evaluate a subscripted variable. MACSYMA first evaluates the subscripts left to right. Then it does an array access to see if the requested array element already has a value. If it does, the value is returned. If it does not, MACSYMA checks to see whether the array has an associated function (see below). If not, the subscripted variable (with the subscripts evaluated) is returned. (This is standard MACSYMA practice - if there is no value for a variable, the variable itself is returned when an evaluation is done.) If there is an associated function, the parameters of the function are bound to the given

1. For efficient translation, the user can also inform MACSYMA of arrays all of whose elements are of a single type, e.g. INTEGER,BOOLEAN,FLOAT. (see 10.8).

subscripts, and the function body is evaluated. The value of the function call is stored in the appropriate array element and returned. Note that once an element is computed by the associated function it is stored so that next time it is needed it will not be recomputed. A consequence of this is that unless the user uses the KILL, REMVALUE, or REMARRAY functions (sec. 10.3) to kill an array element or the entire array, the associated function will never be called a second time on the same arguments. Thus the user should be aware that even if he redefines the associated function, those values which already exist will stay around. Of course individual array elements can be changed by assignment at any time.

These associated functions are defined with the := operator. Their definition looks exactly the same as ordinary function definitions, except that the parameters in the left side of the definition are enclosed in brackets instead of parentheses.

In order to use a subscripted variable as a single entity without it being an array and without ever assigning a value to it, it should be prefixed by an apostrophe to avoid it being confused with a non-subscripted variable of the same name. For example SUBST(0,W,W+'W[0]).

The MACSYMA variable ARRAYS is a list of all the arrays that have been allocated, both declared and undeclared.

DISPFUN (see 10.2) may be used to display the definition of an array associated function.

ARRAYINFO (see 8.1.1) may be used to find out whether an array is declared or undeclared, how large it is, how many subscripts it has, and which elements have values in the case of an undeclared array.

```
(C1) A[N]:=N*A[N-1]$

(C2) A[0]:1$

(C3) A[5];
(D3)                    120

(C4) A[N]:=N$

(C5) A[6];
(D5)                    6
```

(Note that the definition in C4 is being used because A[6] has no value up to this time.)

```
(C6) A[4];
(D6)                    24
```

(Since A[4] was assigned a value as a result of A[5] being computed, the new definition is not used.)

If one is going to define a recursive function which is to be called several times then if may be more efficient to use an array with an associated function for initialization. The reason is that once an element is computed it is stored and thus need not be computed again whereas with a non-subscripted function, each recursive call may cause a repeat of a past computation.

2.6.3 Lambda Notation

The LAMBDA notation is used for unnamed functions in order to indicate the correspondence between the variables of the function and the arguments which are to be substituted for them. It is useful when one desires to pass functional arguments to other functions or when one wants to apply a function just once without having to define it with :=.

```
(C1) F:LAMBDA([X,Y,Z],X^2+Y^2+Z^2);
                                              2     2     2
(D1)                   LAMBDA([X, Y, Z], Z  + Y  + X )

(C2) F(1,2,A);
                                       2
(D2)                                  A  + 5
```

MACSYMA also permits operators to be used in a functional notation; however, in order not to get a syntax error they must be surrounded by "s.

```
(C3) "+"(1,2,A);
(D3)                                  A + 3
```

[2.6.4] Subscripted Functions (Arrays of Functions)

It is possible for the value of an array element to be a lambda expression. Thus if the assignment F[1]:LAMBDA([X],X^2+1) were performed, then F[1] could be used in the ordinary prefix functional sense with its arguments following in parentheses, e.g. F[1](3) would yield the value 10. There is an alternative syntax available for assigning a lambda expression to an array which introduces the notion of a "subscripted function". In the above case one could also type F[1](X):=X^2+1 and this would be entirely equivalent. Other elements of the array could be assigned different lambda expressions (or any MACSYMA expressions). If there is an algorithm for computing the different functions to be stored in an array on the basis of the subscripts alone, then one may use an associated function. For example, F[K]:=LAMBDA([X],X^K+1). Again an alternative syntax of F[K](X):=X^K+1 may be used. The left side of the definition consists of the function name followed by the subscripts, enclosed in brackets, followed by the arguments, enclosed in parentheses. The subscripts (which are not evaluated at definition time) must be either all numeric or all symbolic. Note that subscripted functions are treated exactly like arrays so all of the information in sec. 2.6.2 applies. In particular when a subscripted function is referenced, the element is immediately retrieved and applied to its arguments if it exists; otherwise it is computed (this time only) and then applied. Consequently, two evaluations of the definition are performed. Thus consider the definition F[K](E):=COEFF(E,X,K) and the call F[2](3*X^2-1). Although the user may have thought that this would return the coefficient of X^2 in $3*X^2-1$, i.e. 3, it will return 0. The reason is that F[2] is first computed by evaluating the definition yielding 0, since E has not been bound at this time. Note that F[K](E):= SUBST(K,'J,'(COEFF(E,X,J))) would return the desired result as would F(K,E):=COEFF(E,X,K). Thus the user should be clear about the distinction between subscripted functions (a type of array) and ordinary functions. Also a subscripted function should not be redefined without KILL'ing or REMARRAY'ing it first; otherwise the elements which have already been stored will be used.

The ARRAYS[[]] list (see 2.6.2) also includes subscripted functions.

The function ARRAYINFO (see 8.1.1) may also be used on subscripted functions.

```
(C1) T[N](X):=RATSIMP(2*X*T[N-1](X)-T[N-2](X))$
```

This generates the Chebyshev polynomials.

```
(C2) T[0](X):=1$
```

```
(C3) T[1](X):=X$

(C4) T[4](Y);
                               4       2
(D4)                        8 Y  - 8 Y  + 1


(C5) G[N](X):=SUM(EV(X),I,N,N+2)$

(C6) H(N,X):=SUM(EV(X),I,N,N+2)$

(C7) G[2](I^2);
                               2
(D7)                        3 I

(C8) H(2,I^2);
(D8)                        29
```

The following illustrates a definition for the Legendre polynomials.

```
(C9) P[N](X):=RATSIMP(1/(2^N*N!)*DIFF((X^2-1)^N,X,N))$

(C10) Q(N,X):=RATSIMP(1/(2^N*N!)*DIFF((X^2-1)^N,X,N))$

(C11) P[2];
                                           2
                                        3 X  - 1
(D11)                       LAMBDA([X], --------)
                                           2
(C12) P[2](Y+1);
                                         2
                               3 (Y + 1)  - 1
(D12)                          --------------
                                     2
(C13) Q(2,Y+1);
                                  2
                               3 Y  + 6 Y + 2
(D13)                          --------------
                                     2
(C14) P[2](5);
(D14)                              37
```

```
(C15) Q(2,5);
5
attempt to differentiate wrt a number
```

[2.6.5] Additional Information About Functions

In order to pass a function as an argument to another function you need only give its name in the argument list of the call. It may then be used in the called function by following the name of the corresponding formal parameter with a parenthesized list of arguments. Subscripted functions (see 2.6.4) are passed by giving the name followed by the subscripts in brackets. Arrays can be passed by giving the name of the array in the argument list and they can be referenced by subscripting the corresponding formal parameter.

When passing names of functions or arrays one must take care that there is no atomic variable with the same name which is bound because then that value rather than the name will be passed. In this case the name should be preceded by a ' (see 3.2) to prevent it from being evaluated.

In order to assign to a formal parameter of a function so that the corresponding actual parameter gets changed (and remains changed) when the function is exited, then the :: operator rather than the : operator should be used.

```
(C7) F[I,J](X,Y):=X^I + Y^J;

                                I     J
(D7)          F     (X, Y) := X  +  Y
               I, J

(C8)  G(FUN,ARG1,ARG2):=PRINT(FUN," APPLIED TO ",ARG1," AND ",
                  ARG2," IS ",FUN(ARG1,ARG2))$

(C9)  G(F[2,1],SIN(%PI),2*A);

                    2
LAMBDA([X,Y],Y+X ) APPLIED TO 0 AND 2 A IS 2 A

(D9)                          2 A
```

2.7 Lists

Lists are ordered sets of elements which can be any MACSYMA expressions. They are written enclosed in brackets with elements separated by commas. If the value of a variable is a list, its elements may be obtained or assigned to by subscripting as with arrays. In certain cases lists are treated like vectors (row or column matrices). (see 2.8) Lists are sometimes used as arguments to MACSYMA functions (e.g. MATRIX, SOLVE, etc.). Chapter 8 describes functions for many list operations such as deleting elements, selecting an element, reversing a list, etc.

```
(C1)    [X^2,Y/3,-2]$

(C2)    X[1]*X;
                              3
(D2)                         X

(C3)    [A,D1,D2];
                                   2  Y         3
(D3)                         [A , [X , - , -2] , X ]
                                      3
```

2.8 Matrices

A matrix is a 2-dimensional ordered set of elements. It is represented internally using a list of lists all of the same length which stand for the rows of the matrix. Matrices may be constructed by using the function MATRIX whose arguments are lists representing the rows of the matrix. (The functions ENTERMATRIX and GENMATRIX may also be used to construct a matrix (see 6.4).)

The operators + , - , * , and / may be used between two matrices and take effect elementwise. (A matrix minus itself gives the zero matrix of the same size.) They may also be used between a scalar and a matrix and the scalar will be operated on with each element of the matrix. 1

1. In MACSYMA a scalar is an expression free of lists, matrices, and any atoms declared NONSCALAR.

Matrix multiplication is signified by using the dot operator (non-commutative product). Raising a matrix to a power (multiplying it by itself) is accomplished by use of the ^^ operator. That is, M.M is equivalent to M^^2. The inverse of a matrix may be obtained by using a negative exponent, i.e. M^^-1.

If the switch LISTARITH[TRUE] is TRUE then

1) Lists will behave arithmetically: they can be added to one another, etc.

2) In matrix operations they can be used as row or column vectors and will be converted to such when necessary.

An element of a matrix may be referenced by subscripting the matrix as with arrays but the same name should not be used to stand for both a matrix and an array.

There are many functions for operating on matrices as well as many options which can be set to give the user much flexibility and control over matrix operations (these are described in sec. 6.4). If a matrix is too wide to be displayed all at once, it is displayed column by column or as a list of lists.

```
(C1) M:MATRIX([A,0],[B,1]);

                                  [ A  0 ]
(D1)                              [      ]
                                  [ B  1 ]

(C2) M[1,1]*%;
                                 [  2     ]
(D2)                             [ A    0 ]
                                 [        ]
                                 [ A B  A ]

(C3) M*M;
                                 [  2    ]
                                 [ A   0 ]
(D3)                             [       ]
                                 [  2    ]
                                 [ B   1 ]

(C4) M.M;
```

```
                                [     2        ]
(D4)                            [    A      0  ]
                                [              ]
                                [ A B + B   1  ]

(C5) D2-D4+1;
                                [  1      1 ]
(D5)                            [           ]
                                [ 1 - B   A ]
(C6) M^-1;
DIVISION BY 0


(C7) M^^-1;
                                     [  1      ]
                                     [  -    0 ]
                                     [  A      ]
(D7)                                 [         ]
                                     [   B     ]
                                     [ - -   1 ]
                                     [   A     ]

(C8) [X,Y].M;

(D8)                          [ B Y + A X  Y]
```

2.9 Equations

An equation is formed in MACSYMA simply by using an equal sign between any two expressions. Equations may be added or subtracted, and they may be multiplied or divided by any expression. They may be operated on just as any MACSYMA expression can be and may serve as arguments to functions.

```
(C1) X+1=Y^2$

(C2) X-1=2*Y+1$

(C3) D1+D2;
                                  2
(D3)                    2 X = Y  + 2 Y + 1
```

```
(C4) D1/Y;
                          X + 1
(D4)                      ----- = Y
                            Y

(C5) 1/%;
                            Y     1
(D5)                      ----- = -
                          X + 1   Y
```

[2.10] IF Statement

The IF statement is used for conditional execution. The syntax is

IF *condition* THEN *expression1* ELSE *expression2*.

The result of an IF statement is *expression1* if *condition* is true and *expression2* if it is false. *expression1* and *expression2* are any MACSYMA expressions (including nested IF statements), and *condition* is an expression which evaluates to TRUE or FALSE and is composed of relational and logical operators which are as follows:

Operator name	Symbol	Type
greater than	>	relational infix
equal to	= , EQUAL	" "
not equal to	#	" "
less than	<	" "
greater than or equal to	>=	" "
less than or equal to	<=	" "
and	AND	logical infix
or	OR	" "
not	NOT	logical prefix

The relational operators all have equal priorities which are less than the priorities of the arithmetic operators and greater than that of the logical operators. The priority of NOT is greater than that of AND which is greater than that of OR. The difference between "=" and EQUAL is discussed in sec. 7.1.

If the ELSE clause is omitted, this will be the same as if ELSE FALSE were specified. In order to have several expressions evaluated after the THEN or ELSE clauses, the expressions may be enclosed in a compound statement (see 2.11) but care should be taken to return the desired value. The switch PREDERROR[TRUE] determines the action taken if a clause is not universally true or universally false (see 7.1).

```
(C1)  FIB[N]:= IF N=1 OR N=2 THEN 1
              ELSE FIB[N-1]+FIB[N-2]$

(C2)  FIB[1]+FIB[2];
(D2)                      2
(C3)  FIB[3];
(D3)                      2
(C4)  FIB[5];
(D4)                      5

(C5)  ETA(MU,NU):= IF MU=NU THEN MU
               ELSE  IF  MU>NU  THEN MU-NU
               ELSE MU+NU$

(C6)  ETA(5,6);
(D6)                      11

(C7)  ETA(ETA(7,7),ETA(1,2));

(D7)                      4

(C8)   IF NOT 5>=2 AND 6<=5 OR 4+1>3 THEN A ELSE B;

(D8)                      A
```

[2.11] Compound Statements

In order to execute a sequence of statements in a context where a single statement is permitted then the user may group these statements into a *compound statement* by separating them with commas and enclosing the whole group in parentheses. The value of a compound statement is the value of the last statement in the group.

Compound statements are also useful for grouping together a sequence of related calculations when a computation cannot easily be expressed in a single MACSYMA statement.

```
(C1) IF X=Y THEN (X:X+1, Y:Y-1)
        ELSE (S:0, FOR I:1 THRU X DO (S:S+F(I), Y:Y-G(Y)))$
```

[2.12] Program Blocks

Blocks in MACSYMA are somewhat analogous to subroutines in FORTRAN or procedures in ALGOL or PL/I. Blocks are like compound statements but also enable the user to tag statements within the block and to assign "dummy" variables to values which are local to the block. The syntax is:

BLOCK*([v1, ... vk], statement1,..., statementj)*

where the *vi* are atomic variables which are local to the BLOCK and the *statements* are any MACSYMA expressions. If no variables are to be made local then the list may be omitted.

A block uses these local variables to avoid conflict with variables having the same names used outside of the block (i.e. global to the block). In this case, upon entry to the block, the global values are saved onto a stack and are inaccessible while the block is being executed. The local variables then are unbound so that they evaluate to themselves. They may be bound to arbitrary values within the block but when the block is exited the saved values are restored to these variables. The values created in the block for these local variables are lost. Where a variable is used within a block and is not in the list of local variables for that block it will be the same as the variable used outside of the block.

In order to save and restore other local properties besides VALUE, namely ARRAY (except for complete arrays - (see 10.8)), FUNCTION, DEPENDENCIES, ATVALUE, MATCHDECLARE, ATOMGRAD, CONSTANT, and NONSCALAR[1] (see 8.1), the function LOCAL should be used inside of the block with arguments being the names of the variables (see 10.6).

The value of the block is the value of the last statement or the value of the

1. All of these properties except for FUNCTION are related more closely to the use of the name as a variable rather than as a function

argument to the function RETURN which may be used to exit explicitly from the block. The function GO may be used to transfer control to the statement of the block that is tagged with the argument to GO. To tag a statement, precede it by an atomic argument as another statement in the BLOCK. For example: BLOCK([X],X:1,LOOP,X:X+1,...,GO(LOOP),...). The argument to GO may be any expression which evaluates to a tag. For example GO(IF X>Y THEN PLACE1 ELSE COMPUTEPLACE(X)). One cannot use GO to transfer to a tag in a BLOCK other than the one containing the GO.

Blocks typically appear on the right side of a function definition but can be used in other places as well.

```
(C1) HESSIAN(F):=BLOCK([DFXX,DFXY,DFXZ,DFYY,DFYZ,DFZZ],
      DFXX:DIFF(F,X,2),DFXY:DIFF(F,X,1,Y,1),
      DFXZ:DIFF(F,X,1,Z,1),DFYY:DIFF(F,Y,2),
      DFYZ:DIFF(F,Y,1,Z,1),DFZZ:DIFF(F,Z,2),
      DETERMINANT(MATRIX([DFXX,DFXY,DFXZ],[DFXY,DFYY,DFYZ],
               [DFXZ,DFYZ,DFZZ])))$

(C2) HESSIAN(X^3-3*A*X*Y*Z+Y^3);

                                3          2  3     2  3
(D2)                    - 54 A  X Y Z - 54 A  Y  - 54 A  X

(C3)  SUBST(1,Z,QUOTIENT(%,-54*A^2));

                                  3         3
(D3)                             X  + A Y X + Y
```

The above example computes the Hessian of a cubic curve (the Folium of Descartes) which turns out to be invariant under this transformation, i.e. the result is of the same form.

The example below illustrates the saving and restoring of values described at the beginning of this section.

```
(C4) F(X):=BLOCK([Y], LOCAL(A), Y:4, A[Y]:X, DISPLAY(A[Y]))$

(C5) Y:2$

(C6) A[Y+2]:0$
```

```
(C7) F(9);
                                A  = 9
                                 4

(D7)                            DONE

(C8) A[Y+2];
(D8)                              0
```

If LOCAL(A) had not been used, the value on line D8 would have been 9.

[2.13] The DO Statement

The DO statement is used for performing iteration. Due to its great generality the DO statement will be described in two parts. First the usual form will be given which is analogous to that used in several other programming languages (FORTRAN, ALGOL, PL/I, etc.); then the other features will be mentioned.

[2.13.1] Commonly Used Forms

There are three variants of this form that differ only in their terminating conditions. They are:

(a) FOR *variable : initial-value* STEP *increment*
THRU *limit* DO *body*

(b) FOR *variable : initial-value* STEP *increment*
WHILE *condition* DO *body*

(c) FOR *variable : initial-value* STEP *increment*
UNLESS *condition* DO *body*

(Alternatively, the STEP may be given after the termination condition or limit.)

The *initial-value, increment, limit,* and *body* can be any expressions. To iterate over several statements, the *body* can be made into a compound statement (see 2.11) or a BLOCK (see 2.12). The *condition* is as in the IF statement. If the *increment* is 1 then "STEP 1" may be omitted.

The execution of the DO statement proceeds by first assigning the *initial-value* to the *variable* (henceforth called the control-variable). Then: (1) If the control-variable has exceeded the *limit* of a THRU specification, or if the *condition* of the UNLESS is TRUE, or if the *condition* of the WHILE is FALSE then the DO terminates. (2) The *body* is evaluated. (3) The *increment* is added to the control-variable. The process from (1) to (3) is performed repeatedly until the termination condition is satisfied. One may also give several termination conditions in which case the DO terminates when any of them is satisfied.

In general the THRU test is satisfied when the control-variable is greater than the *limit* if the *increment* was non-negative, or when the control-variable is less than the *limit* if the *increment* was negative. The *increment* and *limit* may be non-numeric expressions as long as this inequality can be determined. However, unless the *increment* is known to be negative (i.e. is a negative number) at the time the DO statement is input, MACSYMA assumes it will be positive when the DO is executed. If it is not positive, then the DO may not terminate properly.

Note that the *limit*, *increment*, and termination *condition* are evaluated each time through the loop. Thus if any of these involve much computation, and yield a result that does not change during all the executions of the *body*, then it is more efficient to set a variable to their value prior to the DO and use this variable in the DO form.

The value normally returned by a DO statement is the atom DONE, as every statement in MACSYMA returns a value. However, the function RETURN (see 2.12) may be used inside the body to exit the DO prematurely and give it any desired value. Note however that a RETURN within a DO that occurs in a BLOCK will exit only the DO and not the BLOCK. Note also that the GO function may not be used to exit from a DO into a surrounding BLOCK.

The control-variable is always local to the DO and thus any variable may be used without affecting the value of a variable with the same name outside of the DO. The control-variable is unbound after the DO terminates.

```
(C1)    FOR A:-3 THRU 26 STEP 7 DO LDISPLAY(A)$
(E1)            A = -3
(E2)            A =  4
(E3)            A = 11
(E4)            A = 18
```

```
(E5)          A = 25
```

The function LDISPLAY generates intermediate labels; DISPLAY does not.

```
(C6)    S:0$
(C7)    FOR I:1 WHILE I<=10 DO S:S+I;
(D7)            DONE
(C8)    S;
(D8)            55
```

Note that the condition in C7 is equivalent to UNLESS I > 10 and also THRU 10

```
(C9)    SERIES:1$
(C10)   TERM:EXP(SIN(X))$
(C11)   FOR P:1 UNLESS P>7 DO
            (TERM:DIFF(TERM,X)/P,
            SERIES:SERIES+SUBST(X=0,TERM)*X^P)$
(C12)    SERIES;
                 7    6      5    4    2
(D12)           X    X      X    X    X
                -- - --- - -- - -- + -- + X + 1
                96   240    15   8    2
```

which gives 8 terms of the Taylor series for $e^{\sin(x)}$.

```
(C13) POLY:0$
(C14) FOR I:1 THRU 5 DO
          FOR J:I STEP -1 THRU 1 DO
              POLY:POLY+I*X^J$

(C15) POLY;
                5      4       3       2
(D15)       5 X  + 9 X  + 12 X  + 14 X  + 15 X

(C16) GUESS:-3.0$

(C17) FOR I:1 THRU 10 DO (GUESS:SUBST(GUESS,X,.5*(X+10/X)),
          IF ABS(GUESS^2-10)<.00005 THEN RETURN(GUESS));

(D17)                   - 3.1622807
```

This example computes the negative square root of 10 using the Newton-Raphson iteration a maximum of 10 times. Had the convergence criterion not been met the value returned would have been "DONE".

[2.13.2] Additional Forms of the DO Statement

Instead of always adding a quantity to the control-variable one may sometimes wish to change it in some other way for each iteration. In this case one may use "NEXT *expression*" instead of "STEP increment". This will cause the control-variable to be set to the result of evaluating *expression* each time through the loop.

```
(C1)  FOR COUNT:2 NEXT 3*COUNT THRU 20
         DO DISPLAY(COUNT)$

                        COUNT = 2
                        COUNT = 6
                        COUNT = 18
```

As an alternative to FOR variable:value ...DO... the syntax FOR variable FROM value ...DO... may be used. This permits the "FROM value" to be placed after the step or next value or after the termination condition. If "FROM value" is omitted then 1 is used as the initial value.

Sometimes one may be interested in performing an iteration where the control-variable is never actually used. It is thus permissible to give only the termination conditions omitting the initialization and updating information as in the following example to compute the square-root of 5 using a poor initial guess.

```
(C1) X:1000
(C2)  THRU 10 WHILE X#0.0 DO X:.5*(X+5.0/X)$
(C3) X;
(D3)                2.236068
```

If it is desired one may even omit the termination conditions entirely and just give "DO body" which will continue to evaluate the body indefinitely. In this case the function RETURN (see 2.11) should be used to terminate execution of the DO.

```
(C1) NEWTON(F,GUESS):=BLOCK([NUMER,Y],LOCAL(DF),NUMER:TRUE,
       DEFINE(DF(X),DIFF(F(X),X)),
        DO (Y:DF(GUESS), IF Y=0.0 THEN ERROR(
         "DERIVATIVE AT",GUESS," IS ZERO"),
        GUESS:GUESS-F(GUESS)/Y,
        IF ABS(F(GUESS))<5.0E-6 THEN RETURN(GUESS)))$

(C2) SQR(X):=X^2-5.0$

(C3) NEWTON(SQR,1000);
(D3)                        2.236068
```

(Note that RETURN, when executed, causes the current value of GUESS to be returned as the value of the DO. The BLOCK is exited and this value of the DO is returned as the value of the BLOCK because the DO is the last statement in the block.)

One other form of the DO is available in MACSYMA. The syntax is:

FOR *variable* IN *list [end-tests]* DO *body*

The members of the *list* (see 2.7) are any expressions which will successively be assigned to the variable on each iteration of the *body*. The optional *end-tests* can be used to terminate execution of the DO; otherwise it will terminate when the *list* is exhausted or when a RETURN is executed in the *body*. (In fact, *list* may be any non-atomic expression, and successive parts are taken.)

```
(C1)  FOR F IN [LOG, RHO, ATAN] DO LDISP(F(1.0))$

(E1)                                0

(E2)                              RHO(1)

                                   %PI
(E3)                               ---
                                    4

(C4) EV(E3,NUMER);

(D4)                           0.78539816
```

[2.14] Syntax Extension

It is possible to add new operators to MACSYMA (infix, prefix, postfix, unary, or matchfix with given precedences), to remove existing operators, or to redefine the precedence of existing operators. Details may be found in Appendix II.

3 What a Serious User Should Know

Usually the user need not be concerned with the internal workings of MACSYMA, but some knowledge of the representation of expressions and of the way in which they are evaluated, simplified, and displayed should be acquired in order to use MACSYMA more easily, efficiently, and effectively.

3.1 Representation

After an expression is read by MACSYMA it is automatically translated (i.e. lexically scanned and parsed) to a LISP "internal" form. This is the form in which MACSYMA's programs deal with expressions. Initially the translated expression is in "general" form but certain functions convert this to other forms.

(1) The **general form** represents non-atomic expressions as LISP lists whose first element is the main operator of the expression and whose remaining elements are the operands also represented in this form. Thus, after simplification, 2*X+3/4 is represented essentially [1] as (PLUS (RAT 3 4) (TIMES 2 X)). F(X)-LOG(X) is represented as (PLUS (F X) (TIMES -1 (LOG X))). Any expression which MACSYMA deals with can be represented in this form.

(2) **Canonical Rational Expressions** constitute a second kind of representation which is especially suitable for expanded polynomials and rational functions (as well as for partially factored polynomials and rational functions when RATFAC[FALSE] is set to TRUE, (see 6.5)). In this CRE **form** an ordering of variables (from most to least main) is assumed for each expression. Polynomials are represented recursively by a list consisting of the main variable followed by a series of pairs of expressions, one for each term of the polynomial. The first member of each pair is the exponent of the main variable in that term and the second member is the coefficient of that term which could be a number or a polynomial in another variable again represented in this form. Thus the principal part of the CRE form of 3*X^2-1 is (X 2 3 0 -1) and that of 2*X*Y+X-3 is (Y 1 (X 1 2) 0 (X 1 1 0 -3)) assuming Y is the main variable, and is (X 1 (Y 1 2 0 1) 0 -3) assuming X is the main variable. "Main"-ness is usually determined by reverse alphabetical order.

The "variables" of a CRE expression needn't be atomic. In fact any

1. Ignoring the flags MACSYMA places on operators

subexpression whose main operator is not + - * / or ^ with integer power will be considered a "variable" of the expression (in CRE form) in which it occurs. For example the CRE variables of the expression X+SIN(X+1)+2*SQRT(X)+1 are X, SQRT(X), and SIN(X+1). If the user does not specify an ordering of variables by using the RATVARS function (see 6.5) MACSYMA will choose an alphabetic one.

In general, CRE's represent rational expressions, that is, ratios of polynomials, where the numerator and denominator have no common factors, and the denominator is positive. The internal form is essentially a pair of polynomials (the numerator and denominator) preceded by the variable ordering list.

If an expression to be displayed is in CRE form or if it contains any subexpressions in CRE form, the symbol /R/ will follow the line label.

(3) An **extended CRE form** is used for the representation of Taylor series. The notion of a rational expression is extended so that the exponents of the variables can be positive or negative rational numbers rather than just positive integers and the coefficients can themselves be rational expressions as described above in (2) rather than just polynomials. These are represented internally by a recursive polynomial form which is similar to and is a generalization of CRE form, but carries additional information such as the degree of truncation.

As with CRE form, the symbol /T/ follows the line label of such expressions.

(4) When RATFAC[FALSE] is TRUE,expressions are brought into **partially factored form**: numerator and denominator are relatively prime products of recursively constructed primitive polynomial kernels. Kernels at the same level within numerator and denominator may not be relatively prime. In the future, kernels may be further specified to be square-free.

(5) Another internal form is used to represent Poisson series. This specialized representation of trigonometric series is described in section 6.6.

3.2 Evaluation

After MACSYMA parses a command line the expression is evaluated and simplified and the result is displayed. Often the two-phase process of evaluation and simplification is referred to simply as "evaluation." In this section though, we use the word "evaluation" to refer only to the evaluation stage proper and not to the simplification stage.

MACSYMA expressions consist of numbers, variables, function calls, and operators. When an expression is read by MACSYMA the parsing program translates it into LISP preserving the order and the result is the value of the current C line. The evaluation phase proceeds by building up an expression which is similar in form to the input expression, but has certain substitutions. The evaluator is recursive, and calls itself on all sub-expressions.

When the evaluator sees a name, it checks to see whether the name has a value assigned to it. If there is a value, that value is returned by the evaluator. If there is no value assigned to the name, the evaluator just returns the name itself. (For the means of assigning values to names refer to 2.4. For a description of the evaluation process as applied to subscripted names see 2.6.2). Note that problems could arise if a variable is bound to an expression containing an occurrence of that variable since each time the variable is evaluated, the entire expression is substituted for each occurrence of the variable. For example if Y has the value [X,Y,Z] and if the value of Y is evaluated the result is [X,[X,Y,Z],Z].

MACSYMA distinguishes between two types of functions - nouns and verbs. Most functions in the system, including all user-defined functions, are initially considered to be verb-type. Undefined functions and some system functions are considered to be noun-type. When the evaluator sees a function call, it evaluates the arguments to the function (unless that function is of a type which doesn't have its arguments evaluated, e.g. BATCH; for a list of such functions see Appendix VI). Then, if the function is verb-type, the evaluator applies the function to the evaluated arguments and returns the value of the function. For noun-type functions the evaluator returns an expression identical to the function call, except that the arguments are replaced by their evaluations.

The user can explicitly declare a name to be noun-type by using the DECLARE function (see 8.1.1). [1] For example, the function INTEGRATE normally tries to integrate its first argument. After the command DECLARE(INTEGRATE,NOUN) is given however, INTEGRATE will not perform the integration. Sometimes the user may give a verb function arguments which it is not equipped to deal with. In certain cases the verb function will return the noun form of itself. If this was because of some undefined functions in the expression,

1. The noun-ness applies to both the function use and array use of the name. Note that nounifying a function name F doesn't affect occurrences of F that existed before F was nounified.

which the user defines at a later time, he can cause the noun-form to be re-evaluated at that time by giving the label of that line followed by the name of the unevaluated function separated with a comma as arguments of the function EV (see 6.1). For example:

```
(C1) DIFF(X*F(X),X);

                          d
(D1)                   X (-- F(X)) + F(X)
                          dX

(C2) F(X):=SIN(X)$

(C3) EV(D1,DIFF);
(D3)                   SIN(X) + X COS(X)
```

Here we see that the expression returned by the evaluator is similar to the input expression. The basic difference is that names which have values are replaced by their values and verb-type function calls are replaced by the result of applying the function to its arguments.

MACSYMA has several special operators which give the user some control over the evaluation process. The single-quote operator ' has the effect of preventing evaluation. Thus an expression preceded by a single-quote evaluates to that expression. A special case is the evaluation of a function call where the name of the function is preceded by a quote as in 'F(X). In this instance the quote causes the function to be treated as though it were noun-type.

To simply prevent evaluation of F(X) without converting F to a noun, use '(F(X)).

The quote-quote operator, ' ' , causes an extra evaluation to occur. It is best considered as a macro character. Inputting an expression preceded by a quote-quote has exactly the same effect as inputting the result of evaluating and simplifying the expression. In other words when an inputted expression contains a sub-expression which begins with a quote-quote that sub-expression is replaced in the input string by the result of evaluating and simplifying the expression following the quote-quote. This occurs at the time an expression is parsed. In the case of evaluating a function call with a '' preceding the name of the function (i.e. ''F(x)), the '' causes the function to be treated as if it were verb-type.

```
(C1) X;
(D1)                    X
(C2) X:3$

(C3) X;
(D3)                    3

(C4) 'X;
(D4)                    X

(C5) F(X):=X^2;
                                       2
(D5)                           F(X) := X

(C6) 'F(2);
(D6)                             F(2)

(C7) EV(%,F);
(D7)                              4

(C8) '(F(2));
(D8)                             F(2)

(C9) ''%;
```

(''atom means evaluate the atom's value)

```
(D9)                              4

(C10) DECLARE(INTEGRATE,NOUN)$

(C11) INTEGRATE(Y^2,Y);
                        /
                       [   2
(D11)                   I Y DY
                        ]
                       /
```

```
(C12) ''INTEGRATE(Y^2,Y);

                              3
                             Y
(D12)                        --
                             3

(C13) F(Y):=DIFF(Y*LOG(Y),Y,2);

(D13)                     F(Y) := DIFF(Y LOG(Y), Y, 2)

(C14) F(Y):=''(DIFF(Y*LOG(Y),Y,2));
                                         1
(D14)                           F(Y) := -
                                         Y

(C15) C14;
                                         1
(D15)                           F(Y) := -
                                         Y
```

(Notice that the input expression has been changed due to the use of ' '.)

Referring to line (C14) above, suppose one wished to define the function F(Y) as DIFF(Y*LOG(Y),Y,I) within another function G(I) where the I in the definition of F(Y) is to be replaced by the argument to G when G is *called*.

G(I):=BLOCK(...,F(Y):=''(DIFF(Y*LOG(Y),Y,I)),...) will *not* do the job because the '' operator will cause the differentiation to be carried out at parse time and thus either an error will result (if I is unbound) or the current global value of I will be used rather than the value of the argument to G when it is called. Omitting the '' is also not desirable in this example because that would force the differentiation to be done each time F is called rather than at the time it is defined. To remedy this one may use the command

DEFINE*(function(arguments),body)*

which is like *function(arguments):=''body* but causes the evaluation of *body* to occur at the time DEFINE is evaluated. Thus G(I):=BLOCK(...,DEFINE(F(Y),DIFF(Y*LOG(Y),Y,I)),...) will work properly. DEFINE may also be used for subscripted functions.

3.3 Simplification

The simplifier takes the output of the evaluator and tries to make it smaller and more manageable, using some built-in algebra. Unless the user takes some special action (like setting the special variable SIMP to FALSE (see 6.1)), MACSYMA will never output an unsimplified expression. The simplifier re-orders expressions in order to obtain a standard form and the result is the value of the current D line. Thus A+B+C or C+A+B or C+B+A if input, will all result in the same internal form, (PLUS A B C) which displays as C + B + A . The simplifier also changes the SQRT function to exponentiation to the 1/2 power and removes the difference and quotient operators from the expression by converting X-Y to X+(-1)*Y and X/Y to X*Y^{-1}.

Roughly speaking, the simplifier orders expressions on the basis of their subexpressions being ordered first. Variables are ordered alphabetically i.e., from A to Z. Constants (%E, %PI, %I and any atoms DECLAREd CONSTANT) come before variables and numbers come before constants. Finally, functions are ordered according to their arguments[1], and according to their names in case their arguments are the same. Thus Y+2*A*X-%PI would become (PLUS (TIMES -1 %PI) (TIMES 2 A X) Y)).

The user should be aware that the line between evaluation and simplification is not clear-cut. For instance, SIN is a noun-function. When the evaluator sees SIN(0), it returns SIN(0). However, the simplifier notices this special case and changes this expression to 0. So simplification will sometimes obscure the difference between noun and verb functions.

[Mo1] mentions these and many other matters dealing with simplification.

1. Comparing first arguments first, second arguments second, etc.

4 Miscellaneous Hints and Facilities

Care should be taken in cases where an expression containing % is re-evaluated since the value of % changes each time a new line is computed. This is shown in the following example.

```
(C1) (X+Y)^3$
(C2) DIFF(%,X);

                                     2
(D2)                       3 (Y + X)
(C3)  Y:X^2+1$
(C4)  ''C2;
(D4)                      2 X
```

In line C4 the user may have intended to re-evaluate C2 thinking that the % still referred to D1 while it actually referred to D3. Note the use of the ' ' operator to re-evaluate a previous expression. (see 3.2)

The following interrupt characters typed *while holding down the control key* have special functions. They may be typed at any time--- even in the middle of a command line---and take effect immediately.

^ (control-shift-N on some terminals) - enters top-level LISP after resetting all locally bound variables and breaking out of all functions. It is not possible to continue an interrupted calculation after a control-^, but typing (CONTINUE) will return to MACSYMA.

A - makes a breakpoint in MACSYMA and suspends the computation. At this point the user is in a MACSYMA break loop. If a user function was being executed at the time of the break, its values may be printed or changed. Aside from this, it is almost like being at top-level MACSYMA. To exit and resume the computation type EXIT; (see 10.1).

X - quits a computation started while in a control-A break without quitting the top-level computation.

] - (control-shift-M on some terminals) prints the time used so far in a computation (without interrupting it).

K - reprints the current input line. This is useful when many rubouts have obscured the line (on hardcopy devices).

Y - gets the last command string.

L - clears the screen on display consoles and reprints the current line.

W - stops printout at the console while the computation continues. (If the user is connected to MACSYMA via the ARPA network, printout will not stop until the Arpanet buffer is emptied.) The switch TTYOFF[FALSE] if set to TRUE also stops the printout. This is useful for temporarily turning off the display for functions which might generate a lot of printing like BATCH. Setting the switch to FALSE causes printing to be resumed.

V - resumes printout at the console turned off by control-W.

G - aborts a computation and returns control to top-level MACSYMA. This is like control-^ immediately followed by (CONTINUE) and is useful for breaking out of infinite loops or for terminating a computation prematurely.

H - (backspace on some consoles) makes a "breakpoint" in MACSYMA, enters LISP, and prints the time used in the current computation. Control-H does not reset any values. Altmode (or Escape) P (for proceed) followed by a space will return to MACSYMA and resume the computation. Control-B also performs this function in NEWIO MACSYMA.

D - causes garbage collection statistics to be printed out each time a garbage collection takes place [Mn1]. See Chapter 16.

C - stops printout of garbage collection statistics turned on by control-D.

Two of the many MACSYMA variables or options mentioned later on are of special interest and will be described here.

(1) The value of LINEL gives the number of characters which are printed on a line. It is initially set by MACSYMA to the line length of the type of terminal being used (as far as is known) but may be reset at any time by the user. The user may have to reset it in DDT with :TCTYP as well. See [Lew1].

(2) If the variable SHOWTIME[FALSE] is TRUE then the computation time will be printed automatically with each output expression.

Sometimes when a user gives a command line the message "... being loaded" will be printed. This means that a function being used in the command line and/or the associated programs are not in the initially loaded MACSYMA but are being loaded in now via the dynamic loader. Infrequently used or inessential functions are not initially loaded into MACSYMA in an effort to save space.

When in LISP typing (CONTINUE) or control-G will return to MACSYMA.

MACSYMA provides the facility for the user to have an initialization file which gets loaded automatically before line (C1) is printed. If the user has a directory then the file should be named MACSYM (INIT). Otherwise he may place the file whose first file name is his login name and whose second file name is MACSYM on the directory called (INIT). This file must be in the format for the LOADFILE function (see 10.4), i.e. it must contain LISP code. It may be created via the SAVE function (see 10.4) or by translating a BATCH file (see 10.8).

A user who knows LISP should note that preceding a name with a ? causes the corresponding LISP atom to be invoked. For example, ?FIXP(4.2); returns FALSE, where FIXP is the name of a LISP system function.

5 Predefined Constants and Functions

5.1 Constants

A number of common mathematical constants have special names in MACSYMA;

%E - the base of the natural logarithms.

%PI - the transcendental number π.

%I - the square root of -1.

INF - real positive infinity.

MINF - real minus infinity.

INFINITY - complex infinity, an infinite magnitude of arbitrary phase angle.[1]

TRUE - the Boolean constant, true. (T in LISP)

FALSE - the Boolean constant, false. (NIL in LISP)

5.2 Functions

All of the functions mentioned below take one argument (shown as X) unless stated otherwise. The default values of MACSYMA variables which affect certain functions are given in brackets with the function.

1. The infinity symbols have meaning only for certain functions, for example, LIMIT, INTEGRATE, SUM.

5.2.1 Simple Functions

ABS*(X)* - absolute value of X

ABSBOXCHAR[!] is the character used to draw absolute value signs around expressions which are more than a single line high.

FLOAT*(exp)* - converts integers, rational numbers and bigfloats in *exp* to floating point numbers.

BFLOAT*(X)* - converts all numbers and functions of numbers to bigfloat numbers. Setting FPPREC[16] to N, sets the bigfloat precision to N digits. If FLOAT2BF[FALSE] is FALSE a warning message is printed when a floating point number is converted into a bigfloat number (since this may lead to loss of precision).

ENTIER*(X)* - largest integer $\leq$ X.

SIGNUM*(X)* - if X<0 then -1 else if X>0 then 1 else 0. If X is not numeric then a simplified but equivalent form is returned. For example, SIGNUM(-X) gives -SIGNUM(X).

POLYSIGN*(X)* - same as SIGNUM but always returns a numerical result by looking at the numerical factor of the highest degree term in X.

MIN*(X1, X2, ...)* yields the minimum of its arguments (or returns a simplified form if some of its arguments are non-numeric).

MAX*(X1, X2, ...)* yields the maximum of its arguments (or returns a simplified form if some of its arguments are non-numeric).

5.2.2 Miscellaneous Functions

SQRT*(X)* - the square root of X. It is represented internally by X^(1/2). Also see ROOTSCONTRACT in section 6.1.1.

RADPRODEXPAND[TRUE] - if TRUE will cause nth roots of factors of a product which are powers of n to be pulled outside of the radical, e.g. SQRT(16*X^2) will become 4*X only if RADPRODEXPAND is TRUE.

ISQRT*(X)* - takes one integer arg and returns the "integer SQRT" of its absolute value.

EXP*(X)* - the exponential function. It is represented internally as %E^X.

DEMOIVRE[FALSE] - if TRUE will cause %E^[A+B*%I] to become %E^A*(COS(B)+%I*SIN(B)) if B is free of %I. A and B are not expanded.

%EMODE[TRUE] - when TRUE %E^[%PI*%I*X] will be simplified as follows: it will become COS(%PI*X)+%I*SIN(%PI*X) if X is an integer or a multiple of 1/2, 1/3, 1/4, or 1/6 and thus will simplify further. For other numerical X it will become %E^[%PI*%I*Y] where Y is X-2*k for some integer k such that ABS(Y)<1. If %EMODE is FALSE no simplification of %E^[%PI*%I*X] will take place.

%ENUMER[FALSE] - when TRUE will cause %E to be converted into 2.718... whenever NUMER is TRUE. The default is that this conversion will take place only if the exponent in %E^X evaluates to a number.

LOG*(X)* - the natural logarithm.

LOGEXPAND[FALSE] - if TRUE will cause LOG(A/B) to become LOG(A)-LOG(B) and LOG(A*B) to become LOG(A)+LOG(B). This does not effect LOG(A^B) which always becomes B*LOG(A).

LOGSIMP[TRUE] - if FALSE then no simplification of %E to a power containing LOG's is done.

LOGNUMER[FALSE] - if TRUE then negative floating point arguments to LOG will always be converted to their absolute value before the log is taken. If NUMER is also TRUE, then negative integer arguments to LOG will also be converted to their absolute value.

The LOGCONTRACT command (see 6.1.1) "contracts" expressions containing LOG.

PLOG*(X)* - the principal branch of the complex-valued natural logarithm with -%PI < X $\leq$ +%PI .

GLOG*(X)* - the generalized logarithm, i.e. all branches. This is sometimes used by the definite integration package.

BINOMIAL*(X, Y)* - the binomial coefficient X*(X-1)*...*(X-Y+1)/Y!. If X and Y are integers, the binomial coefficient is actually computed. If Y or X-Y is an integer, the binomial coefficient is simplified to a polynomial.

RANDOM*(X)* - returns a random integer between 0 and X-1. If no argument is given then a random integer between -2^{35} and $2^{35}-1$ is returned. If X is FALSE then the random sequence is restarted from the beginning.

FIB*(X)* - the Xth Fibonacci number with FIB(0)=0, FIB(1)=1, and FIB(-N)=$(-1)^{(N+1)}$*FIB(N). PREVFIB is FIB(X-1), the Fibonacci number preceding the last one computed.

GENFACT*(X, Y, Z)* is the generalized factorial of X which is: X*(X-Z)*(X-2*Z)*...*(X-(Y-1)*Z). Thus, for integral X, GENFACT(X,X,1)=X! and GENFACT(X,X/2,2)=X!!

GAMMA*(X)* - the gamma function. GAMMA(I)=(I-1)! for I a positive integer.

GAMMALIM[1000000] controls simplification of the gamma function for integral and rational number arguments. If the absolute value of the argument is not greater than GAMMALIM, then simplification will occur. Note that the

FACTLIM switch (see 2.5) controls simplification of the result of GAMMA of an integer argument as well.

BETA*(X, Y)* - same as GAMMA(X)*GAMMA(Y)/GAMMA(X+Y)

ERF*(X)* - the error function, whose derivative is: 2*EXP(-X^2)/SQRT(%PI).

EULER*(X)* - gives the Xth Euler number for integer X.

BERN*(X)* - gives the Xth Bernoulli number for integer X.

ZEROBERN[TRUE] if set to FALSE excludes the zero BERNOULLI numbers.

ZETA*(X)* - gives the Riemann zeta function for certain integer values of X.

PSI*(X)* - derivative of LOG(GAMMA(X)).

5.2.3 Trigonometric Functions

This section outlines the way in which trigonometric functions are called in MACSYMA; for more information on the simplification of trigonometric functions and expressions, the user should read Section 2 of the MACSYMA Primer [Mo5].

Circular Functions

SIN, COS, TAN, COT, SEC, CSC

Inverse Circular Functions

ASIN, ACOS , ATAN , ACOT , ASEC , ACSC

ATAN2*(Y,X)* - yields the value of ATAN(Y/X) in the interval -%PI to %PI.

Hyperbolic Functions

SINH , COSH , TANH , COTH, SECH , CSCH

Inverse Hyperbolic Functions

ASINH , ACOSH ,ATANH , ACOTH , ASECH, ACSCH

TRIGSIGN[TRUE] - if TRUE permits simplification of negative arguments to trigonometric functions. E.g., SIN(-X) will become -SIN(X) only if TRIGSIGN is TRUE.

EXPONENTIALIZE[FALSE] - if TRUE will cause all circular and hyperbolic functions to be converted to exponential form.

LOGARC[FALSE] - if TRUE will cause the inverse circular and hyperbolic functions to be converted into logarithmic form

Examples

```
(C1) SIN(%PI/12)+TAN(%PI/6);


                                    %PI        1
(D1)                            SIN(---) + -------
                                    12     SQRT(3)
(C2) EV(%,NUMER);
(D2)            0.8361693

(C3) BETA(1/2,2/5);

                               2
           SQRT(%PI) GAMMA(-)
                               5
(D3)       ------------------
                        9
              GAMMA(--)
                        10

(C4) EV(%,NUMER);
(D4)               3.6790924
```

```
(C5) DIFF(ATANH(SQRT(X)),X);

                           1
(D5)            -----------------
                2 SQRT(X) (1 - X)

(C6) SOLVE(X^2+10^5*X+1);
SOLUTION

(E6)                  X =  - SQRT(2499999999) - 50000

(E7)                   X = SQRT(2499999999) - 50000

(D7)                              [E6, E7]

(C8) E7,NUMER;
(D8)                         X = 2.9296875E-3

(C9) BFLOAT(E7);
(D9)                    X = - .99999997473787528-5

(C10) FPPREC:25$

(C11) SIN(.5B0);
(D11)                 .479425538604203000273287980
```

The trigonometric simplification routines use declared information in some simple cases. Declarations about variables are used as follows, e.g.

```
(C5) DECLARE(J, INTEGER, E, EVEN, O, ODD)$

(C6) SIN(X + (J*E + 1/2)*%PI)$

(D6)                      COS(X)

(C7) SIN(X + (O + 1/2)*%PI);


(D7)                     - COS(X)
```

5.3 Complex Expressions

A complex expression is specified in MACSYMA by adding the real part of the expression to %I times the imaginary part. Thus the roots of the equation X^2-4*X+13=0 are 2+3*%I and 2-3*%I.

Examples

```
(C1) (SQRT(-4)+SQRT(2.25))^2;

                                 2
(D1)          (2 %I + 1.5)

(C2) EXPAND(%);

              6.0 %I - 1.75

(C3) EXPAND(SQRT(2*%I));

(D3)              %I + 1
```

Note that simplification of products of complex expressions can be effected by expanding the product. Simplification of quotients, roots, and other functions of complex expressions can usually be accomplished by using the REALPART, IMAGPART, RECTFORM, POLARFORM, ABS, CARG functions (see 6.2.3).

6 MACSYMA Functions and Variables

Following is a list of all MACSYMA functions divided into functional classes. MACSYMA variables which affect the operation of some functions are described under the appropriate function with their default value in brackets. These are sometimes referred to as MACSYMA options.

6.1 General Purpose Functions

6.1.1 Evaluation and Simplification Functions

EV*(exp, arg1, ..., argn)* is one of MACSYMA's most powerful and versatile commands. It evaluates the expression *exp* in the environment specified by the *argi*. This is done in steps, as follows:

(1) First the environment is set up by scanning the *argi* which may be as follows:

SIMP causes *exp* to be simplified regardless of the setting of the switch SIMP which inhibits simplification if FALSE.

NOEVAL suppresses the evaluation phase of EV (see step (4) below). This is useful in conjunction with the other switches and in causing *exp* to be resimplified without being reevaluated.

EVAL causes an extra post-evaluation of *exp* to occur. (See step (5) below.)

INFEVAL leads to an "infinite evaluation" mode. EV repeatedly evaluates an expression until it stops changing. To prevent a variable, say X, from being evaluated away in this mode, simply include X='X as an argument to EV. Of course expressions such as EV(X,X=X+1,INFEVAL); will generate an infinite loop. *CAVEAT EVALUATOR.*

EXPAND causes expansion.

EXPAND(*m,n*) causes expansion, setting the values of MAXPOSEX and MAXNEGEX to *m* and *n* respectively. (see the EXPAND function below)

DETOUT causes any matrix inverses computed in *exp* to have their determinant kept outside of the inverse rather than dividing through each element.

DIFF causes all differentiations indicated in *exp* to be performed. (see the DIFF function below.)

DERIVLIST(*var1,...,vark*) causes only differentiations with respect to the indicated variables.

FLOAT causes non-integral rational numbers to be converted to floating point.

NUMER causes some mathematical functions (including exponentiation) with numerical arguments to be evaluated in floating point (see 5.2.1). It causes variables in *exp* which have been given numervals (see 8.1.2) to be replaced by their values. It also sets the FLOAT switch on.

PRED causes predicates (expressions which evaluate to TRUE or FALSE) to be evaluated.

NOUNS converts all nouns occurring in *exp* to verbs.

E where E is an atom declared to be an EVFLAG (see 8.1.1) causes E to be bound to TRUE during the evaluation of *exp*.

V:expression (or alternatively *V=expression*) causes *V* to be bound to the value of *expression* during the evaluation of *exp*. Note that if *V* is a MACSYMA option, then *expression* is used for its value during the evaluation of *exp*. If more than one argument to EV is of this type then the binding is done in parallel. If *V* is a non-atomic expression then a substitution rather than a binding is performed.

E where E is a function name declared to be an EVFUN (see 8.1) causes E to be applied to *exp*.

Any other function names (e.g. SUM) cause evaluation of occurrences of those names in *exp* as though they were verbs (see 3.2).

In addition a function occurring in *exp* (say F(args)) may be defined locally for the purpose of this evaluation of *exp* by giving F(args):=body as an argument to EV.

If an atom not mentioned above or a subscripted variable or subscripted expression was given as an argument, it is evaluated and if the result is an equation or assignment then the indicated binding or substitution is performed. If the result is a list then the members of the list are treated as if they were additional arguments given to EV. This permits a list of equations to be given (e.g. [X=1, Y=A**2]) or a list of names of equations (e.g. [E1,E2] where E1 and E2 are equations) such as that returned by SOLVE. (see 6.3)

The *argi* of EV usually may be given in any order but since they are picked up left to right the order may influence the result. This is strictly true of substitution equations which are handled in sequence, left to right, and EVFUNS which are composed, e.g. EV(*exp*,RATSIMP,RECTFORM) is handled as RECTFORM(RATSIMP(*exp*)). The SIMP, NUMER, FLOAT, PRED, and INFEVAL switches may also be set locally in a block, or globally at the "top level" in MACSYMA so that they will remain in effect until being reset. Setting INFEVAL:TRUE locally will cause all evaluations occurring via explicit calls to EV to be done "infinitely".

If *exp* is in CRE form (see 3.1) then EV will return a result in CRE form provided the NUMER and FLOAT switches are both FALSE.

(2) During step (1), a list is made of the non-subscripted variables appearing on the left side of equations in the *argi* or in the value of some *argi* if the value is an equation. The variables (including subscripted variables) in the expression *exp* are replaced by their global values, except for those appearing in this list. Usually, *exp* is just a label or % (as in (C2) below), so this step simply retrieves the expression named by the label, so that EV may work on it.

(3) If any substitutions are indicated by the *argi*, they are carried out now.

(4) The resulting expression is then re-evaluated (unless one of the *argi* was NOEVAL) and simplified according the the *argi*. Note that any function calls in *exp* will be carried out after the variables in it are evaluated and that EV(F(X)) thus may behave like F(EV(X)).

(5) If one of the *argi* was EVAL, steps (3) and (4) are repeated.

Examples

```
(C1) SIN(X)+COS(Y)+(W+1)**2+'DIFF(SIN(W),W);
                                 d             2
(D1)  COS(Y) + SIN(X) + --SIN(W) + (W + 1)
                                 dW
(C2) EV(%,SIN,EXPAND,DIFF,X=2,Y=1);
                          2
(D2)       COS(W) + W  + 2 W + COS(1) + 1.90929742
```

An alternate "top level" syntax has been provided for EV, whereby one may just type in its arguments, without the EV(). That is, one may write simply *exp,arg1,...,argn*. (This is not permitted as part of another expression, i.e. in functions, blocks, etc.). *exp,RESCAN* is equivalent to EV(*exp*).

```
(C4) X+Y,X:A+Y,Y:2;
(D4)                Y + A + 2
```

(Notice the parallel binding process)

```
(C5) 2*X-3*Y=3$

(C6) -3*X+2*Y=-4$

(C7) SOLVE([D5,D6]);
solution
                                    1
(E7)                     Y =  - -
                                    5
                                 6
(E8)                     X = -
                                 5

(D8)                [E7, E8]

(C9) D6,D8;
(D9)                - 4 =  - 4

(C10) X+1/X > GAMMA(1/2);

                        1
(D10)               X + - > SQRT(%PI)
                        X
```

```
(C11) %,NUMER,X=1/2;

(D11)              2.5 > 1.7724539

(C12) %,PRED;
(D12)                     TRUE
```

UNKNOWN*(exp)* returns TRUE iff *exp* contains an operator or function not known to the built-in simplifier.

EXPAND*(exp)* causes products of sums and exponentiated sums to be multiplied out, numerators of rational expressions which are sums to be split into their respective terms, and multiplication (commutative and non-commutative) to be distributed over addition at all levels of *exp*. For polynomials one may wish use RATEXPAND which uses a more efficient algorithm (see below).

Terms in *exp* whose exponent is less than MAXNEGEX[1000] or greater than MAXPOSEX[1000] will not be EXPANDed. However,

EXPAND(*exp,p,n*) expands *exp*, using *p* for MAXPOSEX and *n* for MAXNEGEX. This helps the user control how much and what kinds of expansion are to take place.

EXPON[0] - the exponent of the largest negative power which is automatically expanded (independent of calls to EXPAND). For example if EXPON is 4 then (X+1)**(-5) will not be automatically expanded.

EXPOP[0] - the highest positive exponent which is automatically expanded. Thus (X+1)**3, when typed, will be automatically expanded only if EXPOP is greater than or equal to 3. If it is desired to have (X+1)**n expanded where n is greater than EXPOP then executing EXPAND((X+1)**n) will work only if MAXPOSEX is not less than n.

```
(C1) (1/(X+Y)**4-3/(Y+Z)**3)**2;

                          1          3     2
(D1)                 (-------- - --------)
                             4          3
                     (Y + X)    (Z + Y)
```

```
(C2) EXPAND(X,2,0);

                              6                 9          1
(D2)            - ----------------- + -------- + --------
                          4       3           6          8
                   (Y + X)  (Z + Y)    (Z + Y)    (Y + X)

(C3) EXPAND(A.(B+C.(D+E)+F));

(D3)            A . F + A . C . E + A . C . D + A . B
```

RATEXPAND*(exp)* expands *exp* by multiplying out products of sums and exponentiated sums, combining fractions over a common denominator, cancelling the greatest common divisor of the numerator and denominator, then splitting the numerator (if a sum) into its respective terms divided by the denominator. This is accomplished by converting *exp* to CRE form (see 3.1) and then back to general form.

RATEXPAND[FALSE] - if TRUE will cause CRE expressions to be fully expanded when they are converted back to general form or displayed, while if it is FALSE then they will be put into a recursive form. (see RATSIMP below)

RATDENOMDIVIDE[TRUE] - if FALSE will stop the splitting up of the terms of the numerator of RATEXPANDed expressions from occurring.

KEEPFLOAT[FALSE] if set to TRUE will prevent floating point numbers from being rationalized when expressions which contain them are converted to CRE form.

GCD[EZ] if FALSE will prevent the greatest common divisor from being taken when expressions are converted to CRE form. This will sometimes speed the calculation if gcds are not required. (cf. the function GCD in 6.5)

```
(C1) RATEXPAND((2*X-3*Y)**3);

                  3          2        2         3
(D1)        - 27 Y  + 54 X Y  - 36 X  Y + 8 X

(C2) (X-1)/(X+1)**2+1/(X-1);
```

```
                           X - 1       1
(D2)                      -------- + -----
                                 2   X - 1
                          (X + 1)

(C3) EXPAND(D2);

                       X            1            1
(D3)              ------------ - ------------ + -----
                   2              2             X - 1
                  X  + 2 X + 1   X  + 2 X + 1

(C4) RATEXPAND(D2);

                          2
                       2 X             2
(D4)               -------------- + --------------
                    3    2           3    2
                   X  + X  - X - 1  X  + X  - X - 1
```

RATSIMP*(exp)* "rationally" simplifies (similar to RATEXPAND) the expression *exp* and all of its subexpressions including the arguments to non-rational functions. The result is returned as the quotient of two polynomials in a recursive form, i.e. the coefficients of the main variable are polynomials in the other variables. Variables may, as in RATEXPAND, include non-rational functions (e.g. SIN(X**2+1)) but with RATSIMP, the arguments to non-rational functions are rationally simplified. Note that RATSIMP is affected by some of the variables which affect RATEXPAND.

RATSIMPEXPONS[FALSE] - if TRUE will cause exponents of expressions to be RATSIMPed automatically during simplification.

RATSIMP*(exp,v1,...,vn)* enables rational simplification with the specification of variable ordering as in RATVARS.

```
(C1) SIN(X/(X^2+X))=%E^((LOG(X)+1)**2-LOG(X)**2);


                                            2                  2
                       X          - LOG (X) + (LOG(X) + 1)
(D1)             SIN(------) = %E
                     2
                    X  + X

(C2) RATSIMP(%);
                                        1         2
(D2)                           SIN(-----) = %E X
                                      X + 1

(C3) ((X-1)**(3/2)-(X+1)*SQRT(X-1))/SQRT((X-1)*(X+1));

                         3/2
                 (X - 1)    - SQRT(X - 1) (X + 1)
(D3)             --------------------------------
                      SQRT(X - 1) SQRT(X + 1)

(C4) RATSIMP(%);
                                    2
(D4)                        - -----------
                               SQRT(X + 1)

(C5)  X**(A+1/A),RATSIMPEXPONS:TRUE;

                    2
                   A  + 1
                   ------
                     A
(D5)              X
```

RADCAN*(exp)* simplifies *exp*, which can contain logs, exponentials, and radicals, by converting it into a form which is canonical over a large class of expressions and a given ordering of variables; that is, all functionally equivalent forms are mapped into a unique form. For a somewhat larger class of expressions, RADCAN produces a regular form [Fa2]. Two equivalent expressions in this class will not necessarily have the same appearance, but their difference will be simplified by RADCAN to zero. For some expressions RADCAN can be quite time consuming. This is the cost of exploring certain relationships among the components of the expression for simplifications based on factoring and partial-fraction expansions of exponents.

RADPRODEXPAND[TRUE] when set to FALSE will inhibit certain transformations: RADCAN(SQRT(1-X)) will remain SQRT(1-X) and will not become %I SQRT(X-1). RADCAN(SQRT(X^2-2*X+1)) will remain SQRT(X^2-2*X + 1) and will not be transformed to X-1.

```
(C1) (LOG(X**2+X)-LOG(X))**A/LOG(X+1)**(A/2);

                          2          A
                   (LOG(X  + X) - LOG(X))
(D1)               ----------------------
                               A/2
                       LOG(X + 1)

(C2) RADCAN(%);
                             A/2
(D2)                 LOG(X + 1)

(C3) LOG(A**(2*X)+2*A**X+1)/LOG(A**X+1);

                  2 X      X
              LOG(A    + 2 A  + 1)
(D3)          --------------------
                        X
                  LOG(A  + 1)

(C4) RADCAN(%);
(D4)                  2

(C5) (%E**X-1)/(%E**(X/2)+1);

                       X
                     %E  - 1
(D5)                ---------
                      X/2
                    %E    + 1

(C6) RADCAN(%);
```

```
                              X/2
(D6)                        %E    - 1
```

COMBINE*(exp)* simplifies the sum exp by combining terms with the same denominator into a single term.

MULTTHRU*(exp)* multiplies a factor (which should be a sum) of *exp* by the other factors of *exp*. That is *exp* is f1*f2*...*fn where at least one factor, say fi, is a sum of terms. Each term in that sum is multiplied by the other factors in the product. (Namely all the factors except fi). MULTTHRU does not expand exponentiated sums. This function is the fastest way to distribute products (commutative or noncommutative) over sums. Since quotients are represented as products (see 3.3) MULTTHRU can be used to divide sums by products as well.

MULTTHRU*(exp1, exp2)* multiplies each term in *exp2* (which should be a sum or an equation) by *exp1*. If *exp1* is not itself a sum then this form is equivalent to MULTTHRU(*exp1*exp2*).

```
(C1) X/(X-Y)**2-1/(X-Y)-F(X)/(X-Y)**3;

                  1         X          F(X)
(D1)          - ----- + -------- - --------
                X - Y          2          3
                        (X - Y)    (X - Y)

(C2) MULTTHRU((X-Y)**3,%);

                          2
(D2)          - (X - Y)  + X (X - Y) - F(X)

(C3) RATEXPAND(D2);
                                2
(D3)                        - Y  + X Y - F(X)

(C4) ((A+B)**10*S**2+2*A*B*S+(A*B)**2)/(A*B*S**2);
```

```
                        10  2                 2  2
                   (B  + A )  S  + 2 A B S + A  B
(D4)               ------------------------------
                                     2
                               A B S

(C5) MULTTHRU(%);

                                          10
                          2   A B   (B  + A)
(D5)                      - + --- + -------
                          S    2      A B
                              S
```

(notice that (B+A)**10 is not expanded)

```
(C6) MULTTHRU(A.(B+C.(D+E)+F));

(D6)                A . F + A . (C . (E + D)) + A . B
```

(compare with similar example under EXPAND)

XTHRU*(exp)* combines all terms of *exp* (which should be a sum) over a common denominator without expanding products and exponentiated sums as RATSIMP does. XTHRU cancels common factors in the numerator and denominator of rational expressions but only if the factors are explicit. Sometimes it is better to use XTHRU before RATSIMPing an expression in order to cause explicit factors of the gcd of the numerator and denominator to be canceled thus simplifying the expression to be RATSIMPed.

```
(C1) ((X+2)**20-2*Y)/(X+Y)**20+(X+Y)**-19-X/(X+Y)**20;

                                                   20
                        1           X       (X + 2)   - 2 Y
(D1)               --------- - --------- + ---------------
                          19          20               20
                   (Y + X)     (Y + X)          (Y + X)

(C2) XTHRU(%);
```

```
                                  20
                           (X + 2)    - Y
(D2)                       ------------
                                     20
                              (Y + X)
```

PARTFRAC*(exp, var)* expands the expression *exp* in partial fractions with respect to the main variable, *var*. Each power of a different denominator will be represented by only a single term (i.e. the decomposition is not "complete"). The algorithm employed is based on the fact that the denominators of the partial fraction expansion (the factors of the original denominator) are relatively prime. The numerators can be written as linear combinations of denominators, and the expansion falls out.

```
(C1) 2/(X+2)-1/(X+1)-X/(X+1)**2$

(C2) RATSIMP(%);
                                   X
(D2)                  - -------------------
                         3      2
                        X  + 4 X  + 5 X + 2

(C3) PARTFRAC(%,X);
                        - 2 X - 1      2
(D3)                   ---------- + -----
                               2    X + 2
                        (X + 1)
```

FACTOR*(exp)* factors the expression *exp*, containing any number of variables or functions, into factors irreducible over the integers.

FACTOR*(exp, p)* factors *exp* over the field of integers with an element adjoined whose minimum polynomial is *p*.

FACTORFLAG[FALSE] if FALSE suppresses the factoring of integer factors of rational expressions.

DONTFACTOR may be set to a list of variables with respect to which factoring is not to occur. (It is initially empty). Factoring also will not take place with respect to any variables which are less important (using the variable ordering assumed for CRE form) than those on the DONTFACTOR list. (see 6.5)

SAVEFACTORS[FALSE] if TRUE causes the factors of an expression which is a product of factors to be saved by certain functions in order to speed up later factorizations of expressions containing some of the same factors.

BERLEFACT[TRUE] if FALSE then the Kronecker factoring algorithm will be used otherwise the Berlekamp algorithm, which is the default, will be used. (see [Be1, Wa4])

INTFACLIM[1000] is the largest divisor which will be tried when factoring a bignum integer. If set to FALSE (this is the case when the user calls FACTOR explicitly), or if the integer is a fixnum (i.e. fits in one machine word), complete factorization of the integer will be attempted. The user's setting of INTFACLIM is used for internal calls to FACTOR. Thus, INTFACLIM may be reset to prevent MACSYMA from taking an inordinately long time factoring large integers.

GCFACTOR*(n)* factors the gaussian integer n over the gaussians, i.e. numbers of the form a + b i where a and b are rational integers. Factors are normalized by making a and b non-negative.

```
(C1) FACTOR(2**63-1);

                                                     2
(D1)                   73 127 337 92737 649657 7

(C2) FACTOR(Z**2*(X+2*Y)-4*X-8*Y);

(D2)                   (2 Y + X) (Z - 2) (Z + 2)

(C3)  X**2*Y**2+2*X*Y**2+Y**2-X**2-2*X-1;

                     2  2         2    2     2
(D3)                X  Y  + 2 X Y  + Y  - X  - 2 X - 1

(C4) DONTFACTOR:[X]$
```

```
(C5) FACTOR(D3/36/(Y**2+2*Y+1));

                                 2
                             (X  + 2 X + 1) (Y - 1)
(D5)                         ----------------------
                                   36 (Y + 1)

(C6) FACTOR(%E**(3*X)+1);
                             X          2 X      X
(D6)                    (%E  + 1) (%E     - %E  + 1)

(C7) FACTOR(X**4+1,A**2-2);

                        2              2
(D7)                (X  + A X + 1) (X  - A X + 1)
```

When FACTOR is applied to integers, note that the value returned by FACTOR when used in other computations may not lead to a simplified result. Using D1 above, the user can check that D1 + 1; will not return $2^6 3$.

FACTORSUM*(exp)* tries to group terms in factors of *exp* which are sums into groups of terms such that their sum is factorable. It can recover the result of EXPAND((X+Y)^2+(Z+W)^2) but it can't recover EXPAND((X+1)^2+(X+Y)^2) because the terms have variables in common.

```
(C1) (X+1)*((U+V)^2+A*(W+Z)^2),EXPAND;

            2      2                             2       2
(D1) A X Z  + A Z  + 2 A W X Z + 2 A W Z + A W  X + V  X

                         2        2    2          2
           + 2 U V X + U  X + A W  + V  + 2 U V + U

(C2) FACTORSUM(%);
                                          2         2
(D2)                   (X + 1) (A (Z + W)  + (V + U) )
```

FACTOROUT*(exp,var1,var2,...)* rearranges the sum *exp* into a sum of terms of the form f(var1,var2,...)*g where g is a product of expressions not containing the vari's and f is factored.

Another technique of factoring complex expressions uses the function SCANMAP (see Chapter 8).

SQFR*(exp)* is similar to FACTOR except that the polynomial factors are "square-free." That is, they have factors only of degree one. This algorithm, which is also used by the first stage of FACTOR, utilizes the fact that a polynomial has in common with its n*th* derivative all its factors of degree > n. Thus by taking gcds with the polynomial of the derivatives with respect to each variable in the polynomial, all factors of degree > 1 can be found.

```
(C1) SQFR(4*X**4+4*X**3-3*X**2-4*X-1);

                                 2              2
(D1)                           (X  - 1) (2 X + 1)
```

GFACTOR*(exp)* factors the polynomial *exp* over the Gaussian integers (i. e. with SQRT(-1) = %I adjoined). This is like FACTOR(*exp*,A**2+1) where A is %I.

```
(C1)  GFACTOR(X**4-1);
(D1)         (X - 1) (X + 1) (X + %I) (X - %I)
```

GFACTORSUM*(exp)* is similar to FACTORSUM but applies GFACTOR instead of FACTOR.

IRREDUCIBLE*(exp)* returns *exp* flagged as being irreducible, i.e. it doesn't factor. *Exp* must be a sum. If FACTOR is ever called on an expression marked as irreducible it returns immediately. For example, if the value of H is a large expression which the user knows to be irreducible and the expression G*H is to be factored (where the value of G is arbitrary) then FACTOR(G*IRREDUCIBLE(H)) is faster than FACTOR(G*H).

PARTITION*(exp, var)* returns a list of two expressions. They are (1) the factors of *exp* (if it is a product) or the terms of *exp* (if it is a sum) which don't contain *var* and, (2) the factors or terms which do.

```
(C1) PARTITION(2*A*X*F(X),X);

(D1)                    [ 2 A , X F(X) ]

(C2) PARTITION(A+B,X);

(D2)                    [ A + B , 0 ]
```

LOGCONTRACT*(exp)* recursively scans an exp, transforming subexpressions of the form a1*LOG(b1) + a2*LOG(b2) + c into LOG(RATSIMP(b1^a1 * b2^a2)) + c

```
(C1) 2*(A*LOG(X) + 2*A*LOG(Y))$

(C2) LOGCONTRACT(%);
                                           2  4
(D3)                                A LOG(X  Y )
```

ROOTSCONTRACT*(exp)* converts products of roots into roots of products. For example, ROOTSCONTRACT(SQRT(X)*Y^(3/2)); gives SQRT(X*Y^3). Currently it only knows about rational number exponents whose denominators are 2, but extension to other roots will follow.

6.1.2 Sums and Products

SUM*(exp, ind, lo, hi)* performs a summation of the values of *exp* as the index *ind* varies from *lo* to *hi*. If the upper and lower limits differ by an integer then each term in the sum is evaluated and added together. Otherwise the summand is evaluated with the index of summation unbound and (if SIMPSUM [FALSE] is TRUE) the result is simplified. This simplification may sometimes be able to produce a closed form. If SIMPSUM is FALSE or if 'SUM is used, the value is a sum noun form which is a representation of the sigma notation used in mathematics.

Sums may be differentiated, added, subtracted, or multiplied with some automatic simplification being performed.

CAUCHYSUM[FALSE] when TRUE causes the Cauchy product to be used when multiplying sums together rather than the usual product. In the Cauchy product the index of the inner summation is a function of the index of the outer one rather than varying independently.

GENINDEX[I] is the alphabetic prefix used to generate the next variable of summation when necessary.

GENSUMNUM[0] is the numeric suffix used to generate the next variable of summation. If it is set to FALSE then the index will consist only of GENINDEX with no numeric suffix.

```
(C1) SIMPSUM:TRUE$

(C2) SUM(I**2+2**I,I,0,N);

                           3      2
                N + 1   2 N  + 3 N  + N
(D2)           2      + --------------- - 1
                               6

(C3) SUM(3**(-I),I,1,INF);

                                1
(D3)                            -
                                2

(C4) SUM(I^2,I,1,4)*SUM(1/I^2,I,1,INF);

                                              2
(D5)                                     5 %PI
```

NUSUM*(exp,var,low,high)* performs indefinite summation of *exp* with respect to *var* using a decision procedure due to R.W. Gosper. *exp* and the potential answer must be expressible as products of nth powers, factorials, binomials, and rational functions.

UNSUM*(fun,n)* is the first backward difference *fun(n) - fun(n-1)*.

```
(C1) G(P):=P*4^N/BINOMIAL(2*N,N);
                                               N
                                            P 4
(D1)                         G(P) := ----------------
                                     BINOMIAL(2 N, N)

(C2) G(N^4);
                                       4  N
                                      N  4
(D2)                              ----------------
                                  BINOMIAL(2 N, N)
(C3) NUSUM(D2,N,0,N);
                         4        3       2            N
          2 (N + 1) (63 N  + 112 N  + 18 N  - 22 N + 3) 4        2
(D3)      ---------------------------------------------------- - ------
                         693 BINOMIAL(2 N, N)                     3 11 7

(C4) UNSUM(%,N);
                                       4  N
                                      N  4
(D4)                              ----------------
                                  BINOMIAL(2 N, N)
```

PRODUCT*(exp, ind, lo, hi)* gives the product of the values of *exp* as the index *ind* varies from *lo* to *hi*. The evaluation is similar to that of SUM. No simplification of products is available at this time.

```
(C1)   PRODUCT(X+I*(I+1)/2,I,1,4);

(D1)              (X + 1) (X + 3) (X + 6) (X + 10)
```

6.1.3 Differentiation and Integration Functions

DIFF*(exp, v1, n1, v2, n2, ...)* differentiates *exp* with respect to each *vi*, *ni* times. If just the first derivative with respect to one variable is desired then the form DIFF(*exp,v*) may be used. If the noun form of the function is required (as, for example, when writing a differential equation), 'DIFF should be used and this will display in a two dimensional format.

DERIVABBREV[FALSE] if TRUE will cause derivatives to display as subscripts.

DIFF(*exp*) gives the "total differential", that is, the sum of the derivatives of *exp* with respect to each of its variables times the function DEL of the variable. No further simplification of DEL is offered.

```
(C1) DIFF(EXP(F(X)),X,2);
                                  2
                         F(X)   d                F(X)  d       2
(D1)                   %E     (--- F(X)) + %E      (-- F(X))
                                 2                     dX
                               dX


(C2) DERIVABBREV:TRUE$


(C3) 'INTEGRATE(F(X,Y),Y,G(X),H(X));
                                    H(X)
                                   /
                                   [
(D3)                               I     F(X, Y) dY
                                   ]
                                   /
                                    G(X)
```

```
(C4) DIFF(%,X);
          H(X)
         /
         [
(D4)     I     F(X, Y)  dY + F(X, H(X)) H(X)  - F(X, G(X)) G(X)
         ]            X                      X                 X
         /
          G(X)
```

DEPENDS*(funlist1,varlist1,funlist2,varlist2,...)* declares functional dependencies for variables to be used by DIFF. DEPENDS([F,G],[X,Y],[R,S],[U,V,W],U,T) informs DIFF that F and G depend on X and Y, that R and S depend on U,V, and W, and that U depends on T. The arguments to DEPENDS are evaluated. The variables in each *funlist* are declared to depend on all the variables in the next *varlist*.[1] A *funlist* can contain the name of an atomic variable or array. In the latter case, it is assumed that all the elements of the array depend on all the variables in the succeeding *varlist*. Initially, DIFF(F,X) is 0; executing DEPENDS(F,X) causes future differentiations of F with respect to X to give DF/DX or Y_X (if DERIVABBREV:TRUE).

```
(C1) DEPENDS([F,G],[X,Y],[R,S],[U,V,W],U,T);
(D1)           [F(X, Y), G(X, Y), R(U, V, W), S(U, V, W), U(T)]

(C2) DEPENDENCIES;
(D2)           [F(X, Y), G(X, Y), R(U, V, W), S(U, V, W), U(T)]
(C3) DIFF(R.S,U);
                                 dR          dS
(D3)                             -- . S + R . --
                                 dU          dU
```

Since MACSYMA knows the chain rule for symbolic derivatives, it takes advantage of the given dependencies as follows:

```
(C4) DIFF(R.S,T);
                             dR dU             dS dU
(D4)                        (-- --) . S + R . (-- --)
                             dU dT             dU dT
```

If we set

1. In this command, lists of length one can be typed in directly as atoms.

```
(C5) DERIVABBREV:TRUE;
(D5)                                TRUE
```

then re-executing the command C4, we obtain

```
(C6) ''C4;
(D6)                   (R  U ) . S + R . (S  U )
                         U  T               U  T
```

To eliminate a previously declared dependency, the REMOVE command can be used. For example, to say that R no longer depends on U as declared in C1, the user can type REMOVE(R,DEPENDENCY). This will eliminate all dependencies that may have been declared for R.

```
(C7) REMOVE(R,DEPENDENCY);
(D7)                                DONE

(C8) ''C4;
(D8)                           R . (S  U )
                                      U  T
```

CAVEAT: DIFF *is the only* MACSYMA *command which uses* DEPENDENCIES *information. The arguments to* INTEGRATE,LAPLACE,*etc. must be given their dependencies explicitly in the command, e.g.,* INTEGRATE(F(X),X).

GRADEF*(f(x1, ..., xn), g1, ..., gn)* defines the derivatives of the function *f* with respect to its n arguments. That is, d*f*/d*xi* = *gi*, etc. If fewer than n gradients, say i, are given, then they refer to the first i arguments of *f*. The *xi* are merely dummy variables as in function definition headers and are used to indicate the i*th* argument of *f*. All arguments to GRADEF except the first are evaluated so that if *g* is a defined function then it is invoked and the result is used.
Gradients are needed when, for example, a function is not known explicitly but its first derivatives are and it is desired to obtain higher order derivatives. GRADEF may also be used to redefine the derivatives of MACSYMA's predefined functions (e.g. GRADEF(SIN(X),SQRT(1-SIN(X)**2))). It is not permissible to use GRADEF on subscripted functions.

GRADEFS is a list of the functions which have been given gradients by use of the GRADEF command.

PRINTPROPS([*f1,f2,...*],GRADEF) (see 8.1.1) may be used to display the gradefs of the functions *f1,f2,..*

REMOVE([*f1,f2,...*],GRADEF) may be used to eliminate the GRADEF property from the functions *f1,f2,...*.

```
(C1) DEPENDS(Y,X)$

(C2) GRADEF(F(X,Y),X**2,G(X,Y))$

(C3) DIFF(F(X,Y),X);
                                        dY    2
(D3)                          G(X, Y) -- + X
                                        dX

(C4) GRADEF(J(N,Z), 'DIFF(J(N,Z),N),
RATSIMP(J(N-1,Z)-N/Z*J(N,Z)))$

(C5) DIFF(J(2,X),X,2);

                        2
               J(0, X) X  - 3 J(1, X) X + 6 J(2, X)
(D5)           ------------------------------------
                                 2
                                X
```

(The example above computes the second derivative of a Bessel function of order two. A subscripted function e.g. J[N], could not have been used because a gradient for it cannot be defined using GRADEF.)

GRADEF*(a,v,exp)* may be used to state that the derivative of the atomic variable *a* with respect to *v* is *exp*. This automatically does a DEPENDS(*a,v*). For examples, see example 2 of Appendix III.

PRINTPROPS([*a1,a2,...*],ATOMGRAD) (see 8.1.1) may be used to display the atomic gradient properties of *a1,a2,...*

REMOVE([*a1,a2,...*],ATOMGRAD) may be used to eliminate the ATOMGRAD property from *a1,a2,...* .

INTEGRATE*(exp, var)* integrates *exp* with respect to *var* or returns an integral expression (the noun form) if it cannot perform the integration. Roughly speaking three stages are used:

(1) INTEGRATE sees if the integrand is of the form F(G(X))*DIFF(G(X),X) by testing whether the derivative of some subexpression (i.e. G(X) in the above case) divides the integrand. If so it looks up F in a table of integrals and substitutes G(X) for X in the integral of F. This may make use of gradients in taking the derivative. (If an unknown function appears in the integrand it must be eliminated in this stage or else INTEGRATE will return the noun form of the integrand.)

(2) INTEGRATE tries to match the integrand to a form for which a specific method can be used, e.g. trigonometric substitutions.

(3) If the first two stages fail it uses the Risch algorithm. (see [Mo2, Mo4])

CAVEAT: INTEGRATE *knows only about explicit dependencies.*

INTEGRATE*(exp, var, low, high)* finds the definite integral of *exp* with respect to *var* from *low* to *high*. Several methods are used,including direct substitution in the indefinite integral and contour integration (see [Wa3]). Improper integrals may use the names INF for positive infinity and MINF for negative infinity. If an integral "form" is desired for manipulation (for example, an integral which cannot be computed until some numbers are substituted for some parameters), the noun form 'INTEGRATE may be used and this will display with an integral sign.
ABCONVTEST[FALSE] when TRUE causes INTEGRATE to test for absolute convergence.

The function LDEFINT uses LIMIT (see 6.1.3) to evaluate the indefinite integral at the lower and upper limits.

Sometimes during integration the user may be asked what the sign of an expression is. Suitable responses are POS; , ZERO; , or NEG; . (see 7.1)

```
(C1) INTEGRATE(SIN(X)**3,X);

                          3
                       COS (X)
(D1)                   ------- - COS(X)
                          3

(C2) INTEGRATE(X**A/(X+1)**(5/2),X,0,INF);
IS  A + 1  POSITIVE, NEGATIVE, OR ZERO?

POS;
IS  2 A - 3  POSITIVE, NEGATIVE, OR ZERO?

NEG;
                                   3
(D2)                BETA(A + 1, - - A)
                                   2

(C3) GRADEF(Q(X),SIN(X**2));
(D3)                               Q(X)

(C4) DIFF(LOG(Q(R(X))),X);

                                d             2
                               (-- R(X)) SIN(R (X))
                                dX
(D4)                           --------------------
                                    Q(R(X))

(C5) INTEGRATE(%,X);
(D5)                             LOG(Q(R(X)))
```

RISCH*(exp, var)* integrates *exp* with respect to *var* using the Risch algorithm. This currently handles the cases of nested exponentials and logarithms which the main part of INTEGRATE can't do. INTEGRATE will automatically apply RISCH if given these cases.

ERFFLAG[TRUE] - if FALSE prevents RISCH from introducing the ERF function in the answer if there were none in the integrand to begin with.

```
(C1) RISCH(X^2*ERF(X),X);

              2      2
           - X      X              3            2
        %E      (%E    SQRT(%PI) X  ERF(X) + X  + 1)
(D1)    -----------------------------------------
                       3 SQRT(%PI)

(C2) DIFF(%,X),RATSIMP;
                                        2
(D2)                                   X  ERF(X)
```

CHANGEVAR*(exp,f(x,y),y,x)* makes the change of variable given by f(x,y) = 0 in all integrals occurring in *exp* with integration with respect to x; y is the new variable.

```
(C1) 'INTEGRATE(%E**SQRT(A*Y),Y,0,4);

                       4
                      /
                      [    SQRT(A) SQRT(Y)
(D1)                  I (%E                ) DY
                      ]
                      /
                      0

(C2) CHANGEVAR(D1,Y-Z^2/A,Z,Y);

                             2 SQRT(A)
                            /
                            [              Z
                          2 I          Z %E   dZ
                            ]
                            /
                             0
(D4)                      --------------------
                                     A
```

LIMIT*(exp, var, val, dir)* finds the limit of *exp* as the real variable *var* approaches the value *val* from the direction *dir*. *Dir* may have the value PLUS for a limit from above, MINUS for a limit from below, or may be omitted (implying a two-sided limit is to be computed). For the method see [Wa3]. LIMIT uses the following special symbols: INF (positive infinity) and MINF (negative infinity). On output it may also use UND (undefined), IND (indefinite but bounded) and INFINITY (complex infinity).

LHOSPITALLIM[4] is the maximum number of times L'Hospital's rule is used in LIMIT. This prevents infinite looping in cases like LIMIT(COT(X)/CSC(X),X,0).

TLIMSWITCH[FALSE] when true will cause the limit package to use Taylor series when possible.

```
(C1) LIMIT(X*LOG(X),X,0,PLUS);

(D1)                    0

(C2) LIMIT((1+X)**(1/X),X,0);

(D2)                    %E

(C3) LIMIT(%E**X/X,X,INF);

(D3)                    INF

(C4) LIMIT(SIN(1/X),X,0);

(D4)                    IND
```

TLIMIT*(exp,var,val,dir)* is just the function LIMIT with TLIMSWITCH set to TRUE.

LDEFINT*(exp,var,low,high)* yields the definite integral of *exp* by using LIMIT to evaluate the indefinite integral of *exp* with respect to *var* at the upper limit *high* and at the lower limit *low*.

TLDEFINT*(exp,var,low,high)* is just LDEFINT with TLIMSWITCH set to TRUE.

RESIDUE*(exp, var, val)* computes the residue in the complex plane of the expression *exp* when the variable *var* assumes the value *val*. The residue is the coefficient of (*var-val*)**(-1) in the Laurent series for *exp*.

```
(C1) RESIDUE(S/(S**2+A**2),S,A*%I);

                                1
(D1)                            -
                                2

(C2) RESIDUE(SIN(A*X)/X**4,X,0);

                                3
                               A
(D2)                          - --
                               6
```

ODE2*(diffeq,depvar,indvar)* solves ordinary differential equations,*diffeq*, of first or second order. The dependent and independent variables are specified as the second and third arguments. When successful ODE2 returns either an explicit or implicit solution for the dependent variable. The symbol *%C* is used to represent the constant in the case of first order equations and *%K1,%K2* represent the constants for second order equations. If for some reason ODE2 cannot obtain a solution, it returns FALSE, sometimes printing an error message to the user.

```
(C1) X^2*'DIFF(Y,X) + 3*X*Y = SIN(X)/X;
                           2 dY            SIN(X)
(D1)                      X  -- + 3 X Y = ------
                             dX              X

(C2) ODE2(%,Y,X);


                                 %C - COS(X)
(D2)                         Y = -----------
                                      3
                                     X
```

6.2 Part Selection and Substitution

The functions in this section are used to extract or replace parts of expressions.

6.2.1 The Part Functions

The Part functions make it possible to reference or replace any part of any MACSYMA expression. A part of a displayed expression is referred to by a set of indices which are non-negative integers. For example, in exponentiation the base is considered part 1 and the exponent part 2. In a quotient the numerator is part 1 and the denominator part 2. In a sum or product the *ith* term or factor is part i. In any expression the main operator is part 0. For -X the 0th part is -, for A^B it is ^, for DIFF(F(X),X) it is DIFF, etc. Note that unary minus is considered an operator.

In MACSYMA the user has some control of the way in which expressions are displayed. The ordering of factors in a product or terms in a sum may be changed by the user (see 10.5, 6.5). The ordering of parts in the displayed form of an expression may differ from the ordering in the internal representation of the expression.

PART*(exp, n1, ..., nk)* deals with the displayed form of *exp*. It obtains the part of *exp* as specified by the indices *n1,...,nk*. First part *n1* of *exp* is obtained, then part *n2* of that, etc. The result is part *nk* of ... part *n2* of part *n1* of *exp*. Thus PART(Z+2*Y,2,1) yields 2. PART can be used to obtain an element of a list, a row of a matrix, etc.

```
(C1)  X+Y/Z**2;

                          Y
(D1)                     -- + X
                          2
                         Z
(C2) PART(D1,1,2,2);

(D2)                      2

(C3) 'INTEGRATE(F(X),X,A,B)+X;
```

```
                             B
                            /
                            [
(D3)                        I F(X)dX + X
                            ]
                            /
                            A

(C4) PART(%,1,1);
(D4)                         F(X)
```

INPART*(exp,n1,...,nk)* is similar to PART but works on the internal representation of the expression (see 3.3) rather than the displayed form and thus may be faster since no formatting is done. Care should be taken with respect to the order of subexpressions in sums and products (since the order of variables in the internal form is often different from that in the displayed form) and in dealing with unary minus, subtraction, and division (since these operators are removed from the expression). PART(X+Y,0) or INPART(X+Y,0) yield +, though in order to refer to the operator it must be enclosed in "s. For example ...IF INPART(D9,0)="+" THEN ...

```
(C1)  X+Y+W*Z;

(D1)                        W Z + Y + X

(C2)  INPART(D1,3,2);

(D2)                        Z
(C3)  PART(D1,1,2);

(D3)                        Z

(C4) 'LIMIT(F(X)**G(X+1),X,0,MINUS);
                                              G(X + 1)
(D4)                             LIMIT   F(X)
                                 X ->0-

(C5) INPART(%,1,2);
(D5)                                  G(X + 1)
```

DISPFORM*(exp)* returns the external representation of *exp* (wrt its main operator). This should be useful in conjunction with PART which also deals with the external representation. Suppose EXP is -A . Then the internal representation of EXP is "*"(-1,A), while the external representation is "-"(A). LENGTH(EXP) gives 2, while LENGTH(DISPFORM(EXP)) gives 1. MAP(F,EXP) gives F(-1)*F(A), while MAP(F,DISPFORM(EXP)) gives -F(A).
DISPFORM(*exp*,ALL) converts the entire expression (not just the top-level) to external format. For example, if EXP:SIN(SQRT(X)), then FREEOF(SQRT,EXP) and FREEOF(SQRT,DISPFORM(EXP)) give TRUE, while FREEOF(SQRT,DISPFORM(EXP,ALL)) gives FALSE.

NOUNIFY*(f)* returns the noun form of the function name *f*. This is needed if one wishes to refer to the name of a verb function as if it were a noun. Note that some verb functions will return their noun forms if they can't be evaluated for certain arguments. This is also the form returned if a function call is preceded by a quote.

```
(C6) IS(INPART(D4,0)=NOUNIFY(LIMIT));

(D6)                              TRUE
```

VERBIFY*(f)* returns the function name *f* in its verb form.

BOX*(exp)* returns *exp* enclosed in a box. The box is actually part of the expression. BOX(*exp,label*) encloses *exp* in a labeled box. *label* is a name which will be truncated in display if it is too long. Simplification will occur within and outside of a BOXed expression but simplifications which require interactions across the box boundary will not take place.

BOXCHAR["] - is the character used to draw the box in this and in the DPART and LPART functions.

DPART*(exp, n1, ..., nk)* selects the same subexpression as PART, but instead of just returning that subexpression as its value, it returns the whole expression with the selected subexpression displayed inside a box. The box is actually part of the expression.

```
(C1) DPART(X+Y/Z**2,1,2,1);

                          Y
(D1)                    ---- + X
                          2
                        *****
                        * Z *
                        *****
```

LPART*(label, exp, n1, ..., nk)* is similar to DPART but uses a labeled box. A labeled box is similar to the one produced by DPART but it has a name in the top line.

REMBOX*(exp, arg)* removes boxes from *exp* according to *arg*. If *arg* is UNLABELED then all unlabeled boxes are removed. If *arg* is the name of some label then only boxes with that label are removed. If *arg* is omitted then all boxes labeled and unlabeled are removed.

6.2.2 The Substitution Functions

SUBST*(a, b, c)* substitutes *a* for *b* in *c*. *b* must be an atom or a complete subexpression of *c*. For example, X+Y+Z is a complete subexpression of 2*(X+Y+Z)/W while X+Y is not. When *b* does not have these characteristics, one may sometimes use SUBSTPART or RATSUBST (see below). Alternatively, if *b* is of the form e/f then one could use SUBST(a*f,e,c) while if *b* is of the form e**(1/f) then one could use SUBST(a**f,e,c). The SUBST command also discerns the X^Y in X^(-Y) so that SUBST(A,SQRT(X),1/SQRT(X)) yields 1/A.

a and *b* may also be operators of an expression (enclosed in "s) or they may be function names. If one wishes to substitute for the independent variable in derivative forms then the AT function (see below) should be used.

SUBST(*eq1,exp*) or SUBST([*eq1,...,eqk*],*exp*) are other permissible forms. The *eqi* are equations indicating substitutions to be made. For each equation, the right side will be substituted for the left in the expression *exp*.

For expressions in CRE representation (see 3.1), SUBST, like many of MACSYMA's general simplification commands, works on the RATDISREPed form of the expression.

EXPTSUBST[FALSE] if TRUE permits substitutions such as Y for %E**X in %E**(A*X) to take place.

```
(C1) SUBST(A,X+Y,X+(X+Y)**2+Y);
                                        2
(D1)                        Y + X + A

(C2) SUBST(-%I,%I,A+B*%I);
(D2)                                A - %I B
```

(Note that C2 is one way of obtaining the complex conjugate of an expression.) The following examples illustrate the difference between substitution (as performed by SUBST) and binding (as performed by EV).

```
(C3) %PI*R,%PI:-%I;
%PI improper value assignment

(C4) SUBST(X=0,DIFF(SIN(X),X));

(D4)                              1

(C5) DIFF(SIN(X),X),X=0;
0
attempt to differentiate wrt a number


(C6) MATRIX([A,B],[C,D]);
                                  [ A  B ]
(D6)                              [      ]
                                  [ C  D ]

(C8) SUBST("[",MATRIX,%);
(D8)                          [[A, B], [C, D]]
```

RATSUBST*(a, b, c)* substitutes *a* for *b* in *c*. *b* may be a sum, product, power, etc. RATSUBST knows something of the meaning of expressions whereas SUBST does a purely syntactic substitution. Thus SUBST(A,X+Y,X+Y+Z) returns X+Y+Z whereas RATSUBST would return Z+A.

RADSUBSTFLAG[FALSE] if TRUE permits substitutions such as U for SQRT(X) in X.

```
(C1) RATSUBST(A,X*Y^2,X^4*Y^8+X^4*Y^3);

                      3     4
(D1)               A X  Y + A

(C2) 1 + COS(X) + COS(X)^2 + COS(X)^3 + COS(X)^4;
             4          3          2
(D2)      COS (X) + COS (X) + COS (X) + COS(X) + 1

(C3) RATSUBST(1-SIN(X)^2,COS(X)^2,%);
             4                    2            2
(D3)      SIN (X) + COS(X) (2 - SIN (X)) - 3 SIN (X) + 3
```

SUBSTPART*(x, exp, n1, ..., nk)* substitutes *x* for the subexpression picked out by the rest of the arguments as in PART. It returns the new value of *exp*. *x* may be some operator to be substituted for an operator of *exp*. In this case it is enclosed in "s.

```
(C1) 1/(X^2+2);
                                      1
(D1)                                ------
                                     2
                                    X  + 2

(C2) SUBSTPART(3/2,%,2,1,2);
                                      1
(D2)                               --------
                                    3/2
                                   X    + 2

(C3) A*X+F(B,Y);
(D3)                            A X + F(B, Y)
```

```
(C4) SUBSTPART("+",%,1,0);
(D4)                          X + F(B, Y) + A

(C5) X^2 + X + 1$

(C6) SUBSTPART("[",%,0);
                                    2
(D6)                             [X , X, 1]
```

SUBSTINPART(*x, exp, n1, ...)* is like SUBSTPART but works on the internal representation of *exp.*

```
(C1) X.'DIFF(F(X),X,2);
                                 2
                                d
(D1)                      X . (--- F(X))
                                  2
                                dX

(C2) SUBSTINPART(D^2,%,2);
                                    2
(D2)                           X . D

(C3) SUBSTINPART(F1,F[1](X+1),0);

(D3)                          F1(X + 1)
```

Additional Information

If the last argument to a Part function is a list of indices then several subexpressions are picked out, each one corresponding to an index of the list. Thus PART(X+Y+Z,[1,3]) is Z+X.

PIECE holds the last expression selected when using the Part functions. It is set during the execution of the function and thus may be referred to in the function itself as shown below.

If PARTSWITCH[FALSE] is set to TRUE then END is returned when a selected part of an expression doesn't exist, otherwise an error message is given.

```
(C1)  27*Y**3+54*X*Y**2+36*X**2*Y+Y+8*X**3+X+1;

            3          2       2             3
(D1)    27 Y  + 54 X Y  + 36 X  Y + Y + 8 X  + X + 1

(C2)  PART(D1,2,[1,3]);

                   2
(D2)         54 Y

(C3)  SQRT(PIECE/54);

(D3)           Y

(C4)  SUBSTPART(FACTOR(PIECE),D1,[1,2,3,5]);

                             3
(D4)          (3 Y + 2 X) + Y + X + 1

(C5) 1/X+Y/X-1/Z;
                                      1   Y   1
(D5)                                - - + - + -
                                      Z   X   X

(C6) SUBSTPART(XTHRU(PIECE),%,[2,3]);
```

```
                                 Y + 1   1
(D6)                             ----- - -
                                   X     Z
```

ATVALUE*(form, list, value)* enables the user to assign the boundary value *value* to *form* at the points specified by *list*.

```
(C1) ATVALUE(F(X,Y),[X=0,Y=1],A**2)$
```

The *form* must be a function, f(v1,v2,...) , or a derivative, DIFF(f(v1,v2,...),vi,ni,vj,nj,...) in which the functional arguments explicitly appear (ni is the order of differentiation with respect vi).

The *list* of equations determine the "boundary" at which the *value* is given; *list* may be a list of equations, as above, or a single equation, vi = exp.

The symbols @1, @2,... will be used to represent the functional variables v1,v2,... when atvalues are displayed.

PRINTPROPS([*f1, f2,...*], ATVALUE) will display the atvalues of the functions *f1,f2,...* as specified in previously given uses of the ATVALUE function. (see 8.1.1) If the list contains just one element then the element can be given without being in a list. If a first argument of ALL is given then atvalues for all functions which have them will be displayed.

AT*(exp, list)* will evaluate *exp* (which may be any expression) with the variables assuming the values as specified for them in the *list* of equations or the single equation similar to that given to the ATVALUE function. If a subexpression depends on any of the variables in *list* but it hasn't had an atvalue specified and it can't be evaluated then a noun form of the AT will be returned which will display in a two-dimensional form.

```
(C1) ATVALUE(F(X,Y),[X=0,Y=1],A**2);

                                  2
(D1)                             A

(C2) ATVALUE('DIFF(F(X,Y),X),X=0,Y+1);

(D2)                             @2 + 1
```

```
(C3) PRINTPROPS(ALL,ATVALUE);
                                   !
                       D           !
                      --- F(@1, @2)!         = @2 + 1
                      D@1          !
                                   !@1 = 0

                                          2
                              F(0, 1) = A

(D3)                                 DONE

(C4) DIFF(4*F(X,Y)**2-U(X,Y)**2,X);

                     d                             d
(D4)     8 F(X, Y) (-- F(X, Y)) - 2 U(X, Y) (-- U(X, Y))
                     dX                            dX

(C5) AT(%,[X=0,Y=1]);
                                                 !
                 2                d              !
(D5)        16 A   - 2 U(0, 1) ( -- U(X, Y)!                )
                                  dX             !
                                                 !X = 0, Y = 1
```

6.2.3 More Functions for Part Extraction

LISTOFVARS*(exp)* yields a list of the variables in *exp*.

LISTCONSTVARS[FALSE] if TRUE will cause LISTOFVARS to include %E, %PI, %I, and any variables declared constant (see 8.1) in the list it returns if they appear in *exp*. The default is to omit these.

```
(C1) LISTOFVARS(F(X[1]+Y)/G**(2+A));

(D1)                          [X[1], Y, A, G]
```

COEFF*(exp, v, n)* obtains the coefficient of *v**n* in *exp*. *n* may be omitted if it is 1. *v* may be an atom, or complete subexpression of *exp* e.g., X, SIN(X), A[I+1], X+Y, etc. (In the last case the expression (X+Y) should occur in *exp*). Sometimes it may be necessary to expand or factor *exp* in order to make v^n explicit. This is not done automatically by COEFF.

```
(C1) COEFF(2*A*TAN(X)+TAN(X)+B=5*TAN(X)+3,TAN(X));

(D1)                         2 A + 1 = 5

(C2) COEFF(Y+X*%E**X+1,X,0);
(D2)                            Y + 1
```

RATCOEF*(exp, v, n)* returns the coefficient, C, of the expression *v**n* in the expression *exp*. *n* may be omitted if it is 1. C will be free (except possibly in a non-rational sense) of the variables in *v*. If no coefficient of this type exists, zero will be returned. RATCOEF expands and rationally simplifies its first argument and thus it may produce answers different from those of COEFF which is purely syntactic. Thus RATCOEF((X+1)/Y+X,X) returns (Y+1)/Y whereas COEFF returns 1. RATCOEF(*exp,v*,0) is currently the same as RATSUBST(0,*v,exp*). Therefore if *v* occurs to any negative powers, RATCOEF should not be used. Since *exp* is rationally simplified before it is examined, coefficients may not appear quite the way they were envisioned.

```
(C1) S:A*X+B*X+5$

(C2) RATCOEF(S,A+B);

(D2)                            X
```

BOTHCOEF*(exp, var)* returns a list whose first member is the coefficient of *var* in *exp* (as found by RATCOEF if *exp* is in CRE form otherwise by COEFF) and whose second member is the remaining part of *exp*. That is, [A,B] where *exp*=A* *var*+B.

```
(C1) ISLINEAR(EXP,VAR):=BLOCK([C],
        C:BOTHCOEF(RAT(EXP,VAR),VAR),
        IS(FREEOF(VAR,C) AND C[1]#0))$

(C2) ISLINEAR((R**2-(X-R)**2)/X,X);

(D2)                                TRUE
```

ISOLATE*(exp, var)* returns *exp* with subexpressions which are sums and which do not contain *var* replaced by intermediate expression labels (these being atomic symbols like E1, E2, ...). This is often useful to avoid unnecessary expansion of subexpressions which don't contain the variable of interest. Since the intermediate labels are bound to the subexpressions they can all be substituted back by evaluating the expression in which they occur.

EXPTISOLATE[FALSE] if TRUE will cause ISOLATE to examine exponents of atoms (like %E) which contain *var*.

```
(C1) (A+B)^4*(1+X*(2*X+(C+D)^2));

                               4              2
(D1)                    (B + A)  (X (2 X + (D + C) ) + 1)

(C2) ISOLATE(%,X);

                                            2
(E2)                                 (D + C)

                                            4
(E3)                                 (B + A)

(D3)                          E3 (X (2 X + E2) + 1)

(C4) RATEXPAND(D3)$

(C5) EV(%);

                 4  2          4        2          4
(D5)  2 (B + A)  X  + (B + A)  (D + C)  X + (B + A)

(C6) (A+B)*(X+A+B)^2*EXP(X^2+A*X+B);
```

```
                                                  2
                                             2   X  + A X + B
(D6)                (B + A) (X + B + A)  %E

(C7) ISOLATE(%,X),EXPTISOLATE:TRUE;

(E7)                               B + A

                                    B
(E8)                              %E

                                                  2
                                             2   X  + A X
(D8)                   E7 E8 (X + E7)  %E
```

PICKAPART*(exp,depth)* will assign E labels to all subexpressions of *exp* down to the specified integer *depth*. This is useful for dealing with large expressions and for automatically assigning parts of an expression to a variable without having to use the Part functions.

```
(C1) INTEGRATE(1/(X^3+2),X)$

(C2) PICKAPART(D1,1);

                                  1/3
                         LOG(X + 2   )
(E2)                     -------------
                                2/3
                             3 2

                                     1/3
                             2 X - 2
                       ATAN(------------)
                                1/3
                               2    SQRT(3)
(E3)                   ------------------
                             2/3
                            2    SQRT(3)
```

```
                                   2    1/3      2/3
                              LOG(X  - 2    X + 2   )
(E4)                        - ----------------------
                                        2/3
                                     6 2

(D4)                             E4 + E3 + E2
```

REVEAL*(exp,depth)* will display *exp* to the specified integer *depth* with the length of each part indicated. Sums will be displayed as SUM(n) and products as PRODUCT(n) where n is the number of subparts of the sum or product. Exponentials will be displayed as EXPT.

```
(C1) INTEGRATE(1/(X^3+2),X)$

(C2) REVEAL(%,2);
(D2)              PRODUCT(3) + PRODUCT(3) + PRODUCT(3)

(C3) REVEAL(D1,3);
                   EXPT LOG                       EXPT LOG
(D3)             - -------- + EXPT EXPT ATAN + --------
                      6                              3
```

NUMFACTOR*(exp)* gives the numerical factor multiplying the expression *exp* which should be a single term. If the gcd of all the term coefficients in a sum is desired the CONTENT function (see 6.5) may be used.

```
(C1) GAMMA(7/2);

(D1)              15 SQRT(%PI)
                  ------------
                        8

(C2) NUMFACTOR(%)
                   15
(D2)               --
                   8
```

HIPOW*(exp, v)* gives the highest explicit exponent of *v* in *exp*. Sometimes it may be necessary to expand *exp* since this is not done automatically by HIPOW. Thus HIPOW(Y**3*X**2+X*Y**4,X) is 2.

LOPOW*(exp, v)* gives the lowest exponent of *v* which explicitly appears in *exp*. Thus LOPOW((X+Y)**2+(X+Y)**A,X+Y) is MIN(A,2).

DERIVDEGREE*(exp, dv, iv)* finds the highest degree of the derivative of the dependent variable *dv* with respect to the independent variable *iv* occurring in *exp*.

```
(C1) 'DIFF(Y,X,2)+'DIFF(Y,Z,3)*2+'DIFF(Y,X)*X**2$
(C2) DERIVDEGREE(%,Y,X);
(D2)                          2
```

The next several functions deal with complex variables. The user should note the following conventions.

1) all variables are assumed to take on real values exclusively;

2) all functions are assumed to be real-valued;

3) the complex argument is maintained in the half-open interval $(-\pi,\pi]$ whenever possible;

4) the argument of 0 is (arbitrarily) assumed to be 0, although normally the user need not worry about this, since 0*%E^(%I*0) is simplified to 0;

5) trigonometric functions are normally assumed to take on their principal values.

REALPART*(exp)* gives the real part of *exp*. REALPART and IMAGPART will work on expressions involving trigonometic and hyperbolic functions, as well as SQRT, LOG, and exponentiation.

IMAGPART*(exp)* returns the imaginary part of the expression *exp*.

The real or imaginary part of an expression of the form Z^N, where Z is not purely real, will be algebraic if n <= MAXPOSEX; otherwise, for compactness,it will be expressed as $ABS(Z)^N * COS(N*ARG Z)$ or $ABS(Z)^N * SIN(N*ARG Z)$.

RECTFORM*(exp)* returns an expression of the form A + B*%I, where A and B are purely real.

POLARFORM*(exp)* returns R*%E^(%I*THETA) where R and THETA are purely real.

CAVEAT: Simplification of algebraic and transcendental functions of a complex variable may give rise to apparent factors, %I. For example, SQRT(-C+D) may be transformed to %I*SQRT(C-D).

CABS*(exp)* returns the complex absolute value (the complex modulus) of *exp*.

CARG*(exp)* returns the argument (phase angle) of *exp*. Due to the conventions and restrictions (described above), principal value cannot be guaranteed.

```
(C1) RECTFORM(SIN(2*%I+X));

(D1)                 COSH(2) SIN(X) + %I SINH(2) COS(X)

(C2) POLARFORM(%);
                 2         2            2        2
(D2) SQRT(COSH (2) SIN (X) + SINH (2) COS (X))

                                 %I ATAN2(SINH(2) COS(X), COSH(2) SIN(X))
                               %E
(C3) RECTFORM(LOG(3+4*%I));

(D3)                           LOG(5) + %I ATAN2(4,3)
(C4) POLARFORM(%);

                  2            2            %I ATAN2(ATAN2(4, 3), LOG(5))
(D4)  SQRT(LOG (5) + ATAN2 (4, 3)) %E
(C5) RECTFORM((2+3.5*%I)^.25),NUMER;
```

```
(D5)                      0.36825881 %I + 1.36826627

(C6) POLARFORM(D5);
                                     0.26291253 %I
(D6)                       1.416957 %E
```

LHS*(eqn)* returns the left side of the equation *eqn*. If *eqn* is not an equation, then LHS(*eqn*) = *eqn*.

RHS*(eqn)* returns the right side of the equation *eqn*. If *eqn* is not an equation, then RHS(*eqn*) = 0.

NUM*(exp)* obtains the numerator, exp1, of the rational expression *exp* = exp1/exp2.

DENOM*(exp)* returns the denominator, exp2, of the rational expression *exp* = exp1/exp2.

The above two commands do not alter the internal representations of expressions and have the desirable property that for all expressions NUM(exp)/DENOM(exp) is the same as exp.

FIRST*(exp)* yields the first part of *exp* which may result in the first element of a list, the first row of a matrix, the first term of a sum, etc. Note that FIRST and the following two functions work on the form of *exp* which is displayed not the form which is typed on input. If the variable INFLAG[FALSE] is set to TRUE however, these functions will look at the internal form of *exp*. Note that the simplifier re-orders expressions (see 3.3). Thus FIRST(X+Y) will be X if INFLAG is TRUE and Y if INFLAG is FALSE. (FIRST(Y+X) gives the same results).

REST*(exp, n)* yields *exp* with its first *n* elements removed if *n* is positive and its last -*n* elements removed if *n* is negative. If *n* is 1 it may be omitted. *Exp* may be a list, matrix, or other expression. If INFLAG:TRUE the internal form of *exp* will be used.

LAST*(exp)* yields the last part (term, row, element, etc.) of the *exp*. If INFLAG:TRUE the internal form of *exp* will be used.

DELETE*(exp1, exp2)* removes all occurrences of *exp1* from *exp2*. *Exp1* may be a term of *exp2* (if it is a sum) or a factor of *exp2* (if it is a product).

```
(C1)  DELETE(SIN(X),X+SIN(X)+Y);

(D1)                Y + X
```

LENGTH*(exp)* gives the number of parts in the internal form of *exp*. For lists this is the number of elements, for matrices it is the number of rows, and for sums it is the number of terms. However for products it may not always yield the number of factors that would be displayed because of the fact that -E is represented internally as -1*E and A/B is represented internally by A*B^(-1). (cf. DISPFORM)

NTERMS*(exp)* gives the number of terms that *exp* would have if it were fully expanded out and no cancellations or combinations of terms occurred. Note that expressions like SIN(E), SQRT(E), EXP(E), etc. count as just one term regardless of how many terms E has (if it is a sum).

6.3 SOLVE and Related Functions

The following functions obtain the roots of equations or yield information concerning the roots.

NROOTS*(poly, low, high)* finds the number of real roots of the real univariate polynomial *poly* in the half-open interval (*low,high*]. The endpoints of the interval may also be MINF,INF respectively for minus infinity and plus infinity. NROOTS(*poly*) is equivalent to NROOTS(*poly*,MINF,INF). The method of Sturm sequences is used. (see Heindel in [A1].)

```
(C1) POLY1:X**10-2*X**4+1/2$
(C2) NROOTS(POLY1,-6,9.1);
RAT REPLACED 0.5 BY 1/2 = 0.5
(D2)                            4
```

REALROOTS*(poly, bound)* finds all of the real roots of the real univariate polynomial *poly* within a tolerance of *bound* which, if less than 1, causes all integral roots to be found exactly. The parameter *bound* may be arbitrarily small in order to achieve any desired accuracy. The first argument may also be an equation. REALROOTS(*poly*) is equivalent to REALROOTS(*poly*,ROOTSEPSILON). ROOTSEPSILON[1.0E-7] is a real number used to establish the confidence interval for the roots.

```
(C1) REALROOTS(X**5+X+1,5.0E-6);

                                  395773
(E1)                        X = - ------
                                  524288

(D1)                             [E1]

(C2) E1,FLOAT;
(D2)                      X = - 0.75487709

(C3) PART(C1,1);
                             5
(D3)                        X  + X + 1
```

```
(C4) %,D2;
(D4)                          1.50687992E-6
```

ALLROOTS*(poly)* finds all the real and complex roots of the real polynomial *poly* which must be univariate and may be an equation. For complex polynomials an algorithm by Jenkins and Traub is used;[1] for real polynomials the algorithm used is due to Jenkins.[2] The flag POLYFACTOR[FALSE] when true causes ALLROOTS to factor the polynomial over the real numbers if the polynomial is real, or over the complex numbers, if the polynomial is complex.

```
(C1) (2*X+1)**3=13.5*(X**5+1);

                                 3          5
(D1)                    (2 X + 1)  = 13.5 (X  + 1)

(C2) ALLROOTS(%);

(E2)                         X =  - 1.0157555

(E3)                           X = 0.829675

(E4)                X =  - 0.96596254 %I - 0.40695972

(E5)                  X = 0.96596254 %I - 0.40695972

(E6)                            X = 1.0

(D6)                      [E2, E3, E4, E5, E6]
```

LINSOLVE*([exp1, exp2, ...], [var1, var2, ...])* solves the list of simultaneous linear equations for the list of variables. The *expi* must each be linear in the variables and may be equations. LINSOLVE does no error checking to assure linearity.

1. Algorithm 419, Comm. ACM, vol. 15, (1972), p. 97

2. Algorithm 493, TOMS, vol. 1, (1975), p.178.

If GLOBALSOLVE[FALSE] is set to TRUE then variables which are SOLVEd for will be set to the solution of the set of simultaneous equations.

BACKSUBST[TRUE] if set to FALSE will prevent back substitution after the equations have been triangularized. This may be necessary in very big problems where back substitution would cause the storage capacity to be exceeded.

```
(C1) X+Z=Y$

(C2) 2*A*X-Y=2*A**2$

(C3) Y-2*Z=2$

(C4) LINSOLVE([D1,D2,D3],[X,Y,Z]),GLOBALSOLVE:TRUE;
SOLUTION

(E4)                          X : A + 1

(E5)                           Y : 2 A

(E6)                          Z : A - 1

(D6)                        [E4, E5, E6]
```

ALGSYS*([exp1, exp2, ...], [var1, var2, ...])* solves the list of simultaneous polynomials or polynomial equations (which can be non-linear) for the list of variables. The symbols %R1, %R2, etc. will be used to represent arbitrary parameters when needed for the solution. In the process described below, ALGSYS is entered recursively if necessary.

The method is as follows:

(1) First the equations are FACTORed and split into subsystems.

(2) For each subsystem *Si*, an equation *E* and a variable *var* are selected (the *var* is chosen to have lowest nonzero degree). Then the resultant of *E* and *Ej* with respect to *var* is computed for each of the remaining equations *Ej* in the subsystem *Si*. This yields a new subsystem *S'i* in one fewer variables (*var* has been eliminated). The process now returns to (1).

(3) Eventually, a subsystem consisting of a single equation is obtained. If the equation is multivariate and no approximations in the form of floating point numbers have been introduced, then SOLVE is called to find an exact solution. [3]

If the equation is univariate and is either linear, quadratic, or bi-quadratic, then again SOLVE is called if no approximations have been introduced. If approximations have been introduced then if the switch REALROOTS[TRUE]:TRUE, the function REALROOTS is called to find the real-valued solutions. If REALROOTS:FALSE then ALLROOTS is called which looks for real and complex-valued solutions. If ALGSYS produces a solution which has fewer significant digits than required, the user can change the value of ALGEPSILON[10^8] to a higher value.

(4) Finally, the solutions obtained in step (3) are re-inserted into previous levels and the solution process returns to (1).

The user should be aware of several caveats.

When ALGSYS encounters a multivariate equation which already contains floating point approximations, then, presently, it does not attempt to apply exact methods to such equations and prints the message: ALGSYS CANNOT SOLVE---SYSTEM TOO COMPLICATED.

Interactions with RADCAN can produce large or complicated expressions. In that case, the user may use PICKAPART or REVEAL to analyze the solution. Occasionally, RADCAN may introduce an apparent %I into a solution which is actually real-valued. To prevent the omission of possible solutions, the user may prefer to set REALONLY[TRUE]:FALSE.

```
(C1) X+Z-Y$

(C2) X+B*Y*Z -A$

(C3) X^2+C*Z -D;

(C4) ALGSYS([D1,D2,D3],[X,Y,Z]);

(D4)                              []
```

3. The user should realize that SOLVE may not be able to produce a solution or if it does the solution may be a very large expression.

```
(C5) X+Z-Y^2$

(C6) ALGSYS([D1,D2,D5],[X,Y,Z]);
                  B - A            A - 1
(D6)      [[X = -----, Y = 1, Z = -----], [X = A, Y = 0, Z = - A]]
                  B - 1            B - 1

(C7) X+B*Y*Z*X -A$

(C8) ALGSYS([D1,D5,D7],[X,Y,Z]);
                   2
          SQRT(B  - 4 A B + 2 B + 1) + B + 1
(D8) [[X = ---------------------------------, Y = 1,
                       2 B

          2
     SQRT(B  + (2 - 4 A) B + 1) + (1 - 2 A) B + 1
Z = - --------------------------------------------],
                  2                    2
        B SQRT(B  + (2 - 4 A) B + 1) + B  + B

           2
    - SQRT(B  - 4 A B + 2 B + 1) + B + 1
[X = ------------------------------------, Y = 1,
                  2 B

          2
     SQRT(B  + (2 - 4 A) B + 1) + (2 A - 1) B - 1
Z = - --------------------------------------------],
                  2                    2
        B SQRT(B  + (2 - 4 A) B + 1) - B  - B


[X = A, Y = 0, Z = - A]]
```

SOLVE*(exp, var)* solves the algebraic equation *exp* for the variable *var* and returns a list of solution equations in *var*. If *exp* is not an equation, it is assumed to be an expression to be set equal to zero. *Var* may be a function (e.g. F(X)), or other non-atomic expression except a sum or product. It may be omitted if *exp* contains only one variable. *Exp* may be a rational expression, and may contain trigonometric functions, exponentials, etc.

The following method is used:

Let E be the expression and X be the variable. If E is linear in X then it is trivially solved for X. Otherwise if E is of the form A*X**N+B then the result is (-B/A)**(1/N) times the Nth roots of unity.

If E is not linear in X then the gcd of the exponents of X in E (say N) is divided into the exponents and the multiplicity of the roots is multiplied by N. Then SOLVE is called again on the result.

If E factors then SOLVE is called on each of the factors. Finally SOLVE will use the quadratic, cubic, or quartic formulas where necessary.

In the case where E is a polynomial in some function of the variable to be solved for, say F(X), then it is first solved for F(X) (call the result C), then the equation F(X)=C can be solved for X provided the inverse of the function F is known.

BREAKUP[TRUE] if FALSE will cause SOLVE to express the solutions of cubic or quartic equations as single expressions rather than as made up of several common subexpressions which is the default.

MULTIPLICITIES[NOT%SET%YET] - will be set to a list of the multiplicities of the individual solutions returned by SOLVE, REALROOTS, or ALLROOTS.

SOLVEFACTORS[TRUE] - if FALSE then SOLVE will not try to factor the expression. The FALSE setting may be desired in some cases where factoring is not necessary.

SOLVERADCAN[FALSE] - if TRUE then SOLVE will use RADCAN which will make SOLVE slower but will allow certain problems containing exponentials and logs to be solved.

When SOLVing polynomials with large integer coefficients, it may be useful to reset INTFACLIM.

SOLVE*([eq1, ..., eqn], [v1, ..., vn])* solves a system of simultaneous (linear or non-linear) polynomial equations by calling LINSOLVE or ALGSYS and returns a list of the solution lists in the variables. In the case of LINSOLVE this list would contain a single list of solutions. It takes two lists as arguments. The first list *(eqi,* i=1,...,n) represents the equations to be solved; the second list is a list of the unknowns to be determined. If the total number of variables in the equations is equal to the number of equations, the second argument-list may be omitted. For linear systems if the given equations are not compatible, the message INCONSISTENT will be displayed; if no unique solution exists, then SINGULAR will be displayed. DISPFLAG[TRUE] when set to FALSE within a BLOCK will inhibit the display of output generated by the Solve functions called from within the BLOCK.

SOLVETRIGWARN[TRUE] if set to FALSE will inhibit printing by SOLVE of the warning message saying that it is using inverse trigonometric functions to solve the equation, and thereby losing solutions.

SOLVEDECOMPOSES[TRUE] if TRUE, will induce SOLVE to use POLYDECOMP (see below) in attempting to solve polynomials. [4]

PROGRAMMODE[FALSE] when TRUE will inhibit SOLVE from printing E-labels and will force SOLVE to return its answers explicitly as elements in a list.

SOLVEEXPLICIT[FALSE] if FALSE, inhibits SOLVE from returning implicit solutions i.e. of the form F(x)=0.

```
(C1) SOLVE(ASIN(COS(3*X))*(F(X)-1),X);
Solution

                                       %PI
(E1)                               X = ---
                                        6
The roots of

(E2)                               F(X) = 1

(D2)                               [E1,E2]

(C3) SOLVERADCAN:TRUE$
(C4) SOLVE(5**F(X)=125,F(X));
(D4)                              F(X) = 3

(C5) [4*X**2-Y**2=12,X*Y-X=2]
```

4. Under certain circumstances (e.g. if there is a variable in the exponent), the implicit "solution" may quite complex

```
                                   2     2
(D5)                         [4 X  - Y  = 12, X Y - X = 2]

(C6) SOLVE(D5,[X,Y]);

(D6)  [[Y =  - 0.15356758,

          X =  - 1.733752], [Y = 2.0, X = 2.0]]

(C7) SOLVE(X^3+A*X+1,X);

                                              3
                                    SQRT(4 A  + 27)
(E7)                                ---------------
                                      6 SQRT(3)

                                              1 1/3
(E8)                                  (E7 - -)
                                              2
SOLUTION

                                              %I SQRT(3)    1
                                             (---------- - -) A
               %I SQRT(3)    1                    2         2
(E9) X = ( - ---------- - -) E8 - ------------------
                  2         2                  3 E8

                                              %I SQRT(3)    1
                                        ( - ---------- - -) A
             %I SQRT(3)    1                     2         2
(E10) X = (---------- - -) E8 - ---------------------
                2         2                    3 E8

                                                A
(E11)                                 X = E8 - ----
                                               3 E8

(D11)                                 [E9, E10, E11]
```

POLYDECOMP*(poly,var)* returns a list of polynomials [f1(*var*),f2(*var*),...fn(*var*)] such that *poly* = f1(f2(...fn(*var*)...)). There is no other decomposition which involves more polynomials excepting linear fi.

6.4 The Matrix Functions

Matrix multiplication is effected by using the dot operator, ".", which is also convenient if the user wishes to represent other non-commutative algebraic operations (see 6.4.1). The exponential of the . operation is "^^" .

Thus, for a matrix A, A.A = A^^2 and, if it exists, A^^-1 is the inverse of A.

The operations +,-,*,** are all element-by-element operations; all operations are normally carried out in full, including the . (dot) operation. Many switches exist for controlling simplification rules involving dot and matrix-list operations (see below).

ENTERMATRIX*(m, n)* allows one to enter a matrix element by element with MACSYMA requesting values for each of the *m***n* entries.

```
(C1) ENTERMATRIX(2,1);

ROW 1 COLUMN 1  X+Y/2;

ROW 2 COLUMN 1  34;

MATRIX-ENTERED
                                        [ Y     ]
                                        [ - + X ]
(D1)                                    [ 2     ]
                                        [       ]
                                        [   34  ]
```

MATRIX*(row1, ..., rown)* defines a rectangular matrix with the indicated rows. Each row has the form of a list of expressions, e.g. [A, X**2, Y, 0] is a list of 4 elements.

GENMATRIX*(array, i2, j2, i1, j1)* generates a matrix from the array using *array(i1,j1)* for the first (upper-left) element and *array(i2,j2)* for the last (lower-right) element of the matrix. If j1=i1 then j1 may be omitted. If j1=i1=1 then i1 and j1 may both be omitted. If a selected element of the array doesn't exist a symbolic one will be used.

```
(C1) H[I,J]:=1/(I+J-1)$

(C2) GENMATRIX(H,3,3);
                                   [   1  1]
                                   [1  -  -]
                                   [   2  3]
                                   [       ]
                                   [1  1  1]
(D2)                               [-  -  -]
                                   [2  3  4]
                                   [       ]
                                   [1  1  1]
                                   [-  -  -]
                                   [3  4  5]
```

COPYMATRIX*(M)* creates a copy of the matrix *M*. This is the only way to make a copy aside from recreating *M* elementwise. Copying a matrix may be useful when SETELMX is used (see below).

COPYLIST*(L)* creates a copy of the list L.

ADDROW*(M,l)* appends the row given by the list *l* onto the matrix *M*.

IDENT*(n)* produces an *n* by *n* identity matrix.

DIAGMATRIX*(n, x)* returns a diagonal matrix of size *n* by *n* with the diagonal elements all *x*. An identity matrix is created by DIAGMATRIX(*n*,1), or one may use IDENT(n).

EMATRIX*(m, n, x, i, j)* will create an *m* by *n* matrix all of whose elements are zero except for the *i,j* element which is *x*.

MATRIXMAP*(fn, M)* will map the function *fn* onto each element of the matrix *M*.

SETELMX*(x, i, j, M)* changes the *i,j* element of *M* to *x*. The altered matrix is returned as the value. The notation *M[i,j]:x* may also be used, altering *M* in a similar manner, but returning *x* as the value.

COEFMATRIX*([eq1, ...], [var1, ...])* the coefficient matrix for the variables *var1,...* of the system of linear equations *eq1,...*

AUGCOEFMATRIX*([eq1, ...], [var1, ...])* the augmented coefficient matrix for the variables *var1,...* of the system of linear equations *eq1,...*. This is the coefficient matrix with a column adjoined for the constant terms in each equation (i.e. those not dependent upon *var1,...*).

```
(C1) [2*X-(A-1)*Y=5*B,A*X+B*Y+C=0]$

(C2) AUGCOEFMATRIX(%,[X,Y]);

                          [2  1 - A  5 B ]
(D2)                      [              ]
                          [A    B    - C]
```

COL*(M,i)* gives a matrix of the *i*th column of the matrix *M*.

ROW*(M, i)* gives a matrix of the *i*th row of matrix *M*.

SUBMATRIX*(m1, ..., M, n1, ...)* creates a new matrix composed of the matrix *M* with rows *mi* deleted, and columns *ni* deleted.

MINOR*(M, i, j)* computes the *i,j* minor of the matrix *M*. That is, *M* with row *i* and column *j* removed.

TRANSPOSE*(M)* produces the transpose of *M*.

ECHELON*(M)* produces the echelon form of *M*. That is, *M* with elementary row operations performed on it such that the first non-zero element in each row in the resulting matrix is a one and the column elements under the first one in each row are all zero.

```
(C3) ECHELON(D2);      (D2 is as above)

                     [      A - 1        5 B     ]
                     [1   - -----        ---     ]
                     [        2           2      ]
(D3)                 [                           ]
                     [               2 C + 5 A B ]
                     [0     1      - ------------]
                     [                      2    ]
                     [               2 B + A  - A]
```

TRIANGULARIZE*(M)* produces the upper triangular form of the matrix *M* which needn't be square.

```
(C4) TRIANGULARIZE(D2);

                   [2     1 + A           5 B      ]
(D4)               [                               ]
                   [          2                    ]
                   [0  2 B + A  - A   - 2 C - 5 A B]
```

RANK*(M)* computes the rank of the matrix *M*. That is, the order of the largest non-singular subdeterminant of M.

```
(C5) RANK(D2);
(D5)                          2
```

DETERMINANT*(M)* computes the determinant of *M* by a method similar to Gaussian elimination. The form of the result depends upon the setting of the switch RATMX (see below). There is a special routine for dealing with sparse determininants which can be used by setting the switches RATMX:TRUE and SPARSE:TRUE.

NEWDET*(M,n)* also computes the determinant of *M* but uses the Johnson-Gentleman tree minor algorithm [Ge1]. *M* may be the name of a matrix or array. The argument *n* is the order; it is optional if *M* is a matrix.

CHARPOLY*(M, var)* computes the characteristic polynomial for *M* with respect to *var*. That is, DETERMINANT(*M* - DIAGMATRIX(LENGTH(*M*),*var*)).

```
(C2) A:MATRIX([3,1],[2,4]);
                                    [3  1]
(D2)                                [    ]
                                    [2  4]

(C3) CHARPOLY(A,LAMBDA);
                                  2
(D3)                        LAMBDA   - 7 LAMBDA + 10

(C4) SOLVE(%);
SOLUTION

(E4)                             LAMBDA = 2

(E5)                             LAMBDA = 5

(D5)                              [E4, E5]

(C6) X:MATRIX([X1],[X2]);
                                    [X1]
(D6)                                [  ]
                                    [X2]

(C7) A.X-LAMBDA*X,E5;
                                 [ X2 - 2 X1  ]
(D7)                             [            ]
                                 [ - X2 + 2 X1]
```

```
(C8)  D7[1,1]=0;
(D8)                           X2 - 2 X1 = 0

(C9) X1**2+X2**2=1;
                                 2      2
(D9)                           X2  + X1  = 1

(C10) SOLVE([D8,D9],[X1,X2]);

(D10) [[X2 =  - 0.89442714, X1 =  - 0.447213571,

        [X2 = 0.89442714, X1 = 0.44721357]]
```

Options Relating to Matrices

Note: MX stands for Matrix and SC stands for Scalar

By resetting the options LMXCHAR and RMXCHAR (with the defaults [and] respectively), the user can specify the delimiters used in the display of matrices.

RATMX[FALSE] - if FALSE will cause determinant and matrix addition, subtraction, and multiplication to be performed in the representation of the matrix elements and will cause the result of matrix inverse to be left in the representation of the matrix elements. If it is TRUE, the 4 operations mentioned above will be performed in CRE form and the result of matrix inverse will be in CRE form. Note that this may cause the elements to be expanded (depending on the setting of RATFAC) which might not always be desired.

LISTARITH[TRUE] - if FALSE causes any arithmetic operations with lists to be suppressed; when TRUE, list-matrix operations are contagious causing lists to be converted to matrices yielding a result which is always a matrix. However, list-list operations should return lists.

DETOUT[FALSE] - if TRUE will cause the determinant of a matrix whose inverse is computed to be kept outside of the inverse. For this switch to have an effect DOALLMXOPS and DOSCMXOPS should be FALSE (see below). Alternatively this switch can be given to EV which causes the other two to be set correctly.

DOALLMXOPS[TRUE] - if TRUE all operations relating to matrices are carried out. If it is FALSE then the setting of the following switches govern which operations are performed.

DOMXMXOPS[FALSE] - if TRUE then all matrix-matrix or matrix-list operations are carried out (but not scalar-matrix operations); if this switch is FALSE they are not.

DOSCMXOPS[FALSE] - if TRUE then scalar-matrix operations are performed.

DOSCMXPLUS[FALSE] - if TRUE will cause SCALAR + MATRIX to give a matrix answer.

SCALARMATRIXP[TRUE] - if TRUE causes a square matrix of dimension one (when produced as a result of a computation) to be converted to a scalar (i.e. its only element).

SPARSE[FALSE] - if TRUE and if RATMX:TRUE then DETERMINANT will use special routines for computing sparse determinants.

6.4.1 The Dot Operator

In some applications one would like to work with expressions that contain variables which are to behave like matrices (e.g. are non-commutative). The variables may be declared to be non-scalar by using the DECLARE function (sect. 6.12). After the expressions are manipulated into a particular form then perhaps actual matrices will be substituted for them. Several options are provided in order to control how MACSYMA treats such expressions. (The options are checked for and utilized by MACSYMA in the order in which they are presented here).

In the following discussion A, B, and C are any expressions, and SC is a scalar expression (i.e. one free of lists, matrices, and any atoms declared non-scalar).

DOTASSOC[TRUE] - when TRUE causes (A.B).C to simplify to A.(B.C)

DOTSCRULES[FALSE] - when TRUE will cause A.SC or SC.A to simplify to SC*A and A.(SC*B) to simplify to SC*(A.B)

DOTCONSTRULES[TRUE] - is similar to DOTSCRULES but with constants instead of scalars.

DOTEXPTSIMP[TRUE] - when TRUE causes A.A to simplify to A^^2

DOTDISTRIB[FALSE] - if TRUE will cause A.(B+C) to simplify to A.B+A.C

```
(C1) DECLARE([M1,M2,M3],NONSCALAR)$

(C2) (1-L*M1).(1-L*M2).(1-L*M3),DOTCONSTRULES:TRUE,EXPAND;

(D2)  - L M3 + L M2 . L M3 - L M2 + L M1 . L M3

         - L M1 . (L M2 . L M3) + L M1 . L M2 - L M1 + 1

(C3) %,DOTSCRULES:TRUE;

                 2                          2
(D3)  - L M3 + L  (M2 . M3) - L M2 + L  (M1 . M3)

                      3                          2
         - L  (M1 . (M2 . M3)) + L  (M1 . M2) - L M1 + 1

(C4) RAT(%,L);
                                    3
(D4)/R/   - (M1 . (M2 . M3)) L  + (M2 . M3 + M1 . M3

                                          2
        + M1 . M2) L  + ( - M3 - M2 - M1) L + 1
```

6.5 Functions for Rational Expressions

A rational expression is the quotient of two polynomials. MACSYMA provides a special internal representation (called CRE for canonical rational expression form - (see 3.1)) which can be used for rational expressions (and polynomials as special cases) and which requires less storage than the general representation. In addition CRE manipulations are usually faster. Therefore it may be desirable to use these whenever the problem of interest can be expressed largely in terms of polynomials or rational expressions. The symbol /R/ following the line label in the display of an expression indicates that either the expression is in CRE form or that some subexpression of it is.

CRE form is "contagious" in that any time a CRE expression is added to or multiplied by another compatible expression, the result is in CRE form. Thus by initially multiplying by RAT(1) one can force his entire calculation to be done in CRE form. However, if CRE are mixed into an expression containing general forms e.g. SIN(RAT(X**2)), such that the result is not totally in CRE form, then the result is automatically converted into general representation.

Some functions (e.g. RATSIMP, FACTOR, etc.) use CRE form internally in the implementation of their algorithms. This fact however is usually transparent to the user.

RATVARS*(var1, var2, ..., varn)* forms its n arguments into a list in which the rightmost variable *varn* will be the main variable of future rational expressions in which it occurs, and the other variables will follow in sequence. If a variable is missing from the RATVARS list, it will be given lower priority than the leftmost variable *var1*. The arguments to RATVARS can be either variables or non-rational functions (e.g. SIN(X)).
The variable RATVARS is a list of the arguments which have been given to this function.

RAT*(exp, v1, ..., vn)* converts *exp* to CRE form by expanding and combining all terms over a common denominator and cancelling out the greatest common divisor of the numerator and denominator as well as converting floating point numbers to rational numbers within a tolerance of RATEPSILON[2.0E-8]. The variables are ordered according to the *v1,...,vn* as in RATVARS, if these are specified. RAT does not generally simplify functions other than + , - , * , / , and exponentiation to an integer power and it does not deal with equations whereas RATSIMP does handle these cases. Note that atoms (numbers and names) in CRE form are not the same as they are in the general form. Thus RAT(X)-X results in RAT(0) which has a different internal representation than 0.

RATFAC[FALSE] when TRUE invokes a partially factored form for CRE rational expressions. During rational operations the expression is maintained as fully factored as possible without an actual call to the factor package. This should always save space and may save some time in some computations. The numerator and denominator are still made relatively prime, for example

```
RAT((X^2 -1)^4/(X+1)^2);  yields (X-1)^4*(X+1)^2),
```

but the factors within each part may not be relatively prime.

RATPRINT[TRUE] if FALSE suppresses the printout of the message informing the user of the conversion of floating point numbers to rational numbers.

KEEPFLOAT[FALSE] if TRUE prevents floating point numbers from being converted to rational numbers.[1]

BFTORAT[FALSE] controls the conversion of bfloats to rational numbers. If BFTORAT:FALSE, RATEPSILON will be used to control the conversion (this results in relatively small rational numbers). If BFTORAT:TRUE, the rational number generated will accurately represent the bfloat.

(Also see the RATEXPAND and RATSIMP functions. sec. 6.1.1)

```
(C1) ((X-2*Y)**4/(X**2-4*Y**2)**2+1)*(Y+A)*(2*Y+X)
        /(4*Y**2+X**2);
                                        4
                               (X - 2 Y)
              (Y + A) (2 Y + X) (------------ + 1)
                                 2     2 2
                               (X  - 4 Y )
(D1)          ---------------------------------
                                2    2
                             4 Y  + X
```

1. As with all other switches, various MACSYMA algorithms may override the setting of this switch if they are unable to operate in that mode.

```
(C2) RAT(%,Y,A,X);
                                2 A + 2 Y
(D2)/R/                         ---------
                                 X + 2 Y
```

RATDISREP*(exp)* changes its argument from CRE form to general form. This is sometimes convenient if one wishes to stop the "contagion", or use rational functions in non-rational contexts (see the example at the beginning of this section). Most CRE functions will work on either CRE or non-CRE expressions, but the answers may take different forms. If RATDISREP is given a non-CRE for an argument, it returns its argument unchanged.

TOTALDISREP*(exp)* converts every subexpression of *exp* from CRE to general form. If *exp* is itself in CRE form then this is identical to RATDISREP but if not then RATDISREP would return *exp* unchanged while TOTALDISREP would "totally disrep" it. This is useful for ratdisrepping expressions e.g., equations, lists, matrices, etc. which have some subexpressions in CRE form.

NUM*(exp)* obtains the numerator of the rational expression *exp*.

DENOM*(exp)* returns the denominator of the rational expression *exp*.

The above two commands do not alter the internal representations of expressions and have the desirable property that for all expressions NUM(exp)/DENOM(exp) is the same as exp. This is not true of the following two commands which return expressions in CRE form.

RATNUMER*(exp)* obtains the numerator of the rational expression *exp*. If *exp* is in general form then the NUM function should be used instead, unless one wishes to get a CRE result.

RATDENOM*(exp)* obtains the denominator of the rational expression *exp*. If *exp* is in general form then the DENOM function should be used instead, unless one wishes to get a CRE result.

RATWEIGHT(*v1, w1, ..., vn, wn*) assigns a weight of *wi* to the variable *vi*. This causes a monomial to be replaced by 0 if its weight exceeds the value of the variable RATWTLVL [FALSE] (for the default value FALSE no truncation occurs). The weight of a monomial is the sum of the products of the weight of a variable in the term times its power. Thus the weight of 3*v1**2*v2 is 2*w1+w2. This truncation occurs only when multiplying or exponentiating CRE forms of expressions.

RATWEIGHTS[[]] returns a list of weight assignments, as does RATWEIGHT(); KILL(...,RATWEIGHTS), SAVE(...,RATWEIGHTS) eliminate and save weight assignments ((see 10.3),(see 15.3)).

```
(C5) RATWEIGHT(A,1,B,1);
(D5)                          [[B, 1], [A, 1]]

(C6) EXP1:RAT(A+B+1)$

(C7) %**2;
                         2                2
(D7)/R/                 B  + (2 A + 2) B + A  + 2 A + 1

(C8) RATWTLVL:1$

(C9) EXP1**2;
(D9)/R/                       2 B + 2 A + 1
```

HORNER(*exp, var*) will convert *exp* into a rearranged representation as in Horner's rule, using *var* as the main variable if it is specified. *Var* may also be omitted in which case the main variable of the CRE form of *exp* is used. HORNER sometimes improves stability if *expr* is to be numerically evaluated. It is also useful if MACSYMA is used to generate programs to be run in FORTRAN (see **STRINGOUT - 10.4**)

```
(C1) 1.0E-20*X^2-5.5*X+5.2E20;
                                 2
(D1)                    1.0E-20 X  - 5.5 X + 5.2E+20

(C2) HORNER(%,X),KEEPFLOAT:TRUE;
(D2)                    X (1.0E-20 X - 5.5) + 5.2E+20

(C3) D1,X=1.0E20;
```

```
ARITHMETIC OVERFLOW

(C4) D2,X=1.0E20;
(D4)                           6.9999999E+19
```

FASTTIMES*(p1, p2)* multiplies the polynomials *p1* and *p2* by using a special algorithm for multiplication of polynomials. They should be multivariate, dense, and nearly the same size. Classical multiplication is of order N*M where N and M are the degrees. FASTTIMES is of order MAX(N,M)**1.585.

The rest of the functions in this section return their results in general representation only if all of their principal arguments are in that form. If any of their principal arguments are in CRE form then the result is returned in CRE form.

DIVIDE*(p1, p2, var1, ..., varn)* computes the quotient and remainder of the polynomial *p1* divided by the polynomial *p2*, in a main polynomial variable, *varn*. The other variables are as in the RATVARS function. The result is a list whose first element is the quotient and whose second element is the remainder.

```
(C1) DIVIDE(X+Y,X-Y,X);
(D1)                         [1, 2 Y]

(C2) DIVIDE(X+Y,X-Y);
(D2)                       [ - 1, 2 X]
```

(Note that Y is the main variable in C2)

QUOTIENT*(p1, p2, var1, ...)* computes the quotient of the polynomial *p1* divided by the polynomial *p2*.

REMAINDER*(p1, p2, var1, ...)* computes the remainder of the polynomial *p1* divided by the polynomial *p2*.

CONTENT*(p1, var1, ..., varn)* returns a list whose first element is the greatest common divisor of the coefficients of the terms of the polynomial *p1* in the variable *varn* (this is the content) and whose second element is the polynomial *p1* divided by the content.

```
(C1) CONTENT(2*X*Y+4*X**2*Y**2,Y);

(D1)            [2*X, 2*X*Y**2+Y].
```

GCD*(p1, p2, var1, ...)* computes the greatest common divisor of *p1* and *p2*. The flag GCD[EZ] determines which algorithm is employed. Setting GCD to EZ,RED, or MOD selects the EZGCD [Mo6], reduced, or modular [Br1] algorithm, respectively. If GCD:FALSE then GCD(p1,p2,var) will always return 1 for all x. Many functions (e.g. RATSIMP, FACTOR, etc.) cause gcd's to be taken implicitly. For homogeneous polynomials it is recommended that GCD:RED be used. To take the gcd when an algebraic is present, e.g. GCD(X^2-2*SQRT(2)*X+2,X-SQRT(2)); , ALGEBRAIC must be TRUE and GCD must not be EZ.

EZGCD*(p1, p2, ...)* gives a list whose first element is the g.c.d of the polynomials *p1,p2,...* and whose remaining elements are the polynomials divided by the g.c.d. This always uses the EZGCD algorithm (not recommended for homogeneous polynomials).

MOD*(p)* converts the polynomial *p* to a modular representation with respect to the current modulus which is the value of the variable MODULUS.

If MODULUS[FALSE] is set to a positive prime p, then all arithmetic in the rational function routines will be done modulo p. That is all integers will be reduced to less than p/2 in absolute value (if p=2 then all integers are reduced to 1 or 0). This is the so called "balanced" modulus system, e.g. N MOD 5 = -2, -1, 0, 1, or 2.

RESULTANT*(p1, p2, var)* computes the resultant of the two polynomials *p1* and *p2*, eliminating the variable *var*. The resultant is a determinant of the coefficients of *var* in *p1* and *p2* which equals zero if and only if *p1* and *p2* have a non-constant factor in common.

MODRESULT[FALSE] if TRUE causes the modular resultant algorithm to be used, otherwise the reduced (which is the default) will be used (see [Col]).

```
(C1)    RESULTANT(A*Y+X**2+1,Y**2+X*Y+B,X);

             4       3                2    2
(D1)        Y  + A Y  + (2 B + 1) Y  + B
```

RATDIFF*(exp, var)* differentiates the rational expression *exp* (which must be a ratio of polynomials or a polynomial in the variable *var*) with respect to *var*. For rational expressions this is much faster than DIFF. The result is left in CRE form. However, RATDIFF should not be used on factored CRE forms; use DIFF instead for such expressions.

```
(C1) (4*X**3+10*X-11)/(X**5+5);
                                        3
                                     4 X  + 10 X - 11
(D1)                                 ----------------
                                          5
                                         X  + 5

(C2) MODULUS:3$

(C3) MOD(D1);
                              2
                             X  + X - 1
(D3)                     --------------------
                          4    3    2
                         X  + X  + X  + X + 1

(C4) RATDIFF(D1,X);
                          5    4    3
                         X  - X  - X  + X - 1
(D4)                 ----------------------------
                      8    7    5    4    3
                     X  - X  + X  - X  + X  - X + 1
```

6.5.1 Algebraic Integers

An algebraic integer is a solution of a univariate monic polynomial equation with integer coefficients. Examples of algebraic integers are 2+3*%I, SQRT(7), and 6^(1/3)-5^(1/7). In addition to the factorization of polynomials over the ring of integers with an algebraic integer adjoined, MACSYMA provides simplification of expressions involving algebraic integers by representing them in a canonically simplified form, in which there are no radicals in the denominators of fractions.

TELLRAT*(poly1,...,polyn)* adds to the ring of algebraic integers known to MACSYMA, the elements which are the solutions of the univariate, monic polynomials *polyj* (integer coefficients). MACSYMA initially knows about %I and all roots of integers. TELLRAT() returns a list of the polynomials given to TELLRAT. To SAVE or KILL all of one's TELLRATs, just do SAVE (...,TELLRATS,...) or KILL(...,TELLRATS,...). To undo a TELLRAT(p(X)), simply do TELLRAT(X).

ALGEBRAIC[FALSE] must be set to TRUE in order for the simplfication of algebraic integers to take effect.

RATALGDENOM[TRUE] if TRUE allows rationalization of denominators wrt. radicals to take effect. To do this one must use CRE form in algebraic mode.

```
(C1) ALGEBRAIC:RATALGDENOM:TRUE$

(C2) RATDIS(E):=RATDISREP(RAT(E))$

(C3) 10*(1+%I)/(3^(1/3)+%I);
                                     10 (%I + 1)
(D3)                                 -----------
                                      1/3
                                     3    + %I

(C4) RATDIS(%);
                       2/3                 1/3
(D4)    (4 %I + 2) 3      + (4 - 2 %I) 3      - 4 %I - 2
```

```
(C5) TELLRAT(A^2+A+1)$

(C6) A/(SQRT(2)+SQRT(3))+1/(A*SQRT(2)-1);
                              1                A
(D6)                    ------------- + -----------------
                        SQRT(2) A - 1   SQRT(3) + SQRT(2)

(C7) RATDIS(%);
       (7 SQRT(3) - 10 SQRT(2) + 2) A - 2 SQRT(2) - 1
(D7)  ------------------------------------------------
                               7
```

6.5.2 Functions for Extended Rational Expressions

An extended rational expression is a truncated power series with rational functions for coefficients (as generated by TAYLOR). The truncation capability (RATWEIGHT,RATWTLVL) described above (see 6.5) is utilized by extended CRE forms as well as by CRE forms.

TAYLOR*(exp,[var1,pt1,ord1],[var2,pt2,ord2],...)* returns a truncated power series in the variables *vari* about the points *pti*, truncated at *ordi*. For further details see 6.7.

PSEXPAND[FALSE] if TRUE will cause extended rational function expressions to display fully expanded. (RATEXPAND will also cause this.) If FALSE, multivariate expressions will be displayed just as in the rational function package. If PSEXPAND:MULTI, then terms with the same total degree in the variables are grouped together.

SRRAT*(exp)* converts *exp* from extended rational form to CRE form, i.e. it is like RAT(RATDISREP(*exp*)) although much faster.

```
(C1)                    TAYLOR(1 + X, [X, 0, 3]);

(D1)/T/                        1 + X + . . .
```

```
(C2) 1/%;

                                        2     3
(D2)/T/                      1 - X + X  - X  + . . .

(C3) TAYLOR(1 + X + Y + Z, [X, 0, 3], [Y, 1, 2],
        [Z, 2, 1]);

(D3)/T/               4 + (Z - 2) + (Y - 1) + X + . . .


(C4) 1/%;
```

```
               1   Z - 2     1    Z - 2
(D4)/T/  - - ----- + (- -- + ----- + . . .) (Y - 1)
               4    16        16    32

    1    3 (Z - 2)                 2
 + (-- - --------- + . . .) (Y - 1)
    64      256

     1    Z - 2    1    3 (Z - 2)
 + (- -- + ----- + (-- - --------- + . . .) (Y - 1)
     16    32      32      128

      3    3 (Z - 2)                 2
 + (- --- + --------- + . . .) (Y - 1)  + . . .) X
     256      256

    1    3 (Z - 2)       3    3 (Z - 2)
 + (-- - --------- + (- --- + --------- + . . .) (Y - 1)
    64      256         256     256

     3    15 (Z - 2)                 2         2
 + (--- - ---------- + . . .) (Y - 1)  + . . .) X
    512      2048

       1    Z - 2      1     5 (Z - 2)
 + (- --- + ----- + (--- - --------- + . . .) (Y - 1)
      256    256      256     1024

       5     15 (Z - 2)                 2          3
 + (- ---- + ---------- + . . .) (Y - 1)  + . . .) X
      2048      4096

 + . . .
```

[6.6] Poisson Series Functions

A Poisson series is a finite sum where each term has the form p*trig(q) where "trig" is either SIN or COS . Usually, p is a polynomial with rational number or floating point coefficients, or a general MACSYMA expression. The argument q is a linear combination of no more than 6 variables, whose names are literally U, V, W, X, Y, and Z. (These restrictions are not vital, but apparently present no difficulty in usual applications. They could be altered easily).

Conversion to a Poisson series expands all products or powers of sines and/or cosines into sums. In order to display the result, it is usually necessary to convert an expression in Poisson encoding into general MACSYMA representation using the OUTOFPOIS function, or to print it using the PRINTPOIS function.

POISSIMP*(A)* converts A into a Poisson series for A in general representation.

INTOPOIS*(A)* converts A into a Poisson encoding.

OUTOFPOIS*(A)* converts A from Poisson encoding to general representation. If A is not in Poisson form, it will make the conversion, i.e. it will look like the result of OUTOFPOIS(INTOPOIS(A)). This function is thus a canonical simplifier for sums of powers of SIN's and COS's of a particular type.

PRINTPOIS*(A)* prints a Poisson series in a readable format. In common with OUTOFPOIS, it will convert A into a Poisson encoding first, if necessary.

POISTIMES*(A, B)* is functionally identical to INTOPOIS(A*B).

POISTRIM*()* is a reserved function name which (if the user has defined it) gets applied during Poisson multiplication. It is a predicate function of 6 arguments which are the coefficients of the U, V,..., Z in a term. Terms for which POISTRIM is TRUE (for the coefficients of that term) are eliminated during multiplication.

POISPLUS*(A, B)* is functionally identical to INTOPOIS(A+B).

POISEXPT*(A, B)* (B a positive integer) is functionally identical to INTOPOIS(A**B).

POISDIFF*(A, B)* differentiates A with respect to B. B must occur only in the trig arguments or only in the coefficients.

POISINT*(A, B)* integrates in a similarly restricted sense (to POISDIFF). Non-periodic terms in B are dropped if B is in the trig arguments.

POISSUBST*(A, B, C)* substitutes A for B in C. C is a Poisson series.
(1) Where B is a variable U, V, W, X, Y, or Z then A must be an expression linear in those variables (e.g. 6*U+4*V).

(2) Where B is other than those variables, then A must also be free of those variables, and furthermore, free of sines or cosines.

POISSUBST*(A, B, C, D, N)* is a special type of substitution which operates on A and B as in type (1) above, but where D is a Poisson series, expands COS(D) and SIN(D) to order N so as to provide the result of substituting A+D for B in C. The idea is that D is an expansion in terms of a small parameter. For example, POISSUBST(U,V,COS(V),E,3) results in COS(U)*(1-E^2/2) - SIN(U)*(E-E^3/6).

POISMAP*(series, sinfn, cosfn)* will map the functions *sinfn* on the sine terms and *cosfn* on the cosine terms of the poisson series given. *sinfn* and *cosfn* are functions of two arguments which are a coefficient and a trigonometric part of a term in series respectively.

```
(C1) PFEFORMAT:TRUE$

(C2) (2*A^2-B)*COS(X+2*Y)-(A*B+5)*SIN(U-4*X);

(D2)          - (A B + 5) SIN(U - 4 X)

                          2
                   + (2 A  - B) COS(2 Y + X)

(C3) POISEXPT(%,2)$
```

```
(C4) PRINTPOIS(D3);

    2
(2 A  - B) ( - A B - 5) SIN( - 2 Y - 5 X + U)

    2
(2 A  - B) ( - A B - 5) SIN(2 Y - 3 X + U)

                     2
- 1/2 ( - A B - 5)  COS(2 U - 8 X)

        2     2                      2
1/2 (2 A  - B)  + 1/2 ( - A B - 5)

        2     2
1/2 (2 A  - B)  COS(4 Y + 2 X)

(D4)                          DONE

(C5) POISINT(D3,Y)$

(C6) POISSIMP(%);

               2     2
(D6) 1/8 (2 A  - B)

          SIN(4 Y + 2 X) - 1/2 (2 A  - B) ( - A B - 5)
                                     2
                                        2
          COS(2 Y - 3 X + U) + 1/2 (2 A  - B) ( - A B - 5)

          COS( - 2 Y - 5 X + U)

(C7) OUTOFPOIS(SIN(X)^5+COS(X)^5);

(D7) 1/16 SIN(5 X) + 1/16 COS(5 X) - 5/16 SIN(3 X)

          + 5/16 COS(3 X) + 5/8 SIN(X) + 5/8 COS(X)
```

One or two final points: the coefficients in the arguments of the trig functions must fit in a pre-arranged domain. Initially this is set to [-15,16] but can be set to $[-2^{n-1}+1, 2^{n-1}]$ by the variable POISLIM[5]. This should not be done in the middle of a calculation. Also, it is possible to define the coefficient arithmetic to be almost anything. The user (probably in conjunction with a LISP-MACSYMA programmer) must define the programs needed to add, multiply, substitute, encode and decode the coefficients. The encoding for +1 and -1 and a program to test for 0 (zero), completes each package. These packages are available for coefficients being CRE form, polynomials with floating point coefficients, and polynomials with rational number coefficients, in addition to the default general MACSYMA form.

If all coefficients of trig terms are desired in CRE form, the user should LOADFILE(POIS3,FASL,DSK,MACSYM) and LOADFILE(RATPOI,FASL,DSK,MACSYM). Only those variables on the RATVARS list can be used in the coefficients. In many instances this is a much more efficient technique in terms of speed.

6.7 Taylor Series

TAYLOR*(exp, var, pt, pow)* expands the expression *exp* in a truncated Taylor series (or Laurent series, if required) in the variable *var* around the point *pt*. The terms through *(var-pt)**pow* are generated. If *exp* is of the form f(*var*)/g(*var*) and g(*var*) has no terms up to degree *pow* then TAYLOR will try to expand g(*var*) up to degree 2**pow*. If there are still no non-zero terms TAYLOR will keep doubling the degree of the expansion of g(*var*) until reaching *pow**2**n where n is the value of the variable TAYLORDEPTH[3].
If MAXTAYORDER[TRUE] is TRUE, then during algebraic manipulation of (truncated) Taylor series, TAYLOR will try to retain as many terms as are certain to be correct.

```
(C1) TAYLOR(SQRT(1+A*X+SIN(X)),X,0,3);


                                 2             2
                 (A + 1) X   (A   + 2 A + 1) X
(D1)/R/   1 + --------- - -----------------
                    2                8

                      3      2             3
                (3 A   + 9 A   + 9 A - 1) X
              + -------------------------- + . . .
                           48

(C2) %**2;
                                   3
                                  X
(D2)/R/       1 + (A + 1) X - -- + . . .
                                  6

(C3) PRODUCT((X**I+1)**2.5,I,1,INF)/(X**2+1);

                           INF
                          /===\
                           ! !      I     2.5
                           ! !  (X   + 1)
                           ! !
                           ! !
                          I = 1
(D3)                      -----------------
                                  2
                                 X  + 1

(C4) TAYLOR(%,X,0,3),KEEPFLOAT:TRUE;

                                           2           3
(D4)/R/       1.0 + 2.5 X + 3.375 X  + 6.5625 X  + . . .

(C5) TAYLOR(1/LOG(1+X),X,0,3);

                                         2        3
                       1    1    X     X     19 X
(D5)/R/                - + - - -- + -- - ----- + . . .
                       X    2   12    24    720
```

Multivariate Taylor Series Expansions

For multivariate functions, there are several ways of obtaining Taylor series expansions. If the variables are truly independent and all singularities involve only one variable at a time then the expansion may be done as follows:

TAYLOR(*exp, var1, pt1, ord1, var2, pt2, ord2,...)*

or

TAYLOR(*exp,[var1,pt1,ord1],[var2,pt2,ord2],...)*

Naturally the two techniques may be intermixed.

However, if the variables are interdependent or singularities involving some of the variables together can occur then the following scheme is to be used:

TAYLOR(*exp, [var1, var2, . . .], pt, ord) where each of pt* and *ord* may be replaced by a list which will correspond to the list of variables. that is, the nth items on each of the lists will be associated together.

The user should be warned that this scheme uses the RATWTLEVEL scheme implicitly whenever the variables are expanded to different orders. In this case the user must not be trying to use RATWTLEVEL simultaneously.

Internally this is done in the following manner; for each X_i substitute

$$X_i \text{ -----> } T^{n_i} W_i.$$

Then a term like $X^2 Y^3 Z$ would become

$$T^{2 n_1 + 3 n_2 + n_3} W_1{}^2 W_2{}^3 W_3$$

and truncation is done on T. The W variables as well as T are not seen by the user. The following are examples of the various modes of Taylor expansions.

```
(C5) TAYLOR(SIN(X+Y),X,0,3,Y,0,3);
```

```
                     3       2        2     3  2     3
                    Y       Y  X   Y X     Y  X     X
(D5)/R/         Y - -- + X - ---- - ---- + ----- - --
                    6         2      2      12      6

                         2  3
                        Y  X
                      + ----- + . . .
                         12

(C6) TAYLOR(SIN(X+Y),[X,Y],0,3);
                                 3      2       2     3
                                X  + 3 Y X  + 3 Y  X + Y
(D6)/R/                 Y + X - ------------------------ + . . .
                                           6

(C7) TAYLOR(1/SIN(X+Y),X,0,3,Y,0,3);
                                                2    3
                       1   Y   X     X   X     X
(D7)/R/                - + - - -- + - + -- - -- + . . .
                       Y   6    2   6    3    4
                               Y        Y    Y
(C8) TAYLOR(1/SIN(X+Y),[X,Y],0,3);
                 1      X + Y
(D8)/R/        ----- + -----
               X + Y     6

          3        2          2        3
       7 X  + 21 Y X  + 21 Y X  +  7 Y
     + -------------------------------- + . . .
                     360
```

If one wants to handle asymptotic expansions a facility exists to some extent. It may be invoked as follows.

TAYLOR*(exp, [x,pt,ord,ASYMP])* will give an expansion of *exp* in negative powers of (*x-pt*). The highest order term will be

$$(x - pt)^{-ord}$$

The *ASYMP* is a syntactic device and not to be assigned to; for example, one types TAYLOR(F(X),[X,0,4,ASYMP]).

If the user is expanding polynomials he may specify a truncation level of INF in which case the expansion will never truncate.

DEFTAYLOR*(function, exp)* allows the user to define the Taylor series (about 0) of an arbitrary *function* of one variable as *exp* which may be a polynomial in that variable or which may be given implicitly as a power series using the SUM function.

In order to display the information given to DEFTAYLOR one can use POWERSERIES(F(X),X,0). (see below).

```
(C1) DEFTAYLOR(F(X),X**2+SUM(X**I/(2**I*I!**2),
        I,4,INF));
(D1)                        [F]

(C2) TAYLOR(%E**SQRT(F(X)),X,0,4);

                    2        3          4
                   X    3073 X    12817 X
(D2)/R/    1 + X + -- + ------- + -------- + . . .
                   2     18432     307200
```

TAYLORINFO*(exp)* returns FALSE if *exp* is not a Taylor series. Otherwise, a list of lists is returned describing the particulars of the Taylor expansion. For example,

```
(C3) TAYLOR((1-Y^2)/(1-X),X,0,3,[Y,A,INF]);
                 2                        2
(D3)/R/ 1 - A   - 2 A (Y - A) - (Y - A)

                    2                       2
          + (1 - A   - 2 A (Y - A) - (Y - A) ) X

        2                       2  2
 + (1 - A   - 2 A (Y - A) - (Y - A) ) X

                    2                       2  3
          + (1 - A   - 2 A (Y - A) - (Y - A) ) X

 + . . .
(C4) TAYLORINFO(D3);
(D4)                     [[Y, A, INF], [X, 0, 3]]
```

POWERSERIES*(exp, var, pt)* generates the general form of the power series expansion for *exp* in the variable *var* about the point *pt* (which may be INF for infinity). In cases in which POWERSERIES is unable to expand *exp* the TAYLOR function may give the first several terms of the series.

VERBOSE[FALSE] - if TRUE will cause comments about the progress of POWERSERIES to be printed as the execution of it proceeds.

```
(C2) VERBOSE:TRUE$

(C3) POWERSERIES(LOG(SIN(X)/(1-X^2)),X,0);

CAN'T EXPAND

                                  LOG(X - 1)
```

```
SO WE WILL TRY AGAIN AFTER APPLYING THE RULE:

                                   d
                                 / -- (X - 1)
                                 [ dX
                LOG(X - 1) = I -------- dX
                                 ]  X - 1
                                 /

CAN'T EXPAND

                           LOG(SIN(X))

SO WE WILL TRY AGAIN AFTER APPLYING THE RULE:
                                   d
                                 / -- SIN(X)
                                 [ dX
                LOG(SIN(X)) = I ------- dX
                                 ] SIN(X)
                                 /


IN FIRST SIMPLIFICATION WE HAVE RETURNED:

         /              /
         [              [   1
         I COT(X) dX - I ----- dX - LOG(X + 1) + LOG(- 1)
         ]              ] X - 1
         /              /


IS  I2  ZERO OR NONZERO?
ZERO;
```

```
TRYING TO DO A RATIONAL FUNCTION EXPANSION OF

                                          1
                                        -----
                                        X - 1

USING A SPECIAL RULE FOR EXPRESSIONS OF FORM

                                              M  - N
                                   (A + C VAR   )

HERE WE HAVE

                          [N = 1, A = - 1, C = 1, M = 1]

        INF
        ====        I3  2 I3                             I3 + 1
        \      (- 1)   2      BERN(2 I3) LOG(X)        X
(D4)  ( >      ------------------------------------ + -------)
        /                   (2 I3)!                    I3 + 1
        ====
        I3 = 0


                          INF
                          ====          I2  I2
                          \        (- 1)   X
                      + ( >        -----------)  +  LOG(-1)
                          /            I2
                          ====
                          I2 = 1
```

6.8 Trigonometric Simplification

TRIGEXPAND*(exp)* expands trigonometric and hyperbolic functions of sums of angles and of multiple angles occurring in *exp*. For best results, *exp* should be expanded. To enhance user control of simplification, this function expands only one level at a time, expanding sums of angles or multiple angles. To obtain full expansion into sines and cosines immediately, set the switch TRIGEXPAND:TRUE.

TRIGEXPAND[FALSE] if TRUE causes expansion of all expressions containing SINs and COSs occurring subsequently.

HALFANGLES[FALSE] - if TRUE causes half-angles to be simplified away.

```
(C1) X+SIN(3*X)/SIN(X),TRIGEXPAND=TRUE,EXPAND;

                                2            2
(D1)                     - SIN (X) + 3 COS (X) + X
(C2) TRIGEXPAND(SIN(10*X+Y));

(D2)               COS(10 X) SIN(Y) + SIN(10 X) COS(Y)
```

TRIGREDUCE*(exp, var)* combines products and powers of trigonometric and hyperbolic SINs and COSs of *var* into those of multiples of *var*. It also tries to eliminate these functions when they occur in denominators. If *var* is omitted then all variables in *exp* are used. Also see the POISSIMP function (6.6).

```
(C4) TRIGREDUCE(D1);
(D4)                         2 COS(2 X) + X + 1
```

The trigonometric simplification routines will use declared information in some simple cases. Declarations about variables (see 10.8) are used as follows, e.g.

```
(C5) DECLARE(J, INTEGER, E, EVEN, O, ODD)$

(C6) SIN(X + (E + 1/2)*%PI)$

(D6)                    COS(X)

(C7) SIN(X + (O + 1/2) %PI);

(D7)                   - COS(X)
```

There are a number of ways the user may also explicitly invoke identities such as $\sin^2(x) + \cos^2(x) = 1$. The simplest method is substitution.

```
(C8) D1;
                                       2           2
(D8)                           - SIN (X) + 3 COS (X) + X

(C9) %,SIN(X)^2=1-COS(X)^2;
                                           2
(D9)                                 4 COS (X) + X - 1
```

Often one wishes to recognize that $\sin^4(x)$ can be transformed using the same rule. For this one needs the added power of RATSUBST.

```
(C10) RATSUBST(1-COS(X)^2,SIN(X)^2,SIN(X)^4);

                                    4           2
(D10)                            COS (X) - 2 COS (X) + 1
```

In general RATSUBST will perform a RATSIMP (and thus an expansion) as well as apply the substitution. One can similarly use the LETSIMP and DEFRULE commands together with additional declarations to define more intricate rules.

Although not as powerful as RATSUBST, the TELLSIMP command enables the automatic application of a rule.

```
(C11) TELLSIMP(SIN(X)^2,1-COS(X)^2)$

(C12) (SIN(X)+1)^2;
                                                2
(D12)                                (SIN(X) + 1)

(C13) EXPAND(%);
                                                2
(D13)                            2 SIN(X) - COS (X) + 2

(C14) SIN(X)^2;
```

```
                                           2
(D14)                                1 - COS (X)
```

6.9 Laplace Transforms

LAPLACE*(exp, ovar, lvar)* takes the Laplace transform of *exp* with respect to the variable *ovar* and transform parameter *lvar*. *Exp* may only involve the functions EXP, LOG, SIN, COS, SINH, COSH, and ERF. It may also be a linear, constant coefficient differential equation in which case ATVALUE (see 6.2.2) of the dependent variable will be used. These may be supplied either before or after the transform is taken. Since the initial conditions must be specified at zero, if one has boundary conditions imposed elsewhere he can impose these on the general solution and eliminate the constants by solving the general solution for them and substituting their values back. *exp* may also involve convolution integrals. Functional relationships must be explicitly represented in order for LAPLACE to work properly. That is, if F depends on X and Y it must be written as F(X,Y) wherever F occurs as in LAPLACE('DIFF(F(X,Y),X),X,S).

```
(C1) LAPLACE(%E**(2*T+A)*SIN(T)*T,T,S);

                          A
                      2 %E  (S - 2)
(D1)                 ---------------
                               2     2
                      ((S - 2)  + 1)


(C2) LAPLACE(DELTA(T-A)*SIN(B*T),T,S);

Is A positive,negative or zero?

POS;
                                                 - A S
(D2)                                 SIN(A B) %E
```

ILT*(exp, lvar, ovar)* takes the inverse Laplace transform of *exp* with respect to *lvar* and parameter *ovar*. *exp* must be a ratio of polynomials whose denominator has only linear and quadratic factors. By using the functions LAPLACE and ILT together with the SOLVE or LINSOLVE functions the user can solve a single differential or convolution integral equation or a set of them.

```
(C1) 'INTEGRATE(SINH(A*X)*F(T-X),X,0,T)+B*F(T)=T**2;

                 T
                /
                [                                         2
(D1)            I (SINH(A X) F(T - X)) DX + B F(T) = T
                ]
                /
                0

(C2) LAPLACE(%,T,S);

             A LAPLACE(F(T), T, S)                                    2
(D2)         --------------------- + B LAPLACE(F(T), T, S) = --
                    2    2                                            3
                   S  - A                                             S


(C3) LINSOLVE([%],['LAPLACE(F(T),T,S)]);
SOLUTION

                                                  2      2
                                               2 S  - 2 A
(E3)            LAPLACE(F(T), T, S) = --------------------
                                           5          2     3
                                        B S  + (A - A  B) S

(D3)                                  [E3]

(C4) ILT(E3,S,T);

Is  A B (A B - 1)  positive, negative, or zero?

POS;
```

```
                                              2
                         SQRT(A) SQRT(A B  - B) T
                   2 COSH(------------------------)
                                     B
(D4)  F(T) =  - ------------------------------
                                  A

               2
             A T             2
         + ------- + ------------------
           A B - 1    3 2       2
                     A  B  - 2 A  B + A
```

Laplace Transforms of Special Functions

The following function for taking Laplace transforms of Special Functions is available in MACSYMA. The user must type LOADFILE(SPECFN,LISP,DSK,SHARE); to load in the special routines. Since the latter take up a great deal of space, it is recommended that these computations be carried out in a fresh MACSYMA.

LAPINT*(exp,ovar)* takes the Laplace transform of *exp* with respect to the variable *ovar*. *exp* may involve

1) Special Functions of linear or quadratic argument multiplied by

a) arbitrary powers of the argument, or
b) trigonometric and exponential functions of linear argument

2) Products of two Special Functions of linear or quadratic argument taken from only one of the following groups:

a) Any kind of Bessel, Modified Bessel, or Hankel functions,
b) Orthogonal polynomials,
c) Confluent Hypergeometric Functions

In this second category, factors of type 1a or 1b are also permitted.

The basic method is to rewrite the expression in terms of Generalized Hypergeometric Functions (GHF), apply a general formula for taking the Laplace transform of GHF's, and then, if possible, present the result in terms of elementary functions or "common" Special Functions. For further details, see *Symbolic Laplace Transforms of Special Functions* [Av].

```
(C1) T^(1/2)*GAMMAINCOMPLETE(1/2,A*T)*%E^(-P*T);

                                   1                 - P T
(D1)                GAMMAINCOMPLETE(-, A T) SQRT(T) %E
                                   2
(C2) LAPINT(%,T);

                        %PI                            2
(D2)       -------------------------- - -------------------------
                    3/2      A   3/2           3/2      A   3/2
           2 (P + A)    (1 - -----)    (P + A)    (1 - -----)
                             P + A                     P + A

(C3) T^(1/2)*J[1](2*A^(1/2)*T^(1/2))*%E^(-P*T);

                                                  - P T
(D3)                 J (2 SQRT(A) SQRT(T)) SQRT(T) %E
                      1

(C4) LAPINT(%,T);

                                       - A/P
                               SQRT(A) %E
(D4)                           ---------------
                                       2
                                      P

(C5) T^2*J[1](A*T)*%E^(-P*T);

                                     2   - P T
(D5)                         J (A T) T  %E
                              1
```

```
(C6) LAPINT(%,T);

                                    3 A
(D6)                           --------------
                                 2
                                A      5/2  4
                               (-- + 1)    P
                                 2
                                P
(C7) T^(3/2)*Y[1](A*T)*%E^(-T);

                                      3/2   - T
(D7)                         Y (A T) T    %E
                              1
(C8) LAPINT(%,T);

                                         %I      1        3/4
                15 %I SQRT(2) P       (- --) (------ - 1)
                               - 2, 1/2  A      2
                                              A  + 1
(D8)            --------------------------------------------
                               2     2    2     2     1/4
                 8 SQRT(%PI) (A  + 1)  ((A  + 1)  - 1)
```

6.10 Combinatorial Functions

MINFACTORIAL*(exp)* examines *exp* for occurrences of two factorials which differ by an integer. It then turns one into a polynomial times the other.

```
(C1) N!/(N+1)!;
                                   N!
(D1)                            --------
                                (N + 1)!

(C2) MINFACTORIAL(%);
                                    1
(D2)                              -----
                                  N + 1
```

FACTCOMB*(exp)* tries to combine the coefficients of factorials in *exp* with the factorials themselves by converting, for example, (N+1)*N! into (N+1)!.

SUMSPLITFACT[TRUE] if set to FALSE will cause MINFACTORIAL to be applied after a FACTCOMB.

```
(C1) (N+1)^B*N!^B;
                                          B   B
(D1)                               (N + 1)  N!

(C2) FACTCOMB(%);
                                              B
(D1)                                 (N + 1)!
```

MAKEFACT*(exp)* transforms occurrences of binomial,gamma, and beta functions in *exp* to factorials.

MAKEGAMMA*(exp)* transforms occurrences of binomial,factorial, and beta functions in *exp* to gamma functions.

BERNPOLY*(v, n)* generates the *n*th Bernoulli polynomial in the variable *v*.

6.11 Continued Fractions

CF*(exp)* converts *exp* into a continued fraction. *exp* is an expression composed of arithmetic operators and lists which represent continued fractions. A continued fraction a+1/(b+1/(c+...)) is represented by the list [a,b,c,...]. a,b,c,.. must be integers. *Exp* may also involve SQRT(*n*) where *n* is an integer. In this case CF will give as many terms of the continued fraction as the value of the variable CFLENGTH[1] times the period. Thus the default is to give one period.

CFDISREP*(list)* converts the continued fraction represented by *list* into general representation.

```
(C1) CF([1,2,-3]+[1,-2,1]);

(D1)                   [1, 1, 1, 2]

(C2) CFDISREP(%);
                             1
(D2)                 1 + ---------
                                 1
                          1 + -----
                                   1
                              1 + -
                                   2
```

CFEXPAND*(x)* gives a matrix of the numerators and denominators of the next-to-last and last convergents of the continued fraction *x*.

```
(C1) (CFLENGTH:4, CF(SQRT(3));
(D1)                   [1, 1, 2, 1, 2, 1, 2, 1, 2]

(C2) CFEXPAND(%);
                                 [265  97]
(D2)                             [       ]
                                 [153  56]

(C3) D2[1,2]/D2[2,2],NUMER;
(D3)                             1.73214285
```

6.12 Number-Theoretic Functions

PRIME*(n)* gives the *n*th prime. MAXPRIME[489318] is the largest number accepted as argument.

DIVSUM*(n,k)* adds up all the factors of *n* raised to the *k*th power. If only one argument is given then *k* is assumed to be 1.

TOTIENT*(n)* is the number of integers less than *n* which are relatively prime to *n*. Also *n* - DIVSUM(n,0) + 1.

JACOBI*(p,q)* is the Jacobi symbol of p and q.

QUNIT*(n)* gives the principal unit of the real quadratic number field SQRT(*n*) where *n* is an integer, i.e. the element whose norm is unity. This amounts to solving Pell's equation A**2-*n**B**2=1.

```
(C1) QUNIT(17);
(D1)                SQRT(17)+4

(C2) EXPAND(%*(SQRT(17)-4));

(D2)                1
```

7 Declaring and Using Mathematical Information

The commands in this chapter deal with the communication, use, and manipulation of mathematical information about objects and functions in MACSYMA. Taken as a whole, this information comprises MACSYMA's *relational data base*. Facts take the form of either "predicates" or "features". A predicate is a well-formed formula consisting of a relation and its arguments, e.g. A>B, or is a composition of predicates using the logical operators NOT, AND, and OR. Alternatively, certain facts about mathematical objects and functions can be expressed more naturally as "features", i.e. unary predicates. For example, one can say that a certain constant is an INTEGER or that a function is INCREASING. Any feature, e.g. the linearity of F, can also be expressed as a predicate via the relation KIND, as in KIND(F,LINEAR).

7.1 Declaring and Assuming

The predicates and features communicated to MACSYMA with the ASSUME and DECLARE commands may be tested with IS and FEATUREP and removed with FORGET and REMOVE. The facts in the relational data base are used by the simplifier and several commands, like SIGN, the IF statement, and INTEGRATE (certain integrations require sign information). MACSYMA has a rudimentary inference capability enabling limited deductions from the data base. It excels at taxonomic deductions, e.g. KIND(N,EVEN) implies KIND(N,INTEGER), and simple expression comparisons, e.g. X<0 and KIND(N,EVEN) imply $Y^2+X^N>0$. The only sort of inequality information used by the inference mechanism at the moment are relations between variables and other variables and numbers.

The operator "=" is a total relation that holds between two expressions if and only if the expressions are syntactically identical. It is not a mathematical comparison. Thus, IS((X+1)^2=X^2+2*X+1) would return FALSE. The relation EQUAL, on the other hand, is a mathematical comparison of its two arguments. A predicate involving EQUAL is true if and only if its arguments are mathematically equivalent in light of the current data base. Thus, IS(EQUAL((X+1)^2,X^2+2*X+1)) would return TRUE. The operators ">", ">=", "<", and "<=" are also mathematical comparisons.

MACSYMA currently recognizes and uses the following features of objects: EVEN, ODD, INTEGER, RATIONAL, IRRATIONAL, REAL, IMAGINARY, and COMPLEX. The useful features of functions include: INCREASING, DECREASING, ODDFUN (odd function), EVENFUN (even function), COMMUTATIVE (or SYMMETRIC), ANTISYMMETRIC, ASSOCIATIVE.

ASSUME*(pred1, pred2, ...)* first checks the specified predicates for redundancy and consistency with the current data base. If the predicates are consistent and non-redundant, they are added to the data base; if inconsistent or redundant, no action is taken. ASSUME returns a list whose entries are the predicates added to the data base and the atoms REDUNDANT or INCONSISTENT where applicable. ASSUME also accepts lists of predicates as arguments.

FORGET*(pred1, pred2, ...)* removes the specified predicates from the data base. Note that it does not guarantee that equivalent facts are removed. FORGET also accepts lists of predicates as arguments.

DECLARE*(a1, f1, a2, f2, ...)* declares each ai to have the corresponding feature *fi*. DECLARE(F, INCREASING) is in all respects equivalent to ASSUME(KIND(F,INCREASING)). The *ai* and *fi* may also be lists of objects or features.

REMOVE*(a1, f1, a2, f2, ...)* removes each feature fi from the corresponding object ai. The ai and fi may also be lists of objects or features.

IS*(pred)* attempts to determine whether the specified predicate is provable from the facts in the current data base. IS returns TRUE if the predicate is true for all values of its variables consistent with the data base and returns FALSE if it is false for all such values. Otherwise, its action depends on the setting of the switch PREDERROR[TRUE]. IS errs out if the value of PREDERROR is TRUE and returns UNKNOWN if PREDERROR is FALSE.

FEATUREP*(a,f)* attempts to determine whether the object *a* has the feature *f* on the basis of the facts in the current data base. If so, it returns TRUE, else FALSE.

```
(C1) DECLARE(J,EVEN)$

(C2) FEATUREP(J,INTEGER);
```

```
(D2)                                TRUE
```

SIGN*(exp)* attempts to determine the sign of its specified expression on the basis of the facts in the current data base. It returns one of the following answers: POS (positive), NEG (negative), ZERO, PZ (positive or zero), NZ (negative or zero), PN (positive or negative), or PNZ (positive, negative, or zero, i.e. nothing known).

```
(C3) ASSUME(A>=B,B>=C,C>=D,D>=A);
(D3)                   [A >= B, B >= C, C >= D, D >= A]

(C4) SIGN(B-C);
(D4)                                ZERO

(C5) DECLARE(K,INTEGER,L,ODD,F,INCREASING)$

(C6) ASSUME(X>0);
(D6)                               [X > 0]

(C7) F(X+3*Y^(L+24*K+1))-F(0);
                                 L + 24 K + 1
(D7)                       F(3 Y             + X) - F(0)

(C8) SIGN(%);
(D8)                                POS
```

ASKSIGN*(exp)* first attempts to determine whether the specified expression is positive, negative, or zero. If it cannot, it asks the user the necessary questions to complete its deduction.[1] The user's answer is recorded in the data base for the duration of the current computation (one "C-line"). The value of ASKSIGN is one of POS, NEG, or ZERO.

The following function, when applicable, gives the user relational information. However, it does *NOT* use the data base.

1. If the user wishes to look at the expression more closely before replying, the variable ASKEXP is set to it. Typing control-A results in a MACSYMA break (see 4); the user may now analyze the expression in order to give an appropriate answer.

ZEROEQUIV*(exp,var)* tests whether the expression *exp* in the single variable *var* is equivalent to zero. It returns either TRUE, FALSE, or DONTKNOW. For example ZEROEQUIV(SIN(2*X) - 2*SIN(X)*COS(X),X) returns TRUE and ZEROEQUIV(%E^X+X,X) returns FALSE. On the other hand ZEROEQUIV(LOG(A*B) - LOG(A) - LOG(B),A) will return DONTKNOW because of the presence of an extra variable. The restrictions are:

(1) Do not use functions that MACSYMA does not know how to differentiate and evaluate.

(2) If the expression has poles on the real line, there may be errors in the result (but this is unlikely to occur).

(3) If the expression contains functions which are not solutions to first order differential equations there may be incorrect results.

(4) The algorithm uses floating-point evaluation at randomly chosen points using a corresponding "epsilon" for carefully selected subexpressions. This is always somewhat hazardous, although the algorithm tries to minimize the potential for error.

7.2 Contexts

The context mechanism makes it possible for a user to bind together and name a selected portion of his data base, called a *context*. Once this is done, the user can have MACSYMA assume or forget large numbers of facts merely by activating or deactivating their context. Any atom can be a context, and the facts contained in that context will be retained in storage until the user destroys them individually by using FORGET or destroys them as a whole by using KILL to destroy the context to which they belong.

FACTS*(context)* returns a list of the facts in the specified *context*.

ACTIVATE*(cont1, cont2, ...)* causes the facts in the specified contexts *contk* to be added to the current data base.

DEACTIVATE*(cont1, cont2, ...)* causes the facts in the specified contexts *contk* to be removed from the current data base, unless specified by some other active context.

CONTEXT[GLOBAL]. Whenever a user assumes a new fact, it is placed in the context named as the current value of the variable CONTEXT. Similarly, FORGET references the current value of CONTEXT. To add or delete a fact from a different context, one must bind CONTEXT to the intended context and then perform the desired additions or deletions. The context specified by the value of CONTEXT is automatically activated. All of MACSYMA's built-in relational knowledge is contained in the default context GLOBAL.

CONTEXTS[[]] is a list of the currently active contexts, not including the value of CONTEXT.

```
(C9) CONTEXT:CON1$

(C10) DECLARE(M,INTEGER)$

(C11) FEATUREP(M,INTEGER);
(D11)                            TRUE

(C12) CONTEXT:CON2$

(C13) FEATUREP(M,INTEGER);
(D13)                           FALSE

(C14) DECLARE(N,INTEGER);
(D14)                            DONE

(C15) CONTEXT:CON1$

(C16) FEATUREP(M,INTEGER);
(D16)                            TRUE

(C17) FEATUREP(N,INTEGER);
(D17)                           FALSE

(C18) ACTIVATE(CON2)$

(C19) FEATUREP(N,INTEGER);
(D19)                            TRUE
```

```
(C13) CONTEXTS;
(D13)                                   [CON2]
```

LOCAL*(a1, a2, ...)* causes the external facts about the objects *a1, a2, ...* to be forgotten for the duration of the enclosing BLOCK, independent of context. Any facts assumed about *a1, a2, ...* within the BLOCK containing the LOCAL will be forgotten upon exit, again independent of context.

```
(C14) DECLARE(P,INTEGER)$

(C15) BLOCK(LOCAL(P),PRINT(FEATUREP(P,INTEGER)),ASSUME(P,IRRATIONAL))$
FALSE

(C16) FEATUREP(P,INTEGER);
(D16)                                   TRUE

(C17) FEATUREP(P,IRRATIONAL);
(D17)                                   FALSE
```

8 List Handling and LISP-like functions

APPLY*(function, list)* gives the result of applying the *function* to the *list* of its arguments. This is useful when it is desired to compute the arguments to a function before applying that function. For example, if L is the list [1, 5, -10.2, 4, 3], then APPLY(MIN,L) gives -10.2. APPLY is also useful when calling functions which do not have their arguments evaluated if it is desired to cause evaluation of them. For example, if FILESPEC is a variable bound to the list [TEST, CASE] then APPLY(CLOSEFILE,FILESPEC) is equivalent to CLOSEFILE(TEST,CASE). In general the first argument to APPLY should be preceded by a ' to to make it evaluate to itself. Since some atomic variables have the same name as certain functions the values of the variable would be used rather than the function because APPLY has its first argument evaluated as well as its second.

FUNMAKE*(name,[arg1,...,argn])* returns *name*(arg1,...,argn) without calling the function *name*.

ARRAYMAKE*(name,[i1,i2,...])* returns *name*[*i1,i2,...*].

MAP*(fn, exp1, exp2, ...)* returns an expression whose leading operator is the same as that of the *expi* but whose subparts are the results of applying *fn* to the corresponding subparts of the *expi*. *Fn* is either the name of a function of n arguments (where n is the number of *expi*) or is a LAMBDA form of n arguments.

MAPERROR[TRUE] - if FALSE will cause all of the mapping functions to (1) stop when they finish going down the shortest *expi* if not all of the *expi* are of the same length and (2) apply *fn* to [*exp1, exp2,...*] if the *expi* are not all the same type of object. If MAPERROR is TRUE then an error message will be given in the above two instances.

One of the uses of this function is to MAP a function (e.g. PARTFRAC) onto each term of a very large expression where it ordinarily wouldn't be possible to use the function on the entire expression due to an exhaustion of list storage space in the course of the computation.

```
(C1) MAP(F,X+A*Y+B*Z);
(D1)                          F(B Z) + F(A Y) + F(X)
(C2) MAP(LAMBDA([U],PARTFRAC(U,X)),X/(X^3+4*X^2+5*X+2));
                                       1        X
(D2)                             X (----- - --------)
                                    X + 2          2
                                            (X + 1)

(C3) MAP(RATSIMP, X/(X**2+X)+(Y**2+Y)/Y);

                              1
(D3)                  Y + ----- + 1
                          X + 1

(C4) MAP("=",[A,B],[-.5, 3,2.5]);
MAP IS TRUNCATING.
(D4)                    [A = -.5, B = 3]
```

MAPATOM*(expr)* is TRUE if and only if *expr* is treated by the MAPping routines a as an "atom", a unit. "Mapatoms" are atoms, numbers (including rational numbers), and subscripted variables.

MAPLIST*(fn, exp1, exp2, ...)* yields a list of the applications of *fn* to the parts of the *expi*. This differs from MAP(*fn,exp1,exp2,...*) which returns an expression with the same main operator as expi has (except for simplifications and the case where MAP does an APPLY). *Fn* is of the same form as in MAP.

FULLMAP*(fn, exp1, ...)* is similar to MAP but it will keep mapping down all subexpressions until the main operators are no longer the same.
The user should be aware that FULLMAP is used by the MACSYMA simplifier for certain matrix manipulations; thus, the user might see an error message concerning FULLMAP even though FULLMAP was not explicitly called by the user.

```
(C1) A+B*C$

(C2) FULLMAP(G,%);

(D2)                G(B) G(C) + G(A)
```

```
(C3)  MAP(G,D1);

(D3)                   G(B C) + G(A)
```

FULLMAPL*(fn, list1, ...)* is similar to FULLMAP but it only maps onto lists and matrices

```
(C1) FULLMAPL("+",[3,[4,5]],[[A,1],[0,-1.5]]);

(D1)                        [[A + 3, 4], [4, 3.5]]
```

SCANMAP*(function,exp)* recursively applies *function* to *exp*. This is most useful when "complete" factorization is desired, for example:

```
(C1) EXP:(A^2+2*A+1)*Y + X^2$

(C2) SCANMAP(FACTOR,EXP);
                                        2       2
(D2)                             (A + 1)  Y + X
```

Note the way in which SCANMAP applies the given function FACTOR to the constituent subexpressions of *exp*; if another form of *exp* is presented to SCANMAP then the result may be different. Thus, D2 is not recovered when SCANMAP is applied to the expanded form of *exp*:

```
(C3) SCANMAP(FACTOR,EXPAND(EXP));
                                 2                2
(D3)                            A  Y + 2 A Y + Y + X
```

Here is another example of the way in which SCANMAP recursively applies a given function to all subexpressions, including exponents:

```
(C4) EXPR : U*V^(A*X+B) + C$

(C5) SCANMAP('F, EXPR);
```

```
                           F(F(F(A) F(X)) + F(B))
(D5) F(F(F(U) F(F(V)                              )) + F(C))
```

APPEND*(list1, list2, ...)* returns a single list of the elements of *list1* followed by the elements of *list2*,...

```
(C1) APPEND([Y+X,0,-3.2],[2.5E20,X]);

(D1)                [Y+X, 0, -3.2, 2.5E20, X]
```

CONS*(exp, list)* returns a new list constructed of the element *exp* as its first element, followed by the elements of *list*.

ENDCONS*(exp, list)* returns a new list consisting of the elements of *list* followed by *exp*.

MEMBER*(exp, list)* returns TRUE if *exp* occurs as a member of *list* (not within a member). Otherwise FALSE is returned.

REVERSE*(list)* reverses the order of the members of *list* (not the members themselves).

The functions **FIRST, REST, LAST, DELETE, LENGTH (6.2.3)** also work on lists.

Examples

```
(C1) UNION(X,Y):=IF X = [] THEN Y ELSE
            IF MEMBER(T:FIRST(X),Y) THEN UNION(REST(X),Y)
            ELSE CONS(T,UNION(REST(X),Y)$

(C2) UNION([A,B,1,1/2,X**2],[-X**2,A,Y,1/2]);

                           2          2         1
(D2)                     [X , 1, B, - X , A, Y, -]
                                                2
```

In this example T is assigned the value of FIRST(X) in the call to MEMBER and is referenced later in CONS(T,UNION(...)).

```
(C3) BERNPOLY(X,5);
                                4      3
                        5    5 X    5 X    X
(D3)                    X  - ---- + ---- - -
                              2      3     6

(C4) MAPLIST(NUMFACTOR,%);
                              5  5   1
(D4)                    [1, - -, -, - -]
                              2  3   6

(C5) APPLY(MIN,%);
                                5
(D5)                          - -
                                2
```

[8.1] Property Specification Functions

The functions in this section are used to specify properties for atomic variables. A property is a piece of information which may be utilized during the user's session with MACSYMA. MODEDECLARE (sec. 10.8) is one example of a property specification function. Also, the subsequent section deals with functions for pattern matching and includes the function MATCHDECLARE. As these are somewhat complicated they are described in other sections. However, along with DECLARE (see below) and other functions, they all perform the task of setting up information which is used later.

For most types of properties there exists a name which is an indicator of that property. For example the command GRADEF(F(X),SIN(X**2)); causes F to receive a "gradef" property of LAMBDA([X],SIN(X**2)). (The indicator is GRADEF and the property is the lambda expression).

The presence of some properties is signified merely by the presence of the indicator and requires no additional information. These indicators are known as flags. For example %I has associated with it the flag CONSTANT. We can also speak of "constant" as being a property of %I.

There are three principal operations which are needed by the user in dealing with properties. He must be able to set up the property, to display it, and to remove it. Also he may sometimes want to know what properties he set up in his MACSYMA. There are several lists (known as information lists) which contain all of the atoms that possess a particular property. Moreover, the value of the variable INFOLISTS is a list of the names of all of the information lists in MACSYMA. These are:

LABELS - all bound C,D, and E labels.

VALUES - all bound atoms (set up by : , :: , or functional binding).

FUNCTIONS - all user defined functions (set up by f(x):=...).

ARRAYS - declared and undeclared arrays (set up by : , :: , or a[x]:=...)

MYOPTIONS - all options ever reset by the user (whether or not they get reset to their default value).

RULES - user defined pattern matching and simplification rules (set up by TELLSIMP, TELLSIMPAFTER, DEFMATCH, or, DEFRULE.)

ALIASES - atoms which have a user defined alias (set up by the ALIAS, ORDERGREAT, ORDERLESS functions or by DECLAREing the atom a NOUN).

DEPENDENCIES - atoms which have functional dependencies (set up by the DEPENDS or GRADEF functions).

GRADEFS - functions which have user defined derivatives (set up by the GRADEF function).

PROPS - atoms which have any property other than those mentioned above, such as atvalues, matchdeclares, etc. as well as properties specified in the DECLARE function.

In addition to these information lists similar lists may be generated by the function PROPVARS*(prop)* which yields a list of those atoms on the PROPS list which have the property indicated by *prop*. Thus PROPVARS(ATVALUE) will yield a list of atoms which have atvalues.

8.1.1 Functions for Handling MACSYMA Properties

DECLARE*(a1, f1, a2, f2, ...)* gives the atom *ai* the flag *fi*. The *ai*'s and *fi*'s may also be lists of atoms and flags respectively in which case each of the atoms gets all of the properties. The possible flags and their meanings are:

CONSTANT - makes *ai* a constant as is %PI.

NONSCALAR - makes *ai* behave as does a list or matrix with respect to the dot operator. (see 6.4.1)

NOUN - makes the function *ai* a noun so that it won't be evaluated automatically. (see 3.2)

ALPHABETIC - adds *ai* to MACSYMA's alphabet (initially A-Z,%). Thus, DECLARE("_",ALPHABETIC) enables NEW_VALUE to be used as a name.

EVFUN - makes *ai* known to the EV function so that it will get applied if its name is mentioned. Initial evfuns are FACTOR, TRIGEXPAND, TRIGREDUCE, BFLOAT, RATSIMP, RATEXPAND, RADCAN, and LOGCONTRACT.

EVFLAG - makes *ai* known to the EV function so that it will be bound to TRUE during the execution of EV if it is mentioned. Initial evflags are FLOAT, PRED, SIMP, NUMER, DETOUT, EXPONENTIALIZE, DEMOIVRE, KEEPFLOAT, LISTARITH, TRIGEXPAND, SIMPSUM, ALGEBRAIC, RATALGDENOM, FACTORFLAG, and LOGEXPAND.

BINDTEST - causes *ai* to signal an error if it ever is used in a computation unbound (see Chapter 19).

ARRAYINFO*(a)* returns a list of information about the array *a*. For hashed arrays it returns a list of "HASHED", the number of subscripts, and the subscripts of every element which has a value. For declared arrays it returns a list of "DECLARED", the number of subscripts, and the bounds that were given the the ARRAY function when it was called on *a*.

```
(C1) B[1,X]:1$
(C2) ARRAY(F,2,3)$
(C3) ARRAYINFO(B);
```

```
(D3)            [HASHED, 2, [1, X]]
(C4) ARRAYINFO(F);

(D4)            [DECLARED, 2, [2, 3]]
```

PROPERTIES*(a)* will yield a list showing the names of all the properties associated with the atom *a*.

PRINTPROPS*(a, i)* will display the property with the indicator *i* associated with the atom *a*. *a* may also be a list of atoms or the atom ALL in which case all of the atoms with the given property will be used. For example, PRINTPROPS([F,G],ATVALUE). PRINTPROPS is for properties that cannot otherwise be displayed, i.e. for ATVALUE,ATOMGRAD,GRADEF, and MATCHDECLARE.

REMOVE*(a1, p1, a2, p2, ...)* removes the property *pi* from the atom *ai*. *Ai* and *pi* may also be lists as with DECLARE. *Pi* may be any property e.g. FUNCTION, MODEDECLARE, etc. It may also be TRANSFUN implying that the translated LISP version of the function is to be removed. This is useful if one wishes to have the MACSYMA version of the function executed rather than the translated version. *Pi* may also be OP or OPERATOR to remove a syntax extension given to *ai* (see Appendix II). If *ai* is "ALL" then the property indicated by *pi* is removed from all atoms which have it. Unlike the more specific remove functions (REMVALUE, REMARRAY, REMFUNCTION, and REMRULE) REMOVE does not indicate when a given property is non-existent; it always returns "DONE".

8.1.2 Functions for Handling Users' Properties

PUT*(a, p, i)* associates with the atom *a* the property *p* with the indicator *i*. This enables the user to give an atom any arbitrary property.

QPUT*(a, p, i)* is similar to PUT but it doesn't have its arguments evaluated.

GET*(a, i)* retrieves the user property indicated by *i* associated with atom *a* or returns FALSE if *a* doesn't have property *i*.

```
(C1) PUT(%E,TRANSCENDENTAL,TYPE);
(D1)                              TRANSCENDENTAL
(C2) PUT(%PI,TRANSCENDENTAL,TYPE)$

(C3) PUT(%I,ALGEBRAIC,TYPE)$

(C4) TYPEOF(X) := BLOCK([Q], IF NUMBERP(X) THEN RETURN(ALGEBRAIC),
                   IF NOT ATOM(X) THEN RETURN(MAPLIST(TYPEOF, X)),
                   Q : GET(X, TYPE),
                   IF Q=FALSE THEN ERROR("NOT NUMERIC") ELSE Q)$

(C5) TYPEOF(2*%E+X*%PI);

NOT NUMERIC
QUIT
(C6) TYPEOF(2*%E+%PI);
(D6)           [[ALGEBRAIC, TRANSCENDENTAL], TRANSCENDENTAL]
```

REM*(a, i)* removes the property indicated by *i* from the atom *a*.

NUMERVAL*(var1, exp1, var2, exp2, ...)* declares *vari* to have a numerical value of *exp1* which is evaluated and substituted for the variable in any expressions in which the variable occurs if the NUMER flag is TRUE. (see the EV function, sec. 6.1.1).

9 Pattern Matching and Related Functions

[9.1] Type Testing Functions

ATOM*(exp)* is TRUE if *exp* is atomic (i.e. a number or name) else FALSE. Thus ATOM(5) is TRUE while ATOM(A[1]) and ATOM(SIN(X)) are FALSE. (Assuming A[1] and X are unbound.)

SUBVARP*(exp)* is TRUE if *exp* is a subscripted variable, for example A[I].

CONSTANTP*(exp)* is TRUE if *exp* is a constant (i.e. composed of numbers and %PI, %E, %I or any variables bound to a constant or DECLAREd constant (see 8.1.1) else FALSE. Any function whose arguments are constant is also considered to be a constant.

NONSCALARP*(exp)* is TRUE if *exp* is a non-scalar, i.e. it contains atoms declared as non-scalars (see 8.1), lists, or matrices.

INTEGERP*(exp)* is TRUE if *exp* is an integer else FALSE.

EVENP*(exp)* is TRUE if *exp* is an even integer. FALSE is returned in all other cases.

ODDP*(exp)* is TRUE if *exp* is an odd integer. FALSE is returned in all other cases.

FLOATNUMP*(exp)* is TRUE if *exp* is a floating point number else FALSE.

BFLOATP*(exp)* is TRUE is *exp* is a bigfloat number else FALSE.

NUMBERP*(exp)* is TRUE if *exp* is an integer, a rational number, a floating point number or a bigfloat else FALSE.

RATNUMP*(exp)* is TRUE if *exp* is a rational number (includes integers) else FALSE.

LISTP*(exp)* is TRUE if *exp* is a list else FALSE.

MATRIXP*(exp)* is TRUE if *exp* is a matrix else FALSE.

RATP*(exp)* is TRUE if *exp* is in CRE or extended CRE form else FALSE.

FREEOF*(x1, x2, ..., exp)* yields TRUE if the *xi* do not occur in *exp* and FALSE otherwise. The *xi* are atoms or they may be subscripted names, functions (e.g. SIN(X)), or operators enclosed in "s. They may also be lists of these objects.

```
(C1) FREEOF(Y,SIN(X+2*Y));

(D1)                     FALSE

(C2) FREEOF(COS(Y),"*",SIN(Y)+COS(X));

(D2)                          TRUE
```

[9.2] General Pattern Matching Functions

The pattern matching functions permit the user to test expressions for combinations of syntactic and semantic patterns and to automatically have variables set to parts of expressions which fit the patterns. This enables one to transform an expression as well as to see if it fits a pattern.

It is also possible to add simplification rules which apply to user or system defined functions or operators. In order to choose the particular rule which applies, a pattern match must usually be performed on the operands of the expression which is to be simplified.

MATCHDECLARE*(patternvar, predicate, ...)* associates a *predicate* with a *pattern variable* so that the *variable* will only match expressions for which the *predicate* is not FALSE. (The matching is accomplished by one of the functions described below). For example after MATCHDECLARE(Q,FREEOF(X,%E)) is executed, Q will match any expression not containing X or %E. If the match succeeds then the variable is set to the matched expression. The predicate (in this case FREEOF) is written without the last argument which should be the one against which the pattern variable is to be tested. Note that the *patternvar* and the arguments to the *predicate* are evaluated at the time the match is performed.

The odd numbered argument may also be a list of pattern variables all of which are to have the associated predicate. Any even number of arguments may be given.

For pattern matching, predicates refer to functions which are either FALSE or not FALSE (any non FALSE value acts like TRUE).

MATCHDECLARE(var,TRUE) will permit var to match any expression.

PRINTPROPS([*v1,v2*,...],MATCHDECLARE) (see 8.1.1) will display the matchdeclare properties of the variables *v1,v2*,...

TELLSIMPAFTER*(pattern, replacement)* defines a *replacement* for *pattern* which the MACSYMA simplifier uses after it applies the built-in simplification rules. The *pattern* may be anything but a single variable or a number.

TELLSIMP*(pattern, replacement)* is similar to TELLSIMPAFTER but places new information before old so that it is applied before the built-in simplification rules. The *pattern* may not be a sum, product, single variable, or number.
RULES is a list of names having simplification rules added to them by DEFRULE, DEFMATCH, TELLSIMP, or TELLSIMPAFTER.

```
(C1) MATCHDECLARE([XX,A,B],TRUE);
(D1)                                    DONE

(C2)
TELLSIMP(D[XX](A,B),B(XX)*DIFF(A(XX),X)-A(XX)*DIFF(B(XX),XX));
RULE PLACED ON SUBVAR
(D2)                         [SUBVARRULE1, FALSE]
```

SUBVARRULE1 is the name assigned to the TELLSIMP rule from (C34).

```
(C3) D[Z](X,Y);
(D3)                        Y(Z) X(Z)  - X(Z) Y(Z)
                                  X            Z
```

Another example of the use of TELLSIMP is shown in the following:

```
(C4) 0^0;
0^0 HAS BEEN GENERATED
```

To override such default simplification, the user can use the following paradigm:

```
(C5) BLOCK([SIMP],SIMP:FALSE,TELLSIMP(0^0,1));

RULE PLACED ON **
(D5)                         [**RULE1, SIMPEXPT]

(C6) 0^0;
(D6)                                  1

(C7) REMRULE("**","**RULE1");
(D7)                               [ SIMPEXPT ]
```

DEFMATCH(*progname, pattern, parm1, ..., parmn*) creates a function of n+1 arguments with the name *progname* which tests an expression to see if it can match a particular *pattern*. The *pattern* is some expression containing pattern variables and *parameters*. The *parms* are given explicitly as arguments to DEFMATCH while the pattern variables (if supplied) were given implicitly in a previous MATCHDECLARE function. The first argument to the created function *progname*, is an expression to be matched against the "*pattern*" and the other n arguments are the actual variables occurring in the expression which are to take the place of dummy variables occurring in the "*pattern*". Thus the parms in the DEFMATCH are like the dummy arguments to the SUBROUTINE statement in FORTRAN. When the function is "called" the actual arguments are substituted. For example:

```
(C1)  NONZEROANDFREEOF(X,E):=  IF E#0 AND FREEOF(X,E)
              THEN TRUE ELSE FALSE$
```

(IS(E#0 AND FREEOF(X,E)) is an equivalent function definition - see sec. 8.1.1.

```
(C2)  MATCHDECLARE(A,NONZEROANDFREEOF(X),B,FREEOF(X))$
(C3)  DEFMATCH(LINEAR,A*X+B,X)$
```

This has caused the function LINEAR(*exp,var1*) to be defined. It tests *exp* to see if it is of the form A**var1*+B where A and B do not contain *var1* and A is not zero. DEFMATCHed functions return (if the match is successful) a list of equations whose left sides are the pattern variables and parms and whose right sides are the expressions which the pattern variables and parameters matched. The pattern variables, but not the parameters, are set to the matched expressions. If the match fails, the function returns FALSE. Thus LINEAR(3*Z+(Y+1)*Z+Y**2,Z) would return [B=Y**2, A=Y+4, X=Z]. Any variables not declared as pattern variables in MATCHDECLARE or as parameters in DEFMATCH which occur in *pattern* will match only themselves so that if the third argument to the DEFMATCH in (C4) had been omitted, then LINEAR would only match expressions linear in X, not in any other variable.

A pattern which contains no parameters or pattern variables returns TRUE if the match succeeds.

```
(C1) MATCHDECLARE([A,F],TRUE)$

(C2) CONSTINTERVAL(L,H):=CONSTANTP(H-L)$

(C3) MATCHDECLARE(B,CONSTINTERVAL(A))$

(C4) MATCHDECLARE(X,ATOM)$

(C5) BLOCK(REMOVE(INTEGRATE,LINEAR),
              DEFMATCH(CHECKLIMITS,'INTEGRATE(F,X,A,B),
              DECLARE(INTEGRATE,LINEAR))$

    (C6) 'INTEGRATE(SIN(T),T,X+%PI,X+2*%PI)$

    (C7) CHECKLIMITS(%);

    (D7)  [B = X + 2 %PI, A = X + %PI, X = T,

                                      F = SIN(T)]

    (C8) 'INTEGRATE(SIN(T),T,0,X)$

    (C9) CHECKLIMITS(%);

    (D9)                      FALSE
```

DEFRULE*(rulename, pattern, replacement)* defines and names a *replacement* rule for the given *pattern*. If the rule named *rulename* is applied to an expression (by one of the APPLY functions below), every subexpression matching the *pattern* will be replaced by the *replacement*. All variables in the *replacement* which have been assigned values by the pattern match are assigned those values in the *replacement* which is then simplified. The rules themselves can be treated as functions which will transform an expression by one operation of the pattern match and replacement. If the pattern fails, the original expression is returned.

APPLY1*(exp, rule1, ..., rulen)* repeatedly applies the first rule to *exp* until it fails, then repeatedly applies the same rule to all subexpressions of *exp*, left-to-right, until the first rule has failed on all subexpressions. Call the result of transforming *exp* in this manner *exp'*. Then the second rule is applied in the same fashion starting at the top of *exp'*. When the final rule fails on the final subexpression, the application is finished.

APPLY2*(exp, rule1, ..., rulen)* differs from APPLY1 in that if the first rule fails on a given subexpression, then the second rule is repeatedly applied, etc. Only if they all fail on a given subexpression is the whole set of rules repeatedly applied to the next subexpression. If one of the rules succeeds, then the same subexpression is reprocessed, starting with the first rule.

MAXAPPLYDEPTH[10000] is the maximum depth to which APPLY1 and APPLY2 will delve.

APPLYB1*(exp, rule1, ..., rulen)* is similar to APPLY1 but works from the "bottom up" instead of from the "top down". That is, it processes the smallest subexpression of *exp*, then the next smallest, etc.
MAXAPPLYHEIGHT[10000] - is the maximum height to which APPLYB1 will reach before giving up.

[9.3] Pattern Matching for Rational Expressions

LETSIMP*(exp)* will continually apply the substitution rules previously defined by the function LET (see below) until no further change is made to *exp*.

LET*(prod, repl, predname, arg1, arg2, ..., argn)* defines a substitution rule for LETSIMP such that *prod* gets replaced by *repl*. *prod* is a product of positive or negative powers of the following types of terms:

(1) *Atoms* which LETSIMP will search for literally unless previous to calling LETSIMP the MATCHDECLARE function is used to associate a predicate with the atom. In this case LETSIMP will match the atom to any term of a product satisfying the predicate.

(2) *Kernels* such as SIN(X), N!, F(X,Y), etc. As with atoms above LETSIMP will look for a literal match unless MATCHDECLARE is used to associate a predicate with the argument of the kernel.

A term to a positive power will only match a term having at least that power in the expression being LETSIMPed. A term to a negative power on the other hand will only match a term with a power at least as negative. In the case of negative powers in "product" the switch LETRAT must be set to TRUE (see below).

If a predicate is included in the LET function followed by a list of arguments, a tentative match (i.e. one that would be accepted if the predicate were omitted) will be accepted only if *predname(arg1',...,argn')* evaluates to TRUE where *argi'* is the value matched to *argi*. The *argi* may be the name of any atom or the argument of any kernel appearing in *prod*. *repl* may be any rational expression. If any of the atoms or arguments from *prod* appear in *repl* the appropriate substitutions will be made.

LETRAT[FALSE] when FALSE, LETSIMP will simplify the numerator and denominator of *expr* independently and return the result. Substitutions such as N!/N goes to (N-1)! will fail. To handle such situations LETRAT should be set to TRUE, then the numerator, denominator, and their quotient will be simplified in that order.

These substitution functions allow you to work with several rule packages at once. Each rule package can contain any number of LETed rules and is refered to by a user supplied name. To insert a rule into the rule package *name*, do LET(*[prod,repl,pred,arg1,...]*,*name*). To apply the rules in rule package *name*, do LETSIMP(*expr*, *name*). The function LETSIMP(*expr,name1,name2,...*) is equivalent to doing LETSIMP(*expr,name1*) followed by LETSIMP(*%,name2*) etc.

There is a default rule package name which is assumed when no other name is supplied to any of the functions. Whenever a LET includes a rule package name the default rule package is made to look like that rule package.

REMLET*(prod, name)* deletes the substitution rule, *prod* --> *repl*, most recently defined by the LET function. If *name* is supplied the rule is deleted from the rule package *name*. REMLET() and REMLET(ALL,*name*) delete all substitution rules from the default rule package. If *name* is supplied the rule package, *name*, is also deleted.
If a substitution is to be changed using the same product, REMLET need not be called, just redefine the substitution using the same product (literally) with the LET function and the new replacement and/or predicate name. Should REMLET(product) now be called the original substitution rule will be revived.

LETRULES*(name)* and LETRULES() display the rules in the default rule package and the rule package, *name*, respectively. Note that the function LETRULES(*name*) will set the default rule package to the rule package, *name*.

```
(C1) MATCHDECLARE([A1,A2],TRUE)$

(C2) ONELESS(X,Y):=IS(EQUAL(X,Y-1))$

(C3) LET(A1*A2!,A1!,ONELESS,A2,A1);
```

```
(D3)          A1 A2! --> A1! WHERE ONELESS(A2, A1)

(C4) LETRAT:TRUE$

(C5) LET(A1!/A1,(A1-1)!);

                             A1!
(D5)                         --- --> (A1 - 1)!
                             A1

(C6) LETSIMP(N*M!*(N-1)!/M);

(D6)                          (M - 1)! N!
```

The user should be aware that simplification rules for differential operators can be specified using MACSYMA's pattern-matching commands.

Consider a function F(X). To inform MACSYMA that F depends on X, the user must type DEPENDS(F,X); (otherwise, DIFF(F,X) will return 0). We will assume that this has been done and that DERIVABBREV has been set to TRUE in the following example.

Now suppose that the function F(X) satisfies some constraint, say that the d'Alembertian of f(x) is zero:

$$\Box^2 f(x) = 0.$$

In a curved space, this may take the form:

```
(C4) -2*(DIFF(F,X)*X +2)
        *(L*(DIFF(F,X)^3)*X^2
           +(4*DIFF(F,X,2)+4*(DIFF(F,X)^2))*X+8*DIFF(F,X));
                               3   2                  2
(D4)   - 2 (F  X + 2) ((F )  L X  + (4 F   + 4 (F ) ) X + 8 F )
             X            X               X X      X           X
```

One can solve for the second-order term:

```
(C5) SOLVE(%,DIFF(F,X,2));
Solution

                                     3   2          2
                              (F )  L X  + 4 (F )  X + 8 F
                                X             X          X
(E5)               F      = - ----------------------------
                    X X                    4 X
(D5)                                   [E5]
```

which can be restated as a simplification rule:

```
(C6) LET(DIFF(F,X,2),RHS(E5));
(D6)                          (F      --> RHS(E5))
                                X X
```

Then a relatively complicated expression such as

```
            4  2  3                           3     2
(D10) ((F )  L  X  + (8 F  F    L + 8 (F )  L) X
         X                X  X X         X

                2
      + ((F )  (12 L + 16) + 16 F    ) X + 32 F )
           X                     X X          X

                                2    2
                     /(X ((F )  L X  + 4 F  X + 4))
                            X              X
```

can be simplified using the LETSIMP command:

```
(C11) FACTOR(LETSIMP(%));

                                              2
(D11)                               - (F )  L
                                        X
```

10 Utility, Input-Output, and Display Functions

[10.1] Debugging Functions

The functions in this section permit the user to examine his MACSYMA environment and to obtain debugging information. Further detail is given in section 12.0.

TRACE*(name1, name2, ...)* gives a trace printout whenever the functions mentioned are called. TRACE() prints a list of the functions currently under TRACE.

UNTRACE*(name1, ...)* removes tracing incurred by the TRACE function. UNTRACE() removes tracing from all functions.

REMTRACE*()* removes the tracing facilities from MACSYMA thus freeing up some storage. They will be reloaded when TRACE is used again.

DECLARE*([var1, var2, ...], BINDTEST)* causes MACSYMA to give an error message whenever any of the *vari* occur unbound in a computation.

BREAK*(arg1, ...)* evaluates and prints its arguments then enters a MACSYMA break loop.

Options and Variables

%% is the value of the last computation performed while in a (MACSYMA-BREAK).

DEBUGMODE[FALSE] if TRUE causes MACSYMA to enter a MACSYMA break loop whenever a MACSYMA error occurs. If DEBUGMODE:ALL then the user may examine BACKTRACE for the list of functions currently entered.

REFCHECK[FALSE] if TRUE causes a message to be printed each time a bound variable is used for the first time in a computation.

PREDERRCR[TRUE] - if TRUE causes a message to be printed whenever the predicate of an IF statement or an IS function fails to evaluate to either TRUE or FALSE.

SETCHECK[FALSE] - if set to a list of variables (which can be subscripted) will cause a printout whenever the variables, or subscripted occurrences of them, are bound (with : or :: or function argument binding). The printout consists of the variable and the value it is bound to. SETCHECK may be set to ALL or TRUE thereby including all variables.

SETCHECKBREAK[FALSE] - if set to TRUE will cause a (MACSYMA-BREAK) to occur whenever the variables on the SETCHECK list are bound.

BACKTRACE (when DEBUGMODE:ALL has been done) has as value a list of all functions currently entered. (see Chapter 19).

10.2 Functions for Displaying

DISPFUN*(f1, f2, ...)* displays the definition of the user defined functions *f1, f2, ...* which may also be the names of array associated functions, subscripted functions, or functions with constant subscripts which are the same as those used when the functions were defined. DISPFUN(ALL) will display all user defined functions as given on the **FUNCTIONS** and **ARRAYS** lists except subscripted functions with constant subscripts.

DISPRULE*(rulename)* will display a rule with the name *rulename* as was given by DEFRULE, TELLSIMP, or TELLSIMPAFTER or a pattern defined by DEFMATCH. For example, the first rule modifying SIN will be called SINRULE1. (see 9.2)

DISPLAY*(exp1, exp2, ...)* displays equations whose left side is *expi* unevaluated, and whose right side is the value of the expression centered on the line. This function is useful in blocks and FOR statements in order to have intermediate results displayed. The arguments to DISPLAY are usually atoms, subscripted variables, or function calls. (see the DISP function below.)

```
(C1) DISPLAY(B[1,2]);

                                                  2
                                  B       = X - X
                                   1, 2

(D1)                                    DONE
```

LDISPLAY*(exp1,exp2,...)* is like DISPLAY but also generates intermediate labels.

DISP*(exp1,exp2, ...)* is like DISPLAY but only the value of the arguments are displayed rather than equations. This is useful for complicated arguments which don't have names or where only the value of the argument is of interest and not the name.

LDISP*(exp1,exp2,...)* is like DISP but also generates intermediate labels.

PRINT*(exp1, exp2, ...)* evaluates and displays its arguments one after the other "on a line" starting at the leftmost position. If *expi* is unbound or is preceded by a single quote or is enclosed in "s then it is printed literally. For example, PRINT("THE VALUE OF X IS ",X). The value returned by PRINT is the value of its last argument. No intermediate lines are generated.

DISPTERMS*(exp)* displays its argument in parts one below the other. That is, each term in a sum or factor in a product is displayed separately. This is useful if *exp* is too large to be otherwise displayed. For example if P1, P2, ... are very large expressions then the display program may run out of storage space in trying to display P1+P2+... all at once. However, DISPTERMS(P1+P2+...) will display P1, then below it P2, etc. When not using DISPTERMS, if an exponential expression is too wide to be displayed as A**B it will appear as EXPT(A,B) (or as NCEXPT(A,B) in the case of A^^B).

REVEAL*(exp,depth)* will display *exp* to the specified integer *depth* with the length of each part indicated. Sums will be displayed as SUM(n) and products as PRODUCT(n) where n is the number of subparts of the sum or product. Exponentials will be displayed as EXP.

```
(C1) INTEGRATE(1/(X^3+2),X)$

(C2) REVEAL(%,2);
(D2)                PRODUCT(3) + PRODUCT(3) + PRODUCT(3)

(C3) REVEAL(D1,3);
                      EXPT LOG                     EXPT LOG
(D3)                - -------- + EXPT EXPT ATAN + --------
                         6                            3
```

PLAYBACK*(arg)* "plays back" input and output lines. If *arg*=n (a number) the last n expressions (Ci, Di, and Ei count as 1 each) are "played-back", while if *arg* is omitted, all lines are. If *arg*=INPUT then only input lines are played back. If *arg*=[m,n] then all lines with numbers from m to n inclusive are played-back. If m=n then [m] is sufficient for arg. *Arg*=SLOW places PLAYBACK in a slow-mode similar to DEMO's (as opposed to the "fast" BATCH). This is useful in conjunction with SAVE or STRINGOUT (see below) when creating a secondary-storage file in order to pick out useful expressions. If *arg*=TIME then the computation times are displayed as well as the expressions. *arg*=NOSTRING displays all input lines when playing back rather than STRINGing them. If *arg*=GRIND then the display will be in a more readable format. One may include any number of options as in PLAYBACK([5,10],20,TIME,SLOW).

STRING*(exp)* converts *exp* to MACSYMA's linear notation (similar to FORTRAN's) just as if it had been typed in and puts *exp* into the buffer for possible editing (in which case *exp* is usually Ci) (see sec. 14.2). The STRING'ed expression should not be used in a computation.

STRINGOUT*(args)* will output an expression to a file in a linear format. STRINGOUT([filespec],...,FUNCTIONS,...) puts all the user's function definitions in the specified file. (see 10.4)
GRIND[FALSE] if TRUE will cause the STRING, STRINGOUT, and PLAYBACK commands to use "grind" mode instead of "string" mode. For PLAYBACK, "grind" mode can also be turned on (for processing input lines) by specifying GRIND as an option.

GRIND*(arg)* prints out *arg* in a more readable format than the STRING command. It returns a D-line as value.

FORTRAN*(exp)* converts *exp* into a FORTRAN linear expression in legal FORTRAN with 6 spaces inserted at the beginning of each line,continuation lines,and ** rather than ^ for exponentiation. When the option FORTSPACES[FALSE] is TRUE, the FORTRAN command fills out to 80 columns using spaces.

FORTMX*(name,matrix)* converts a MACSYMA *matrix* into a sequence of FORTRAN assignment statements of the form
name(i,j)= <the ij *matrix* element>

DESCRIBE*(function)* prints out the portion of the MACSYMA manual describing the *function*.

EXAMPLE*(function)* does a DEMO of relevant examples involving *function*.

10.3 Functions for Freeing Storage

REMOVE*(args)* will remove some or all of the properties associated with variables or functions. (see 8.1)

REMFUNCTION*(f1, f2, ...)* removes the user defined functions *f1,f2,...* from MACSYMA. If there is only one argument of ALL then all functions are removed.

REMVALUE*(name1, name2, ...)* removes the values of user variables (which can be subscripted) from the system. If name is ALL then the values of all user variables are removed. Values are those items given names by the user as opposed to those which are automatically labeled by MACSYMA as Ci, Di, or Ei.

REMARRAY*(name1, name2, ...)* removes arrays and array associated functions and frees the storage occupied. If name is ALL then all arrays are removed. It may be necessary to use this function if it is desired to redefine the values in a hashed array.

REMRULE*(function, rulename)* will remove a rule with the name *rulename* from the *function* which was placed there by DEFRULE, DEFMATCH, TELLSIMP, or TELLSIMPAFTER. If *rule-name* is ALL, then all rules will be removed. (see example in 9.2)

KILL*(arg1, arg2, ...)* eliminates its arguments from the MACSYMA system. If *argi* is a variable (including a single array element), function, or array, the designated item with all of its properties is removed from core. If *argi*=LABELS then all input, intermediate, and output lines to date (but not other named items) are eliminated. If *argi*=CLABELS then only input lines will be eliminated; if *argi*=ELABELS then only intermediate E-lines will be eliminated; if *argi*=DLABELS only the output lines will be eliminated. If *argi* is the name of any of the other information lists (the elements of the MACSYMA variable INFOLISTS), then every item in that class (and its properties) is KILLed and if *argi*=ALL then every item on every information list previously defined as well as LABELS is KILLed. If *argi*=a number (say n), then the last n lines (i.e. the lines with the last n line numbers) are deleted. If *argi* is of the form [m,n] then all lines with numbers between m and n inclusive are killed. Note that KILL(VALUES) or KILL(variable) will not freeup the storage occupied unless the labels which are pointing to the same expressions are also KILLed. Thus if a large expression was assigned to X on line C7 one should do KILL(D7) as well as KILL(X) to release the storage occupied.

KILL(ALLBUT(*name1,...,namek*)) will do a KILL(ALL) except it will not KILL the names specified.

KILL removes all properties from the given argument; thus KILL(VALUES) will kill <u>all</u> properties associated with <u>every</u> item on the VALUES list. KILL always returns the value "DONE" even if the named item doesn't exist (see 8.1).

The "REMOVE" functions (REMVALUE,REMFUNCTION, REMARRAY,REMRULE) remove a <u>specific</u> property. These functions return a list of names or FALSE (if the specific argument doesn't exist).

MACSYMA options may not be KILLed. The user may do RESET() (see 10.6) to reset all options to their default values.

The error message "NO CORE - FASLOAD" results when either too many FASL files have been loaded in or when allocation level has gotten too high. Note that once this occurs, KILLing expressions will not help. In either of these cases, no amount of killing will cause the size of these spaces to decrease. Killing expressions only causes some spaces to get emptied out but not made smaller.

10.4 Functions Which Reference Disk Files

LOADFILE*(fn1, fn2, DSK, directory)* loads a file as designated by its arguments. This function may be used to bring back quantities that were stored from a prior MACSYMA session by use of the SAVE or STORE functions. If *DSK* and *directory* are omitted then the last directory seen (initially the same as the user's login name or USERS if the user has no file directory) will be used. If *DSK* and *directory* are omitted, *fn2* may also be omitted if *fn1* > is to be loaded in (where > follows the conventions of ITS's file system). *Fn1 fn2* must be a file of LISP functions and expressions, not of MACSYMA command lines, in which case BATCH or DEMO is to be used. (See Chapter 14).

DELFILE*(file-specification)* will delete the file given by the *file-specification*.

BATCH*(file-specification)* reads in and evaluates MACSYMA command lines from a file. (see Chapter 14).

DEMO*(file-specification)* same as BATCH but pauses after each command line and continues when a space is typed. (see Chapter 14).

BATCON*(argument)* continues BATCHing in a file which was interrupted (see 14.4).

WRITEFILE*(DSK, directory)* opens a file for writing.

APPENDFILE*(filename1, filename2, DSK, directory)* is like WRITEFILE(DSK,*directory*) but appends to the file whose name is specified by the first two arguments. A subsequent CLOSEFILE will delete the original file and rename the appended file.

CLOSEFILE*(filename1, filename2)* closes a file opened by WRITEFILE and gives it the name *filename1 filename2*. Thus to save a file consisting of the display of all input and output during some part of a session with MACSYMA the user issues a WRITEFILE, transacts with MACSYMA, then issues a CLOSEFILE. The user can also issue the PLAYBACK function after a WRITEFILE to save the display of previous transactions. (Note that what is saved this way is a copy of the *display* of expressions not the expressions themselves). To save the actual expression in internal form the SAVE function may be used. The expression can then be brought back into MACSYMA via the LOADFILE function. To save the expression in a linear form which may then be BATCHed in later, the STRINGOUT function is used. (see below)

STRINGOUT*(file-specification, A1, A2, ...)* outputs to a file given by *file-specification* ([filename1,filename2,DSK, directory]) the values given by *A1,A2,..* in a MACSYMA readable format. The *file-specification* may be omitted, in which case the default values will be used. (see 15.2 - C) The *Ai* are usually C labels or may be INPUT meaning the value of all C labels. Another option is to make *Ai* FUNCTIONS which will cause all of the user's function definitions to be strungout (i.e. all those retrieved by DISPFUN(ALL)). *Ai* may also be a list [m,n] which means to stringout all labels in the range m through n inclusive. This function may be used to create a file of FORTRAN statements by doing some simple editing on the strungout expressions. The FORTRAN[FALSE] should be set to TRUE, however, to cause exponentiation to be strung as ** rather than as ^, as well as to effect other FORTRAN-like changes. Alternatively, the function FORTRAN can be used (see 10.2)

SAVE*(args)* saves quantities described by its arguments on disk and keeps them in core also. (see 15.3).

STORE*(args)* same as SAVE but doesn't retain quantities in core. (see 15.3).

FASSAVE*(args)* is similar to SAVE but produces a FASL file in which the sharing of subexpressions which are shared in core is preserved in the file created. Hence, expressions which have common subexpressions will consume less space when loaded back from a file created by FASSAVE rather than by SAVE.

UNSTORE*(name1, ...)* brings the named expressions into core that were stored away by use of the STORE function in the current MACSYMA. (see 15.3).

RESTORE*(file-specification)* reinitializes all quantities filed away by a use of the SAVE or STORE functions, in a prior MACSYMA session, from the file given by *file-specification* without bringing them into core. (see 15.4).

REMFILE*()* removes files created by the secondary storage scheme in the MACSYMA under use (see 15.2). REMFILE(ALL) does what REMFILE() does and in addition deletes any files which have been created by the SAVE or STORE functions but which have not been assigned names by the user.

[10.5] Ordering Functions

Aside from declaring a variable to be constant or using options like POWERDISP (see below), the only other way in which a user can alter the ordering of parts of an expression is to set up special aliases for variables which cause them to be alphabetically less than or greater than any other variables. Functions which do this are described below. This technique requires care because although the names have been aliased, they display with their original name. Aside from the input/output phase the two names represent two different symbols and thus expressions which contain both the original name and the alias will not be simplified as the user desires. This is shown in the examples below.

ORDERGREAT(*V1, ..., Vn*) sets up aliases for the variables *V1, ..., Vn* such that *V1* > *V2* > ... > *Vn* > any other variable not mentioned as an argument.

ORDERLESS(*V1, ..., Vn*) sets up aliases for the variables *V1, ..., Vn* such that *V1* < *V2* < ... < *Vn* < any other variable not mentioned as an argument.

Thus the complete ordering scale is:

numerical constants < declared constants <

< first argument to ORDERLESS < ... < last argument to ORDERLESS <

< variables which begin with A < ... < variables which begin with Z <

< last argument to ORDERGREAT < ... < first argument to ORDERGREAT.

ORDERGREATP(*exp1,exp2*) returns TRUE if *exp2* precedes *exp1* in the ordering induced by the variable ordering described above.

ORDERLESSP(*exp1,exp2*) returns TRUE if *exp1* precedes *exp2* in the ordering induced by the variable ordering described above.

UNORDER() removes the aliases created by the last use of the above ordering commands. ORDERGREAT and ORDERLESS may not be used more than one time each without calling UNORDER.

```
(C1) A**2+B*X;
                                        2
(D1)                             B X + A

(C2) ORDERGREAT(A);
(D2)                               DONE

(C3) A**2+B*X;
                                  2
(D3)                             A  + B X

(C4) %-D1;
```

```
                                    2    2
(D4)                               A  - A

(C5) UNORDER();
(D5)                                [A]
```

SORT*(list,optional-predicate)* sorts the *list* using a suitable *optional-predicate* of two arguments (such as "<" or ORDERLESSP). If the *optional-predicate* is not given, then MACSYMA's built-in ordering predicate is used.

10.6 Miscellaneous Functions

TIME*(Di1, Di2, ...)* gives a list of the times in milliseconds taken to compute the *Di*.

LOGOUT*()* causes the user to be logged out and all jobs deleted. This is useful when it is desired to BATCH in a file and have the terminal logged out automatically when the computations are finished. (Equivalent to ^Z and :LOGOUT)

QUIT*()* kills the current MACSYMA but doesn't affect the user's other jobs. (Equivalent to ^Z and :KILL).

READ*(string1, ...)* prints its arguments, then reads in and evaluates one expression. For example: A:READ("ENTER THE NUMBER OF VALUES").

READONLY*(string1,...)* prints its arguments, then reads in an expression (which in contrast to READ is not evaluated).

DEFINE*(f(x1, ...), body)* is equivalent to f(x1,...):="body but when used inside functions it happens at execution time rather than at the time of definition of the function which contains it. (see 3.2)

LOCAL*(v1, v2, ...)* causes the variables *v1,v2,...* to be local with respect to all the properties in the statement in which this function is used (see 2.12). LOCAL may only be used in BLOCKs, in the body of function definitions or LAMBDA expressions, or in the EV function and only one occurrence is permitted in each.

ERROR*(arg1, arg2, ...)* will evaluate and print its arguments and then will cause an error return to top level MACSYMA or to the nearest enclosing ERRCATCH. This is useful for breaking out of nested functions if an error condition is detected, or wherever one can't type control-G.
ERRORFUN[FALSE] - if set to the name of a function of no arguments will cause that function to be executed whenever an error occurs. This is useful in BATCH files where the user may want his MACSYMA killed or his terminal logged out if an error occurs. In these cases ERRORFUN would be set to QUIT or LOGOUT.

ERRCATCH*(exp1, exp2, ...)* evaluates its arguments one by one and returns a list of the value of the last one if no error occurs. If an error occurs in the evaluation of any arguments, ERRCATCH "catches" the error and immediately returns [] (the empty list). This function is useful in BATCH files where one suspects an error might occur which would otherwise have terminated the BATCH if the error weren't caught.

CATCH*(exp1,...,expn)* evaluates its arguments one by one; if the structure of the *expi* leads to the evaluation of an expression of the form THROW(arg), then the value of the CATCH is the value of THROW(arg). This "non-local return" thus goes through any depth of nesting to the nearest enclosing CATCH. There must be a CATCH corresponding to a THROW, else an error is generated. If the evaluation of the *expi* does not lead to the evaluation of any THROW then the value of the CATCH is the value of *expn*.

```
(C1) G(L):=CATCH(MAP(LAMBDA([X],
        IF X<0 THEN THROW(X) ELSE F(X)),L))$

(C2) G([1,2,3,7]);
(D2)                     [F(1), F(2), F(3), F(7)]

(C3) G([1,2,-3,7]);
(D3)                              3
```

The function G returns a list of F of each element of L if L consists only of non-negative numbers; otherwise, G "catches" the first negative element of L and "throws" it up.

THROW*(exp)* evaluates *exp* and throws the value back to the most recent CATCH. THROW is used with CATCH as a structured nonlocal exit mechanism.

BREAK*(arg1, ...)* will evaluate and print its arguments and will then cause a (MACSYMA-BREAK) at which point the user can examine and change his environment. Upon typing EXIT; the computation resumes. (see Chapter 19)

RESET*()* causes all MACSYMA options to be set to their default values.

%TH*(i)* is the *i*th previous computation. That is, if the next expression to be computed is D(j) this is D(j-*i*). This is useful in BATCH files or for referring to a group of D expressions. For example, if SUM is initialized to 0 then FOR I:1 THRU 10 DO SUM:SUM+%TH(I) will set SUM to the sum of the last ten D expressions.

CONCAT*(arg1, arg2, ...)* evaluates its arguments and returns the concatenation of their values resulting in a name or a quoted string (see 2.2 and 2.3) the type being given by that of the first argument. Thus if X is bound to 1 and D is unbound then CONCAT(X,2)="12" and CONCAT(D,X+1)=D2.

GETCHAR*(a, i)* returns the *i*th character of the quoted string or atomic name *a*. This function is useful in manipulating the LABELS list.

STATUS*(arg)* will return miscellaneous status information about the user's MACSYMA depending upon the *arg* given. Permissible arguments and results are as follows:

TIME - the time used so far in the computation.
DAY - the day of the week.
DATE - a list of the year, month, and day.
DAYTIME - a list of the hour, minute, and second.

RUNTIME - accumulated cpu time times the atom "MSEC".
REALTIME -the real time (in sec) elapsed since the user started up his MACSYMA.
WRITEFILE - a list of the device and username for the current writefile or an empty list if no WRITEFILE has been done.
LOADFILE - a list of the first file name, second file name, device, and username for the current BATCH, DEMO, or LOADFILE function.
FILE - a list of the current first file name and second file name.
UNIT - a list of the current device and username.

ALARMCLOCK*(arg1, arg2, arg3)* will execute the function of no arguments whose name is *arg3* when the time specified by *arg1* and *arg2* elapses. If *arg1* is the atom "TIME" then *arg3* will be executed after *arg2* seconds of real-time has elapsed while if *arg1* is the atom "RUNTIME" then *arg3* will be executed after *arg2* milliseconds of cpu time. If *arg2* is negative then the *arg1* timer is shut off.

LABELS*(char)* takes a char C,D,or E as arg and generates a list of all C-labels,D-labels, or E-labels, respectively. (If you've generated many E-labels via SOLVE, then FIRST(REST(LABELS(C))); reminds you what the last C-label was.)

ALIAS*(newname1, oldname1, newname2, oldname2, ...)* provides an alternate name for a (user or system) function,variable,array,etc. Any even number of arguments may be used.

[10.7] Options and Variables for I/O, Status, and Display

GRIND[FALSE] if TRUE will cause the STRING, STRINGOUT, and PLAYBACK commands to use "grind" mode instead of "string" mode. For PLAYBACK, "grind" mode can also be turned on (for processing input lines) by specifying GRIND as an option.

SHOWTIME[FALSE] - if TRUE causes MACSYMA to print the cpu time taken by each computation. This figure does not include I/O time except in the case of the time given at the end of running a batch file. By setting SHOWTIME:ALL, in addition to the cpu time MACSYMA now also prints out (when not zero) the amount of time spent in garbage collection (gc) in the course of a computation. This time is of course included in the time printed out as "time=" .

(It should be noted that since the "time=" time only includes computation time and not any intermediate display time, and since it is difficult to ascribe "responsibility" for gc's, the gctime printed will include all gctime incurred in the course of the computation and hence may in rare cases even be larger than "time=").

LASTTIME - the time to compute the last expression in milliseconds presented as a list of "time" and "gctime" .

OPTIONSET[FALSE] - if TRUE, MACSYMA will print out a message whenever a MACSYMA option is reset. This is useful if the user is doubtful of the spelling of some option and wants to make sure that the variable he assigned a value to was truly an option variable.

NOLABELS[FALSE] - if TRUE then no labels will be bound except for E lines generated by the solve functions (sect. 6.3). This is most useful in the "BATCH" mode where it eliminates the need to do KILL(LABELS) in order to free up storage.

BFTRUNC[TRUE] causes trailing zeroes in non-zero bigfloat numbers not to be displayed. Thus, if BFTRUNC:FALSE, BFLOAT(1); displays as 1.0000000000000000B0. Otherwise, this is displayed as 1.0B0.

EXPTDISPFLAG[TRUE] - if TRUE, MACSYMA displays expressions with negative exponents using quotients e.g., X**(-1) as 1/X.

%EDISPFLAG[FALSE] - if TRUE, MACSYMA displays %E to a negative exponent as a quotient, i.e. %E^-X as 1/%E^X.

SQRTDISPFLAG[TRUE] - if FALSE causes SQRT to display with exponent 1/2.

PFEFORMAT[FALSE] - if TRUE will cause rational numbers to display in a linear form and denominators which are integers to display as rational number multipliers.

DISPFLAG[TRUE] - if set to FALSE within a BLOCK (see 2.12) will inhibit the display of output generated by the solve functions (see 6.3) called from within the BLOCK. Termination of the BLOCK with a dollar sign, $, sets DISPFLAG to FALSE.

LOADPRINT[TRUE] - governs the printing of messages accompanying loading of files. The following options are available: TRUE means always print the message; 'LOADFILE means print only when the LOADFILE command is used; 'AUTOLOAD means print only when a file is automatically loaded in (e.g. the integration file SIN FASL); FALSE means never print the loading message.

NOUNDISP[FALSE] - if TRUE will cause NOUNs to display with a single quote. This switch is always TRUE when displaying function definitions.

POWERDISP[FALSE] - if TRUE will cause sums to be displayed with their terms in the reverse order. Thus polynomials would display as truncated power series, i.e., with the lowest power first.

BOTHCASES[FALSE] - if TRUE will cause MACSYMA to retain lower case text as well as upper case. Note, however, that the names of any MACSYMA special variables or functions must be typed in upper case.

STARDISP[FALSE] - if TRUE will cause multiplication to be displayed explicitly with an * between operands.

DSKGC[FALSE] - if TRUE will cause user defined values, functions, arrays, and line labelled expressions to be automatically stored on disk whenever the system determines that the available in-core space is getting low (see also 15.2).

LABELS - a list of C, D, and E lines which are bound.

INCHAR[C] - the alphabetic prefix of the names of expressions typed by the user.

LINECHAR[E] - the alphabetic prefix of the names of intermediate displayed expressions.

OUTCHAR[D] - the alphabetic prefix of the names of outputted expressions.

LINENUM - the line number of the last expression.

CURSOR[_] is the prompt symbol of the MACSYMA editor, DEMO function, PLAYBACK(SLOW) mode, and (MACSYMA-BREAK). (see chapters 13 and 19).

GENINDEX[I] -the alphabetic prefix of the index of summation for generated sums. (The values of GENINDEX and of the above four variables may be any number of characters though the default is a single character.)

IBASE[10] - the base for inputting numbers.

BASE[10] - the base for display of numbers.

LINEL - the length of the printed line on the terminal. Also used for plotting (see Chapters 17 and 18).

PLOTHEIGHT - the height of the area used for plotting (see Chapters 17 and 18).

VERSION[267] - is the version number of MACSYMA. This could be useful if the user wants to label his output.

INFOLISTS is the list of all the information lists which are in MACSYMA:

```
[LABELS,VALUES,FUNCTIONS,ARRAYS,MYOPTIONS,PROPS,ALIASES,RULES,
                        GRADEFS,DEPENDENCIES]
```

Initially, all these information lists are empty. As the user proceeds with his computation, he may examine these lists when necessary.

[10.8] Functions for Translation and Compilation

MODEDECLARE*(y1, mode1, y2, mode2, ...)* is used to declare the modes of variables and functions for subsequent translation or compilation of functions. Its arguments are pairs consisting of a variable yi, and a mode which is one of BOOLEAN, INTEGER, NUMBER, RATIONAL, FLOAT, POLY (for polynomial), or CRE (for expression in CRE form). Each *yi* may also be a list of variables all of which are declared to have *modei*.

If *yi* is an array, and if every element of the array which is referenced in the function has a value then ARRAY(*yi*, COMPLETE, d1, d2, ...) rather than ARRAY(*yi*, d1, d2, ...) should be used when first declaring the bounds of the array. If all the elements of the array are of mode INTEGER (FLOAT), use INTEGER (FLOAT) instead of COMPLETE. Also if every element of the array is of the same mode, say *m*, then MODEDECLARE(COMPLETEARRAY(*yi*),*m*)) should be used for efficient translation. Also numeric code using arrays can be made to run faster by declaring the expected size of the array, as in:

```
MODEDECLARE(COMPLETEARRAY(A[10,10]),FLOAT)
```

for a floating point number array which is 10 x 10.

Additionally one may declare the mode of the result of a function by using FUNCTION(F1,F2,...) as an argument; here *F1,F2,...* are the names of functions. For example the expression,

```
MODEDECLARE([FUNCTION(F1,F2,...),X],POLY,Q,COMPLETEARRAY(Q),FLOAT)
```

declares that X and the value returned by F1,F2,... are polynomials and that Q is an array of floating point numbers. MODEDECLARE is used either immediately inside of a function definition (see below) or at top-level for global variables.

OPTIMIZE*(exp)* returns an expression that produces the same value and side effects as *exp* but does so more efficiently by avoiding the recomputation of common subexpressions. OPTIMIZE also has the side effect of "collapsing" its argument so that all common subexpressions are shared.

```
(C1) DIFF(%,X,2);

                      2            2               2              2
               2   Y + X        Y + X           Y + X          Y + X
            4 X  %E         2 %E          4 X %E          2 %E
(D1)        ------------ + ---------- - ------------ + ----------
               Y + X         Y + X                2              3
                                          (Y + X)        (Y + X)

(C2) OPTIMIZE(%);
                                                   2          Y + T2
(D2) BLOCK([T1, T2, T3, T4], T1 : Y + X, T2 : X , T3 : %E        ,

                  1                           4 X T3   2 T3
             T4: --,  4 T2 T4 T3 + 2 T4 T3 - ------ + ----)
                 T1                              2       3
                                               T1      T1
```

TRANSLATE*(f1, f2, ...)* translates the user defined functions *f1,f2,...* from the MACSYMA language to LISP (i.e. it makes them EXPRs). This results in a gain in speed when they are called. The functions should include a call to MODEDECLARE at the beginning when possible in order to produce more efficient code. For example:

```
F(X1,X2,...):=BLOCK([v1,v2,...],MODEDECLARE(v1,mode1,v2,mode2,...),...)
```

where the X1,X2,... are the parameters to the function and the v1,v2,... are the local variables. The names of translated functions are added to the PROPS lists (see 8.1). Functions should not be translated unless they are fully debugged.

TRANSLATE(FUNCTIONS) or TRANSLATE(ALL) means translate all functions.

TRANSLATE[FALSE] - If TRUE, causes automatic translation of a user's function to LISP. Note that translated functions may not run identically to the way they did before translation as certain incompatabilities may exist between the LISP and MACSYMA versions. Principally, the RAT function with more than one argument and the RATVARS function (see 6.5) should not be used if any variables are MODEDECLAREd CRE.

SAVEDEF[TRUE] - if TRUE will cause the MACSYMA version of a user function to remain when the function is TRANSLATEd. This permits the definition to be displayed by DISPFUN and allows the function to be edited. If SAVEDEF is FALSE, the names of translated functions are removed from the FUNCTIONS list.

TRANSRUN[TRUE] - if FALSE will cause the interpreted version of all functions to be run (provided they are still around) rather than the translated version.

TRANSBIND[TRUE] - if TRUE removes global declarations in the local context. This applies to variables which are formal parameters to functions.

One can translate functions stored in a file by giving TRANSLATE an argument which is a file specification. This is a list of the form [fn1,fn2,DSK,dir] where fn1 fn2 is the name of the file of MACSYMA functions, and dir is the name of a file directory.

Such a file may contain declarations involving DECLARE,MODEDECLARE, or MATCHDECLARE in addition to the function definitions. The file should not use %, since % is not maintained when the translated file is loaded.

The result returned by TRANSLATE is a list of the names of the functions TRANSLATEd. In the case of a file translation the corresponding element of the list is a list of the first and second new file names containing the LISP code resulting from the translation. This will be fn1 LISP on the disk directory dir. The file of LISP code may be read into MACSYMA by using the LOADFILE function (see 10.4).

COMPFILE*([filespec],f1,f2,...)* will translate (if necessary) and write out MACSYMA function definitions and other expressions into a disk file which can be read into the compiler. The *filespec* (optional) specifies the file to be written. The default for users with a directory is CMPFIL > and the standard MACSYMA default file for other users. The file written contains declarations used by the compiler. When COMPGRIND[FALSE] is TRUE the function definitions are pretty-printed. The remaining arguments are atomic function names.

11 Tensor Manipulation

MACSYMA implements symbolic tensor manipulation of two distinct types: **explicit** tensor manipulation and **indicial** tensor manipulation.

Explicit tensor manipulation means that tensors are represented as arrays or matrices; tensor operations such as contraction or covariant differentiation are carried out by actually summing over repeated (dummy) indices---by explicitly performing operations on the appropriate tensor components stored in an array or matrix.

Indicial tensor manipulation is implemented by representing tensors simply as functions of their covariant and contravariant indices; tensor operations such as contraction or covariant differentiation are represented by manipulating the indices themselves rather than the components which they refer to.

These two approaches to the question of treating differential, algebraic and analytic processes in the context of Riemannian geometry have various advantages and disadvantages which reveal themselves only through the particular nature and difficulty of the user's problem. However, one should keep in mind the following characteristics of the two implementations:

Explicit Tensor Manipulation (ETENSR)

i) The standard representation of tensors and tensor operations explicitly in terms of their components makes ETENSR easy to use: specification of the metric and the computation of the induced tensors and invariants is straightforward.

ii) Although all of MACSYMA's powerful simplification capacity is at hand, a complex metric with intricate functional and coordinate dependencies can easily lead to expressions whose size is excessive and whose structure is hidden.

Indicial Tensor Manipulation (ITENSR)

i) Because of the special way in which tensors and tensor operations are represented in terms of symbolic operations on their indices, expressions which in the *explicit* representation would be unmanageable can be greatly simplified by contraction and reduction to a "canonical" form (for symmetric tensors). In this way the structure of a large expression may be more transparent.

ii) On the other hand, because of the the special indicial representation in ITENSR, in some cases the user must be careful about the specification of the metric, function definition, and the evaluation of differentiated "indexed" objects.

These two tensor manipulation packages, ETENSR and ITENSR, are available to the MACSYMA user on the SHARE directory (see 12). To use the functions in these files, the user can load them in by doing

LOADFILE(ETENSR,FASL,DSK,SHARE); --- for explicit tensor manipulation.

LOADFILE(ITENSR,FASL,DSK,SHARE); --- for indicial tensor manipulation

Both of these packages enable the user to specify a metric and compute the basic quantities of interest---Christoffel symbols, Riemann curvature tensor, curvature invariants---in the study of Riemannian manifolds. These routines were written primarily for research in gravitation theory; however, they may also be of some use in other areas of physics where Riemannian geometry is applied.

11.1 Explicit Tensor Manipulation

TSETUP*()* automatically loads the ETENSR package and presents several options which are self-explanatory. First the user chooses the kind of metric to be used---whether it will be one of the standard metrics already stored in some file, or a power-series approximation, or some new metric to be specified (perhaps only a slight modification of a metric already defined). After the metric has been specified, a number of simplification options are provided which govern the rational simplification and factoring of the tensor components to be computed. The particular quantity to be computed can then be indicated; the user can say whether the results (some of which may be quite lengthy) are to be immediately displayed or not.

Here is a sample protocol:

```
(C2) TSETUP();
DO YOU WANT
1 - TO CONSIDER A METRIC IN THE SPECIAL METRIC FILE?
2 - TO APPROXIMATE A METRIC WITH A POWER SERIES?
3 - TO ENTER A NEW METRIC?
TYPE 1 OR 2 OR 3
3;
SPECIFY THE COORDINATES AS A LIST OF FOUR ELEMENTS
[R,THETA,PHI,T];
DO YOU WANT
1 - TO SPECIFY  A DIAGONAL METRIC?
2 - TO CHANGE A COMPONENT IN A PREVIOUSLY DEFINED
        METRIC?
3 - TO SPECIFY A GENERAL (SYMMETRIC) METRIC?
TYPE 1 2  OR  3
1;

ENTER DIAGONAL MATRIX
[1, 1]
-EXP(M);
[2, 2]
-R^2;
[3, 3]
-R^2*SIN(THETA)^2;
[4, 4]
EXP(N);

INDICATE THE KIND OF SIMPLIFICATION YOU WANT

1 - RATIONAL SIMPLIFICATION ONLY
2 - FACTORING AND RATIONAL SIMPLIFICATION
3 - EXPANSION
TYPE 1 2 OR 3
2;
```

At this point the user has the option of computing various quantities which are described below.

CHRISTOF*(arg)* computes the Christoffel symbols of both kinds; the *arg*, determines which results are to be immediately displayed. The Christoffel symbols of the first kind are stored in the array LCS[I,J,K]. If the argument to CHRISTOF were LCS, all the non-zero values of LCS[I,J,K] would be displayed.

The Christoffel symbols of the second kind (Mixed Christoffel Symbols) are given by the array MCS[I,J,K]; in the example below, the argument MCS was given resulting in the immediate display of all the non-zero mixed Christoffel symbols.

```
(C3) CHRISTOF(MCS);
                                                 M
                                                  R
(E3)                          MCS             = --
                                 1, 1, 1        2
                                                  1
(E4)                           MCS            = -
                                  1, 2, 2         R
                                                  1
(E5)                           MCS            = -
                                  1, 3, 3         R
                                                 N
                                                  R
(E6)                          MCS             = --
                                 1, 4, 4        2
                                                  - M
(E7)                        MCS           = - %E     R
                               2, 2, 1
                                            COS(THETA)
(E8)                       MCS            = ----------
                              2, 3, 3       SIN(THETA)
                                            - M      2
(E9)                    MCS           = - %E     R SIN (THETA)
                           3, 3, 1
(E10)                  MCS           = - COS(THETA) SIN(THETA)
                          3, 3, 2
                                               N - M
                                             %E      N
                                                      R
(E11)                        MCS            = ----------
                                4, 4, 1            2

(D11)                                   DONE
```

MOTION*(dis)* gives the geodesic equations of motion corresponding to a given metric. They are stored in the array EM[I]. If the argument *dis* is TRUE then these equations are displayed.

RICCICOM*(dis)* This function first computes the contravariant components LR[I,J] of the Ricci tensor (LR is a mnemonic for "lower Ricci"). Then the mixed Ricci tensor is computed using the covariant metric tensor. If the value of the argument to RICCICOM is TRUE, then these mixed components, RICCI[I,J] (the index I is covariant (down) and the index J is contravariant (up)), will be displayed directly. Otherwise, RICCICOM(FALSE) will simply compute the entries of the array RICCI[I,J] without presenting the results.

```
(C13) RICCICOM(TRUE);

(E13) RICCI
           1, 1

           - M              2
        %E     ((2 N     + (N )  - M  N ) R - 4 M )
                    R R     R       R  R         R
      = ---------------------------------------------
                             4 R

                          - M                    M
                       %E     ((M  - N ) R + 2 %E  - 2)
                                 R    R
(E14)  RICCI       = - ------------------------------
           2, 2                         2
                                     2 R


                          - M                    M
                       %E     ((M  - N ) R + 2 %E  - 2)
                                 R    R
(E15)  RICCI       = - ------------------------------
           3, 3                         2
                                     2 R
```

```
(E16)  RICCI
            4, 4

                  - M              2
                %E    ((2 N   + (N )  - M  N ) R + 4 N )
                           R R    R      R  R          R
              = -----------------------------------------
                                  4 R

(D16)                             DONE
```

NTERMSRCI*()* returns a list of pairs, whose second elements give the number of terms in the RICCI component specified by the first elements. In this way, it is possible to quickly find the non-zero expressions and attempt simplification.

LRICCICOM*(dis)* computes the covariant components LR[I,J] of the Ricci tensor. If the argument *dis* is TRUE, then the non-zero components are displayed.

EINSTEIN*(dis)* computes the Einstein tensor once the Christoffel symbols and Ricci tensor have been obtained. Again, if the argument evaluates to TRUE, then the non-zero values of the Einstein tensor G[I,J] will be displayed. RATEINSTEIN:TRUE will perform rational simplification on these components; if FACRAT:TRUE then the components will also be factored.

NTERMSG*()* gives the user a quick picture of the "size" of the Einstein tensor. It returns a list of pairs whose second elements give the number of terms in the components specified by the first elements.

Of course a detailed examination of the structure of the Einstein components can be made using the powerful simplification, factoring and extraction functions available in MACSYMA (see 6.1.1,6.2.3).

SCURVATURE*()* returns the scalar curvature (obtained by contracting the Ricci tensor) of the Riemannian manifold with the given metric.

RIEMANN*(dis)* computes the Riemann curvature tensor from the given metric (the Christoffel symbols should be obtained first using CHRISTOF). If *dis* is TRUE, the non-zero components R[I,J,K,L] will be displayed. *All the indicated indices are covariant.* As with the Einstein tensor, various switches set by the user control the simplification of the Riemann components. If RATRIEMAN:TRUE, then rational simplification will be done; if FACRAT:TRUE then each of the components will also be factored.

```
(C27) RIEMANN(TRUE);
                                                M  R
                                                 R
(E27)                       R            = - ----
                             1, 2, 1, 2        2

                                                  2
                                          M  R SIN (THETA)
                                           R
(E28)                  R            = - ----------------
                        1, 3, 1, 3              2

                       N          N   2       N
                 - 2 %E  N    - %E  (N )  + M %E  N
                          R R         R     R      R
(E29) R          = ------------------------------------
       1, 4, 1, 4                         4

                      M        - M  2   2
(E30) R          = - (%E  - 1) %E    R  SIN (THETA)
       2, 3, 2, 3

                                           N - M
                                         %E      N  R
                                                  R
(E31)                  R            = - ------------
                        2, 4, 2, 4             2
```

```
                                 N - M          2
                               %E       N   R SIN (THETA)
                                         R
(E32)    R               = - -----------------------
          3, 4, 3, 4                      2

(D32)                                   DONE
```

RAISERIEMANN*(dis)* returns the *contravariant* components of the Riemann curvature tensor as array elements UR[I,J,K,L]. These are displayed if *dis* is TRUE.

RINVARIANT*()* forms the scalar invariant obtained by contracting R[I,J,K,L]*UR[I,J,K,L].

```
(C34) RINVARIANT();
          -2M            2         2                    4
(D34) %E     ((4 (N    )  + (4 (N )  - 4 M  N ) N   + (N )
                  R R          R         R  R   R R    R

              3       2      2  4          2        2   2
 - 2 M  (N )  + (M )  (N ) ) R  + (8 (N )  + 8 (M ) ) R
      R   R        R     R              R         R

         2 M        M        4
 + 16 %E    - 32 %E  + 16)/(4 R )
```

WEYL*(dis)* computes the Weyl conformal tensor. If the argument *dis* is TRUE, the non-zero components W[I,J,K,L] will be displayed to the user. Otherwise, these components will simply be computed and stored. If the switch RATWEYL is set to TRUE, then the components will be rationally simplified; if FACRAT is TRUE then the results will be factored as well.

DSCALAR*(function)* computes the d'Alembertian of the scalar *function.*

```
(C41) DEPENDS(FIELD,R);
(D41)                          [FIELD(R)]

(C42) DSCALAR(FIELD);

(D43)
    -M
  %E   ((FIELD  N - FIELD  M + 2 FIELD   ) R + 4 FIELD )
              R  R        R  R          R R              R
- ----------------------------------------------------
                                 2 R
```

DALEM*(field,i,j)* computes the d'Alembertian of the i,j - component of the rank 2 tensor *field.*

YT*(f,m,n)* computes the m,n component of the Yilmaz tensor defined by

$$t_m{}^n = -2(\partial_m f_a{}^b \partial^n f_b{}^a - (1/2)\delta_\mu{}^\nu \partial^k f_a{}^b \partial_k f_b{}^a)$$

$$+ \partial_m \underline{f} \partial^n \underline{f} - (1/2)\delta_\mu{}^\nu \partial^k \underline{f} \partial_k \underline{f},$$

where $\underline{f}$ is the trace of $f_a{}^b$ and repeated indices are summed. This reduces to the Newtonian stress-energy tensor.

[11.2] Indicial Tensor Manipulation

In ITENSR a tensor is represented as an "indexed object" . This is a function of 3 groups of indices which represent the covariant, contravariant and derivative indices. The covariant indices are specified by a list as the first argument to the indexed object; the contravariant indices are specified by a list as the second argument. If the indexed object lacks either of these groups of indices, then the empty list, [], is given as the corresponding argument. For example,

G([mu,nu],[])

represents an indexed object called G which has covariant indices mu,nu and no contravariant or derivative indices.

The derivative indices, if they are present, follow as additional arguments to the (symbolic) function representing the tensor. They are usually not explicitly specified by the user but are created in the process of differentiation with respect to some coordinate variable. A derivative index is not the coordinate variable itself, but rather the coordinate *index*. These indices are appended as additional arguments to the function representing the tensor. Since it is assumed that ordinary differentiation is independent of the order in which it is carried out, the derivative indices are sorted alphabetically. This canonical order makes it possible for MACSYMA to recognize that, for example, T([mu],[nu],i,j) is the same as T([mu],[nu],j,i). Differentiation of an indexed object with respect to some coordinate whose index does not appear as an argument to the indexed object would normally yield zero since MACSYMA would not know that the tensor represented by the indexed object might depend implicitly on the corresponding coordinate. This has been remedied by modifying the existing MACSYMA function DIFF so that in the tensor package it assumes that all indexed objects depend on any variable of differentiation unless otherwise stated. This makes it possible for the summation convention to be extended to derivative indices.

To specify that an indexed object is independent of all coordinate variables, it is specified a constant by using the DECLARE function (see 8.1). Usually, DIFF(W([],[I,J]),K) results in W([],[I,J],K); if the command DECLARE(W,CONSTANT) had previously been given, the result of the differentiation would be 0.

The following functions are available in the tensor package for manipulating indexed objects. At present it is assumed that all tensor indices are completely symmetric.

In what follows, general indexed objects will be denoted *tensor, tensor1,tensor2,...* . The letters *L1,L2,...* denote lists which are arguments to indexed objects. Optional arguments are enclosed in angle brackets.

SHOWTEN*(exp)* will display *exp* with the indexed objects in it shown having covariant indices as subscripts,contravariant indices as superscripts. The derivative indices will be displayed as subscripts, separated from the covariant indices by a comma.

COMPONENTS*(tensor,exp)* permits one to assign an expression *exp* giving the values of the components of *tensor*. These are automatically substituted for the *tensor* whenever it occurs with all integer indices.
The *tensor* must be of the form T([...],[...]), which can specify a covariant (second list empty), contravariant (first list empty), or mixed (neither list empty) tensor. *exp* can be a matrix, an array or any expression (except a single variable) involving other tensors.

If *exp* is a matrix, then *tensor* must have exactly two indices. If it is an array, then *exp* and *tensor* must have the same number of indices. If *exp* is some other expression involving tensors, then it must have the same free indices as *tensor*. This expression will be used even if the indices are not integers.

```
(C1) COMPONENTS(G([I,J],[ ]),MATRIX)$

(C2) COMPONENTS(G([ ],[J,K]),ARRAY)$

(C3) COMPONENTS(E([I],[J]),G([I,K])*H([ ],[K,J]))$
```

Thus, the various covariant, contravariant, and mixed components of the <u>same</u> tensor (for example, G above) can be specified using COMPONENTS several times. The appropriate components will be chosen when required in a computation.

INDEXED*(tensor)* must be executed before assigning components to a *tensor* for which a built in value already exists as with CHR1, CHR2, RIEMANN.

REMCOMPS*(tensor)* unbinds all values from *tensor* which were assigned with the COMPONENTS function.

INDICES*(exp)* returns a list of two elements. The first is a list of the free indices in *exp* (those that occur only once); the second is the list of dummy indices in *exp* (those that occur exactly twice).

RENAME*(exp, <count>)* returns an expression equivalent to *exp* but with the dummy indices in each term chosen from the set [#1, #2,...], if the optional second argument is omitted. Otherwise, the dummy indices are indexed beginning at the value of *count*. Each dummy index in a product will be different; for a sum, RENAME will try to make each dummy index in a sum the same. In addition, the indices will be sorted alphanumerically.

DUMMY*(i1,i2,...)* will set each index i1,i2,... to name of the form #n where n is a positive integer. This guarantees that dummy indices which are needed in forming expressions will not conflict with indices already in use.
COUNTER[1] determines the numerical suffix to be used in generating the next dummy index. The prefix is determined by the option DUMMYX[#].

DEFCON*(tensor1,<tensor2,tensor3>)* gives *tensor1* the property that the contraction of a product of *tensor1* and *tensor2* results in *tensor3* with the appropriate indices. If only one argument, *tensor1*, is given, then the contraction of the product of *tensor1* with any indexed object having the appropriate indices (say *tensor*) will yield an indexed object with that name,i.e.*tensor*, and with a new set of indices reflecting the contractions performed.

For example, if METRIC: G, then DEFCON(G) will implement the raising and lowering of indices through contraction with the metric tensor.
More than one DEFCON can be given for the same indexed object; the latest one given which applies in a particular contraction will be used.
CONTRACTIONS is a list of those indexed objects which have been given contraction properties with DEFCON.

DISPCON*(tensor1,tensor2,...)* displays the contraction properties of the *tensori* as were given to DEFCON. DISPCON(ALL) displays all the contraction properties which were defined.

REMCON*(tensor1,tensor2,...)* removes all the contraction properties from the *tensori*. REMCON(ALL) removes all contraction properties from all indexed objects.

CONTRACT*(exp)* carries out all possible contractions in *exp*, which may be any well-formed combination of sums and products. This function uses the information given to the DEFCON function. Since all tensors are considered to be symmetric in all indices, the indices are sorted into alphabetical order. Also all dummy indices are renamed using the symbols #1,#2,... to permit the expression to be simplified as much as possible by reducing equivalent terms to a canonical form. For best results *exp* should be fully expanded.
RATEXPAND (see 6.1.1) is the fastest way to expand products and powers of sums if there are no variables in the denominators of the terms. The TAKEGCD switch should be FALSE if gcd cancellations are unnecessary.

CANTEN*(exp)* reduces *exp* to a canonical form and simplifies the expression as much as possible by renaming and permuting dummy indices. The expression *exp* must be fully expanded.
As an example of the kind of simplification this function achieves, consider the following sum of tensor "monomials":

```
(C4) P([I,J,S,V],[M,N,Q],V)*P1([Q,T],[R,S])*P2([R,L,M,N,U],[I,J,K]) +

     P2([L,N,U],[R,M,I,J,K])*P([M,I,J,V],[N,Q,S],V)*P1([R,S,Q,T],[])$

(C6) SHOWTEN(D4);

        R M I J K  N Q S                M N Q      R S     I J K
(E6) P2            P          P1      + P          P1      P2
        L N U      M I J V,V  R S Q T   I J S V,V  Q T     R L M N U

(D6)                                   E6
```

Recall that all these indexed objects are assumed to be completely symmetric in their indices. The function RENAME renames the dummy indices:

```
(C7) SHOWTEN(RENAME(D4))$


         #6 #7 #8           #2 #3 K         #4 #5
(E7) P                   P2              P1
         #1 #2 #3 #5,#1     #4 #6 #7 L U    #8 T


                      #3 #4 #8                            #2 #5 #6 #7 K
                  + P                     P1          P2
                      #1 #5 #6 #7,#1    #2 #3 #4 T    #8 L U
```

but is not able to notice that by a sequence of raising and lowering operations, the original expression can be transformed into

```
(C8) SHOWTEN(CANTEN(D4))$
                               K                 I J M N   Q R S
(E8)                       2 P2                P          P1
                               I J L M N R U   Q S V,V    T
```

METRIC*(G)* specifies the metric by assigning the variable METRIC:*G*; in addition, the contraction properties of the metric *G* are set up by executing the commands DEFCON(*G*), DEFCON(*G*,*G*,KDELTA).

CHR1*([i,j,k])* yields the Christoffel symbol of the first kind

$$(g_{ik,j} + g_{jk,i} - g_{ij,k})/2$$

The variable METRIC must be assigned the name of a function (which can be either defined or undefined); in the above example, METRIC:g.

CHR2*([i,j],[k])* yields the Christoffel symbol of the second kind.

$$\mathrm{CHR2}([i,j],[k]) = g^{ks}\ \mathrm{CHR1}([i,j,s])$$

LC*(L)* is the permutation (or Levi-Civita) tensor which yields 1 if the list *L* consists of an even permutation of integers, -1 if it consists of an odd permutation, and 0 if some indices in *L* are repeated.

KDELTA*(L1,L2)* is the generalized Kronecker delta function. *L1* and *L2* are lists of indices of the same length.

a) If *L1* and *L2* have a single member, say L1=[a] and L2=[b]. Then

1) if the index a is identical to the index b and they are non-numeric, the value of the function KDELTA is the value of the variable DIMENSION[4] ;

2) if a and b are numeric then the value of the function is 1 if they are equal, else 0;

3) otherwise the noun form of KDELTA.

b) If L1=[a1,a2] and L2=[b1,b2]. Then the value of the function is

```
KDELTA([a1],[b1])*KDELTA([a2],[b2])
                  - KDELTA([a1],[b2])*KDELTA([a2],[b1])
```

c) If L1 and L2 have more than two indices the result generalizes.

RIEMANN*([i,j,k],[l])* yields the Riemann curvature tensor in terms of the Christoffel symbols of the second kind (CHR2).

Suppose the name specified by the value of METRIC corresponds to a tensor which has been given some structure via the COMPONENTS command above; in order to evaluate an *expression* involving the Riemann tensor and incorporate this given structure of the metric explicitly into the result, the user can simply do *expression*,EVAL.

Consider the following example involving a metric, G, expressed in terms of the rank two tensors, P and E. First, the covariant and contravariant forms of the metric are specified:

```
(C9) COMPONENTS(G([M,N],[]),

E([M,N],[]) + L*(2*P([],[])*E([M,N],[])-4*P([M,N],[]))
          L^2*(2*P([],[])^2*E([M,N],[])
              - 8*P([],[])*P([M,N],[])+8*P([M],[Q])*P([Q,N],[])))$

(C10) COMPONENTS(G([],[M,N]),

E([],[M,N]) - L*(2*P([],[])*E([],[M,N])-4*P([],[M,N]))
         + L^2*(2*P([],[])^2*E([],[M,N])
              - 8*P([],[])*P([],[M,N])+8*P([],[M,Q])*P([Q],[N])))$

(C11) SHOWTEN(G([A,B],[]))$

        2     Q                   2
(E11) L  (8 P  P    - 8 P P    + 2 P E    ) + L (2 P E    - 4 P    )
            A  Q B      A B          A B              A B       A B


      + E
         A B
(C12) SHOWTEN(G([],[R,S]));

        2     R Q  S      R S        2  R S          R S     R S
(E12) L  (8 P     P   - 8 P P    + 2 P E    ) - (2 P E    - 4 P    ) L
                  Q


          R S
        + E
(C13) (RATVARS(L),RATWEIGHT(L,1),RATWTLVL:2)$
```

The above metric is an expansion to second order of the Yilmaz exponential metric

$$G = E.e^{2*L*(I.P - 2*\underline{P})}$$

where L is an expansion parameter, $\underline{P}$ is a rank two tensor field viewed as a matrix whose components are $P^{R,S}$ and whose trace is P.

To second order, the covariant and contravariant forms of the metric should be mutually inverse. As an example of the contraction and simplification functions CONTRACT,CANTEN one can verify this as follows:

```
(C14) SHOWTEN(EX:CONTRACT(RATEXPAND(G([M,R],[ ])*G([ ],[R,N]))))$

                      N        N  2  #1        #1 N        2
(E14)           KDELTA  + 16 P   L  P   - 16 P    P       L
                      M        #1    M              #1 M
```

Because RATWTLVL:2, the product of the expansions is truncated; the second order terms are actually equal and should cancel. Using a canonical form, the function CANTEN makes the appropriate simplification which enables the last two terms to be compared and cancelled:

```
(C16) SHOWTEN(CANTEN(EX));

                                            N
(E16)                                 KDELTA
                                            M
```

The Ricci tensor is easily expressed in terms of the Riemann tensor, using the Einstein summation convention:

```
(C18) RICCI:RIEMANN([I,J,K],[K])$

(C19) %,EVAL$

(C20) EXP1:CONTRACT(RATEXPAND(RICCI))$

(C21) SHOWTEN(FACTOR(EXP1))$

              #1          #1 #2
(E21) - (2 P       - 2 E        P
            J,#1 I               I J,#1 #2

                                    #1 #2           #1
                            + P          E     E  + 2 P         ) L
                               ,#1 #2           I J      I,#1 J
```

DIFF*(exp,v1,n1,v2,n2,...)* is the usual MACSYMA differentiation function; it takes the derivative of *exp* wrt *v1 n1* times, wrt *v2 n2* times, etc. For the tensor package, the following modifications have been incorporated:

1) the derivatives of any indexed objects in *exp* will have the variables *vi* appended as additional arguments. Then all the derivative indices will be sorted.

2) the *vi* may be integers from 1 up to the value of the variable DIMENSION[4]. This will cause the differentiation to be carried out wrt the *vi*th member of the list COORDINATES which should be set to a list of the names of the coordinates, e.g., [x,y,z,t] . If COORDINATES is bound to an atomic variable, then that variable subscripted by *vi* will be used for the variable of differentiation. This permits an array of coordinate names or subscripted names like X[1], X[2],... to be used. If COORDINATES has not been assigned a value, then the variables will be treated as in 1) above.

COVDIFF*(exp,v1,v2,...)* yields the covariant derivative of *exp* with respect to the variables *vi* in terms of the Christoffel symbols of the second kind (CHR2). In order to evaluate these, one can use *exp*, eval or EV(*exp*,CHR2).

UNDIFF*(exp)* returns an expression equivalent to *exp* but with all derivatives of indexed objects replaced by the noun form of the DIFF function with arguments which would yield that indexed object if the differentiation were carried out. This is useful when it is desired to replace a differentiated indexed object with some function definition and then carry out the differentiation by saying EV(...,DIFF).

LORENTZ*(exp)* yields *exp* replacing by zero those indexed objects which have a derivative index identical to a contravariant index. This imposes the Lorentz condition.

MAKEBOX*(exp)* will display *exp* in the same manner as SHOWTEN; however, any tensor d'Alembertian occurring in *exp* will be indicated using the symbol []. For example, []P([M],[N]) represents G([],[I,J])*P([M],[N],I,J).

ZERO(*tensor*) will give *tensor* a property such that the command FLUSH(*expression*) will, in *expression*, replace by zero all occurences of the *tensor* involving derivative indices.

12 The SHARE Directory

The SHARE directory contains programs, information files, etc. which are considered to be of interest to the MACSYMA community. Most files on SHARE; are not part of the MACSYMA system per se and must be loaded individually by the user, e.g. LOADFILE(ITENSR,FASL,DSK,SHARE);. Many files on SHARE; were contributed by MACSYMA users, and all MACSYMA users are encouraged to do so.

Names for files on SHARE; should be chosen as appropriate. However, the contributor will probably want to follow the conventions discussed here. A contributor will probably create some but not all of the following files. Examples may be seen on the SHARE directory.

1) NAME > is the file name of the MACSYMA BATCHable programs. The > sign indicates a numeric second filename which is increased whenever a new version is created.

2) NAME LISP is the file name of the LISP code for the programs contained in the file NAME >. This file is loaded into MACSYMA using LOADFILE. It was obtained by using the TRANSLATE command or was written directly in LISP by the contributor.

3) NAME FASL is the file name of the FASL (fast-loadable) version of NAME LISP, and was produced from NAME LISP by using the LISP compiler. It is loaded into MACSYMA using the LOADFILE command.

4) NAME USAGE is the name of the documentation file for the programs in NAME >. It describes how the programs are used, inputs, outputs, options, warnings, error messages, etc. It may mention the algorithms behind the programs, references, and whatever else the user should know. It should certainly indicate who programmed the routines, especially his login name. If the NAME USAGE file does not exist, this information should be given in NAME > or elsewhere.

5) NAME DEMO is the name of the demonstration file which may be used in DEMOing NAME > or NAME LISP.

6) NAME OUTPUT may be used to store sample output obtained from running NAME > on some examples or from DEMOing the NAME DEMO file.

7) Other file names may be used for information files providing some

information on some aspect of the MACSYMA system or for describing some MACSYMA utility or for notes on some MACSYMA issues, etc.

The SHARE > file is an index to the SHARE directory and is intended to contain a short note on each of the programs on the SHARE directory. It should be updated by the SHARE; contributor as appropriate.

Any comments or questions about the use of the SHARE directory should be sent to JPG.

12.1 Simplification for ABS and SIGNUM

The file SHARE;ABSIMP > contains MACSYMA pattern-matching rules that extend the built-in simplification rules for the ABS and SIGNUM functions. Among other things, use is made of global relations established with the built-in ASSUME function or by declarations such as DECLARE(M,EVEN, N,ODD) for even or odd integers. UNITRAMP and UNITSTEP functions are also defined in terms of ABS and SIGNUM. These routines were written by David Stoutemyer.

[12.2] Array Manipulation

The file ARRAY FASL provides various utility functions for handling arrays.

LISTARRAY*(array)* returns a list of the elements of a declared array. the order is row-major. You will get garbage if any of the elements have not been defined yet.

FILLARRAY*(array,list-or-array)* fills *array* from *list-or-array*. If *array* is a floating-point (integer) array then *list-or-array* should be either a list of floating-point (integer) numbers or another floating-point (integer) array. If the dimensions of the arrays are different *array* is filled in row-major order. If there are not enough elements in *list-or-array* the last element is used to fill out the rest of *array*. If there are too many the remaining ones are thrown away. FILLARRAY returns its first argument.

REARRAY*(array,dim1, ... ,dimk)* can be used to change the size or dimensions of an array. The new array will be filled with the elements of the old one in row-major order. If the old array was too small, FALSE, 0.0 or 0 will be used to fill the remaining elements, depending on the type of the array. The type of the array cannot be changed.

ARRAYAPPLY*(array,[sub1, ... ,subk])* is like APPLY except the first argument is an array.

These routines were written by Charles Karney.

[12.3] Solving Differential Equations by Laplace Transforms

DESOLN LISP contains a routine, written by Richard Bogen, for solving differential equations or systems of them by using Laplace transforms. The call is:

DESOLVE*([eq1,...,eqn],[var1,...,varn])* where the *eq*'s are differential equations in the dependent variables var1,...,varn. The functional relationships must be explicitly indicated in both the equations and the variables. For example

```
(C1) 'DIFF(F,X,2)=SIN(X)+'DIFF(G,X);
(C2) 'DIFF(F,X)+X^2-F=2*'DIFF(G,X,2);
```

is not the proper format. The correct way is:

```
(C3) 'DIFF(F(X),X,2)=SIN(X)+'DIFF(G(X),X);
(C4) 'DIFF(F(X),X)+X^2-F(X)=2*'DIFF(G(X),X,2);
```

The quotes are not necessary since DIFF will return the noun forms anyway.

The call is then DESOLVE([D3,D4],[F(X),G(X)]);

If initial conditions at 0 are known, they should be supplied before calling DESOLVE by using ATVALUE.

```
(C11) 'DIFF(F(X),X)='DIFF(G(X),X)+SIN(X);
```

```
                                 D        D
(D11)                            -- F(X) = -- G(X) + SIN(X)
                                 DX       DX
(C12) 'DIFF(G(X),X,2)='DIFF(F(X),X)-COS(X);
                                  2
                                 D         D
(D12)                            --- G(X) = -- F(X) - COS(X)
                                   2       DX
                                 DX
(C13) ATVALUE('DIFF(G(X),X),X=0,A);
(D13)                                       A

(C14) ATVALUE(F(X),X=0,1);
(D14)                                       1

(C15) DESOLVE([D11,D12],[F(X),G(X)]);

                     X                             X
(D16) [F(X)=A %E  - A+1, G(X) = COS(X) + A %E  - A + G(0) - 1]

/* VERIFICATION */
(C17) [D11,D12],D16,DIFF;
                       X       X      X                X
(D17)          [A %E  = A %E , A %E  - COS(X) = A %E  - COS(X)]
```

12.4 Exterior Calculus of Differential Forms

The exterior calculus of differential forms is a basic tool of differential geometry developed by Elie Cartan and has important applications in the theory of partial differential equations. The present implementation is due to F.B. Estabrook and H.D. Wahlquist. The program is self-explanatory and can be accessed by doing

BATCH([CARTAN,START,DSK,SHARE],ON)

which will give a description with examples.

The next six sections describe programs written by David Stoutemyer.

12.5 Vector Analysis

The file VECT > contains a vector analysis package, VECT DEMO contains a corresponding demonstration, and VECT ORTH contains definitions of various orthogonal curvilinear coordinate systems.

The vector analysis package can combine and simplify symbolic expressions including dot products and cross products, together with the gradient, divergence, curl, and Laplacian operators. The distribution of these operators over sums or products is under user control, as are various other expansions, including expansion into components in any specific orthogonal coordinate systems. There is also a capability for deriving the scalar or vector potential of a field.

To establish *indeterminate1, indeterminate2,* ... as vector entities, type

DECLARE([*indeterminate1, indeterminate2,* ...], NONSCALAR) $

Vectors can also be represented as lists of components.

"." is the dot-product operator, "~" is the cross-product operator, GRAD is the gradient operator, DIV is the divergence operator, CURL is the curl or rotation operator, and LAPLACIAN is DIV GRAD.

Most non-controversial simplifications are automatic. For additional simplification, there is a function which can be used in the form

VECTORSIMP*(vectorexpression)*

This function employs additional non-controversial simplifications, together with various optional expansions according to the settings of the following global flags:

EXPANDALL, EXPANDDOT, EXPANDDOTPLUS

EXPANDCROSS, EXPANDCROSSPLUS, EXPANDCROSSCROSS

EXPANDGRAD, EXPANDGRADPLUS, EXPANDGRADPROD

EXPANDDIV, EXPANDDIVPLUS, EXPANDDIVPROD

EXPANDCURL, EXPANDCURLPLUS, EXPANDCURLCURL

EXPANDLAPLACIAN, EXPANDLAPLACIANPLUS, EXPANDLAPLACIANPROD

All these flags have default value FALSE. The PLUS suffix refers to employing additivity or distributivity. The PROD suffix refers to the expansion for an operand that is any kind of product. EXPANDCROSSCROSS refers to replacing p~(q~r) with (p.r)*q-(p.q)*r, and EXPANDCURLCURL refers to replacing CURL CURL p with GRAD DIV p + DIV GRAD p. EXPANDCROSS:TRUE has the same effect as EXPANDCROSSPLUS:EXPANDCROSSCROSS:TRUE, etc. Two other flags, EXPANDPLUS and EXPANDPROD, have the same effect as setting all similarly suffixed flags true. When TRUE, another flag named EXPANDLAPLACIANTODIVGRAD, replaces the LAPLACIAN operator with the composition DIV GRAD. All of these flags are initially FALSE. For convenience, all of these flags have been declared EVFLAG.

For orthogonal curvilinear coordinates, the global variables COORDINATES[[X,Y,Z]], DIMENSION[3], SF[[1,1,1]], and SFPROD[1] are set by the function invocation

SCALEFACTORS*(coordinatetransform)*

Here *coordinatetransform* evaluates to the form [[expression1, expression2, ...], indeterminate1, indeterminat2, ...], where indeterminate1, indeterminate2, etc. are the curvilinear coordinate variables and where a set of rectangular Cartesian components is given in terms of the curvilinear coordinates by [expression1, expression2, ...]. COORDINATES is set to the vector [indeterminate1, indeterminate2, ...], and DIMENSION is set to the length of this vector. SF[1], SF[2], ..., SF[DIMENSION] are set to the coordinate scale factors, and SFPROD is set to the product of these scale factors. Initially, COORDINATES is [X, Y, Z], DIMENSION is 3, and SF[1]=SF[2]=SF[3]=SFPROD=1, corresponding to 3-dimensional rectangular Cartesian coordinates.

To expand an expression into physical components in the current coordinate system, there is a function with usage of the form

EXPRESS*(expression)*

The result uses the noun form of any derivatives arising from expansion of the vector differential operators. To force evaluation of these derivatives, the built-in EV function can be used together with the DIFF evflag, after using the built-in DEPENDS function to establish any new implicit dependencies.

The scalar potential of a given gradient vector, in the current coordinate system, is returned as the result of POTENTIAL*(givengradient)*

The calculation makes use of the global variable POTENTIALZEROLOC[0], which must be NONLIST or of the form [indeterminatej=expressionj, indeterminatek=expressionk, ...], the former being equivalent to the nonlist expression for all right-hand sides in the latter. The indicated right-hand sides are used as the lower limit of integration. The success of the integrations may depend upon their values and order. POTENTIALZEROLOC is initially set to 0.

The vector potential of a given curl vector, in the current coordinate system, is returned as the result of

VECTORPOTENTIAL*(givencurl)*

POTENTIALZEROLOC has a similar role as for POTENTIAL, but the order of the left-hand sides of the equations must be a cyclic permutation of the coordinate variables.

EXPRESS, POTENTIAL, and VECTORPOTENTIAL can have a second argument like the argument of SCALEFACTORS, causing a corresponding invocation of SCALEFACTORS before the other computations.

12.6 Dimensional Analysis

The file DIMEN > contains functions for automatic dimensional analysis, and file DIMEN DEMO contains a demonstration. Usage is of the form

NONDIMENSIONALIZE*(list of physical quantities)*

The returned value is a sufficient list of nondimensional products of powers of the physical quantities. A physical relation between only the given physical quantities must be expressible as a relation between the nondimensional quantities. There are usually fewer nondimensional than physical quantities, which reduces the number of experiments or numerical computations necessary to establish the physical relation to a specified resolution, in comparison with the number if all but one dependent physical variable were independently varied. Also, the absence of any given physical quantity in the output reveals that either the quantity is irrelevant or others are necessary to describe the relation.

The program already knows an extensive number of relations between physical quantities, such as VELOCITY=LENGTH/TIME. The user may over-ride or supplement the prespecified relations by typing

DIMENSION*(equation or list of equations)*

where each equation is of the form indeterminate=expression, where expression is a product or quotient of powers of none or more of the indeterminates CHARGE, TEMPERATURE, LENGTH, TIME, or MASS. To see if a relation is already established type

GET(indeterminate, 'DIMENSION);

The result of NONDIMENSIONALIZE usually depends upon the value of the global variable %PURE, which is set to a list of none or more of the indeterminates ELECTRICPERMITTIVITYOFAVACUUM, BOLTZMANNSCONSTANT, SPEEDOFLIGHT, PLANCKSCONSTANT, GRAVITYCONSTANT, corresponding to the relation between charge and force, temperature and energy, length and time, length and momentum, and the inverse-square law of gravitation respectively. Each included relation is used to eliminate one of CHARGE, TEMPERATURE, LENGTH, TIME, or MASS from the dimensional basis. To avoid omission of a possibly relevant nondimensional grouping, either include the relevant constant in %PURE or in the argument of NONDIMENSIONALIZE if the corresponding physical effect is thought to be relevant to the problem. However, the inclusion of unnecessary constants, especially the latter three, tends to produce irrelevant or misleading dimensionless groupings, defeating the purpose of dimensional analysis. As an extreme example, if all five constants are included in %PURE, all physical quantities are already dimensionless. %PURE is initially set to '[ELECTRICPERMITTIVITYOFVACUUM, BOLTZMANNSCONSTANT], which is best for most engineering work. %PURE must not include any of the other 3 constants without also including these 2.

[12.7] Analytic Optimization

We now describe a package for finding the stationary points of a multivariate objective function, either unconstrained or subject to equality and/or inequality constraints.

RELEVANT FILES: OPTMIZ > is a MACSYMA batch file containing the functions and option settings for optimization. OPTMIZ DEMO is a MACSYMA batch file demonstrating various ways of using the optimization functions. OPTMIZ OUTPUT is a text file listing OPTMIZ DEMO together with the output that it produces when executed.

To use this package from a MACSYMA, first type BATCH(OPTMIZ, ">", DSK, SHARE) Then the following command is available:

STAP*(OBJECTIVE, LEZEROS, EQZEROS, DECISIONVARS)*

OBJECTIVE is an expression denoting the objective function or the label of such an expression. LEZEROS is a list of expressions which are constrained to be less than or equal to zero. Use [] if no such constraints. EQZEROS is a list of expressions which are constrained to equal zero, or the label of such a list. Use [] if there are no such constraints. DECISIONVARS is a list of the decision variables or the label of such a list. One may use [] if all variables in objective and constraintsare decision variables. For convenience, brackets may be omitted from one-expression lists, and trailing [] arguments may be omitted.

ROOTSEPSILON may affect the accuracy of results computed by SOLVE and REALROOTS within STAP. The default value of 1.0E-7 for this MACSYMA global variable is as small as practical for pdp single-precision floating-point arithmetic. Larger values save cpu time.

The class of functions that may be used and the practical limitations on the number of decision variables and constraints is primarily dependent upon the capabilities of the built-in SOLVE function, which is still under development.

[12.8] Variational Optimization

This section describes how to use a MACSYMA variational optimization package to analytically solve problems from the calculus of variations and the maximum principle, including optimal control.

To use this package in a MACSYMA, first type BATCH(OPTVAR,">", DSK, SHARE) or LOADFILE(OPTVAR, LISP, DSK, SHARE).

To derive the Euler-Lagrange equations for a calculus-of-variations problem, type

EL*(F, YLIST, TLIST)*

F is an expression or the label of an expression for the integrand of the stationary functional, augmented by Lagrange multipliers times the integrands of any isoperimetric constraints and/or differential expressions constrained to equal zero. The multipliers should be written as functions of the independent variables in the latter case.

YLIST is a list of the dependent variables, or the label thereof.
TLIST is a list of the independent variables, or the label thereof.

For convenience, square brackets may be omitted from 1-element lists. EL displays one or more E-labeled equations, then returns a list of the E-labels. These equations are the Euler-Lagrange equations, perhaps together with first integrals corresponding to conservation of energy and/or conservation of momentum. The former will contain a constant of integration K[0], whereas the latter will contain constants of integration K[I], with positive I. The latter will immediately follow the corresponding Euler-Lagrange equation.

OPTVAR DEMO or OPTVAR OUTPUT illustrates some ways that the resulting differential equations may be solved analytically.

To derive the Hamiltonian and auxiliary differential equations for an optimal control problem, type

HAM*(ODES)*

ODES is a list of the first-order differential equations that govern the state variables. Each differential equation must be of the form

```
'D(Y,T) = EXPRESSION
```

where Y is one of the dependent variables, T is the independent variable, and EXPRESSION depends upon the independent, dependent, and control variables.

HAM displays two or more E-labeled expressions, then returns a list of the E-labels. The first expression is the Hamiltonian, and the other expressiona are the auxiliary diferential equations, together with their general solutions, AUX[I] = K[I], whenever the Ith differential equation is of the trivial form 'D(AUX[I],T) = 0. The K[I] are undetermined constants of integration.

HAM is directly suitable for the autonomous time-optimal problem. Other problems may be converted to this form by introducing extra state variables, as described in most optimal-control texts or in the report referenced in OPTVAR OUTPUT and OPTVAR DEMO.

[12.9] Qualitative Analysis

QUAL > contains MACSYMA functions for qualitative analysis of an expression, QUAL DEMO contains a demonstration, and QUAL OUT contains the output from executing the demo.

To use the functions do ALLOC(2); LOADFILE(STOUTE,">",DSK,MRG); BATCH(QUAL, ">", DSK, SHARE);

Top-level usage is of the form QUAL*(<expression>, <variables>)*

where <expression> is any given expression, <variables> is a given indeterminate or list of indeterminates. If omitted, this argument defaults to all of the indeterminates in the first argument.

QUAL returns a list of E-labels of displayed equations, each of the form

```
<property name> = <property value>
```

where <property name> is one of the second-level function names below, and <property value> is the value returned by that function. These second-level functions may also be used directly. Usage is of the form

```
REVELATION(<expression>, <minimum>, <maximum>);
BOUNDS(<expression>);
SLOPES(<expression>, <variables>);
CURVATURE(<expression>, <variables>);
SYMMETRY(<expression>, <variables>);
PERIODS(<expression>, <variables>);
ZEROSANDPOLES(<expression>, <variables>);
STATIONARYPOINTS(<expression>, <variables>);
LIMITS(<expression>, <variables>);
```

where <expression>, <variables>, and their defaults are as for QUAL.

[12.10] Units Conversion

The file UNITS > contains assignments for automatic conversion to MKS metric units. Usage example: 5*FT + METER + CM; simplifies to 2.534*METER .

Erroneously dimensionally inhomogeneous expressions are revealed by uncollected terms. For example, 5*FT + SECOND; does not simplify to one term.

The supplied conversions comprise a rather complete set, but it should be clear how to supplement them or produce an analagous set for conversion to other units.

[12.11] The Eigen Package

The BATCH file EIGEN > contains a package of functions is written in top-level MACSYMA. Its purpose is to compute right eigenvectors, right unit eigenvectors, eigenvalues, and similarity transforms. *NOTE: This package currently will not handle systems with multiple eigenvalues. Implementation of this capability will await the ability to handle multiple roots from SOLVE.*

EVEC1*(M,mu,modes)* computes right eigenvectors of the matrix *M*, given the eigenvalues *mu*; mu is a one dimensional array of the eigenvalues of *M*. *modes* is the order of the system. EVEC1 returns a list of lists which are the eigenvectors; i.e., it returns a list of sublists, each of which contains the components of an eigenvector.*

EVEC2*(M,mu,modes)* is exactly the same as EVEC1 except that it constructs the list of eigenvectors differently from EVEC1. It is a toss-up as to which is faster.

INPROD*(x,y)* computes the real inner product of two lists (not vectors). This inner product is the sum of the products of the respective components of the lists. The two lists used as arguments to INPROD must be of the same length.

UEVEC*(M,mu,modes)* computes the unit eigenvectors of the matrix M. That is, the eigenvectors are of unit length as defined by INPROD(x,x). UEVEC merely calls evec1 to compute the eigenvectors, and then calls INPROD to compute the length of each eigenvector in turn and divides by this length. UEVEC returns a list of vectors (not lists) which are the unit eigenvectors. Since UEVEC calls EVEC1, the eigenvalues mu (defined as in EVEC1) must be distinct.

EVALS*(mat,lambda)* computes the eigenvalues of the matrix *mat* and stores them in the one dimensional array *lambda*. It returns the value "done". The array *lambda* must be set up as an array before the call to EVALS. Also, this function will not handle multiple eigenvalues.

SIMTRAN*(mat)* is the function which computes the similarity transformation mentioned above. it merely calls the other functions such as EVALS, UEVEC, etc. to determine the eigenvalues and right unit eigenvectors. For a symmetric matrix with distinct eigenvalues, the matrix formed by taking the unit eigenvectors as columns is an orthogonal matrix, as is its transpose. Thus if A is the original symmetric matrix (with distinct eigenvalues) and Q is the orthogonal matrix constructed as above, and QT is the transpose of Q, then

QT*A*Q = D

where D is the diagonal matrix with the eigenvalues of A on the diagonal. SIMTRAN takes the matrix *mat* and fills the global array *lambda* with the eigenvalues of *mat*. It also fills the globally defined matrices Q and QT with the matrices Q and QT as described above.

The array *lambda* and the matrices Q and QT must exist as an array and matrices (presumably loaded with dummy info) before the call to SIMTRAN. The matrix *mat* should not have multiple eigenvalues for the reasons mentioned in previous comments.

[12.12] Elimination by Resultants

ELIM LISP contains a program, written by Richard Bogen, for eliminating variables from equations (or expressions assumed equal to zero) by taking successive resultants. the call is:

ELIMINATE*([eq1,eq2,...,eqn],[v1,v2,...,vk])*

This returns a list of n-k expressions with the k variables *v1,...,vk* eliminated. First *v1* is eliminated yielding n-1 expressions, then *v2* is, etc. If k=n then a single expression in a list is returned free of the variables *v1,...,vk*. In this case SOLVE is called to SOLVE the last resultant for the last variable. If there are floating point numbers in the input expressions then KEEPFLOAT:TRUE should be done so as not to have them rationalized in taking the resultant.

EXAMPLE

```
(C5) .5*X^4+Y*X+Z;
                                              4
(D5)                              Z +X Y + 0.5 X

(C6) 3*X+5*Y-Z-1;
(D6)                             - Z + 5 Y + 3 X - 1

(C7) Z^2+X^2-Y^2+5;
                                  2     2   2
(D7)                              Z  - Y + X  + 5


(C8) ELIMINATE([D7,D6,D5],[Y,Z]),KEEPFLOAT:TRUE;

                 8           6         5          4          3          2
(D8) [150.0 X - 375.0 X  + 50.0 X  + 275.0 X  + 100 X  + 550 X

                                                              + 1400 X + 3100]
```

[12.13] Integration of Special Forms

INTSCE LISP contains a routine, written by Richard Bogen, for integrating products of sines,cosines and exponentials of the form

```
EXP(A*X+B)*COS(C*X)^N*SIN(C*X)^M
```

The call is INTSCE*(expr,var) expr* may be any expression, but if it is not in the above form then the regular integration program will be invoked if the switch ERRINTSCE[TRUE] is TRUE. If it is FALSE then INTSCE will err out.

12.14 Integral Equations

The following package was written by Richard Bogen based on some routines of David Stoutemeyer. It is still an experimental version, untranslated and uncompiled at present.

To load the package, do BATCH(INTEQN,">",DSK,RAB).

CAVEAT: To free some storage, a KILL(LABELS) is included in this file. Therefore, before loading the integral equation package, the user should give names to any expressions he wants to keep.

The usage is very simple. The main function is called IEQN. It takes five arguments though only the first one is required. If trailing arguments are omitted they will default to preset values which will be announced.

Two types of equations are considered. An integral equation of the *second kind* is of the form:

```
                                B(X)
                               /
                               [
(D3)        P(X) = Q(X, P(X),  I     W(X, U, P(X), P(U)) dU)   .
                               ]
                               /
                                A(X)
```

An integral equation of the *first kind* is of the form:

```
                        B(X)
                       /
                       [
(D4)                   I     W(X, U, P(U)) dU = F(X)
                       ]
                       /
                        A(X)
```

The unknown function in these equations is P(X) while Q,W,A, and B are given functions of the independent variable. Although these are the general forms, most of the solution techniques require particular forms of Q and W.

The techniques used are:

For SECONDKIND equations:

FINITERANK: for degenerate (or separable) integrands.
DIFFEQN: reduction to differential equation.
TRANSFORM: Laplace Transform for convolution types.
FREDSERIES: Fredholm-Carleman series for linear equations.

TAILOR: Taylor series for quasi-linear variable-limit equations.
NEUMANN: Neumann series for quasi-second kind equations.
COLLOCATE: collocation using a power series form for P(X) evaluated at equally spaced points.

For FIRSTKIND equations:

FINITERANK: for degenerate integrands.
DIFFEQN: reduction to differential equation.
ABEL: for singular integrands.
TRANSFORM: see above
COLLOCATE: see above
FIRSTKINDSERIES: iteration technique similar to Neumann series.

Also, differentiation is used in certain cases to transform a FIRSTKIND into a SECONDKIND.

The calling sequence is

IEQN*(ie,unk,tech,n,guess)* where *ie* is the integral equation; *unk* is the unknown function; *tech* is the technique to be tried from those given above (*tech* = FIRST means: try the first technique which finds a solution; *tech* = ALL means: try all applicable techniques); *n* is the maximum number of terms to take for TAYLOR, NEUMANN, FIRSTKINDSERIES, or FREDSERIES (it is also the maximum depth of recursion for the differentiation method); *guess* is the initial guess for NEUMANN or FIRSTKINDSERIES.
Default values for the 2nd thru 5th parameters are:

unk: P(X), where P is the first function encountered in an integrand which is unknown to MACSYMA and X is the variable which occurs as an argument to the first occurrence of P found outside of an integral in the case of SECONDKIND equations, or is the only other variable besides the variable of integration in FIRSTKIND equations. If the attempt to search for X fails, the user will be asked to supply the independent variable;
tech: FIRST;
n: 1;
guess: NONE, which will cause NEUMANN and FIRSTKINDSERIES too use F(X) as an initial guess.

The value returned by IEQN is a list of labels of solution lists. The solution lists are printed as they are found unless the variable IEQNPRINT[TRUE] is set to FALSE. These lists are of the form

[SOLUTION, TECHNIQUE USED, NTERMS, FLAG]

where FLAG is absent if the solution is exact. Otherwise, it is the word APPROXIMATE or INCOMPLETE corresponding to an inexact or non-closed form solution, respectively. If a series method was used, NTERMS gives the number of terms taken (which could be less than the *n* given to IEQN if an error prevented generation of further terms).

For examples, do BATCH(INTEXS,">",DSK,RAB) which will load an array called EQ with about 43 sample integral equations. Then try IEQN(EQ[1]), IEQN(EQ[30],P(X),ALL), for instance.

12.15 Numerical Techniques

[12.15.1] Numerical Integration

ROMBER is a program for Romberg numerical integration written by Richard Fateman. The calling sequence is ROMBERG*(f,a,b,lim)*

where f is a function name (elsewhere, f(x):= .., returns a number which must be floating), a and b are limits of integration, and lim is a limit (e.g. 1.0e-4) for the accuracy of the answer. ROMBERG computes upper and lower bounds on the integral, and goes through a maximum of 10 iterations to satisfy limit. if limit is unsatisfied, then it prints error message. It is much better than Simpson's rule, and is handier in terms of usage. It may be loaded by

```
BATCH(ROMBER,">",DSK,SHARE)
```

Also on ROMBER > is Simpson's rule, called by SIMPSON*(f,a,b,n)*

where n is number of subdivisions, which had better be an even integer. for SIMPSON; f, a, and b, (and maybe n), need not be numbers.

To integrate the exponential function from 0 to 1 to a tolerance of 1.0E-4, try ROMBERG(?exp,0,1,1.0E-4); . The time taken is 18 msec (0.018 sec). If one uses G(X):=EV(%E^X,NUMER) instead, and does ROMBERG(g,0,1,1.0E-4); the time is still only 150 msec.

[12.15.2] Fast Fourier Transforms

The following describes some FFT routines written by Tom Knight. To load the routines do LOADFILE(FFT,FASL,DSK,SHARE); The basic functions are: FFT(Fast Fourier Transform), IFT (Inverse Fast Fourier Transform).

These functions perform a (complex) fast fourier transform on either 1 or 2 dimensional FLOATING-POINT arrays, obtained by: ARRAY(array,FLOAT,dim1); or ARRAY(array,FLOAT,dim1,dim2); for 1d arrays dim1 must equal 2^n-1, and for 2d arrays dim1=dim2=2^n-1 (i.e. the array is square). (Recall that MACSYMA arrays are indexed from a 0 origin so that there will be 2^n and $(2^n)^2$ arrays elements in the above two cases.)

The calling sequence is:

FFT*(real-array,imag-array)*

IFT*(real-array,imag-array)*

The real and imaginary arrays must of course be the same size. The transforms are done in place so that *real-array* and *imag-array* will contain the real and imaginary parts of the transform.

Other functions included in this file are:

POLARTORECT*(magnitude-array,phase-array)* converts and magnitude phase arrays into real/imaginary form putting the real part in the magnitude array and the imaginary part into the phase array

RECTTOPOLAR*(real-array,imag-array)* undoes POLARTORECT

(the above 4 functions return a list of their arguments)

[12.15.3] Roots of Equations by Interpolation

The file INTPOL FASL, created by Charles Karney, contains the function

INTERPOLATE*(func,x,a,b)* which finds the zero of *func* as *x* varies. The last two args give the range to look in. The function must have a different sign at each endpoint. If this condition is not met, the action of the of the function is governed by INTPOLERROR[TRUE]). If INTPOLERROR is TRUE then an error occurs, otherwise the value of INTPOLERROR is returned (thus for plotting INTPOLERROR might be set to 0.0). Otherwise (given that MACSYMA can evaluate the first argument in the specified range, and that it is continuous) INTERPOLATE is guaranteed to come up with the zero (or one of them if there is more than one zero).
The accuracy of INTERPOLATE is governed by INTPOLABS[0.0] and INTPOLREL[0.0] which must be non-negative floating point numbers. INTERPOLATE will stop when the first arg evaluates to something less than or equal to INTPOLABS or if successive approximants to the root differ by no more than INTPOLREL * <one of the approximants>. The defaults values of INTPOLABS and INTPOLREL are 0.0 so INTERPOLATE gets as good an answer as is possible with the single precision arithmetic we have. The first arg may be an equation. The order of the last two args is irrelevant. Thus

```
INTERPOLATE(SIN(X)=X/2,X,%PI,.1);
```

is equivalent to

```
INTERPOLATE(SIN(X)=X/2,X,.1,%PI);
```

The method used is a binary search in the range specified by the last two args. When it thinks the function is close enough to being linear, it starts using linear interpolation.

An alternative syntax has been added to interpolate, this replaces the first two arguments by a function name. The function MUST be TRANSLATEd or compiled function of one argument. No checking of the result is done, so make sure the function returns a floating point number.

```
F(X):=(MODEDECLARE(X,FLOAT),SIN(X)-X/2.0);
INTERPOLATE(SIN(X)-X/2,X,0.1,%PI)       time= 60 msec
INTERPOLATE(F(X),X,0.1,%PI);            time= 68 msec
TRANSLATE(F);
INTERPOLATE(F(X),X,0.1,%PI);            time= 26 msec
INTERPOLATE(F,0.1,%PI);                 time=  5 msec
```

[12.15.4] Special Functions

The file BESSEL FASL contains routines for computing numerical values for various special functions. If they are given non-numeric arguments they return themselves.

Bessel Functions

The following functions compute Bessel functions of integer order for real arguments.

J0*(X)* returns the value of the zeroth order Bessel function at *X*

J1*(X)* returns the value of the Bessel function of first order at *X*

JN*(X,N)* returns the *N*th order Bessel function. In addition it sets up an array JARRAY of N+1 elements, (numbered from 0 to ABS(N)) such that JARRAY[I] gives the value of the I'th order Bessel function with argument X. (If N < 0 then JARRAY[I] gives the (-I)'th Bessel function).

Modified Bessel Functions

The following functions compute the Modified Bessel Functions I of integer orders for real arguments.

I0*(X)* returns the value of the modified Bessel function of zeroth order.

I1*(X)* returns the value of the modified Bessel function of first order.

IN*(X,N)* works the same way as JN(X,N), except that the array is called IARRAY.

Since the modified Bessel function blows up like EXP(ABS(X)) at infinity, they cannot be evaluated directly for ABS(X) > 83 (due to overflow). The following functions avoid this problem:

GO*(X)* returns IO(X)*EXP(-ABS(X)).

G1*(X)* returns I1(X)*EXP(-ABS(X)).

GN*(X,N)* returns IN(X,N)*EXP(-ABS(X)). The array generated by GN is called GARRAY.

Complex Bessel Function of positive fractional order

BESSEL*(Z,A)* returns the Bessel function J for complex Z and real A $\geq$ 0.0 . Also an array BESSELARRAY is set up such that BESSELARRAY[I] = J[I+A-ENTIER(A)](Z)

AIRY*(X)* returns the Airy function Ai of real argument *X*.

Plasma Dispersion Function, NZETA(Z).

This function is related to the complex error function by

```
NZETA(Z) = %I*SQRT(%PI)*EXP(-Z^2)*(1-ERF(-%I*Z))
```

NZETA*(Z)* returns the complex value of the Plasma Dispersion Function for complex Z.

NZETAR*(Z)* returns REALPART(NZETA(Z)).

NZETAI*(Z)* returns IMAGPART(NZETA(Z)).

Normal distribution function

GAUSS*(MEAN,SD)* returns a random floating point number from a normal distribution with mean *MEAN* and standard deviation *SD*

[12.15.5] Polynomial Zeros

REALZERO*(poly,anything)* is a program to isolate zeros of a polynomial based on the Collins-Loos differentiation algorithm. The 2nd argument,currently ignored, is reserved for some "epsilon" . To access it, one must LOADFILE(eb,lap,dsk,rjf)$, BATCH(COLLIN,">",DSK,RJF)$, TRANSLATE(ISOLATE,RR)$. It seems to be slower than REALROOTS for a similar task (occasionally uses NROOTS, though it needn't).

13 The MACSYMA Editor

13.1 Introduction

The major features of the editor are its concise commands (few alphabetic characters), its varied assortment of commands, concatenation of commands as in TECO (the PDP-10 file editor), mnemonics for command names (once you know them, R means "move in the Reverse direction" ; B means "move to the Bottom"), and compatibility with TECO as to command names (in the case of C, D, F, G, I, J, K, L, R, and S).

13.2 Entering the Editor

At any time while the user is inputting a command line to MACSYMA, he may enter the input-stream editor by typing "altmode" or "escape", henceforth denoted by <$>. The editor is given the string of characters typed so far in the current input line. In the case of a detected syntax error, upon typing <$>[1] the entire previous command string will be given to the editor. Alternatively, <control>-Y also retrieves the previous command string (even if other characters have already been typed in the current C-line). Thus, the previous command line is readily available for editing without retyping it.

When MACSYMA detects a syntax error in an input line, the error message sometimes displays the offending expression to help the user pinpoint the source of error. This is inconvenient if the expression is large, especially when the user has a slow terminal. Gradually this scheme is being replaced by a scheme that does not display the offending expression, but rather sets ERREXP to it, and prints out the error message followed by the message "ERREXP contains the offending expression".

One may also request the editor to edit or modify a previously accepted input line by using the STRING function in MACSYMA. Typing STRING(Ci) will restore the expression labeled as Ci as the current input string. This enables the user to modify it by then immediately typing <$>. For a simpler method, see the MYV command below.

1. <$> must be the first character typed on the next command line. Any other character (except "space") causes the edit buffer to be emptied.

All the commands to the editor reference a cursor (displayed as an underscore or back-arrow, depending on the console) which is displayed within or at either end of the string of characters currently being edited (called the "input string" from now on). The value of the variable **CURSOR** determines what character is used (see 10.7).

The editor accepts a command string which must be terminated by <$><$>. A command string is any concatenation of one or more legal commands which will be processed in left-to-right order. Display of the input string occurs at the end of the processing of each command string. <$> is used to enter the editor, to exit from the editor (as <$><$>), and to terminate insert or search substrings. Otherwise, spurious <$>'s are ignored. Rubouts (the rubout or delete key on the console) may be used at any point prior to command termination to delete the last character typed in. ?? deletes the entire command. At any point prior to command termination, the user may type a <control>K, and the editor will reprint the characters of the command typed so far.

Occasionally, one gets a syntax error because of omitting characters from the end of a command (especially right parentheses). By typing <$><$><$> immediately, as the first 3 characters of the next input line, the last command will be automatically reproduced on the current input line at which point one can supply the missing characters or rubout erroneous characters. For example:

```
(C1) (((X+1)*X+2)*X+3$
( ( ( X + 1 ) * X + 2 ) * X + 3 ***$***
SYNTAX ERROR
PLEASE REPHRASE OR EDIT

(C1) <$>
_(((X+1)*X+2)*X+3
<$><$>
(C1) (((X+1)*X+2)*X+3 )*X+4$
```
(In the above line the user typed the
characters after the 3)

13.3 A Description of the Commands

Some commands may be prefixed by an integer (represented below by "n") which usually may be positive or negative; although it may be zero as well in the case of K, L, and W; and it must be non-negative in case of W. The default value of n is +1. Except in the case of R, if n is positive the commands operate toward the right of the cursor, if n is negative they operate toward the left. I and S are two of the few commands which may be followed by other characters, namely the characters which constitute the insert or search strings. An error message will be printed if an illegal command substring is encountered or if any command substring fails. In case of such error, the processing of the current command string will be terminated at that point, with the offending command substring indicated.

Command	*Mnemonic*	*Action*
Commands which move the cursor		
nC	Character	moves the cursor past n characters.
nR	Reverse	moves the cursor past n characters in the reverse direction (nR = -nC).
J or T	Jump to Top	moves the cursor to the beginning of the input string.
B or ZJ	Bottom	moves the cursor to the end of the input string.
nL	Line	moves the cursor to the right of the nth carriage return (0L moves left); e.g., L moves to the next line.
nSstring<$>		moves the cursor to the right (left if n is negative) of the nth occurrence of "string" in the input string.
nS	Search	repeats the last S command given.
) or]	Move	moves the cursor right from the current position over the next balanced pair of parentheses (or brackets).
(or [	Move	similar to) or] but moves left.
Commands which delete characters		
nD	Delete	deletes n characters, and saves them in the "save-register" (see the G command below).

nK	Kill	deletes all the characters through the nth carriage return (0K kills left), and saves them in the "save-register"; e.g., K deletes the remainder of this line.
M) or M]	Delete	similar to) or] but deletes the characters moved over and saves them in the "save-register".
M(or M[	Delete	similar to M) or M] but moves left.
nFRstring<$>		deletes the next n occurrences of string. (This command is a special case of the FR command below and can be used in this way only when it is the last argument in the command string).

Commands which insert characters

Istring<$>	Insert	inserts the characters "string" at the current cursor position. The cursor is positioned at the right of the inserted text. If no argument is given then the string of the last I command which had one is used.
G	Get	inserts at the current cursor position the characters deleted by the last use of D, K, or M. Thus G may be used in combination with D or K to move characters from one place to another in the input string; or to recover from an accidental use of D or K. There is only one "save-register".
nFRstring1<$>string2<$>		replaces the next n occurrences of string1 by string2. If n is 1 it may be omitted. nFR given with no string arguments uses those from the last FR given which had them.
MFRstring1<$>string2<$>		replaces all occurrences of string1 by string2.
YVname<$>	Yank value	puts into the editing buffer, the value of the argument whose name is given, if a label or the name of a user variable, at the current cursor position leaving the cursor at the end of the inserted string.
YFname<$>	Yank function	puts into the editing buffer the definition of

the user function whose name is given (as with YV). If the name is followed by a list of subscripts in brackets, then the named subscripted function is brought into the buffer. This command provides an alternative to DISPFUN and STRING (see 10.2).

Note: If the YV or YF commands are prefixed by the letter M then the editor will clear the buffer before yanking, and also will leave the cursor at the head of the edit string when done.

Commands which control display of results

P	Print	simply reprints the input string. This is useful in case of console problems.
nW	Window	controls the window size of the display, which is the maximum number of characters displayed on each side of the cursor. This is useful in case of slow consoles and large input strings. 0W will cause only the cursor to be displayed. Once set the window size remains at that setting until it is reset. V View restores the display to full view, which is the normal mode (affected only by W).
Q	Quit	exits the editor without reprinting the just edited string.

<$><$> will exit from the editor and is also the command string terminator. Two examples of legal command strings are 4C3DIFOO<$><$> and -2SBAR<$>3R<$><$>. The first moves right over four characters, deletes the next three characters, and inserts FOO. The second searches from the current pointer position to the beginning of the text for the second occurrence of BAR then moves left over three characters.

Example

```
(C1) X:1$

(C2) NATRIX([A,4],[-1,A/2]);

(D2)                    NATRIX([A, 4], [- 1, 1/2 A])
(C3)<$>
```

```
_NATRIX([A,4],[-1,A/2])
DIM<$><$>
M_ATRIX([A,4],[-1,A/2])
]2CD<$><$>
MATRIX([A,4],[_1,A/2])
<$><$>    (In the line below the user typed the ;)
(C3) MATRIX([A,4],[1,A/2]);

                                         [ A    4   ]
(D3)                                     [          ]
                                         [ 1  1/2 A ]
(C4) CHARPOLY(%,X);
(D4)                          (A - 1) (1/2 A - 1) - 4
(C5)<$>
_CHARPOLY(%,X)
S%<$>-DID3<$>CI'<$><$>
CHARPOLY(D3,'_X)
<$><$>    (In the line below the user typed the ;)
(C5) CHARPOLY(D3,'X);
(D5)                          (A - X) (1/2 A - X) - 4
```

14 Batch Functions

14.1 Introduction

The Batch set of functions in MACSYMA, namely BATCH, DEMO, and BATCON (mnemonic for BATch CONtinue), provide a facility for executing command lines stored on a disk file rather than in the usual on-line mode. This facility has several uses, namely to provide a reservoir for working command lines, for giving error-free demonstrations, or helping in organizing one's thinking in complex problem-solving situations where modifications may be done via the PDP-10 TECO file editor.

A batch file consists of a set of MACSYMA command lines, each with its terminating ; or $, which may be further separated by spaces, carriage-returns, form-feeds, and the like. The BATCH and DEMO functions have both a simple and more complicated format, which are described below.

14.2 The Simple Format

BATCH*(filename1, filename2, DSK, directory)*

(The same function format holds for DEMO as well.) The arguments to BATCH (or DEMO) in this format specify the file which is to be batched. Here, each file is specified by two filenames of at most six characters each, the device the file is on (which is normally DSK), and the user file directory. E.g. DEMO(TAYLOR,DEMO,DSK,DEMO) calls for "demonstrating" (see below) the file TAYLOR DEMO on the DEMO disk directory. Latter arguments to the BATCH or DEMO functions may always be omitted if they are known from previous file-manipulating functions.

The BATCH function calls for reading in the command lines from the file one at a time, echoing them on the user console, and executing them in turn. Control is returned to the user console only when serious errors occur or when the end of the file is met. Of course, the user may quit out of the file-processing by typing control-G at any point (see 4) .

DEMO differs from BATCH only in that it pauses after the execution of each command line, waiting for the user to type a space which tells it to go on. If the user types any other character, file-processing will then terminate, giving control over to the user console. (The user may actually continue processing from the file at any time - see the BATCON function below.)

14.3 The More Complicated Format

BATCH*([fn1, fn2, DSK, directory], <delay-switch>, <index-specification>)*

The arguments to BATCH or DEMO in this mode are as follows:

The first argument is the file specification (as above), enclosed in brackets.

The second argument, the delay-switch, may be answered by ON or OFF (the default if it is omitted). This switch has to do with the temporary inability of LISP, the system underlying MACSYMA, to have more than one input file open at a time. If in the course of batching in a file of command lines, execution of a function forces a second file to be input, this would ordinarily cause an error. However, setting the delay-switch to ON causes the entire batch file to be read in before execution of it begins, thus preventing the error. The default for the delay-switch is OFF, as the circumstance described above is not frequent, it takes some time to read in a batch file, and one may always continue batching via the BATCON function. As soon as the inability of LISP is removed, this switch will no longer be needed.

The index-specification is given by one or two arguments, the possibilities being: (In the following, m and n are positive integers.)

(i) *m*. This indicates that processing is to begin with the mth command line in the file. Thus, the default for the index-specification is 1.

(ii) *m*, *n*. This indicates that only the *m*th command line through the *n*th command line are to be processed.

(iii) a variable (say FOO). FOO must be non-numeric and neither TRUE nor FALSE. This causes file-processing to begin at FOO&& (see 14.5) and continue until the end of the file. This makes it unnecessary to count command lines as required by (i) above.

(iv) variable (say FOO), continue-flag. The continue-flag is either ON (the default, and unnecessary) or OFF. If OFF, this enables one to separate a batch file into subfiles by prefixing a command line in the file with FOO&&. By using FOO as the index-specification, one may execute only that subfile which begins with FOO and ends with some other variable&&, or the end of file. If the continue-flag is ON, this causes mode (iv) to operate as (iii) above.

One can see that BATCH(TAYLOR,DEMO,DSK,DEMO) and BATCH([TAYLOR,DEMO,DSK,DEMO], OFF, 1) are equivalent.

14.4 The BATCON Function

The BATCON function is used to continue or change the last BATCH or DEMO function, without it being necessary to mention again BATCH or DEMO, the file specification, or the setting of the delay-switch. Of course, if one wishes to change any of these, a new call to BATCH or DEMO is required.

The possible arguments to BATCON are as follows:

(i) a number

(ii) number1, number2

(iii) a variable

(iv) variable, continue-flag

They are all as in 8.3. The numeric arguments may involve the variable BATCOUNT[0] which is set to the number of the last expression BATCHed in from the file. Thus BATCON(BATCOUNT-1) will resume BATCHing from the expression before the last BATCHed in from before. One other mode is possible:

(v) skip-flag. The skip-flag is useful if an error has occurred while batching, or if the user wishes to interject command lines from the console while in DEMO-mode and then to continue processing from the file. The skip-flag may be either TRUE or FALSE. If FALSE, this indicates that processing is to continue with the last command line attempted (supposedly edited, in case of error); if TRUE, this indicates that processing is to continue with the next (untried) command line in the file.

14.5 Miscellany

(1) Comments may be added to batch files at any point, and will, of course, be treated as such when batching in the file. A comment is any string beginning with /* and ending with */ as in PL/I.

(2) Any command line in a batch file may begin with variable&&. This labels that command line so that the file can be partitioned into subfiles. If not in a subfile mode, this prefix will be treated as a comment.

(3) When using the batch functions, it is inconvenient to keep track of which Di label MACSYMA will assign to a computation; yet later command lines often need to refer to an earlier computation. One way to get around this, of course, is for the user to explicitly label some of his command lines. A function %TH is also provided, such that %TH(i), where i is positive, refers to the result of the *ith* previous command line. E.g., %TH(1) and the variable % both refer to the same computation.

(4) When BATCHing in several files it is possible for one file to unintentionally cause an error to occur in a subsequent one by duplication of names or settings of options. If the variable BATCHKILL[FALSE] is TRUE however, then the effect of all previous BATCH files is nullified because a KILL(ALL) and a RESET() will be done automatically when the next one is read in. If BATCHKILL is bound to any other atom then a KILL(BATCHKILL) will be done. (The default value of BATCHKILL is FALSE meaning to do nothing.)

(5) While BATCHing in a file which takes a lot of time to process the user may leave his terminal unattended. If an error occurs he may want some special action to be taken automatically. By setting the option ERRORFUN to the name of a function of no arguments one can have that function executed when any error occurs. Useful functions are QUIT and LOGOUT. However in the case of LOGOUT the user should also set the switch TTYOFF to TRUE to prevent his job from hanging up in the case it tries to output to the terminal (see 4). In addition if a file has been opened for writing, then a command to close it should be executed before the LOGOUT. Also, the user may wish to set DYNAMALLOC[FALSE] to TRUE (see 16) so that his job will not hang if additional storage space is needed

If the user is executing a function of his own and would like to signal an error he can use the functions ERROR and ERRCATCH (see 10.6).

(6) If the user does not have a directory of his own then he can use the one called USERS to store his files. He should identify them as his in some fashion such as using his login name for the first file names.

(7) The DEMO file directory contains many demonstration files which may be helpful to the user in learning to use MACSYMA.

15 Secondary Storage Functions

15.1 Introduction

There are two different reasons for wanting to use secondary storage while running a MACSYMA. Sometimes the user's intermediate expressions are large, and it is impossible to complete the job if all the intermediate expressions are kept in main memory. In this case the user would like to have his intermediate expressions written automatically onto the disk, in order to free main memory. On the other hand, some users would like to save some expressions onto the disk so that they can be read back into a future MACSYMA at a later time. In this case the user would like to specify certain expressions to be stored and to name the disk file where they are to be stored. MACSYMA offers the user two secondary storage schemes. The user may ask to have his expressions automatically filed away onto the disk, or he may, by means of the SAVE and STORE functions, exercise explicit control over the storage of expressions. These latter functions give the user more power and flexibility at the expense of a greater effort. It is expected that the user whose only concern is to run a big job which would not run without using secondary storage will use the automatic storage scheme, while the user who wishes to save expressions for use in later MACSYMAs will use the SAVE and STORE functions.

15.2 Automatic Storage of Expressions

A - How to use it

To activate the automatic storage scheme the user merely sets the MACSYMA option DSKUSE[FALSE] to TRUE. From this point on labelled expressions will be written out periodically onto the disk. (A labelled expression is one which is referred to by a line label, e.g. D4, C7, E12.) Once an expression is written onto the disk it will no longer reside in main memory and most of the main memory storage taken up by it will be released. When the user attempts to reference an expression which has been stored onto the disk, MACSYMA will retrieve the correct value from the disk file. In this scheme expressions are copied periodically onto the disk whenever there are enough to write out (see

FILESIZE). An alternative heuristic to use in order to free some storage is to write out all labelled expressions, values, functions, and arrays whenever the garbage collector finds that space is getting low. This is the purpose of the DSKGC function (see 10).

If the user is dealing with large expressions then his storage limit may be exceeded before FILESIZE expressions have been generated. In this case the DSKGC method should be used. If this situation does not occur and if the user prefers to have some control over how many expressions are saved in each file then the other scheme should be used.

B - Cleaning up the disk

The automatic storage scheme will in general cause several disk files to be created, which are of no further value after the user has finished running his current MACSYMA. There is a function called REMFILE, which will delete all the files created by the automatic storage scheme. Thus if the user does not want these files to stay around, he should execute REMFILE() before leaving MACSYMA. REMFILE will only delete files created in the same MACSYMA to which the REMFILE function is given. In order to delete files created in previous uses of MACSYMA it is necessary to use the DELFILE function (see 10).

C - Options

The user may specify how often files are written, how large they are, what they will be named, and what gets stored in them, or he may accept the default values for all these. The following MACSYMA options are relevant.

FILENAME - The value of this variable is the first name of the files which are generated by the automatic disk storage scheme. The default value is the first three characters of the user's login name concatenated with a three-digit random number (e.g. ECR864)

STORENUM - The value of this variable, an integer, is the second name of the last file written. Each time a file is written, this value is first increased by 1, so it must always be an integer. It is initially set to 0.

FILESIZE - The value of this variable is the number of expressions written into each file. The default value is 16.

DEVICE - The value of this variable is the default device. It is initialized to DSK.

UNAME - The value of this variable is the default sname. It is initialized to the user's login name, if he has a disk directory, and to USERS otherwise. UNAME determines to what directory disk files will be written.

DIREC - may be used as an alias for UNAME.

DSKALL - If TRUE will cause values, functions, arrays, and rules to be written periodically onto the disk in addition to labelled expressions. TRUE is the default value whereas if DSKALL is FALSE then *only* labelled expresions will be written.

15.3 Explicit Storage of Expressions

15.3.1 Use of the storage functions

The functions SAVE, STORE, and FASSAVE allow the user to explicitly state that certain expressions should be written onto the disk. These functions also allow him to specify the file into which these expressions should be written. They allow the user to store away arrays, function definitions, rules, and any other type of information. The main purpose of these functions is to allow the user to save expressions onto the disk so that they can be read into future MACSYMAs.

SAVE and STORE are identical in all respects but one. When an expression is STOREd it is both written onto the disk and removed from main memory. (When the expression is referenced, of course, the correct value is retrieved from the disk.) When an expression is SAVEd, it is written onto the disk but not removed from main memory. The only difference between these two functions is their effect on main memory storage.

FASSAVE is similar to SAVE but produces a FASL file in which the sharing of subexpressions which are shared in core is preserved in the file created. hence, expressions which have common subexpressions will consume less space when loaded back from a file created by FASSAVE rather than by SAVE. The user should note that FASSAVE files are not as flexible as SAVE files since the RESTORE function (see below) cannot be applied to them. Also if the user's

MACSYMA is already near maximal allocation, FASSAVE , which uses a considerable amount of space and time in doing its job, may not work, whereas SAVE still might.

SAVE, STORE, FASSAVE take any number of arguments. If the first argument is a list it is assumed to be the file specification (e.g. [fn1, fn2, DSK, directory]). In accordance with the standard options for file specifications, the latter arguments may be omitted from the list and the defaults will be assumed. If the first argument is not a list, the expressions will be written into a file with the default filename. The value of the MACSYMA variable FILENAME is the default first filename, and the value of the MACSYMA variable FILENUM is the default second filename. The value of FILENUM is increased by 1 each time a file is written, so its value must always be an integer. FILENUM is initially 0. The value of DEV is the default device, and the value of UNAME is the default username.

All arguments to SAVE or STORE, except possibly the first, must be one of the following:

(1) The name of an "information list" (see 8.1). SAVE(VALUES) will not cause MACSYMA options (e.g. SHOWTIME, RATPRINT, etc.) to be saved. Also ALIASES will be automatically saved with every use of the SAVE function if they exist.

(2) ALL When this atom is an argument every quantity associated with any information list is written.

(3) [m,n] when this list is given as an argument, every label whose line number lies between m and n inclusive gets written.

(4) When any other atom is an argument, it must be either an array, a function, or have a value. It gets written onto the disk.

(5) A=B The effect is similar to the case where the argument is just B, i.e. B gets written onto the disk. The only difference shows up if the file is read into some future MACSYMA. In that case, the expression which is referred to as "B" in the present MACSYMA will be referred to as "A" in the future MACSYMA. For example, suppose the user wishes to save some expression, say D7, for use in a future MACSYMA. He can execute STORE([FOO, BAR], YESTERDAYSD7 = D7). D7 is now stored onto the disk. When he comes back the following day and load in a fresh MACSYMA he merely executes LOADFILE(FOO, BAR, DSK, ECR) and the variable YESTERDAYSD7 will take on the value which D7 had yesterday. This renaming however has no effect on the present MACSYMA, where D7 must still be referred to as "D7". Note that if a SAVEd or STOREd file contains labelled

expressions they may conflict with expressions having the same label in the MACSYMA into which the file is loaded. For example if D9 is in a file which is loaded into a MACSYMA then it would replace the D9 which was already in the MACSYMA (if there was a D9 generated), or it would itself be replaced by D9 when the new D9 was generated. To avoid this difficulty the user should give labelled expressions a name as described in (4) above. He could also set LINENUM in the new MACSYMA or save it from the old one so that line numbers wouldn't conflict.

The user should note that each use of the SAVE or STORE function will cause exactly one file to be written, regardless of the number of arguments the function is given.

REMFILE(TRUE) will perform REMFILE() (see 15.2 - B) and in addition will delete files created by SAVE or STORE which haven't been assigned names explicitly by the user.

Certain MACSYMA variables (i.e. LINENUM, FILESIZE, etc.) are used to communicate to the MACSYMA system that certain options are in effect, or to tell the system to use certain values. These variables should not be STOREd (though they may be SAVEd), since the system programs will not be able to correctly retrieve their values from the disk. In general, one should not attempt to STORE variables whose purpose is to provide information to the system (i.e. MACSYMA options).

15.3.2 Retrieval of expressions stored on disk

1 - In the MACSYMA you are using

Expressions which are written onto the disk using the SAVE function also reside in main memory, so the notion of retrieving them from the disk in the present MACSYMA is not applicable. Expressions written onto the disk using STORE, however, no longer reside in main memory. When such expressions are referenced the system will always retrieve the correct value from the disk. When a STOREd array is referenced, the array will be brought back to main memory. Functions and values will be read from the disk correctly, but will not be returned to main memory. If the user wants to bring an expression back to main memory he may use the function UNSTORE. This function takes any number of arguments. Each argument must be an atomic variable, and if this atomic variable refers to an expression which is stored onto the disk, the expression is returned to main memory. Of course, when an expression is UNSTOREd, either by the user or by the system (as happens when STOREd arrays are accessed), a copy of the expression still remains on the disk in the assigned file.

2 - In future MACSYMAs

Files created by SAVE and STORE can be loaded into future MACSYMAs using the LOADFILE function. This will set up in main memory all those expressions which were written into the file. Some of the expressions will have different names than they had in the MACSYMA where they were created, if the renaming option (i.e. arguments of the form A=B) of the STORE or SAVE function was used. Also, unless the FASSAVE scheme was used, expressions will generally take up more space than they did in the MACSYMA where they were created, as sharing among common subexpressions will be lost.

15.4 Saving a MACSYMA ~~Overnight~~

Often a user in the middle of his work would like to save everything onto the disk so he can go home and resume work tomorrow. When the user decides to save the state of his MACSYMA, he should execute for example: SAVE([fn1,fn2,DSK,directory],ALL)

This will write all his lines, arrays, functions, values, rules, and aliases (if he has created any), and the current value of LINENUM into a single disk file named fn1 fn2 (where these may be any names given by the user). Of course, the user should choose names for his files which are unique. If he does not have his own directory then he should use the USERS directory and his login name for the first file name. If the automatic storage scheme was in effect he should now execute REMFILE(); to delete useless files from the disk. When the user comes back the next day he should load a fresh MACSYMA and execute one of the following two functions:

LOADFILE(*fn1, fn2, DSK, directory*);

or RESTORE(*fn1, fn2, DSK, directory*);

The first command will cause all expressions to be loaded into the present MACSYMA. Whereas all the expressions may have fit into the MACSYMA in which they were generated, they may not fit into a new MACSYMA, because common subexpressions originally shared will not be shared in a new MACSYMA. However if the OPTIMIZE function is used (see 6.2.3), then some sharing of common subexpressions may be obtained. The RESTORE function does not cause the expressions to be loaded into main memory but does permit them to be accessed when needed. (This is as though STORE had been used on the information.) Thus it should be used if it is not desired to bring all the expressions into main memory at the same time.

16 Storage Management

In the LISP system in which MACSYMA resides, the space requirements of the user's programs and data may be increased during the execution of the programs. This is in contrast to static storage allocation systems in which the storage is completely allocated before the programs are executed and consequently the storage requirements must be completely known before execution time and cannot be changed during program execution. If they exceed the capacity of the memory space that has been allocated for them, the programs will not be allowed to run.

With our LISP system, a certain amount of space is initially allocated, the programs are started running, and the amount of space utilized changes during execution. If at some point the limit of available space is exceeded, program execution will be terminated.

This LISP divides up the available memory spaces into several portions on the basis of what kind of data they will contain.

BPS - (binary program space) for compiled functions and arrays.

FIXNUM - for integers which fit into one machine word.

FLONUM - for floating point numbers.

BIGNUM - the first word of numbers bigger than one machine word.

SYMBOL - for atomic symbols.

ARRAY - for array indicators.

LIST - for anything else not in the other spaces, e.g. uncompiled functions, symbolic expressions, etc.

PDL - for several kinds of pushdown lists.

When a MACSYMA is started up, each space is initially allocated some fixed amount. These spaces will grow as the user interacts with MACSYMA, each particular space growing as the user causes more objects to be created which reside in that space. For example, executing a command line which causes an out-of-core file to be loaded mainly increases BPS and LIST space. Also new labelled expressions are created every time a command line is executed and these occupy LIST space. Push down lists are used to store variables, return addresses, and other information related to the function calling mechanism.

When a space (except for BPS and the PDLs) is used up, a process is initiated called "garbage collection" which attempts to free up storage so it can be reused rather than trying to increase the size of the spaces. In very simple terms, it marks every word in a particular storage space which is still being utilized and then links up the unmarked words (termed "garbage") on a chain to be used to store subsequently created data. If this chain is not of a certain assigned minimal size, a special allocation routine is invoked. At this point several possibilities can occur, among which is the possiblity of increasing the size of the spaces. Before these are described however, there is something to be mentioned which the user should take note of. The initial allocations (later to be referred to as "level 0 allocations") are quite reasonable. Many problems run quite well using these allocations. If the user's problem does not run due to running out of storage, most often this is due to one of the following circumstances and not to the insufficiency of the allocations:

(1) The user has organized his problem poorly, thereby either not solving the problem he intended to solve, biting off too much in too short a time, or creating intermediate expression swell of perhaps incredible proportions. He should get a feel for the size of all of his expressions and the behavior of MACSYMA's functions on them.

(2) The user is retaining expressions in core that are useless to him. Since MACSYMA maintains a complete history of the user's session, it does not release the storage occupied by the user's data unless explicitly instructed to do so. This can be accomplished in several ways. One way is by using the functions KILL, REMVALUE, REMFUNCTION, and REMARRAY (see 10.3) which unbind an item from the expression it represents thus freeing up the storage occupied by the expression to be reclaimed on the next garbage collection. It is recommended that the user give a name to all labelled expressions which he wishes to keep around for a time, and then periodically do a KILL(LABELS). He should also KILL functions and arrays which he no longer needs. Killing a name will not accomplish much unless the labelled expression (D line) at which the assignment was done is also KILLed because the two symbols are holding onto the same expresson.

If the above approach is unacceptable because the (intermediate or final) expressions which the user needs occupy a lot of storage, he can store them on the disk (kill the corresponding label if any) and have them retrieved when needed. This can be accomplished in the following two ways (see Chapter 15).

The STORE function can be used to explicitly transfer expressions from main memory to disk. These will be brought back into main memory each time they are referenced. Since the process of retrieving an expression from a large file may be rather time consuming, the user can use the UNSTORE function to bring an expression back to main memory and keep it there. However, in general an UNSTOREd expression will occupy more space then it originally did because several copies of common subexpressions will be created. These were originally represented by pointers to a single copy.

Another way to transfer expressions to the disk is by using the automatic storage mechanism. This will cause expressions to be STOREd automatically in an effort to conserve space. As with the explicit use of the STORE function, the expressions will be retrieved by MACSYMA when they are referenced. There are two heuristics which MACSYMA uses to decide when to store expressions. One is to STORE all values, functions, arrays, and labelled expressions whenever available space becomes low. This is the purpose of the DSKGC function. The other heuristic is to periodically STORE a fixed number of labelled expressions whenever that many get created above a minimum number which are to be kept around. The user can utilize this option by setting the switch DSKUSE to TRUE. In addition if DSKALL is TRUE, then all values, functions, and arrays will be written at this time as well.

(3) A recursive infinite loop has occurred because of a bug in either the user's code or in MACSYMA's code. Such a loop would cause storage spaces to grow indefinitely if possible. It may be difficult for the user to recognize that this situation has occurred, as opposed to a real need for more space, yet such a situation can cause havoc to any dynamic allocation scheme. By typing a control-D the user will be informed whenever a garbage collection occurs and a printout such as the following will occur:

```
;GC DUE TO ... SPACE
;2729[33%] LIST, 1935[94%] FIXNUM, 511[99%] FLONUM,
; 509[99%] BIGNUM, 629[15%] SYMBOL,
; 480[93%] ARRAY WORDS FREE
```

The numbers before the name of each space give the number of words of that space which are available. The percentages refer to the ratio of the available amount of space compared to the total amount (used plus unused). This may be of some help in determining whether the user's computation is doing what he expected. Typing a control-C will turn off this g.c. printout.

If the user feels that he is not in one of the above situations and that there is a real need for more space, he can avail himself of the following scheme:

When a space, e.g. LIST is exhausted, MACSYMA will print:

```
You have run out of LIST space
Do you want more?
Type ALL; NONE; a level-no.; or the name of a space;
```

At this point the user can type a control-A and enter a (MACSYMA-BREAK). He can then follow any of the procedures mentioned in (2) above. After getting out of the break by typing EXIT; he can then type OK; which indicates that the user believes he has freed up enough space thus making reallocation unnecessary. If he is wrong, he will get the above message again. He can also reply with the name of a space, i.e. one of FIXNUM, FLONUM, BIGNUM, SYMBOL, or ARRAY which will cause the size of that space to be expanded. If he replies with LIST or ALL then not only will list space be increased, but so will every other space. This is because if he needs more list space, then he probably needs more of the other spaces as well. Replying LIST also increments the "allocation level" by one. There are 5 equi-spaced levels of allocation, ranging from the initial level 0 to level 4. The user can also boost his MACSYMA up to that particular level of allocation immediately by replying with that level number.

At level 4 the maximal allocation possible on the computer is just about exhausted so that there is no higher level. Also, the amount of core space devoted to the allocatible storage spaces is obviously inversely proportional to the number and size of out-of-core files which are loaded in. If many out-of-core files are loaded in, allocation level 4 will not even be attainable. The user may wish for this reason alone to continue now and then with a fresh MACSYMA. If the user's computation exceeds this level of storage it will error out with the message "...STORAGE CAPACITY EXCEEDED". Since the size of the spaces can't be decreased, it is important not to increase them unless it is necessary.

If the user knows initially that his problem will require much space, the function ALLOC is provided. ALLOC takes any number of arguments which are the same as the replies to the "run out of space" question above. It increases allocations accordingly.

Lastly, if he is running a MACSYMA disowned, or for other reasons wishes storage space to be increased automatically as needed without having any questions asked, the user may set the switch DYNAMALLOC [FALSE] to TRUE which will allocate additional space whenever necessary.

Like the other spaces BPS and the PDLs cannot be decreased. BPS will continue to grow until the MACSYMA runs out of address space so caution should be exercised in causing out-of-core files to be loaded. For example, once the integration file is loaded into the user's MACSYMA it is there to stay even if he no longer uses it. The allocation of the PDLs is sufficiently large, so that if they are ever caused to overflow it is probably due to a recursive infinite loop in the user's programs.

17 Simple Plotting Functions

The MACSYMA functions PLOT,PARAMPLOT,GRAPH, and MULTIGRAPH produce character plots of specified functions and sets of data points. (They can also be used to produce output files for plotting on the Calcomp plotter or XGP attached to the PDP-10 used by the Artificial Intelligence Group). The format of these functions and the variables used by the corresponding routines are described below:

Variables

LINEL - width of graphing area in terms of number of characters.

PLOTHEIGHT - height of graph in terms of number of characters.

XAXIS[FALSE] - if set to TRUE will cause the Y=0 axis to be displayed.

YAXIS[FALSE] - if set to TRUE will cause the X=0 axis to be displayed.

Formats for PLOT and PARAMPLOT

PLOT*(F(x), x, low, high)* - plots the expression F(x) over the domain low < x < high.

PLOT*(F(x), x, low, high, INTEGER)* - as above, but plots F(x) only for integer values of x.

PLOT*(F(x), x, [x1, x2, x3, ..., xn])* - plots the function F(x) for the values x1,x2,x3,...,xn.

The first argument to PLOT may also be a list of functions rather than just a single function. This permits several functions to be plotted on the same set of axes. Three optional final arguments may also be given. They are: (1) an X axis label (quoted string or name), (2) a Y axis label, and (3) a list of plotting characters used for the given function(s) enclosed in "s. (Note also that if a special symbol such as ; , $, etc. is used it must be preceded by a \.) An * will be used to plot any functions which are not given a particular plotting character.

PARAMPLOT*(f1(t),f2(t),t,low,high)* plots the plane curve f(t) = (f1(t) , f2(t)) with parameter t. The syntax is basically like that of plot. For example,

```
PARAMPLOT(COS(T),SIN(T),T,0,2*%PI)
```

plots a circle. Also several curves may be presented at the same time by using the following syntax

PARAMPLOT*([f1(t), g1(t), ... h1(t)], [f2(t), g2(t), ..., h2(t)], t, low, high, [list of plotting characters])* - plots the plane curves f(t) = (f1,f2), g(t) = (g1,g2),...,h(t) = (h1,h2) using the specified plotting characters or the default "*". For example,

```
PARAMPLOT([COS(T),COS(T)+7],[SIN(T),SIN(T)],
                T,0,2*%PI,["@"])
```

plots two circles.

The user may wish to **TRANSLATE** or **COMPFILE** (see 10.8) the functions to be plotted as they might be evaluated many times.

Formats for GRAPH and MULTIGRAPH:

GRAPH*([x1, x2, x3, ..., xn], [y1, y2, y3, ..., yn])* - Graphs the two sets of data points.

GRAPH*([[x1, y1], [x2, y2], ..., [xn, yn]])* - Graphs the points specified by the list of coordinate pairs.

GRAPH*(xset, [yset1, yset2,..., ysetn],optional-args)* - allows graphing of one x-domain with several y-ranges; e.g. GRAPH([0,1],[[0,1],[1,2]],["&"]).

MULTIGRAPH*([[xset1, yset1], ..., [xsetn, ysetn]], optional-args)* - allows the user to produce a scatter-graph involving several x-domains each with a single y-range; e.g. MULTIGRAPH([[[0,1],[0,1]], [[3,4],[1,2]]],["&"]).

The three optional final arguments mentioned above under PLOT may also be used with GRAPH and MULTIGRAPH.

The plot produced by the above functions is a character plot on a coordinate system defined by axes along the minimum x and y values of the plot. The x and y coordinates are independently scaled to optimally use the specified graphing area. Note that this may distort the shape of the graph e.g., a circle could become an ellipse. The origin of the graph (left-hand corner) is given on the graph by the values of XORG and YORG; the computed increments (= one character) are given by the values of XDELTA and YDELTA and the maximum X and Y values are given by XMAX and YMAX. The axes are labeled with the number sequence 0,2,4,6,8,0,2,4,... as an aid in counting the number of increments from the origin.

When a graph is completed, the user must type a single character (on non-printing consoles), such as space or carriage return, to return control to MACSYMA.

Examples

```
(C1) XAXIS:YAXIS:TRUE$

(C2) PLOT([1/(X^2+1),X^2-1],X,-2,2,[@]);
```

```
4[    *                           .
 [                                .                           *
2[                                .
 [     *                          .
0[                                .                          *
 [      *                         .
8[                                .                         *
 [       *                        .
6[                                .                        *
 [        *                       .
4[                                .                       *
 [         *                      .
2[                             @@@@@@@                   *
 [          *              @@@   .   @@@
0[                        @@      .      @@             *
 [           *          @@        .        @@@
8[                   @@@          .           @@       *
 [            *     @@            .             @@@
6[              @@@@              .                @@@*
 [          @@@@                  .                  *@@@@
4[   @@@@@@      *                .                       @@@@@@
 [@@@             *               .                 *           @@@
2[                                .                *
 [.................*...............................................
0[                  *             .               *
 [                   *            .              *
8[                                .             *
 [                    *           .            *
6[                     *          .           *
 [                      *         .          *
4[                       **       .         *
 [                         *      .       **
2[                          **    .      *
 [                            **  .   ***
0[                             *******
 ------------------------------------------------------------------
  0 2 4 6 8 0 2 4 6 8 0 2 4 6 8 0 2 4 6 8 0 2 4 6 8 0 2 4 6 8 0 2 4
XORG=-2.0 YORG=-0.999744 XDELTA=0.062 YDELTA=0.091
XMAX=1.96800017 YMAX=3.0
```

```
(C2) POLARPLOT(RHO,NUMBREV):=BLOCK([THETA, LIMIT, X, Y, P, NUMER,
        RATPRINT], NUMER:TRUE, RATPRINT:FALSE, THETA:0.0, X:Y:[],
          LIMIT:72*NUMBREV, FOR I:1 THRU LIMIT DO
                ( P:RHO(THETA), X:CONS(P*COS(THETA),X),
                  Y:CONS(P*SIN(THETA),Y), THETA:THETA+%PI/36.0),
        GRAPH(X,Y,X,Y))$

(C3) XAXIS:YAXIS:FALSE$

(C4) F(T):=4-COS(8.0/3.0*T)$
```

```
(C5) POLARPLOT(F,3)$
```

```
  6[                      *  *  *  *     *  *  *  *
   [              * *  *         * *  *  * *         *  * *
  4[             *           * *             * *            *
   [      * *  *  *  *  *  ** * * * * * * * * * *** *  *  *  ** * *
Y 2[    *     *         **                         **         *      *
   [ **       *   * * **                             * ** *   *       * *
  0[*       *  ***                                          *** *       *
   [**  * *     *                                           *      * * **
  8[  *          *                                          *         *
   [**  * *     *                                           *      * * **
  6[*       *  ***                                          *** *       *
   [ *        **  * * *                                ** *  **         *
  4[  * *     *        ***                         ***        *      **
   [      * *  *           ** * * * * * * * * * ***          ** * *
  2[             **  *  *    * *             * *    *  *  * *
   [              * *            * *     * *            * *
  0[                   *  *  *  *  *  *  *  *  *  *  *
   ---------------------------------------------------------------------------
    0 2 4 6 8 0 2 4 6 8 0 2 4 6 8 0 2 4 6 8 0 2 4 6 8 0 2 4 6 8 0 2 4 6 8 0 2
XORG=-4.77137667 YORG=-4.77137756 XDELTA=0.14 YDELTA=0.6
XMAX=4.7713773 YMAX=4.77137697
                                   X
```

18 Complex Plotting and Graphics---PLOT2

18.1 Introduction

This is a description of the functions contained in the following 6 files:
APLOT2 > ARC CFFK, TEKPLT > ARC CFFK, PLOT3D > ARC CFFK
WORLD > ARC CFFK, PRINT > ARC CFFK, IFFUN > ARC CFFK.
They will be loaded up as needed when most of the functions below are used. However to get the "complete" PLOT2 environment, LOADFILE(PLOT2,LISP,DSK,SHARE) is recommended. This loads in complete AUTOLOAD properties for the functions described below. The effect of loading this file will be to cause the PLOT2 package to be automatically loaded when you need it. If you use PLOT2 regularly, you should consider including PLOT2 LISP DSK SHARE in your MACSYM (INIT) file; it will take up neglegible room in your MACSYM.

The capabilities of the routines described here include plotting of several curves on a single graph, plotting several graphs in different positions on the screen, saving plots, replotting plots with different scales without having to recompute any points, plotting of 3 dimensional surfaces, plotting of user defined dashed lines and symbols.

The devices supported are: the Tektronix 4010 and 4013, the Imlac PDS 1 and PDS 4 (using ARDS graphics conventions), the XGP and the Gould line-printer (at MIT) and printing and display consoles in a "preview" mode.

18.2 Two-Dimensional Plotting

PLOT2*(y-exprs,variable,var-range,optionals-args)* plots *y-exprs* on the y axis as *variable* (the x axis) takes on values specified by *var-range*.
y-exprs can take one of two forms:

i) *expr* plots a curve of *expr* against *variable*

ii) [*expr1,expr2,* .. *,expri,* .. *,exprn*] plots n curves of *expri* against *variable*. *expri* gets evaluated in the context

FLOAT(EV(*expri,variable=value gotten from var-range,* NUMER)). It is an error if this doesn't result in a floating point number.

var-range can have the following forms:

i) *low,high,* where *low* and *high* evaluate to numbers. *low* may be either greater or less than *high*. *variable* will take on CALCOMPNUM values equally spaced between *low* and *high*.

> Note that the first argument will be evaluated at *low* first e.g. PLOT2(1/X,X,-1,-3); calculates 1/(-1.0) before 1/(-3.0). This will only make a difference if the computation of the first arg changes a variable which changes the value returned by subsequent computation. Whether or not *low* < *high*, min(*low,high*) will be on the left of the plot.

ii) *low,high,*INTEGER. As in i), except *variable* will take on all integer values between *low* and *high* inclusive.

iii) [*val1,val2,* ... , *valn*] . *variable* takes the values specified by the list

iv) *arrayname* where *arrayname* is the name of a declared floating-point one-dimensional array (i.e. declared by ARRAY(*arrayname*,FLOAT, *max-index*);). *variable* takes the values from *arrayname*[0] thru *arrayname*[*max-index*] (*max-index* is the maximum index of *arrayname*

optional-args can be any of the following:
i) X-Label, Y-Label or Title descriptor
ii) Line type descriptor
iii) FIRST, SAME and LAST
iv) POLAR, LOG, LINLOG, LOGLIN

The optional arguments may appear in any order. The rule for evaluation of the optional args is as follows. If the argument is atomic it gets evaluated. The resulting arguments are the ones that get used.

If you want to plot more than 3-4 curves on the same plot investigate using the NOT3D option to PLOT3D (see 18.4)

Examples

`PLOT2(SIN(X),X,-%PI,%PI);`	plots sin(X) against X as X takes on CALCOMPNUM values between -%PI to %PI
`PLOT2(X!,X,0,6,INTEGER);`	plots X! as X takes integral values between 0 and 6
`F(X):=SQRT(X);`	
`PLOT2(F(X),X,[-2,3,100.12]);`	plots F(X) as X takes the values in the values in the list
`PLOT2([X+1,X^2+1],X,-1,1);`	plots 2 curves on top of each other

PLOT2*(y-funs,var-range,optionals-args)* is the alternative form for PLOT2. *y-funs* must be a function of one argument or a list of functions of one argument. The functions must be either translated or compiled functions which return a floating point number when it is given floating point arg (or integer arg if the INTEGER arg to PLOT2 is given). This form of PLOT2 acts as though you had given a argument to the *y-funs*, and also specified that argument as the *variable* in the form above. E.g. PLOT2(F,-2,2); acts like PLOT2(F(X),X,-2,2); This is supposed to provide a quicker evaluation of the first arg and for that reason NO checking is done on the result. If the wrong kind of number is returned, the resulting plot will not be meaningful.

```
TRANSLATE:TRUE;
F(X):=(MODEDECLARE(X,FLOAT),EXP(-X*X));
PLOT2(F,-2,2);
PLOT2(F,[-2,-1,0,1,2]);
ARRAY(V,FLOAT,10);
FOR I FROM 0 THRU 10 DO V[I]:FLOAT(I*I);
PLOT2(F,V);
```

GRAPH2*(x-lists,y-lists,optional-args)* plots points specified by the first *x-lists* and *y-lists*. The format for *x-lists* can be one of

i) [*x-pt1,x-pt2,* .. *,x-pti,* .. *,x-ptn*] where *x-pti* evaluates to a number

ii) *arrayname* where *arrayname* is the name of a declared one-dimensional array of floating point numbers

iii) *2d-arrayname* where *2d-arrayname* is the name of a declared two-dimensional array of floating point numbers (i.e. by ARRAY(*2d-arrayname*, FLOAT, *max-row-index,max-col-index*);)

iv) [*x-list1,x-list2, .. ,x-listi, .. ,x-listk*] where *x-listi* can have the form of either i) or ii).

The format of *y-lists* is similar. The format of *optional-args* is the same as for PLOT2.

Note that GRAPH2 performs the same job as the MACSYMA function MULTIGRAPH and that GRAPH2 is thus slightly incompatible with GRAPH FASL.

PARAMPLOT2*(x-exprs,y-exprs,variable,var-range,optional-args)* plots *x-exprs* as the x coordinate against *y-exprs* as the y coordinate.

The format for the first two arguments is the same as that for the first argument to PLOT2. Thus if *x-exprs* is [*x-expr1, x-expr2, .. ,x-expri, .. ,x-exprn*] and *y-exprs* is [*y-expr1,y-expr2, .. ,y-expri, .. ,y-exprk*], then max(n,k) curves will be plotted. They will be (assuming n > k):

x-expr1 vs. *y-expr1*, .. ,*x-exprk* vs. *y-exprk*,
x-expr(k+1) vs. *y-exprk*, .. ,*x-exprn* vs. *y-exprk*

The format for the remaining arguments is the same as for PLOT2.

PARAMPLOT2*(x-funs.y-funs,var-range,optional-args)* efficiently evaluates its first 2 arguments in the same way that the alternative form of PLOT2 works.

Examples

```
TRANSLATE:TRUE;        causes automatic translation
F(X):=(MODEDECLARE(X,FLOAT),COS(X));
G(X):=(MODEDECLARE(X,FLOAT),SIN(X));
PARAMPLOT2(F,G,0,2*%PI); plots F(x) vs G(x) as x goes from 0 to 2*%PI

PARAMPLOT2(SIN(T),COS(T),T,0,2*%PI);
```

plots sin(T) for the x-axis and cos(T) for the y-axis as T takes on CALCOMPNUM (see 18.2) values between 0 and 2*%PI. (If EQUALSCALE is TRUE this draws a circle.)

CALCOMPNUM[20] is the number of points PLOT2 and PARAMPLOT2 plot when given the *low,high* type of variable range. The default if you load up PLOT2,LISP,DSK,SHARE is 20, which is sufficient for trying things out. 100 is a suitable value for a final hard copy.

18.3 What to Type When PLOT2 has Finished Plotting

When a plot is finished the bell on your terminal will be dinged. (In fact, this happens only if WAIT is TRUE and if you're plotting on the terminal.) The plotting function is now waiting for you to type something before it exits and prints the next C-Label on your nice plot. It does nothing with most characters; they are left to be part of the next C-Line. However, the following characters are read and interpreted specially.

space exits the plotting function.

return clears the screen and then exits.

tab causes the previous plot to be replotted.
This is useful if the line is noisy, or, in conjunction with *control-A* if various plotting parameters need to be changed.

linefeed sends out a hardcopy signal.
At present this can only be used to generate hardcopy on the Tektronix hardcopy unit or on the Gould line-printer.

rubout names the plot.
PLOT2 types out "Enter name of plot" you reply *plotname*; this is identical to typing a space instead of a rubout, followed by NAMEPLOT(*plotname*);.

control-A
This is not really read by the plotting function, but enables you to enter a *control-A* break and change various options, before typing tab to have the plot replotted. Make sure the plot has finished before you type this.

PLOTBELL[TRUE] when FALSE inhibits the dinging of the bell.

While your plot is coming out your terminal is in a rather strange state (e.g. not echoing characters). Thus it is OK to type ahead to MACSYMA, but the ONLY way you should interrupt the plot is with *control-^*. E.g. do not use *control-A* until the plot has finished.

18.4 Three-Dimensional Plotting

PLOT3D*(z-exprs,x-var,var-range,y-var,var1-range,optional-args)* makes a 3-dimensional plot of *z-exprs* against *x-var* and *y-var*. The plot consists of curves of *y-exprs* against *x-var* (the x coordinate) with *y-var* (the y coordinate) held fixed. Perspective is used and curves further away from the viewer have those parts of them which are hidden by the closer curves removed.
The format of *y-exprs* is the same as for PLOT2. The context of evaluation is FLOAT(EV(*expri,x-var=value gotten from var-range,y-var=value gotten from var1-range*,NUMER)).

The format for *var-range* and *var1-range* is the same as for PLOT2 except that if *var1-range* is of the form *low,high* then *y-var* will take on CALCOMPNUM1 values.

The format of *optional-args* is the same as for PLOT2 except that additional options **NOT3D, \3D, CONTOUR** are available.

PLOT3D*(z-funs,var-range,var1-range,optional-args)* is analogous to the alternative form for PLOT2. *z-funs* must be a function or list of functions of 2 arguments, which must return a floating point argument when given floating point (integer, if the INTEGER argument is used for either *var-range* or *var1-range*) arguments. The functions must be translated or compiled. If you expect to make several 3D plots this form is recommended.
A simple example is

```
TRANSLATE:TRUE;        causes automatic translation
G(X,Y):=(MODECLARE(X,FLOAT),EXP(-X*X-Y*Y));   defines a funtion G
PLOT3D(G,-2,2,-2,2);  plots it
```

GRAPH3D*(x-lists,y-lists,z-listsoptional-args)* takes 3 arguments (GRAPH2 takes 2) and interprets them as lists of x, y, and z points which it uses to draw lines using the 3d transformations. It can be used to add lines (e.g. axes) to your 3D plot. The hidden line routines are not used.

NOT3D

In this section we describe the option NOT3D. As an example, consider

```
PLOT3D(SIN(X)+A,X,-%PI,%PI,A,[-2,3,4,6],NOT3D)
```

which plots sin(X)+A for X from -%PI to %PI (CALCOMPNUM[20] points) and A taking the values in the list. This is equivalent to:

```
PLOT2([SIN(X)-2,SIN(X)+3,SIN(X)+4,SIN(X)+6],X,-%PI,%PI)
```

but requires less typing.

The additional NOT3D argument to PLOT3D, causes exactly the same points as in the bare PLOT3D to be calculated. Instead of plotting a 3-dimensional representation of the data, the data is plotted in a 2D one. Specifically 1 2D curve of z vs. x for each y value, and so is a convenient way to plot several curves on the same plot.

PERSPECTIVE, REVERSE, VIEWPT, and CENTERPLOT

The following options govern the type of perspective view given.

PERSPECTIVE[TRUE], if FALSE causes a non-perspective view to be taken. This is equivalent to extending the viewing position out to infinity along a line connecting the origin and VIEWPT.

REVERSE[FALSE], if TRUE cause a left-handed coordinate system to be assumed.

VIEWPT and CENTERPLOT determine the perspective view taken. They are defaulted to be unbound. VIEWPT may be set to a list of 3 numbers and gives the point from which the projection should be made. CENTERPLOT may likewise be set to a list of 3 numbers and gives a point on the line of sight. The projection will be made onto a plane perpendicular to a line joining VIEWPT and CENTERPLOT.

If VIEWPT and CENTERPLOT are unbound (the default) then they will be chosen as follows: the extreme values of the coordinates are determined. This gives the two points min: [xmin,ymin,zmin], max:[xmax,ymax,zmax]. CENTERPLOT is chosen as (min+max)/2, and VIEWPT is chosen as max+3*(max-min). The view is then one in which the z axis is vertical, the x axis is increasing towards you to the left and the y axis is increasing towards you to the right.

If CENTERPLOT is FALSE then the old type of perspective view will be given (like setting the x and z components of CENTERPLOT to the corresponding components of VIEWPT).

18.5 Using the XGP from PLOT2

To get plots out on the XGP, simply do PLOTMODE(XGP, ...) where ... is the correct plot mode for your terminal (i.e. GR for Grinnell TV's, T for Tektronix, or D for character display terminals like VT52's). Then you can use linefeed at the end of the display of a plot, or HARDCOPY(); to cause a hardcopy to be submitted to the XGP (it is processed by the Gould spooler first). If MIT-AI[1] is up and the queuing for the XGP is successful, you will receive a message from the XGP spooler when your plot is printed. If MIT-AI is down, your plot will be processed into an XGP scan file, .GLPT.; > SCN, and the Gould spooler will send you a warning message telling you that MIT-AI is down. You must copy the SCN file to MIT-AI yourself when MIT-AI comes up and queue it by doing

```
:XGP ;SCAN AI:dir;* SCN
```

where dir is the directory on MIT-AI to which you copied the files (use MACSYM; if you don't know of any other directory to use).

If you use the NAMEFILE[2] command instead of HARDCOPY or linefeed, you can print the plot file later by doing

```
:GTPL dir;fn1 fn2/a/o[x]
```

where dir, fn1, fn2 are the directory, first file name, and second file name of the plot file stored with NAMEFILE. PLOTLFTMAR[128] and PLOTBOTMAR[320]

1. The computer at the MIT AI Lab

2. NAMEFILE(filespec) simply copies the PLOT2 scratch file to filespec

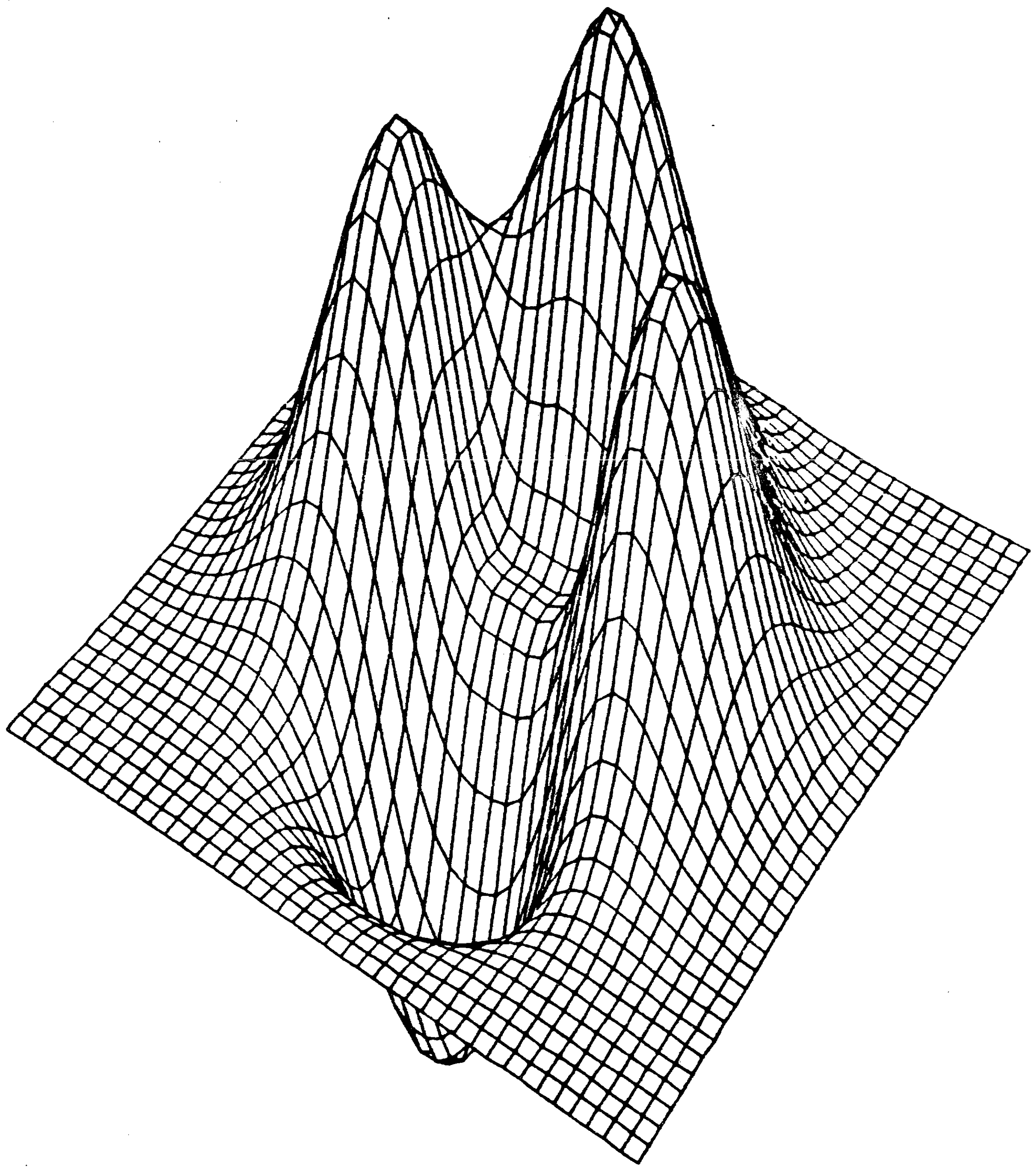
adjust the bottom margin and left margin for the XGP plots. These default to values[3] such that the plots will fit comfortably on an 8 1/2 x 11 page.

There are many other features of PLOT2 (such as three-dimensional and contour plotting) which the user can learn about by reading SHARE;PLOT2 USAGE and SHARE;PLOT2 RECENT. As an example, consider

```
(PLOTMODE(XGP,D),WINDOW:[100,900,0,950])$

(CALCOMPNUM:CALCOMPNUM1:40,VIEWPT:[-30,-20,5])$

PLOT3D((X^3+Y^4-0.2*X)*EXP(-X^2-Y^2)+0.3*EXP(-(X-1.225)^2-Y^2),
                X,-3,3,Y,-3,3);

NAMEFILE(PLOT,TEST,DSK,CFFK);
```

The resulting plot is shown on the next page

3. Whose units are given in increments of 1/200 th of an inch



19 Debugging in MACSYMA

When the user's command lines, especially functions and BLOCK programs, do not do what is expected or generate errors, MACSYMA offers several debugging alternatives:

(1) The user may trace function calls by typing TRACE(*fun1,fun2,...*), where the *funi* are either MACSYMA or user-defined functions. This will cause a printout of the function name and its arguments each time it is entered, and of the function name and the value it returns each time it is exited. A count which is the level of recursion is also printed. Usually, this is all the tracing power the user will need, although MACSYMA offers the full capabilities of the LISP tracing package including conditional and breakpoint tracing. This will not be described here - for information see [Mn1]. MACSYMA uses trace-syntax very similar to that of LISP.

To check which functions are currently under trace, the user may type TRACE(). To remove tracing of functions use UNTRACE(*fun1,fun2,...*). To untrace all previously traced functions type UNTRACE(). Since the TRACE package takes up some of the user's workspace in core, when finished with it the user should type REMTRACE(). It can always be reloaded again if necessary.

(2) The assignment of variables can be traced by setting the variable SETCHECK to a list of variables (which can be subscripted). When a variable on the list is bound (either with : or :: or function argument binding) then a message -- variable "SET TO" value -- will be printed. If the variable SETCHECKBREAK is set to TRUE then a (MACSYMA-BREAK) will be caused each time a variable on the SETCHECK list is bound.

(3) By setting the variable REFCHECK[FALSE] to TRUE, the user will be informed when each of his variables which has a value comes up for evaluation for the first time during the course of a computation. This has a dual purpose. The user will be informed of evaluations he may not have been aware of which are the result of assignments he made long ago. It also gives him a sort of chronological trace of his computations which may be helpful in finding out where an error has occurred.

(4) By setting the variable PREDERROR to TRUE, the user will be informed of predicates of IF-THEN-ELSE statements which failed to evaluate to either TRUE or FALSE.

(5) The user may have variables which he intends not to use purely

symbolically, i.e. they are to have values all the time. By typing DECLARE([*var1,var2,...*],BINDTEST) MACSYMA will give the user an error whenever any of the vari appear in a computation unbound. To remove a BINDTEST declaration, the user may use the function REMOVE. (see 8.1)

(6) When an error occurs in the course of a computation, MACSYMA prints out an error message and terminates the computation. At times the user may find it helpful to investigate the environment at the place of the error. To do so, type DEBUGMODE:TRUE or DEBUGMODE:ALL and repeat the computation. This enters a special debugging mode which will "break" or pause when an error occurs. This mode may be terminated by typing DEBUGMODE:FALSE. When an error occurs in debugging mode, (ERROR-BREAK) is printed. MACSYMA is then waiting for the user to type something. He may type any command line just as if he were at "top level". The command lines will be evaluated in the environment of the error. If the user had done DEBUGMODE:ALL, he may now type BACKTRACE;, and MACSYMA will print out a backtrace, which is a list of the function calls the user is currently in together with the arguments they were called with, ordered from most recent to earliest i.e., when reversed, this list shows a trace beginning from the initial function and ending at the last call entered including only those function calls from which the user still has not exited. To exit from the MACSYMA error-break and return to "top-level", type EXIT;.

The user may also enter the error-break at any point, by typing control-A or by executing the function BREAK (see 10.6). This will simply cause his computation to pause, while he investigates at will. %% refers to the last computed result while in the MACSYMA break. % still refers to the last result computed at top-level. Upon typing EXIT;, the computation will resume. If he wants to quit a computation begun in a control-A break without quitting the top-level suspended computation, the user can type control-X.

During a break one may type TOPLEVEL;. This will cause top-level MACSYMA to be entered recursively. Labels will now be bound as usual. Everything will be identical to the previous top-level state except that the computation which was interrupted is saved. The function TOBREAK() will cause the break which was left by typing TOPLEVEL; to be re-entered. If TOBREAK is given any argument whatsoever, then the break will be exited, which is equivalent to typing TOBREAK() immediately followed by EXIT;.

In the following example, an attempt is made to define a function ROOT which finds an approximate root to an expression using Newton-Raphson iteration.

```
(C1) ROOT(F,V):=BLOCK([VAL,FUN,DER],DER:DIFF(F,V),VAL:0.0,
         TEST,FUN:SUBST(VAL,V,F),IF ABS(FUN)<5.0E-7 THEN
         RETURN(VAL),DER:SUBST(VAL,V,DER),VAL:VAL-FUN/DER,
         GO(TEST))$

(C2) NUMER:TRUE$

(C3) F:SIN(%PI*X)-%PI*(X-1)$

(C4) ROOT(F,X);

3.3116898E+8 ARG TOO BIG FOR ACCURACY - SIN

(C5) DEBUG:TRUE$

(C6) DEBUGMODE:TRUE$

(C7) ''C4;
F has value
V has value
VAL has value
FUN has value
DER has value

3.3116898E+8 ARG TOO BIG FOR ACCURACY - SIN

_(ERROR-BREAK)

_VAL;
1.0541436E+8

_DER;
- 2.98023224E-8

_TRACE(SUBST)$

_''C4;
```

(The numerical value of %Pi is present below due to NUMER being set to TRUE above.)

```
1 ENTER SUBST [0.0, X, SIN(3.1415927 X) - 3.1415927 (X - 1)]
1 EXIT SUBST: 3.1415927
1 ENTER SUBST [0.0, X, 3.1415927 COS(3.1415927 X) - 3.1415927]
1 EXIT SUBST: - 2.98023224E-8
1 ENTER SUBST [1.0541436E+8, X, SIN(3.1415927 X) - 3.1415927 (X - 1)]

3.3116898E+8 ARG TOO BIG FOR ACCURACY - SIN

try again
```

(This message is due to an error-break occurring within another error break.)

```
_EXIT;
EXITED FROM THE BREAK

(C8) <$>
_''C4
MYFROOT<$><$>
```

(The user uses the MYF command of the MACSYMA editor to insert the definition of ROOT into the edit buffer. The editor is then used to insert an IF statement to test for DER being close to 0. The actual editing work is not shown.)

```
(C8) ROOT(F,V):=BLOCK([VAL,FUN,DER],DER:DIFF(F,V,1),VAL:0.0,
        TEST,FUN:SUBST(VAL,V,F),IF ABS(FUN)<5.0E-7 THEN
         RETURN(VAL),DER:SUBST(VAL,V,DER),IF ABS(DER)<5.E-8
         THEN ERROR("DERIVATIVE IS ZERO"),VAL:VAL-FUN/DER,
        GO(TEST))$

(C9) UNTRACE();
(D9)                                [SUBST]

(C10) DEBUG:FALSE$

(C11) ''C4;
DERIVATIVE IS ZERO
QUIT      (This is due to the ERROR function being executed.)
```

Appendix I - Multics MACSYMA

MACSYMA is also implemented on the Multics operating system which runs on Honeywell Series 60 machines (the 68/80 in particular). It is the same MACSYMA as on the ITS system; the only differences that exist are due to the interaction of the LISP in which the MACSYMA is imbedded with a different operating system. Since this manual is mainly written for using a MACSYMA on ITS, one has to be aware of these differences if one is to use the manual while using a MACSYMA on Multics.

Procedures for logging in and out will not be given here. It is assumed that if one has access to a Multics one will either know the procedures for logging in or will be able to obtain adequate documentation in order to learn how to do so.

1) To use a MACSYMA on a Multics one simply invokes the "macsyma" command. However since this command may not be in one's default search rules, it may be necessary to link to the command or to refer to it by its full pathname. For example, on the Multics at MIT, the full pathname of the command is ">udd>ap>library>macsyma".

When the command is invoked, the MACSYMA system is loaded and prompts the user in the standard manner.

2) All input must end with a newline (return) after the semi-colon or dollar-sign is typed.

3) The terminator character for the editor is "&" instead of altmode.

4) Control characters are not entered in the manner described in the rest of this manual but are instead entered by the conventions of the Multics implementation of LISP. This is done by using the "attention" or "break" button on the console followed by the letter of the alphabet for the control character you want, followed by a newline character. i.e. when you hit the attention key, the system will type "CTRL/", and you should respond by typing the appropriate character. If you want a control-Z, for example, you would type the letter "Z" followed by a newline. The various control characters have the same meaning as on ITS MACSYMA. The only one that is different is control-Z which rather than "returning" to a monitor level instead calls a new invocation of the monitor at a higher level i.e. it is the same thing as a normal Multics quit signal. A "start" command will start the macsyma moving again.

5) There are two different ways of referring to files in the Multics hierarchy. MACSYMA commands that take file names as arguments can be given the Multics file name enclosed in double quotes as a single argument. An alternative form of reference is one that maps the ITS way of referring to files with four arguments into a reasonable Multics filename. Thus:

```
(cl) batch("random.demo");
(cl) batch(">udd>Project>JRNurd>random.demo");
(cl) batch(random,demo);
(cl) batch(random,demo,dsk,">udd>Project>JRNurd");
```

would all refer to the same file if the current working directory were ">udd>Project>JRNurd".

(6) To use the Multics plotting package, set the variable MULTGRAPH to TRUE.

Appendix II - A MACSYMA Grammar Primer

The concepts MACSYMA deals with are primarily mathematical, and its grammar has been designed to reflect this emphasis by making the representation of expressions as natural as possible. All the usual mathematical operators are predefined, and MACSYMA commands are expressed exclusively in functional notation. In addition, MACSYMA provides a flexible syntax extension capability for users who require additional operators.

For the purposes of this appendix, the internal representation of a MACSYMA expression can best be described in terms of function calls. All expressions are represented as appropriately nested function calls; all actions are the result of function evaluations. This primer is intended to introduce the reader to MACSYMA's syntax and and syntax extension capabilities and to help him utilize these features most effectively in syntactically expressing the semantics of the job he wants done.

It is assumed here that the reader is already familiar with the rules of formation for lexemes, i.e. integers, real numbers, atoms, and strings. At present these rules are somewhat confusing and should be mastered before proceeding (see 2.3) .

The Parser

When a user types a string of characters to MACSYMA, it is first broken up into lexemes by a lexical-scanning program. For example the input "IF X>0 THEN X ELSE -X" becomes (IF X > 0 THEN X ELSE - X). This sequence of lexemes is then passed to an "extended operator precedence parser with types" and converted into MACSYMA's internal representation, i.e. suitably nested function calls. In this case, the result would be as follows:

```
"IF"(">"(X,0),X,"-"(X))
```

Such functional notation is always legal MACSYMA syntax, and it will be used throughout this appendix, as above, to represent the meaning of various syntactic constructions.

Every lexeme in MACSYMA is either an "operator", a "delimiter", or an "operand". The operators and delimiters taken together are sometimes referred to as the "keywords" of MACSYMA. With each operator is associated a specific parsing function which prescribes how its arguments are to be selected from the input. Thus, for example, the arguments of an infix operator like ">" are to be found, one to the left and one to the right of the operator. The resulting internal representation is a function call of the operator on its arguments. For example, in the sentence above, ">" has as arguments "X" and "0", and the internal representation of "X>0" is ">"(X,0).

A delimiter is a reserved lexeme used by certain operators to mark their arguments. A delimiter may not be used as an operand but has no special parsing function associated with it. In the sentence above, "THEN" and "ELSE" are delimiters, used by the operator "IF" to mark its second and third arguments.

An operand is a lexeme with no special parsing properties. All lexemes, unless otherwise specified, are operands. Operands serve as the arguments of operators to form function calls which may then in term serve as arguments to other operators. In the example sentence, "X" and "0" are operands; ">"(X,0), X, and "-"(X) also appear as operands to the operator "IF". Note that any operator may be used as an operand by enclosing it in double quotes, e.g. INFIX("&").

The process of parsing is one of recognizing the operators, delimiters, and operands in a sequence of lexemes and correctly identifying the arguments to the operators in order to construct the function nesting implicit in the sequence. There are only seven types of operators in MACSYMA, most of which should be familiar to mathematicians. These seven types are described below.

PREFIX OPERATORS

A PREFIX operator is one which signifies a function of one argument, which argument immediately follows an occurrence of the operator. Some examples are:

NOT TRUE	means	"NOT" (TRUE)
- A	means	"-" (A).

This resembles the usual functional notation except that the parentheses surrounding the argument are here unnecessary. Of course any expression may be embedded to arbitrary depth within another as in the following.

NOT NOT TRUE	means	"NOT" ("NOT" (TRUE)) = TRUE

POSTFIX OPERATORS

POSTFIX operators like the PREFIX variety denote functions of a single argument, but in this case the argument immediately precedes an occurrence of the operator in the input string.

3 !	means	"!" (3)
A !!	means	"!!" (A)

INFIX OPERATORS

INFIX operators are used to denote functions of two arguments, one given before the operator and one after. Again some examples:

A^2	means	"^" (A,2)
3^3 > 10	means	">" ("^" (3,3), 10)

A variation of the INFIX operator is the NARY.

NARY OPERATORS

An NARY operator is used to denote a function of any number of arguments, each of which is separated by an occurrence of the operator.

A*B*C	means	"*" (A,B,C)
A>B AND TRUE AND C<D	means	"AND"(">"(A,B),TRUE,"<"(C,D))

are all examples of NARY operators.

SPECIAL OPERATORS

NARY operators are useful for functions whose arguments are in one way or other homogeneous. For other functions of many arguments, special forms are required. A familiar example is the conditional statement.

IF A>2 THEN A-1 ELSE A means "IF"(">"(A,2),"-"(A,1),A)

Here the operator "IF" denotes a function of three arguments; the first is found immediately after the "IF"; the others are introduced arguments signalled by the occurrence of the delimiters associated with "IF", namely "THEN" and "ELSE". Once again, delimiters are not in themselves operators but are merely used by operators to mark introduced arguments. Using a delimiter out of the context of the operators for which it was defined will result in a syntax error.

Another example of an operator with introduced arguments is the iteration statement. Here the delimiters precede their defining operator.

```
FOR I FROM 2 STEP 3 THRU 10 UNLESS A>10 DO PRINT(A)
```

In this example each of the indicated segments contributes an argument to the "DO" function. It happens that in MACSYMA any of these arguments may be omitted, or if given they can be given in any order. Thus the following are all legal, though not necessarily equivalent, sentences.

```
THRU 10 DO S:S^2
FOR I FROM 2 THRU 5 DO PRINT (A[I])
THRU 5 UNLESS A>1000 DO A:A!
```

When arguments are omitted as above or like the "ELSE" argument of the "IF" operator, the corresponding "holes" are filled with predetermined default values. These are listed in the dictionary below. Also the "DO" statement has some additional flexibility. The "STEP" argument can be replaced by a "NEXT" expression which denotes what the iteration variable is to be set to on each pass thru the loop rather than the value by which it is to be incremented; and there are permitted arbitrarily many "WHILE" or "UNLESS" clauses as termination conditions. Some examples with answers:

```
THRU 3 DO PRINT (A)
A
A
A

                         DONE
```

```
FOR I STEP 2 THRU 3 DO PRINT (I)
1
3
                                          DONE
FOR I NEXT I+2 UNLESS I>3 DO PRINT (I)
1
3
                                          DONE
```

NOFIX OPERATORS

NOFIX operators are used to denote functions of no arguments. The mere presence of such an operator in a sentence will cause the corresponding function to be evaluated.

QUIT	means	"QUIT" ()
LOGOUT	means	"LOGOUT" ()

Care should be taken in using these operators, however, since they tend to look much like variables but semantically are very different.

MATCHFIX OPERATORS

MATCHFIX operators are used to denote functions of any number of arguments which are passed to the function as a list. The arguments occur between the main operator and its "matching" delimiter. For example:

[) A, B, C (]	means	[)" (A,B,C)

A legal sentence in MACSYMA is a correct sequence of operators from these seven categories and their operands. By "correct" here we mean that due respect has been shown the type of the operator, e.g. not giving two arguments to a PREFIX operator, and that the two sole grammatical rules in the language have not been violated. These rules concern the "binding powers" of MACSYMA's keywords and the "parts of speech" legal in each argument slot.

BINDING POWERS

The binding powers of keywords are used to resolve ambiguities of argument association such as that in the following example.

- 233 !

Is this "-"("!"(233)) or "!"("-"(233)); or, in other words, which operator gets the operand "233" and which, the resulting function call? It is a convention in MACSYMA that the keyword with the higher binding power gets the disputed argument and the other is then applied to the result. In this case, the "left binding power" of "!" (160) is greater than the "right binding power" of "-" (100); and so "233" is associated with "!" and the resulting function call becomes the argument for "-".

Each keyword must possess a left and a right binding power to resolve such conflicts. Some of these numbers are superfluous, such as the left binding power of a prefix operator; and in such cases the binding power is arbitrarily taken to be 200. Currently the range of binding powers is 0 to 200.

PARTS OF SPEECH

From natural language the notion of "part of speech" should be familiar. MACSYMA also has parts of speech and constraints on which parts of speech are legal in various contexts. Whereas binding powers are necessary to resolve ambiguities af argument assignment, parts of speech exist solely to detect unintentional syntax errors.

> *Every operator possesses a part of speech constraint on each of its argument slots. Any operand filling a slot must satisfy the associated constraint, or a syntax error will result.*

There are only three parts of speech predefined in MACSYMA, namely EXPR, CLAUSE, and ANY. An EXPR is essentially a mathematical expression; a CLAUSE, a mathematical predicate or a command. Thus "A+B" is an EXPR but not a CLAUSE; and "A+B>2" is a CLAUSE but not an EXPR. The part of speech ANY is used to signify objects which may be either CLAUSEs or EXPRs, such as "F(X)". The parts of speech required by MACSYMA's predefined operators are listed in the dictionary below.

SYNTAX EXTENSION

While MACSYMA's syntax should be adequate for most ordinary applications, it is possible to define new operators or eliminate predefined ones that get in the user's way. The extension mechanism is rather straightforward and should be evident from the examples below.

```
(C1) PREFIX("DDX")$

(C2) DDX Y$              means              "DDX"(Y)

(C3) INFIX("<-")$

(C4) A<-DDX Y$           means              "<-"(A,"DDX"(Y))
```

An appreciation of the concepts and rules introduced in this primer should be all that is necessary to use the syntax extension capabilities successfully. The only form of syntax extension available is the definition of new operators. For each of the types of operator except SPECIAL, there is a corresponding creation function that will give the lexeme specified the corresponding parsing properties. Thus "PREFIX("DDX")" will make "DDX" a prefix operator just like "-" or "NOT". Of course, certain extension functions require additional information such as the matching keyword for a matchfix operator. In addition, binding powers and parts of speech must be specified for all keywords defined. This is done by passing additional arguments to the extension functions. If a user does not specify these additional parameters, MACSYMA will assign default values. The six extension functions with binding powers and parts of speech defaults (enclosed in brackets) are summarized below.

PREFIX*(operator, rbp[180], rpos[ANY], pos[ANY])*

POSTFIX*(operator, lbp[180], lpos[ANY], pos[ANY])*

INFIX*(operator, lbp[180], rbp[180], lpos[ANY], rpos[ANY], pos[ANY])*

NARY*(operator, bp[180], argpos[ANY], pos[ANY])*

NOFIX|**operator***(pos[ANY]|)*

MATCHFIX*(operator, match, argpos[ANY], pos[ANY])*

The defaults have been provided so that a user who does not wish to concern himself with parts of speech or binding powers may simply omit those arguments to the extension functions. Thus the following are all equivalent.

```
PREFIX("DDX",180,ANY,ANY)$
PREFIX("DDX",180)$
PREFIX("DDX")$
```

It is also possible to remove the syntax properties of an operator by using the functions REMOVE or KILL. Specifically, "REMOVE("DDX",OP)" or "KILL("DDX")" will return "DDX" to operand status; but in the second case all the other properties of "DDX" will also be removed.

The following is an example of syntax extension to permit the use of set notation.

```
(C1) MATCHFIX("{","}")$

(C2) INFIX("|")$

(C3) {X|X>0};
(D3)                         {X|X>0}
(C4) {X|X<2};
(D4)                         {X|X<2}

(C5) INFIX(".U.")$

(C6) INFIX(".I.")$
```

Now assuming the functions ".U." and ".I." have been appropriately defined as union and intersection, the following interaction can occur.

```
(C7) D3.U.D4;
(D7)                              UNIVERSE

(C8) D3.I.D4;
(D8)                              {X|X>0 AND X<2}

(C9) {1,2,3}$

(C10) {3,4,5}$

(C11) D9.U.D9.I.D10;
(D11)                             {3}
```

Line C11 was parsed as ((D9.U.D9).I.D10) whereas the usual convention would call for the alternate parsing (D9.U.(D9.I.D10)), which would have resulted in {1,2,3} as value. The problem here is that the default binding powers for ".U." and ".I." are identical; so the parser associates them in left to right order. To obtain the usual parsing, the syntax definitions in lines C5 and C6 must give ".I." a higher left binding power than ".U."'s right binding power as in the following.

```
(C12) INFIX(".U.",100,100)$

(C13) INFIX(".I.",120,120)$

(C14) D9.U.D9.I.D10;
(D14)                             {1,2,3}

(C15) REMOVE(".U.",OPERATOR)$

(C16) D9.U.D10;
Syntax error
D9 .U. ***$*** D10
Please rephrase or edit
```

A DICTIONARY OF MACSYMA'S KEYWORDS

The following is a list of all the keywords in MACSYMA, categorized with respect to type. With each keyword is given the information necessary to recreate its syntactic behavior. The abbreviations "lbp", "rbp", "lpos", "rpos", "bp", and "pos" stand for "left binding power", "right binding power", left part of speech", "right part of speech", "binding power", and "part of speech". The reader should consult the text of this primer to understand the significance of these parameters. It should also be noted that some lexemes, like "-", have two syntactic types.

PREFIX OPERATORS

	rbp	rpos	pos
'	190	ANY	ANY
''	190	ANY	---
+	100	EXPR	EXPR
-	100	EXPR	EXPR
NOT	70	CLAUSE	CLAUSE

POSTFIX OPERATORS

	lbp	lpos	pos
!	160	EXPR	EXPR
!!	160	EXPR	ANY

INFIX OPERATORS

	lbp	lpos	rbp	rpos	pos
#	80	ANY	80	ANY	CLAUSE
**	140	EXPR	139	EXPR	EXPR
.	130	ANY	129	ANY	ANY
:	180	ANY	20	ANY	ANY
::	180	ANY	20	ANY	ANY
:=	180	ANY	20	ANY	ANY
<	80	EXPR	80	EXPR	CLAUSE
<=	80	EXPR	80	EXPR	CLAUSE
=	80	EXPR	80	EXPR	CLAUSE
>	80	EXPR	80	EXPR	CLAUSE
>=	80	EXPR	80	EXPR	CLAUSE
^	140	EXPR	139	EXPR	EXPR
^^	135	ANY	134	ANY	ANY

NARY OPERATORS

	bp	argpos	pos
*	120	EXPR	EXPR
+	100	EXPR	EXPR
,	10	ANY	ANY
-	100	EXPR	EXPR
/	120	EXPR	EXPR
AND	60	CLAUSE	CLAUSE
OR	50	CLAUSE	CLAUSE

SPECIAL OPERATORS

	lbp	lpos	rbp	rpos	pos
(	200	ANY	MATCHFIX for right arg		ANY
[	200	ANY	MATCHFIX for right arg		ANY
DO	200		25	ANY	ANY
	FOR		optional		
	FROM		optional		
	IN		optional		
	STEP or NEXT			optional	
	THRU	optional			
	WHILE		any number of occurrences		
	UNLESS		any number of occurrences		
IF	200		45	CLAUSE	ANY
	THEN				
	ELSE		optional		

MATCHFIX OPERATORS

	match	argpos	pos
(	)	ANY	ANY
[	]	ANY	ANY

DELIMITERS

	op	lbp	rbp	rpos
ELSE	IF	5	25	ANY
FOR	DO	30	200	ANY
FROM	DO	30	95	EXPR
IN	DO	30	95	ANY
NEXT	DO	30	45	ANY
STEP	DO	30	95	EXPR
THEN	IF	5	25	ANY
THRU	DO	30	95	EXPR
UNLESS	DO	30	45	CLAUSE
WHILE	DO	30	45	CLAUSE

Appendix III - Illustrative Examples

This appendix shows a complete interaction with MACSYMA. Three examples are given. First an ordinary second-order differential equation is solved by two methods. (1) by using pattern matching and solving the characteristic equation and (2) by using Laplace transforms. The second example shows the conversion of an expression from one coordinate system to another and the third example shows a truncated power series solution of a differential equation.

Example 1

```
(C1) BATCH(SOLDER,DEMO,DSK,DEMO);

(C2) /* THE FOLLOWING ROUTINE RETURNS THE HOMOG.-PART SOLN.
TO 2ND ORDER LINEAR DIFF'L EQNS. WITH CONST. COEFFS. */

MATCHDECLARE([B,C],RATNUMP)$

MATCOM FASL DSK MACSYM BEING LOADED
LOADING DONE

(C3) MATCHDECLARE(F,FREEOF(U))$

(C4) ALIAS(D,DIFF)$

(C5) DEFMATCH(SOLDE,'D(U,X,2) + B*'D(U,X) + C*U = F,U,X)$
```

```
(C6) SOLDER(EQN,U,X) :=
   BLOCK([B,C,F,DISC,R1,R2,ALPHA,BETA],
         IF SOLDE(EQN,U,X) = FALSE THEN RETURN(FALSE),
         DISC: B^2 - 4*C, ALPHA: -B/2,
         IF DISC=0 THEN RETURN(%E^(ALPHA*X)*(A1+A2*X)),
         BETA: SQRT(DISC)/2,
         IF DISC > 0
            THEN (R1: ALPHA + BETA, R2: ALPHA - BETA,
                  RETURN(A1*%E^(R1*X) + A2*%E^(R2*X)))
            ELSE (BETA: SQRT(-1)*BETA,
                  RETURN(%E^(ALPHA*X) * (A1*COS(BETA*X)
                               + A2*SIN(BETA*X)))))$

(C7) /* AN EXAMPLE - THE METHOD OF UNDETERMINED COEFFS. FOR
       OBTAINING THE PARTICULAR SOLN. AS WELL */

    DE: 'D(Y,X,2) - 'D(Y,X) - 6*Y = SIN(X);

                          2
                         D Y   DY
(D7)                     --- - -- - 6 Y = SIN(X)
                           2   DX
                         DX

(C8) YH(X) := ''(SOLDER(%,Y,X));

                                   - 2 X        3 X
(D8)               YH(X) := A2 %E       + A1 %E

(C9) YP(X) := B1*SIN(X) + B2*COS(X)$

(C10) YG(X) := YH(X) + YP(X)$

(C11) PLUGIN: EV(DE,DIFF,EXPAND,Y=YP(X));

(D11) B2 SIN(X) - 7 B1 SIN(X) - 7 B2 COS(X)

                   - B1 COS(X) = SIN(X)

(C12) EQN1: COEFF(PLUGIN,SIN(X));

(D12)                    B2 - 7 B1 = 1
```

```
(C13) EQN2: COEFF(PLUGIN,COS(X));

(D13)                          - 7 B2 - B1 = 0

(C14) GLOBALSOLVE: TRUE$

(C15) SOLN: LINSOLVE([EQN1,EQN2],[B1,B2]);
SOLUTION

                                     7
(E15)                          B1 : - --
                                     50


                                    1
(E16)                            B2 : --
                                    50

(D16)                          [E15, E16]

(C17) YG(X);

             7 SIN(X)   COS(X)        3 X          - 2 X
(D17)      - -------- + ------ + A1 %E    + A2 %E
                50        50

(C18) /* PLUGGING IN INITIAL CONDITIONS OF Y(0)=1
                AND Y'(0)=0 */

EQN1: YG(0) = 1;
                                    1
(D18)                        A2 + A1 + -- = 1
                                    50

(C19) DIFF(YG(X),X);

           SIN(X)   7 COS(X)          3 X            - 2 X
(D19)    - ------ - -------- + 3 A1 %E    - 2 A2 %E
             50        50

(C20) EQN2: EV(%,X=0) = 0;
                                           7
(D20)                     - 2 A2 + 3 A1 - -- = 0
                                           50
```

```
(C21) SOLN: LINSOLVE([EQN1,EQN2],[A1,A2]);
SOLUTION

                                        21
(E21)                             A1 : --
                                        50

                                        14
(E22)                             A2 : --
                                        25
(D22)                           [E21, E22]

(C23) YG(X);
                                               3 X           - 2 X
              7 SIN(X)    COS(X)    21 %E           14 %E
(D23)       - -------- + ------ + -------- + ----------
                 50         50         50            25

(C24) /* RESETTING OF OPTIONS */

      GLOBALSOLVE: FALSE$

(D25)                           BATCH DONE




(C26) "SOLUTION BY LAPLACE TRANSFORMS"$

(C27) SUBST(Y(X),Y,DE);

              2
             D          D
(D27)        --- Y(X) - -- Y(X) - 6 Y(X) = SIN(X)
               2        DX
             DX

(C28) [ATVALUE(Y(X),X=0,1), ATVALUE('DIFF(Y(X),X),X=0,0)];

(D28)                     [1, 0]
```

```
(C29) LAPLACE(D29,X,S);

LAPLAC FASL DSK MACSYM being loaded
loading done


        2
(D29) S   LAPLACE(Y(X), X, S) - S LAPLACE(Y(X), X, S)


                                                           1
                   - 6 LAPLACE(Y(X), X, S) - S + 1 = ------
                                                          2
                                                         S  + 1

(C30) LINSOLVE([%],['LAPLACE(Y(X),X,S)]);
Solution

                                          3    2
                                         S  - S  + S
(E30)     LAPLACE(Y(X), X, S) = ----------------------
                                      4    3      2
                                     S  - S  - 5 S  - S - 6

(D30)                                  [E30]

(C31) ILT(E30,S,X);
                                                      3 X          -2X
                          7 SIN(X)    COS(X)    21 %E         14 %E
(D31)    Y(X) =  - -------- + ------ + -------- + --------
                             50         50        50          25
```

Example 2

```
(C1) BATCH(C2CYL,DEMO,DSK,DEMO);

(C2) /* CONVERSION OF THE LAPLACIAN FROM CARTESIAN COORDS. TO
   CYLINDRICAL COORDS. */

/* CAUSE DERIVATIVES TO DISPLAY WITH SUBSCRIPTS */

DERIVABBREV:TRUE$

(C3) /* ORDER X,Y, AND Z SO THEY WILL BE GROUPED NICELY */

ORDERLESS(Z,Y,X)$

(C4) /* U(X,Y,Z) BECOMES U(R,T,Z) IN CYLINDRICAL COORDINATES
                R STANDS FOR RHO AND T FOR THETA */

DEPENDS(U,[R,T,Z])$

(C5) /* INPUT THE TRANSFORMATION RULES FROM THE
   CARTESIAN SYSTEM TO THE CYLINDRICAL SYSTEM */

GRADEF(R,X,X/R)$

(C6) GRADEF(R,Y,Y/R)$

(C7) GRADEF(T,X,-Y/R^2)$

(C8) GRADEF(T,Y,X/R^2)$

(C9) /* SET EXPOP TO CAUSE PARENTHESIZED EXPRESSIONS
TO BE EXPANDED AUTOMATICALLY */

EXPOP:1$
```

```
(C10) /* NOW JUST INPUT THE LAPLACIAN IN CART. COORDS.,
   AND LET THE CHAIN RULE DO ITS THING */

DIFF(U,X,2)+DIFF(U,Y,2)+DIFF(U,Z,2);

         2         2         2         2               2       2
        X  U      Y  U      X  U      Y  U     2 U    X  U    Y  U
            T T       T T       R R       R R     R       R       R
(D10)  ------- + ------- + ------- + ------- + ---- - ----- - ----- + U
          4         4         2         2       R      3       3      Z Z
         R         R         R         R              R       R

(C11) SUBST(R^2-X^2,Y^2,%);

                                U                U
                                 T T              R
(D11)                           ---- + U     +  -- + U
                                  2     R R      R     Z Z
                                 R

(D12)                                 BATCH DONE
```

Example 3

The following differential equation:

T^4*B(T)^3*DIFF(B(T),T,2)+(1-K*T^2)*B(T)^4-T^4=0
is known to have a solution of the form:
B(T)=T+A3*T^3+A5*T^5+...+A11*T^11+...
valid for small T. The problem is to find the coefficients A3 through A11 as functions of K. We use RATWEIGHT and RATWTLVL to truncate on powers of T above 14. (This problem originated in "Bessel Functions for Large Arguments" by Goldstein and Thaler, in Math. Tables and Other Aids to Computation, (now called Mathematics of Computation) XII, no. 61, p.18, January 1958.)

```
(C3) EQ:T^4*B(T)^3*DIFF(B(T),T,2)+(1-K*T^2)*B(T)^4-T^4;

                             2
                  4  3      D                    2   4       4
(D3)              T  B (T) (--- B(T)) + (1 - K T ) B (T) - T
                              2
                            DT

C4) TRIAL:T+SUM(A[2*I+1]*T^(2*I+1),I,1,5);

                     11        9        7        5        3
(D4)              A   T   + A  T  + A  T  + A  T  + A  T  + T
                   11        9        7        5        3

(C5) POWERDISP:TRUE$

(C6) RATWEIGHT(T,1)$

(C7) RATWTLVL:14$
```

```
(C8) EQ,B(T)=TRIAL,DIFF;

          4    4                  3          5          7            9
(D8)   - T  + T  (6 A  T + 20 A  T  + 42 A  T  + 72 A  T  + 110 A   T )
                     3         5          7          9           11

                         3       5       7       9        11 3
                (T + A  T  + A  T  + A  T  + A  T  + A   T  )
                      3       5       7       9       11

                       2           3       5       7       9        11 4
             + (1 - K T ) (T + A  T  + A  T  + A  T  + A  T  + A   T  )
                                3       5       7       9       11

a
(C9) EXPANDEDEQ:RAT(%,T);
                      6               2                      8
(D9)/R/  (4 A   - K) T  + (6 A   + 6 A   + 4 A   - 4 A   K) T
             3                3       3       5       3

        2       3                                      2              10
 + (18 A   + 4 A   + (20 + 12 A ) A   + 4 A   + ( - 6 A   - 4 A ) K) T
        3       3              3   5       7           3       5

        3     4                  2             2
 + (18 A   + A   + (78 A   + 12 A   ) A   + 6 A   + (42 + 12 A ) A   + 4 A
        3     3         3        3     5       5              3   7       9

              3                         12
     + ( - 4 A   - 12 A  A  - 4 A ) K) T
              3        3  5      7

           4         2       3                        2
     + (6 A   + (96 A   + 4 A   ) A   + (60 + 12 A ) A
           3         3       3     5              3   5

                     2
     + (144 A  + 12 A   + 12 A ) A   + (72 + 12 A ) A   + 4 A
             3       3        5   7              3   9       11

            4        2         2                         14
     + ( - A   - 12 A  A  - 6 A   - 12 A  A  - 4 A ) K) T
            3        3  5      5         3  7      9
    (C10) COEFF(EXPANDEDEQ,T,6);
    (D10)/R/                      4 A   - K
                                     3
```

```
                                            3
(C11) ANS3:SOLVE(%,A[3]);
Solution


                                              K
(E11)                                     A = -
                                           3  4

(D11)                                     [E11]

(C12) COEFF(EXPANDEDEQ,T,8);
                                         2
(D12)/R/                6 A  + 6 A   + 4 A  - 4 A  K
                           3      3       5      3

(C13) %,ANS3;
                                                     2
                                 - 32 A  - 12 K + 5 K
                                       5
(D13)/R/                       = ---------------------
                                           8

(C14) SOLVE(%,A[5]);
Solution

                                                   2
                                       - 12 K + 5 K
(E14)                            A  =  -------------
                                  5         32

(D14)                                    [E14]

(C15) /* ETC*/

FOR I:3 THRU 11 STEP 2 DO
 COEFFICIENT[I]:COEFF(EXPANDEDEQ,T,I+3)$

(C16) FOR I:3 THRU 11 STEP 2 DO
       (SOL[I]:ANS:SOLVE(COEFFICIENT[I],A[I]),
        FOR J:I+2 STEP 2 THRU 11 DO
        COEFFICIENT[J]:EV(COEFFICIENT[J],ANS))$

(C21) RATEXPAND:TRUE$
```

```
(C22) FOR I:3 THRU 11 STEP 2 DO PRINT(RATSIMP(EV(SOL[I])))$

       K
[A   = -]
  3    4

                    2
        3 K    5 K
[A   =  - --- + ----]
  5      8      32

                    2         3
      15 K    37 K      15 K
[A   = ---- - ----- + -----]
  7     8       32       128

                           2           3          4
        315 K   1821 K     611 K     195 K
[A   =  - ----- + ------- - ------ + ------]
  9        16       128       256      2048

                          2           3           4          5
      2835 K   2223 K    29811 K    4199 K    663 K
[A    = ------ - ------- + -------- - ------- + ------]
  11     8        8         512        1024       8192
```

Appendix IV - Glossary For The Programming Novice

algorithm - a method, specified with sufficient precision to be programmed for a computer, to resolve any one of a well-defined class of problems.

arguments - the expressions which are the values of the formal parameters when a function is called.

assignment - the process of associating a value with a variable.

atomic - (in the sense of high level programming languages) cannot be broken down into smaller parts, e.g. a number, a string, or a name.

binding - the process of assigning values to the formal parameters in a function definition or to the local parameters in a block in such a way that, upon exit from the function or block, the previous values of the parameters are restored.

bound variable - a variable which has been assigned a value (see binding).

break point - a point at which a computation is temporarily suspended and control returned to the console, permitting the user to explore the state of the computation.

bug - an error in a program caused by improper coding which may be due to unanticipated types of arguments being given to the program, faulty logic, etc.

command line - the input line typed to MACSYMA, terminated by ; or $.

constant - any number or atomic symbol whose value does not vary, or an expression made up only of such quantities.

CRE form - canonical rational expression form. This is one of the several internal representations of MACSYMA expressions (see 4.1). It is especially suitable for rational expressions (polynomials or ratios of polynomials). It is also contagious in that whenever any expression is added to or multiplied by a CRE form the result will be in CRE form.

DDT - originally a program used for debugging of other programs but modified in ITS to include the functions of a monitor.

default value - the initial value of a variable before any assignments to it.

evaluation - the process of replacing variables and function calls in an expression by their values.

expansion - the transformation of a product of sums into a sum of products by applying the distributive laws.

expression - a syntactically legal sequence of characters composed of constants, variables, functions, and operators.

flag - a variable whose value is usually either TRUE or FALSE, e.g. NUMER.

formal parameters - the atomic variables appearing in the function header (the left-hand side of a function definition).

hashing - a method of storing sparse vectors and arrays through the use of a function, which for given arguments produces a number in a range of possible values.

indicator - the name of a property, e.g. GRADEF.

internal representation - the representation of MACSYMA expressions in LISP.

ITS - the time sharing system used on the PDP-10 at Project MAC.

LISP - a list processing programming language used extensively in non-numerical applications. The LISP in which MACSYMA resides is called MACLISP.

local parameters - atomic variables which are bound within a function or block.

property - a piece of information known about or associated with a variable. Some specific properties are used in MACSYMA for simplification and evaluation, e.g. the GRADEF property for SIN.

property list - for a given variable, the set of all properties associated with the variable.

quoted string - a sequence of characters enclosed in quotation marks, used as a comment, message, etc.

rational expression - a polynomial or the ratio of two polynomials. Sometimes used as a synonym for CRE form.

scalar - an expression which is not (or is assumed not to be) a list or matrix.

semantics - rules for determining the meaning of any legal (syntactically correct) sentence in a language.

simplification - the process of reducing the "complexity" of an expression (relative to some criterion or measure) by applying known (or assumed) relations in the form of rules which transform the original expression into an "equivalent" one.

string - a sequence of characters consisting of digits, letters, special characters ($,%,#) or break characters (space,tab).

subscripted function - a type of array each of whose elements is a function expression.

subscripted variable - a variable, e.g. A[0], in subscripted form.

switch - a variable which can take on only a small number of values (usually just two). It is used to determine which branch of a condition to follow.

syntax - rules for determining whether a sequence of characters is a legal sentence in that language. If not, then a parsing error results. In MACSYMA, the rules are implemented as a parsing procedure which converts an input string into MACSYMA's internal representation.

terminator - a character which signals the end of a sequence of characters. In a MACSYMA command line the terminators are semi-colon and dollar sign.

variable - an atomic symbol. A variable that evaluates to a value is an assigned variable. If no value has been assigned to a variable, the variable itself is returned as the result of evaluation.

Appendix V - Bibliography and References

[A1] A.C.M. Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation, Los Angeles, Calif., March, 1971. +

[A2] A.C.M. Communications - August 1971 Vol. 14 No. 8 +

[A3] A.C.M. Journal - October 1971 Vol. 18 No. 4 +

[Av] Avgoustis, Yannis. "Symbolic Laplace Transforms of Special Functions." Proceedings of the MACSYMA Users' Conference, NASA, Berkeley, CA, July 1977.

[Ba1] Barton,D.A. and R. Zippel "A Polynomial Decomposition Algorithm" Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation, August 10-12, 1976. +

[Be1] Berlekamp, E. R. "Factoring Polynomials Over Large Finite Fields" - Mathematics of Computation Vol. 24 No. 111 July 1970

[Br1] Brown, W. S. "On Euclid's Algorithm and the Computation of Polynomial Greatest Common Divisors" in [A3]

[Ca1] Caviness,B.F. and R. Fateman, "Simplification of Radical Expressions" Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation, August 10-12, 1976. +

[Co1] Collins, G. E. "The Calculation of Multivariate Polynomial Resultants" in [A3]

[Do] Doohovskoy, Alexander. "Varieties of Operator Manipulation." Proceedings of the MACSYMA Users' Conference, NASA, Berkeley, CA, July 1977.

[Ea1] Eastlake, D., et. al. "ITS 1.5 Reference Manual" A.I. Memo 161A July 1969 *

[Fa1] Fateman, R. J. "The User-Level Semantic Matching Capability in MACSYMA" - in [A1]

[Fa2] --- "Essays in Algebraic Simplification" - Ph.D. Thesis MAC TR-95 April 1972 **

[Fa3] --- "Rationally Simplifying Non-Rational Expressions" SIGSAM Bulletin July 1972 No. 23 *

[Fa4] --- "On the Implementation of Modular Algorithms" SIAM National Meeting. June 1973

[Fa5] --- "On the Computation of Powers of Sparse Polynomials" MIT Studies in Applied Mathematics Vol. 53 No. 2 June 1974 *

[Fa6] --- "Polynomial Multiplication, Powers, and Asymptotic Analysis, Some Comments" - SIAM Journal of Computing Vol. 3 Sept. 1974, pp.196-213.

[Fa7] --- "On the Multiplication of Poisson Series" , Celestial Mechanics, Vol 10, No. 2, Oct. 1974, pp. 243-247.

[Fa8] --- "A Case History in Interactive Problem Solving" SIGSAM Bulletin No. 28, Dec. 1973 +

[Fa9] --- "The MACSYMA Big-Floating-Point Arithmetic System" Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation, August 10-12, 1976. +

[Fa10] --- "An Approach to Automatic Asymptotic Expansions" Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation, August 10-12, 1976. +

[Gene1] Genesereth, Michael R. "The Difficulties of Using MACSYMA and the Function of User Aids." Proceedings of the MACSYMA Users' Conference, NASA, Berkeley, CA, July 1977.

[Gene2] --- "An Automated Consultant for MACSYMA." Proceedings of the MACSYMA Users' Conference, NASA, Berkeley, CA, July 1977.

[Gent1] Gentleman, W.M. and S.C. Johnson, "The Evaluation of Determinants by Expansion by Minors and the General Problem of Substitution" Mathematics of Computation,Volume 28,Number 126,April 1974,pp. 543-548.

[Go1] Golden, Jeffrey P. "MACSYMA's Symbolic Ordinary Differential Equation

Solver." Proceedings of the MACSYMA Users' Conference, NASA, Berkeley, CA, July 1977.

[Go2] --- "The Evaluation of Atomic Variables in MACSYMA." Proceedings of the MACSYMA Users' Conference, NASA, Berkeley, CA, July 1977.

[Kn1] Knuth, D. "Seminumerical Algorithms" in The Art of Computer Programming, Vol. 2 Addison-Wesley Publishers 1969

[Lew1] Lewis, Ellen, "An Introduction to ITS for the MACSYMA User", September, 1977. *

[Lew2] --- "User Aids for MACSYMA." Proceedings of the MACSYMA Users' Conference, NASA, Berkeley, CA, July 1977.

[Ma1] Martin, W. A. "Computer Input/Output of Mathematical Expressions" in [A1]

[Ma2] --- "Determining the Equivalence of Algebraic Expressions by Hash Coding" in [A1, A3]

[Ma3] --- "Symbolic Mathematical Laboratory" - Ph.D. Thesis MAC TR-36 January 1967 AD-657-283 ***

[Ma4] --- and Fateman, R. J. - "The MACSYMA System" in [A1]

[Mn1] Moon, D., et. al "LISP Reference Manual" *

[Mo1] Moses, J. "Algebraic Simplification - A Guide for the Perplexed" - in [A1, A2]

[Mo2] --- "Symbolic Integration - The Stormy Decade" - in [A1, A2]

[Mo3] --- "Towards a General Theory of Special Functions" Communications of the Assocation for Computing Machinery July 1972 Vol. 15 No. 7 +

[Mo4] --- "Symbolic Integration" - Ph.D. Thesis - MAC TR-47 December 1967 AD-662-666 ***

[Mo5] --- "MACSYMA Primer", October 1977 *

[Mo6] --- and Yun, D. Y. "The EZ GCD Algorithm" - Procedings of the ACM convention August 1973. +

[Mo7] --- "The Evolution of Algebraic Manipulation Algorithms" Proceedings of IFIP conference August 1974, Stockholm.

[Mo8] --- "MACSYMA - The Fifth Year" SIGSAM Bulletin - August 1974. *

[Mo9] --- "An Introduction to the Risch Integration Algorithm." Proceedings of the ACM Annual Conference, ACM, Houston, Tx., Oct. 1976.

[Mo10] --- "The Variety of Variables in Mathematical Expressions." Proceedings of the MACSYMA Users' Conference, NASA, Berkeley, CA, July 1977.

[Mo11] --- and Cohen, Jacques. "Summation of Rational Exponential Expressions in Closed Form." Proceedings of the MACSYMA Users' Conference, NASA,

[Os1] Osman, E. "DDT Reference Manual" - A.I. Memo 147A Sept 1971 *

[St1] Steele, Guy L., Jr. "Data Representations in PDP-10 MACLISP." Proceedings of the MACSYMA Users' Conference, NASA, Berkeley, CA, July 1977.

[St2] --- "Fast Arithmetic in MACLISP." Proceedings of the MACSYMA Users' Conference, NASA, Berkeley, CA, July 1977.

[Tr1] Trager,B. "Algebraic Factoring and Rational Function Integration" Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation, August 10-12, 1976. +

[Tr2] --- and D.Y.Y.Yun "Completing nth Powers of Polynomials" Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation, August 10-12, 1976. +

[Wa1] Wang, P. "Automatic Computation of Limits" - in [A1]

[Wa2] --- "Application of MACSYMA to an Asymptotic Expansion Problem" - Proceedings of the ACM Annual Conference August 1972 Vol. 2 +

[Wa3] --- "Evaluation of Definite Integrals by Symbolic Manipulation" - Ph.D. Thesis - MAC TR-92 October 1971 **

[Wa4] --- and Rothschild, L. "Factoring Multivariate Polynomials over the Integers" - Mathematics of Computation, vol. 29 (131) , pp. 935-950. *

[Wa5] Wang,P. "Multivariate Polynomials over Algebraic Number Fields" - Mathematics of Computation, Vol. 30, (134),pp. 324-336.

[Wa6] --- "On the Expansion of Sparse Symbolic Determinants" Proceedings of the International Confference on System Sciences, Jan. 4-6,1977 Honolulu, Hawaii

[Wa7] --- "An Improved Multivariate Polynomial Factoring Algorithm" Mathematics of Computation (to appear)

[Wa8] --- and T. Minamikawa "Taking Advantage of Zero Entries in the Exact Inverse of Sparse Matrices, Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation, August 10-12, 1976. +

[Wa9] --- "Preserving Sparseness in Multivariate Polynomial Factorization." Proceedings of the MACSYMA Users' Conference, NASA, Berkeley, CA, July 1977.

[Wa10] --- "Matrix Computations in MACSYMA." Proceedings of the MACSYMA Users' Conference, NASA, Berkeley, CA, July 1977.

[Wh1] White, Jon L. "Lisp: Program is Data - A Historical Perspective on MACLISP." Proceedings of the MACSYMA Users' Conference, NASA, Berkeley, CA, July 1977.

[Wh2] --- "Lisp: Data is Program - A Tutorial in LISP." Proceedings of the MACSYMA Users' Conference, NASA, Berkeley, CA, July 1977.

[Yu1] Yun, D. Y. "The Hensel Lemma in Symbolic Manipulation" Ph.D. Thesis - MAC TR-138, November 1974 *

[Yu2] --- "On Algorithms for Solving Systems of Polynomial Equations" SIGSAM Bulletin - Sept. 1973 +

[Zi1] Zippel, R. "Power Series Expansions in MACSYMA" Proceedings of the Conference on Mathematical Software II Purdue University, May 29, 1974

[Zi2] --- "Univariate Power Series Expansions in Algebraic Manipulation" Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation, August 10-12, 1976. +

[Zi3] --- "Radical Simplification Made Easy." →Proceedings of the MACSYMA Users' Conference←, NASA, Berkeley, CA, July 1977.

Where to Obtain Publications

* Available from:

Mathlab Group
MIT Laboratory for Computer Science NE43-828
545 Technology Square
Cambridge, Mass. 02139

** Available for a charge from:
MIT Laboratory for Computer Science
Publications - NE43-112
545 Technology Square
Cambridge, Mass. 02139

+ Available from:
The Association for Computing Machinery (ACM)
1133 Avenue of the Americas
New York, N.Y. 10036

Appendix VI - MACSYMA Functions and Argument Evaluation

Most MACSYMA functions, including all user-defined functions,[1] are processed by MACSYMA's evaluator in a straightforward manner: the arguments to the function are evaluated (left-to-right), the function is applied to the evaluated arguments, and then the result is returned.

However, there are two classes of functions which are not subsumed under this simple scheme. In the first class, some or all of the arguments are NOT evaluated. This class includes

```
ALIAS ALLOC APPENDFILE ARRAY BATCH BATCON CLOSEFILE
COMPFILE DECLARE DEFINE DELFILE DEMO DISPFUN DISPLAY
FASSAVE GRADEF KILL LABELS LOADFILE LOCAL MATCHDECLARE
MODEDECLARE PLAYBACK QPUT REMARRAY REMFILE REMFUNCTION
REMOVE REMRULE REMVALUE RESTORE SAVE STORE STRING STRINGOUT
TIME TRACE TRANSLATE UNSTORE UNTRACE WRITEFILE
```

There is another class of functions which *control* the evaluation of their arguments; "control" involves the order of evaluation of the arguments as well as their form. For example, order of evaluation is important in SUM,PROD, and the plotting functions; on the other hand, the EV command takes advantage of the structure of the arguments as well. This class includes

```
CATCH ERRCATCH EV FORTRAN FORTMX FULLMAP FULLMAPL
GRIND PROD SUBSTINPART SUBSTPART SUM
```

and the plotting functions.

In addition, certain MACSYMA "operators" (see Appendix II) do not evaluate some of their arguments, such as ":", ":=", "'". On the other hand, MACSYMA constructs such as BLOCK, the various forms of DO, and logical operators such as AND,OR, IF...THEN...ELSE control the evaluation of their arguments.

1. More flexibility for user-defined functions will soon be available